UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ANÁLISE E OTIMIZAÇÃO DO APACHE HADOOP EM ARQUITETURAS PARALELAS COM MEMÓRIA COMPARTILHADA

TRABALHO DE GRADUAÇÃO

Éder John Scheid

Santa Maria, RS, Brasil

2014

ANÁLISE E OTIMIZAÇÃO DO APACHE HADOOP EM ARQUITETURAS PARALELAS COM MEMÓRIA COMPARTILHADA

Éder John Scheid

Trabalho de Graduação apresentado ao Curso de Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de

Bacharel em Ciência da Computação

Orientadora: Profª. Drª. Andrea Schwertner Charão

389
Santa Maria, RS, Brasil

2014

Universidade Federal de Santa Maria
Centro de Tecnologia
Curso de Ciência da Computação

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Graduação

ANÁLISE E OTIMIZAÇÃO DO APACHE HADOOP EM ARQUITETURAS PARALELAS COM MEMÓRIA COMPARTILHADA

elaborado por
Éder John Scheid

como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação

COMISSÃO EXAMINADORA:

Andrea Schwertner Charão, Drª.
(Presidente/Orientadora)

Guilherme Weigert Cassales, Prof. (UFSM)

João Vicente Ferreira Lima, Prof. Dr. (UFSM)

Santa Maria, 9 de Dezembro de 2014.

AGRADECIMENTOS

Gostaria primeiramente de agradecer à minha família que sempre esteve ao meu lado: ao meu pai Ademar, o qual me incentivou à sempre manter a curiosidade de conhecer como as coisas funcionam e como, eventualmente, consertá-las e também sendo um modelo de pai e homem à ser seguido; à minha mãe Neusa, a qual me fez sempre buscar mais conhecimento e aperfeiçoamento, provendo dicas preciosas para o meu crescimento como pessoa, pelo carinho e pela dádiva da vida.

Ao meu irmão Amir, merecendo um parágrafo novo, que sempre foi um exemplo de dedicação e que também esteve do meu lado em todos os momentos, seja com um café quente ou com uma cerveja gelada.

Aos meus grandes amigos que conheci durante todos estes anos e que ainda hoje fazem parte da minha vida e contribuíram com este trabalho de alguma maneira.

Ao Programa de Educação Tutorial de Ciência da Computação (PET-CC), provendo um ambiente de crescimento pessoal e acadêmico durante os anos de minha participação no programa.

À minha orientadora e coordenadora do curso, professora Andrea Charão, pelos conselhos durante toda a graduação e em especial durante o desenvolvimento deste trabalho.

Ao curso de Ciência da Computação como um todo, corpo docente e funcionários que ajudam na formação de novos cientistas da computação de uma maneira exemplar.

“Many dreams come true and some have silver linings. I live for my dreams and a pocket full of gold.”

— Led Zeppelin, Over The Hills And Far Away

RESUMO

Trabalho de Graduação
Curso de Ciência da Computação
Universidade Federal de Santa Maria

ANÁLISE E OTIMIZAÇÃO DO APACHE HADOOP EM ARQUITETURAS PARALELAS COM MEMÓRIA COMPARTILHADA

AUTOR: ÉDER JOHN SCHEID
ORIENTADORA: ANDREA SCHWERTNER CHARÃO
Local da Defesa e Data: Santa Maria, 9 de Dezembro de 2014.

O processamento de grande volumes de dados foi sempre um obstáculo na computação. O surgimento do paradigma da computação paralela aliado com a ideia de distribuir a computação em diversos computadores ajudou a resolver uma parte considerável deste obstáculo. Muitos *frameworks* foram criados baseados nessa premissa, um deles é o *framework* Apache Hadoop. Voltado para ambientes onde os dados estão distribuídos entre vários computadores, o Apache Hadoop, oferece uma ótima solução para o processamento de *big data*, mas a literatura sobre como este *framework* se comporta em um ambiente onde os dados estão alocados em um único local ainda é pequena. O foco deste trabalho é analisar e otimizar este *framework* em uma arquitetura paralela onde os dados não estão distribuídos, podendo assim conseguir resultados que demonstrem qual sua eficiência neste ambiente.

Palavras-chave: Apache Hadoop. memória compartilhada. máquina NUMA.

ABSTRACT

Undergraduate Final Work
Undergraduate Program in Computer Science
Federal University of Santa Maria

ANALYSIS AND OPTIMIZATION OF THE APACHE HADOOP IN PARALLEL ARCHITECTURES WITH SHARED MEMORY

AUTHOR: ÉDER JOHN SCHEID
ADVISOR: ANDREA SCHWERTNER CHARÃO
Defense Place and Date: Santa Maria, December 9th, 2014.

The act of processing large volumes of data has always been an obstacle in computing. The emergence of the paradigm of parallel computing combined with the idea of distributing the computation across multiple computers helped to solve a considerable part of this obstacle. Many *frameworks* have been created based on this premise, one of them is the Apache Hadoop *framework*. Aiming environments where the data is distributed among several computers, the Apache Hadoop provides an optimal solution for processing *big data*, but the literature on how this *framework* behaves in an environment where the data is allocated on a single machine is still small. The focus of this work is to analyze and optimize this *framework* in a paralel architecture where the data is not distributed, and thus achieving results that demonstrates what is its efficiency under those circumstances.

Keywords: Apache Hadoop. shared memory. NUMA machine.

LISTA DE FIGURAS

Figura 2.1 – Arquitetura simplificada de uma máquina NUMA (Database Technology Group, Technische Universität Dresden, 2011) 15
Figura 2.2 – Arquitetura mestre/escravo HDFS (Apache Software Foundation, 2014a) ... 17
Figura 2.3 – Arquitetura do YARN (Apache Software Foundation, 2014b) 17
Figura 3.1 – Nós principais (0 e 1) da SGI UV2000 21
Figura 3.2 – Gráfico do Número de Pontos por Tempo Levado para execução no Hadoop 2.5.0 Standalone 28
Figura 3.3 – Comparação entre um *map* executando no Hadoop Standalone e um executando no Hadoop Pseudo-Distribuído 28
Figura 3.4 – Diferença de tempo entre os algoritmos CapacityScheduler e FairScheduler (PiEstimator com até 48 *mappers*). 30
Figura 3.5 – Diferença de tempo entre os algoritmos CapacityScheduler e FairScheduler (PiEstimator com até 500 *mappers*). 31
Figura 3.6 – Aumento no desempenho do algoritmo PiEstimator entre o Hadoop configurado para 48 *containers* e 3 *containers*. 34
Figura 3.7 – Aumento no desempenho do algoritmo TeraSort ordenando 5Gb entre o Hadoop configurado para 48 *containers* e 3 *containers*. 37
Figura 3.8 – Aumento no desempenho do algoritmo TeraSort ordenando 10Gb entre o Hadoop configurado para 48 *containers* e 3 *containers*. 37

LISTA DE TABELAS

Tabela 3.1 – Resultados do algoritmo PiEstimator executando no Hadoop 2.5.0 em modo *standalone* 25
Tabela 3.2 – Configurações do Hadoop geradas pelo Script (Anexo B) 26
Tabela 3.3 – Resultados do algoritmo PiEstimator executando no Hadoop 2.5.0 em modo Pseudo-Distribuído 27
Tabela 3.4 – Configurações Otimizadas do Hadoop para gerar 48 *containers* de 10 *gigabytes* 32
Tabela 3.5 – Resultados do algoritmo PiEstimator executando no Hadoop 2.5.0 Otimizado 33
Tabela 3.6 – Resultados do algoritmo TeraSort Ordenando 5Gb com a Configuração gerada pelo Script (Anexo B) 35
Tabela 3.7 – Resultados do algoritmo TeraSort Ordenando 5Gb com a Configuração Otimizada 35
Tabela 3.8 – Resultados do algoritmo TeraSort Ordenando 10Gb com a Configuração gerada pelo Script (Anexo B) 36
Tabela 3.9 – Resultados do algoritmo TeraSort Ordenando 10Gb com a Configuração Otimizada 36

LISTA DE ANEXOS

ANEXO A – Algortimo PiEstimator 44
ANEXO B – Script provido pelo site HortonWorks (Hortonworks, Inc., 2013) para configuração do *Apache Hadoop* 51

SUMÁRIO

1 INTRODUÇÃO 12
1.1 Objetivos 13
1.2 Justificativa 13
2 FUNDAMENTOS E REVISÃO DE LITERATURA 15
2.1 Arquitetura NUMA 15
2.2 Apache Hadoop 16
2.2.1 Arquitetura do *Apache Hadoop* 16
2.2.2 *Paradigma MapReduce* 17
2.3 Trabalhos Relacionados 18
3 DESENVOLVIMENTO 20
3.1 Máquina SGI UV2000 20
3.2 Configuração do *Apache Hadoop* 21
3.3 Algoritmos Usados 22
3.3.1 PiEstimator 22
3.3.2 TeraSort 23
3.4 Experimentos e Resultados Iniciais 23
3.4.1 Hadoop *Standalone* 24
3.4.2 Hadoop em modo Pseudo-Distribuído 25
3.4.3 Comparação e discussão dos resultados 27
3.5 Otimização da Configuração 29
3.5.1 Escalonadores (*Schedulers*) 29
3.5.1.1 CapacityScheduler 29
3.5.1.2 FairScheduler 29
3.5.2 Número de *containers* e Memória Alocada 31
3.6 Experimentos com o Hadoop Otimizado 32
3.6.1 Discussão com Algoritmo PiEstimator 32
3.6.2 Discussão com Algoritmo TeraSort 34
3.6.2.1 Resultados com ordenação de 5GB de dados 34
3.6.2.2 Resultados com ordenação de 10GB de dados 35
3.6.2.3 Comparação dos Resultados 36
4 CONCLUSÃO 39
REFERÊNCIAS 40
ANEXOS 43

1 INTRODUÇÃO

Durante os últimos anos a computação foi marcada pelo termo *big data*, que representa o crescimento exponencial de dados e a disponibilidade dos mesmos (SAS Institute Inc., 2014). Com este crescimento a necessidade de um poder computacional capaz de processar esta quantidade de dados também foi crescendo, surgindo assim *clusters* de computadores capazes de completar esta tarefa. Algoritmos tradicionais de agrupamento e ordenação tiveram de ser adaptados para usarem toda a capacidade deste *cluster*, gerando assim uma vasta comunidade especializada nisto.

Paralelamente a isto, sistemas computacionais também tiveram um crescimento no poder computacional e uma redução de preços. Tarefas que antes eram processadas em um *cluster* composto por 10 máquinas hoje podem ser processadas em apenas um servidor, possivelmente, custando a metade do preço. Um exemplo destes servidores, são máquinas de memória compartilhada com arquitetura NUMA (non-uniform memory access), estas estão ganhando espaço no setor de processamento de grande volumes de dados e a atenção da comunidade científica, nos últimos anos (Bligh, Martin J. and Dobson, Matt and Hart, Darren, 2004). Logo, a premissa que os dados precisam estar distribuídos entre vários computadores para se obter um tempo de execução aceitável, precisa ser revista (Kumar, K. Ashwin and Gluck, Jonathan and Deshpande, Amol and Lin, Jimmy, 2014).

A solução atualmente para qualquer processamento de grandes volumes de dados é montar justamente um *cluster* composto por vários servidores, mas muitas vezes o volume de dados pode ser processado em um único servidor. Tendo apenas um servidor para realizar estas tarefas, corta-se custos e mão de obra. Mas e como utilizar esta tecnologia que tem foco em *clusters*, como o *framework Apache Hadoop* (Apache Hadoop, 2014), em apenas uma máquina?

Ao responder esta pergunta encontramos problemas como a falta de dados sobre como estes *frameworks*, voltados para *clusters*, se comportam em apenas uma máquina, visto que ainda é uma área em exploração. Portanto, avaliar como estes *frameworks*, mais precisamente o *Apache Hadoop*, trabalham com dados que não estão distribuídos se demonstra muito viável e passível de investigação sobre seu desempenho nestas circunstâncias, podendo assim contribuir com um ambiente de processamento estável e simples.

1.1 Objetivos

Os objetivos gerais deste trabalho são analisar e otimizar o *framework Apache Hadoop* em um ambiente para o qual o mesmo não foi projetado, ou seja em uma máquina NUMA (Non-uniform memory access), que apresenta uma arquitetura paralela.

1.2 Justificativa

Para o processamento de grandes volumes de dados, atualmente usa-se o artifício de alocar diversas máquinas, um *cluster*, com poder computacional para realizar determinada tarefa. Com isto geram-se diversos problemas, como a manutenção de cada nó (máquina) e gerenciamento do sistema. Ainda podemos citar o problema da necessidade do desenvolvimento de aplicações paralelas voltadas para estes sistemas, uma tarefa que exige um esforço maior por parte do programador (Carissimi, Alexandre and Dupros, Fabrice and Méhaut, Jean-François and Polanczyk, Rafael Vanoni, 2014).

Máquinas NUMA aproveitam o fato que processadores *multicore* vêm se popularizando cada vez mais e se tornam uma alternativa para o processamento de alto desempenho em uma única máquina, sem criar os problemas citados. Adicionalmente, no sistema de armazenamento de dados de uma máquina NUMA todos os dados são armazenados em um só local, sem a necessidade de transferência pela rede.

Paralelamente, com a capacidade de processamento e de memória dos servidores atuais, a premissa de que é necessário mais de um servidor para o processamento de tarefas que demandam relativamente maior tempo computacional é passível de revisão. Por exemplo em 2008 um nó de um servidor *Apache Hadoop* era constituído por dois processadores dual-core e 4 GB de RAM. Hoje em dia, um único servidor pode ter dois processadores com oito núcleos cada e 256 GB de RAM, logo um único servidor atual possui mais núcleos e memória do que um *cluster* em 2008 (Kumar, K. Ashwin and Gluck, Jonathan and Deshpande, Amol and Lin, Jimmy, 2014).

Além disto, com o aumento no interesse da comunidade científica em *mineração de dados* e *aprendizado de máquina* os dados a serem processados estão se tornando cada vez mais refinados, visto que uma vez estes são derivados de buscas mais específicas e com isto o tamanho destes dados também diminui, contribuindo para que os mesmos possam ser processados em um único servidor.

A proposta do presente trabalho é relevante pois experimenta o uso de uma tecnologia (*Apache Hadoop*) projetada para um ambiente específico (*cluster* de computadores) em um ambiente totalmente diferente do seu propósito (única máquina NUMA), podendo assim gerar resultados satisfatórios e contribuir para a comunidade com dados e análises e ainda otimizar a tecnologia para o uso no ambiente em questão.

2 FUNDAMENTOS E REVISÃO DE LITERATURA

O presente capítulo é destinado à definição de conceitos teóricos sobre as ferramentas e paradigmas que foram utilizados no trabalho, sendo eles: *Arquitetura NUMA*, *Framework Apache Hadoop*, paradigma de MapReduce, assim como trabalhos relacionados.

2.1 Arquitetura NUMA

A sigla NUMA significa, em inglês, *non-uniform memory access*, isto significa que o tempo de acesso a memória depende da distância na qual ela esta do processador. Cada processador possui sua própria memória. Em contrapartida, o acesso à memória de outros processadores tende a demorar, devido à latência do acesso remoto. A programação para estas máquinas geralmente é focada no princípio da localidade, usando recursos próximos ao sistema e evitando o tráfego entre nós (Bligh, Martin J. and Dobson, Matt and Hart, Darren, 2004). A Figura 2.1 apresenta a arquitetura simplificada de uma máquina com esta arquitetura.

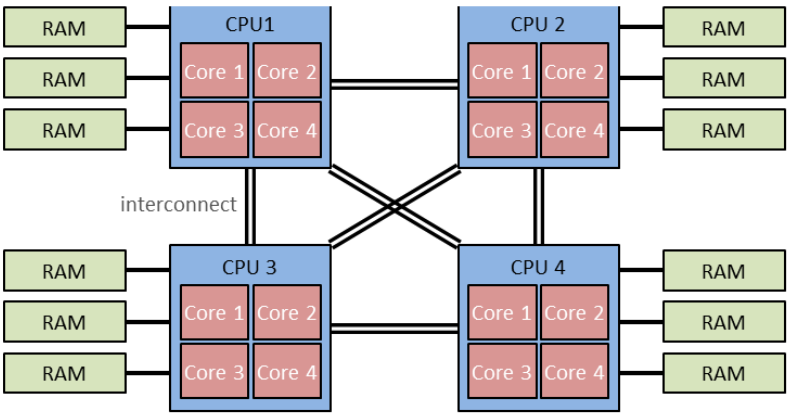

Figura 2.1 – Arquitetura simplificada de uma máquina NUMA (Database Technology Group, Technische Universität Dresden, 2011)

A Silicon Graphics International (SGI) é uma empresa especializada no desenvolvimento de computadores de alto desempenho desde a década de 90 (Silicon Graphics International Corp, 2014a). Dentro dos diversos produtos oferecidos pela empresa, destaca-se a plataforma SGI UV (Silicon Graphics International Corp, 2014b), na qual é possível gerenciar até 2.048 núcleos e operar com até 64 *terabytes* de memória, graças a arquitetura NUMA. A

Informática da UFSM visando projetos científicos, aprendizado e ensino sobre este tipo de máquina, adquiriu uma SGI UV2000 com 8 nós NUMA, cada um com 6 núcleos, totalizando 48 núcleos e com um total de 496 *gigabytes* de memória RAM. O diferencial desta máquina, além da arquitetura, é a capacidade de armazenamento em disco com 1 *terabyte*, uma capacidade relativamente pequena comparado a *clusters* atuais.

2.2 Apache Hadoop

O *framework Apache Hadoop* foi desenvolvido pela Yahoo! (Yahoo, 2014) na sua essência para apoiar outro projeto da mesma empresa, o *Nutch* (Apache Nutch™, 2014), um motor de busca. A necessidade de processamento de muitas páginas Web fez com que os criadores do *Nutch* buscassem alternativas para a paralelização do processamento destes dados, como o paradigma de *MapReduce*.

2.2.1 Arquitetura do *Apache Hadoop*

A arquitetura do *Apache Hadoop* consiste em duas partes, o sistema de arquivos, chamado de *Hadoop Distributed File System* (HDFS) (Shvachko, Konstantin et al., 2010) e a parte que cuida da execução das aplicações escritas no paradigma *MapReduce*, esta parte sendo chamada de *Yet Another Resource Negotiator* (YARN) (Vavilapalli, Vinod Kumar et al., 2013).

O sistema de arquivos desenvolvido para uso com o Hadoop (HDFS) consiste em uma arquitetura mestre/escravo (Figura 2.2), sendo o nó mestre chamado de *NameNode*. Este nó (*NameNode*) é responsável por gerenciar todo o acesso aos dados e armazenar informações sobre onde estes dados estão alocados, é possível existir um *NameNode* por *cluster*. Os nós escravos são chamados de *DataNodes*, estes são responsáveis por armazenar os dados propriamente ditos e provém chamadas de leitura de escrita dentro dos mesmos. Além disto, os *DataNodes* realizam a criação, remoção e replicação caso ordenados pelo *NameNode*.

Para o gerenciamento de recursos do *cluster* foi desenvolvido um *framework* (YARN). A ideia por trás do YARN é existir uma aplicação que gerencia os recursos globais do cluster (ResourceManager) e outra para cada nó do *cluster* (NodeManager), estes se comunicam enviando sinais sobre o estado do nó. Para cada aplicação é criado um gerenciador chamado de ApplicationMaster, este é responsável por aceitar tarefas e delegar estas para os chamados *Containers*, que são recursos (CPU, memória, discos, rede) gerenciados pelos NodeManagers.

A Figura 2.3 apresenta como estas aplicações se comunicam.

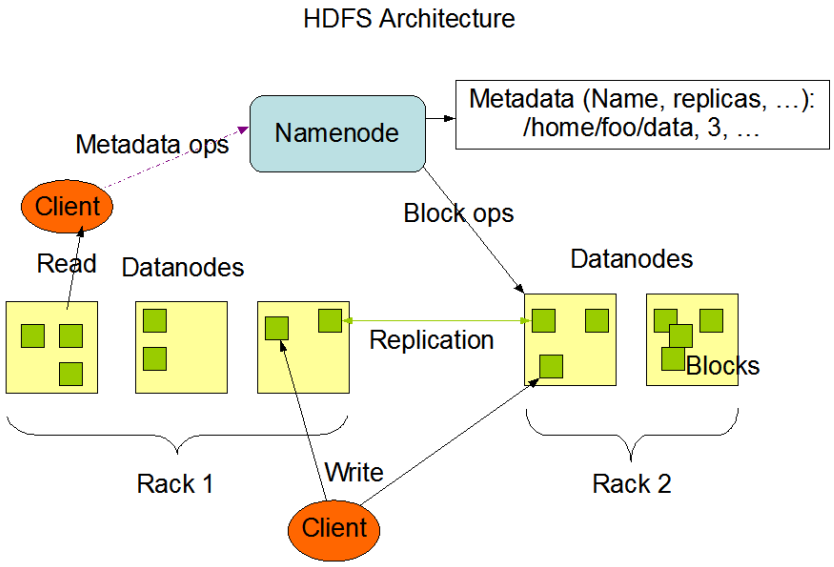

Figura 2.2 – Arquitetura mestre/escravo HDFS (Apache Software Foundation, 2014a)

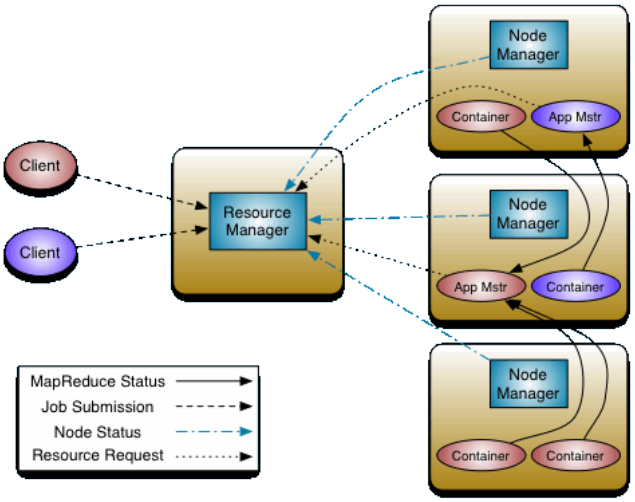

Figura 2.3 – Arquitetura do YARN (Apache Software Foundation, 2014b)

2.2.2 *Paradigma MapReduce*

O paradigma de MapReduce foi originalmente proposto pela Google (Dean, Jeffrey and Ghemawat, Sanjay, 2008) para ambientes que são compostos por várias máquinas interligadas, ou seja, *clusters*. Este paradigma se mostrou simples e eficaz no quesito de processamento de grandes volumes de dados em paralelo. A lógica por trás do *MapReduce* consiste em duas funções escritas pelo programador, *map* e *reduce*, a primeira, *map*, deve produzir um par <*chave, valor*> que será agrupado com outros pares para formar um par <*chave, lista de valores*>, que

posteriormente será processado pela função *reduce*. O que torna este processamento rápido é a característica de independência entre as várias instâncias das mesmas funções, ou seja não existe espera no término de outras funções, possibilitando a paralelização das mesmas.

2.3 Trabalhos Relacionados

Ao longo do desenvolvimento deste trabalho foi feita uma pesquisa bibliográfica com o propósito de estudar o que a comunidade acadêmica havia desenvolvido envolvendo máquinas de memória compartilhada e processadores multi-core com o *Hadoop*. Buscou-se levantar qual é o estado da arte neste tipo de integração. Foram encontrados trabalhos que seguem esta linha de investigação, contribuindo para o aprendizado sobre o comportamento do *Hadoop* em um ambiente não-nativo.

Hone (*Hadoop One*) (KUMAR et al., 2013), a proposta dos autores deste trabalho é refatorar o *Apache Hadoop* para que o mesmo tenha um comportamento eficiente em máquinas multi-core com memória compartilhada. Para isto foi criado um protótipo de *runtime* (Hone), com foco em manter os algoritmos desenvolvidos para o paradigma de *MapReduce* sem nenhuma alteração, mas com um desempenho relativamente melhor neste tipo de ambiente. A diferença do Hone para o presente trabalho se encontra na máquina usada para experimentos e desenvolvimento, bem como no foco do desenvolver uma API para melhorar o desempenho do *Apache Hadoop* nestes casos.

Phonenix (The Phoenix System) (Yoo, Richard M. and Romano, Anthony and Kozyrakis, Christos, 2009), o objetivo deste trabalho é implementar uma versão do paradigma de *MapReduce* em sistemas de memória compartilhada na qual o programador não tem a necessidade de se preocupar com a gerência de concorrência. Foi desenvolvida uma API na qual toda a parte de paralelização, gerenciamento de recursos e recuperação de falha é feita pela API. Neste caso o *Hadoop* não foi utilizado para experimentos ou desenvolvimento, mas o fato do trabalho explorar o paradigma *MapReduce* o torna válido.

M3R (Main Memory Map Reduce) (Shinnar, Avraham and Cunningham, David and Saraswat, Vijay and Herta, Benjamin, 2012), é uma nova implementação da API do *MapReduce* do *Apache Hadoop*. Esta implementação tem como alvo a análise online de *clusters* onde os dados conseguem ocupar a memória RAM total do *cluster*. A API permite que os *jobs* do tipo *Hadoop MapReduce* executem sem nenhuma alteração estrutural, fazendo com que o desempenho no ambiente *Apache Hadoop* não seja afetado.

HJ-Hadoop (*HabaneroJava-Hadoop*) (Zhang, Yunming, 2013), a justificativa deste trabalho é baseada na premissa de que para aplicações com uso intensivo de memória executando no Hadoop, a máquina virtual Java acaba por duplicar estruturas, ambas estáticas e dinâmicas, quando a mesma executa funções de *MapReduce*. Neste sentindo o HJ-Hadoop visa paralelizar as próprias funções *MapReduce* dentro das mesmas, evitando assim a criação de novas funções *MapReduce*.

Ao final do estudo destes trabalhos, percebemos que existe um aumento no desempenho do *Apache Hadoop* em máquinas multi-core e/ou com memória compartilhada. Mas, devemos frisar que este aumento apenas foi atingido para alguns tipos de dados e dentro de uma faixa de tamanho, nenhuma das implementações oferece uma solução genérica eficaz, isto devido a peculiaridade das máquinas.

Podemos notar, também, que a bibliografia sobre como o *Apache Hadoop* se comporta em ambientes com memória compartilhada e multi-core, bem como em máquinas com arquitetura NUMA, é relativamente pequena, confirmando assim que esta área carece de investigação.

3 DESENVOLVIMENTO

Este capítulo está voltado para descrever toda sequência de passos que foram necessários para atingir os objetivos deste trabalho. Como a natureza do trabalho é experimental, ou seja, determinar como o *Apache Hadoop* se comporta em um sistema com uma arquitetura NUMA, este capítulo foi dividido em seis sessões que visam corroborar com as decisões tomadas ao final dos experimentos. A primeira sessão apresenta a máquina (SGI UV2000) com todos os detalhes do *hardware* presente na máquina e a diferença desta para um *cluster* comum. A segunda sessão foca em mostrar a configuração do *framework* Hadoop (2.5.0 YARN) na máquina em questão (SGI UV2000), mostrando quais dependências foram necessárias para o funcionamento correto e qual a expectativa do desempenho. Os algoritmos que foram escolhidos para a realização dos experimentos estão resumidos na terceira sessão. Partindo para uma abordagem mais prática, a quarta sessão apresenta resultados obtidos a partir de experimentos inicias feitos em duas configurações do Hadoop, a primeira sendo sem nenhuma alteração nos parâmetros e outra com mudanças na utilização da memória RAM pelo mesmo. A quinta sessão apresenta considerações e configurações na qual, em teoria, podem levar a um aumento no desempenho. Finalizando com a sexta sessão que discute os resultados dos experimentos realizados com o Hadoop otimizado executando os algoritmos apresentados.

3.1 Máquina SGI UV2000

A máquina SGI UV 2000 na qual todos os experimentos deste trabalho foram conduzidos apresenta 4 blades com arquitetura *Non-Uniform Memory Access*, isto significa que cada nó (processador) possui seu nó de memória. Para a abstração do *hardware* foi usado o software *hwloc* (Open MPI, 2014), que recria a topologia hierárquica da arquitetura da máquina, todas as figuras apresentadas nesta sessão foram produzidas pelo *hwloc*. A Figura 3.1 mostra os nós número 0 e número 1 da máquina, cada um deles possui 62 *gigabytes* de memória RAM, formando um grupo com 124 *gigabytes* de RAM, o nó número 0 é responsável por gerenciar a rede (eth0 e eth1) e outros dois barramentos PCI, já o nó número 1 gerencia o disco (sda). Os outros nós (2,3,4,5,6,7) são apenas processadores dedicados.

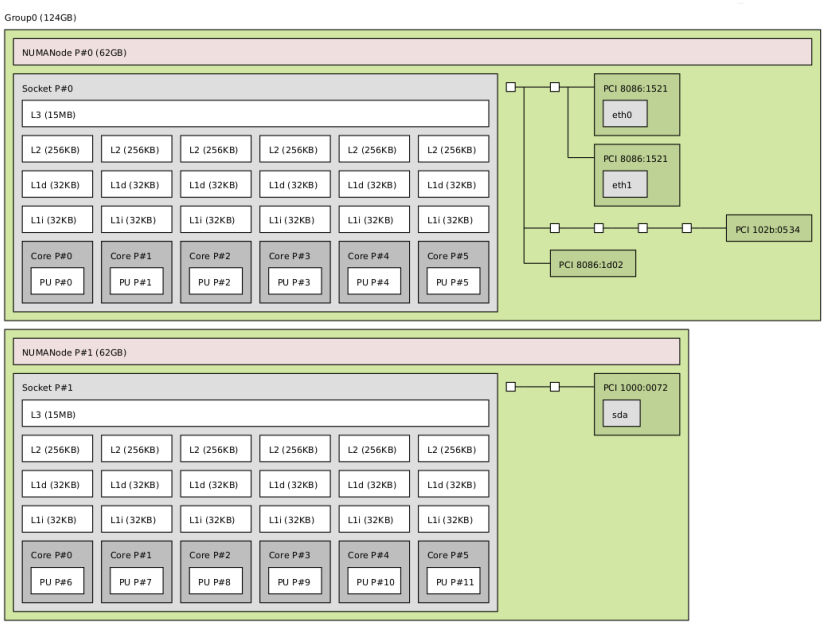

Figura 3.1 – Nós principais (0 e 1) da SGI UV2000

Para ilustrar o fator NUMA da máquina, ou seja, o gasto que um processador tem para acessar a memória de outro processador, foi usado dois algoritmos de *benchmark* sendo eles, *lmbench* (McVoy, Larry and Staelin, Carl, 2014) e *STREAM benchmark* (McCalpin, John D., 2014). Ao executar o algoritmo *lmbench*, que mede a latência do acesso remoto à memória de um processador para outro, o fator NUMA calculado foi de 7.52632. Já para o algoritmo *STREAM benchmark*, o qual calcula o fator NUMA em relação à banda de memória, o valor máximo de gasto foi de 3.79569.

3.2 Configuração do *Apache Hadoop*

A versão do *framework Apache Hadoop* escolhida para o desenvolvimento deste trabalho foi a versão Hadoop 2.5.0 YARN. A escolha da versão foi baseada no estado da arte do *framework*, sendo que a versão (2.5.0) até o momento (Agosto de 2014) era a mais nova.

Durante o desenvolvimento deste trabalho foram alterados alguns parâmetros do *Apache Hadoop*. Os arquivos e seus respectivos parâmetros alterados estão brevemente explicados abaixo.

O arquivo *mapred-site.xml* é responsável por armazenar as configurações sobre a parte encarregada do *MapReduce*. Alguns dos parâmetros alterados foram:

- *mapreduce.map.memory.mb*: Configura o limite de memória para tarefas *map*.

- *mapreduce.reduce.memory.mb*: Altera o limite de memória para tarefas *reduce*.
- *mapreduce.map.java.opts*: Tamanho da *heap* para as máquinas virtuais Java dos filhos de tarefas *map*.
- *mapreduce.reduce.java.opts*: Tamanho da *heap* para as máquinas virtuais Java dos filhos de tarefas *reduce*.
- *mapreduce.task.io.sort.mb*: Configura o limite de memória para ordenar os dados.

Já o arquivo *yarn-site.xml* configura os recursos disponíveis no *cluster* através do YARN. Os valores que passarem dos valores configurados não serão levados em consideração pelo *framework*. Para configurar estes recursos, foram alterados os seguintes parâmetros:

- *yarn.scheduler.minimum.allocation-mb*: Alocação mínima para cada pedido de criação de um *container*.
- *yarn.scheduler.maximum.allocation-mb*: Alocação máxima para cada pedido de criação de um *container*.
- *yarn.nodemanager.resource.memory-mb*: Quantidade de memória física disponível para o *NodeManager* alocar *containers*.
- *yarn.app.mapreduce.am.resource.mb*: Quantidade de memória física alocada para o *ApplicationMaster*.
- *yarn.app.mapreduce.am.command-opts*: Memória *heap* para as máquinas virtuais Java do *AplicationMaster*.

3.3 Algoritmos Usados

A escolha dos algoritmos foi baseada nas peculiaridades da máquina, como por exemplo, no poder computacional relativamente maior em comparação à máquinas convencionais, assim como na grande quantidade de memória RAM e pouco espaço em disco.

3.3.1 PiEstimator

Para os primeiros experimentos, foi executado o algoritmo de exemplo PiEstimator (The Apache Software Foundation, 2009), que aplica o método de quasi-Monte Carlo (Caflish, Rus-

sel E., 1998). O algoritmo PiEstimator recebe dois argumentos, o primeiro sendo o número de tarefas *map* que serão iniciadas e o segundo o número de pontos utilizados para estimar o valor do Pi. A escolha deste algoritmo se deve ao fato de que não existe um uso extensivo de disco para calcular o valor do Pi, mas sim um uso maior de memória e processamento destes dados.

3.3.2 TeraSort

O algoritmo TeraSort (Apache Software Foundation, 2014c) é um dos algoritmos usados para testar o desempenho do *Apache Hadoop* tanto no quesito de processamento (tarefas *map* e *reduce*), quanto no quesito de sistema de arquivos (*HDFS*). A meta deste algoritmo é ordenar 1 *terabyte* de dados no menor tempo possível. No ano de 2008, Owen O'Malley quebrou o recorde da época ao ordenar 1 *terabyte* de dados em 209 segundos, usando um *cluster* composto de 910 nós (O'Malley, Owen, 2008), hoje em dia existem competições que se tratam apenas disto (Chris Nyberg, 2014). O conjunto de *benchmark* combina três algoritmos:

- TeraGen: usado para gerar um conjunto de dados randômico, com tamanho informado pelo usuário.
- TeraSort: o algoritmo *MapReduce* de *benchmark* que ordena os dados gerados pelo algoritmo TeraGen.
- TeraValidate: usado para validar se os dados ordenados.

A ordem de execução é a mesma da ordem dos item acima. Primeiro cria-se os dados, na qual não é necessário ser 1 *terabyte*, este algoritmo é executado apenas uma vez, pois os dados são os mesmos para todos os experimentos. Após os dados serem criados, o algoritmo de ordenação TeraSort é executado, gerando assim um tempo de execução para tal configuração. Podemos validar os dados, usando o TeraValidate, mas partimos do princípio que os dados estão ordenados corretamente, evitando um gasto relativamente maior de tempo.

3.4 Experimentos e Resultados Iniciais

Para termos uma base de comparação inicial, foi escolhido usar o *framework* Hadoop na sua configurações *default*, ou seja, sem nenhuma alteração e no modo *standalone*, na qual nem o sistema de arquivos do Hadoop (HDFS) ou o gerenciador de recursos (YARN) foram configurados. Além disto foram feitos experimentos com o Hadoop configurado para usar o

HDFS e o YARN, como também configurado para usar 80% da memória disponível no sistema no caso 397 *gigabytes*.

O algoritmo utilizado nos experimentos iniciais foi o PiEstimator, visando apresentar resultados mais rapidamente para uma análise mais rasa. Foram feitos, para cada configuração, um total de 25 experimentos com número de pontos aumentando gradualmente, como também o número de *maps*. O número de pontos variou de 1 milhão de pontos até 10 bilhões de pontos, já o número de *maps* variou de apenas 1 *mapper* até 48 (número de cores) *mappers*.

3.4.1 Hadoop *Standalone*

Neste modo o *Hadoop* executa como um simples processo Java, em modo não distribuído. O sistema de disco local é usado, não há replicação de dados e não existe nenhum outro processo gerenciando todos os componentes do *framework*. Nenhum dos arquivos de configuração do Hadoop (*mapred-site.xml*, *yarn-site.xml* e *hdfs-site.xml*) foram alterados, garantindo uma base para comparações.

A Tabela 3.1 mostra os resultados obtidos em todos os experimentos nesta configuração.

Tabela 3.1 – Resultados do algoritmo PiEstimator executando no Hadoop 2.5.0 em modo *standalone*

Número de *mappers*	Número de Pontos	Tempo Decorrido (segundos)	Valor do PI
1	1,000,000	1.618	3.14155200000000
8	1,000,000	2.688	3.14159800000000
16	1,000,000	2.707	3.14159125000000
32	1,000,000	4.868	3.14158612500000
48	1,000,000	5.894	3.14159633333333
1	10,000,000	1.621	3.14158440000000
8	10,000,000	3.692	3.14159445000000
16	10,000,000	5.728	3.14159155000000
32	10,000,000	9.914	3.14159237500000
48	10,000,000	13.973	3.14159285000000
1	100,000,000	3.636	3.14159256000000
8	100,000,000	17.044	3.14159287000000
16	100,000,000	33.21	3.14159239750000
32	100,000,000	65.65	3.14159270625000
48	100,000,000	99.403	3.14159282083333
1	1,000,000,000	22.712	3.14159272000000
8	1,000,000,000	150.772	3.14159279700000
16	1,000,000,000	296.938	3.14159270875000
32	1,000,000,000	564.268	3.14159267450000
48	1,000,000,000	845.148	3.14159267166667
1	10,000,000,000	209.901	3.14159267760000
8	10,000,000,000	1462.126	3.14159265100000
16	10,000,000,000	2896.077	3.14159265117500
32	10,000,000,000	5696.377	3.14159264358750
48	10,000,000,000	8354.487	3.14159264505833

3.4.2 Hadoop em modo Pseudo-Distribuído

Neste experimento, o *framework* Hadoop simula um processamento distribuído em uma única máquina fazendo com que cada parte do Hadoop execute em um processo Java diferente. Foram configurados os seguintes arquivos: *core-site.xml*, *hdfs-site.xml*, *mapred-site.xml* e *yarn-site.xml*. Para a decisão de quais propriedades do *framework* deveriam ser configuradas para que o Hadoop executasse com 80% da memória física disponível na máquina, foi usado um *script* (Anexo B) escrito em Python baseado nos cálculos apresentados no website Hortonworks (Hortonworks, Inc., 2013) em seu manual de instalação do Hadoop. Na Tabela 3.2 estão as propriedades e seus respectivos valores gerados pelo script.

Tabela 3.2 – Configurações do Hadoop geradas pelo Script (Anexo B)

Propriedade	Valor
yarn.scheduler.minimum-allocation-mb	135168
yarn.scheduler.maximum-allocation-mb	405504
yarn.nodemanager.resource.memory-mb	405504
mapreduce.map.memory.mb	135168
mapreduce.map.java.opts	-Xmx108134m
mapreduce.reduce.memory.mb	135168
mapreduce.reduce.java.opts	-Xmx108134m
yarn.app.mapreduce.am.resource.mb	135168
yarn.app.mapreduce.am.command-opts	-Xmx108134m
mapreduce.task.io.sort.mb	54067

Após a configuração do Hadoop e seus arquivos de propriedades foi realizado os mesmos experimentos na qual o ambiente *standalone* foi submetido, os resultados obtidos estão descritos na Tabela 3.3.

Tabela 3.3 – Resultados do algoritmo PiEstimator executando no Hadoop 2.5.0 em modo Pseudo-Distribuído

Número de *mappers*	Número de Pontos	Tempo Decorrido (segundos)	Valor do PI
1	1,000,000	17.522	3.14155200000000
8	1,000,000	35.732	3.14159800000000
16	1,000,000	66.137	3.14159125000000
32	1,000,000	126.905	3.14158612500000
48	1,000,000	187.605	3.14159633333333
1	10,000,000	21.529	3.14158440000000
8	10,000,000	39.803	3.14159445000000
16	10,000,000	70.198	3.14159155000000
32	10,000,000	130.173	3.14159237500000
48	10,000,000	190.617	3.14159285000000
1	100,000,000	22.518	3.14159256000000
8	100,000,000	51.949	3.14159287000000
16	100,000,000	94.5	3.14159239750000
32	100,000,000	180.294	3.14159270625000
48	100,000,000	263.594	3.14159282083333
1	1,000,000,000	40.701	3.14159272000000
8	1,000,000,000	162.214	3.14159279700000
16	1,000,000,000	316.91	3.14159270875000
32	1,000,000,000	616.279	3.14159267450000
48	1,000,000,000	916.692	3.14159267166667
1	10,000,000,000	219.586	3.14159267760000
8	10,000,000,000	1241.763	3.14159265100000
16	10,000,000,000	2486.278	3.14159265117500
32	10,000,000,000	4932.161	3.14159264358750
48	10,000,000,000	7401.152	3.14159264505833

3.4.3 Comparação e discussão dos resultados

A Figura 3.2, mostra o gráfico resultante dos experimentos com o Hadoop operando em modo *standalone*. Podemos notar que quanto mais pontos e quanto mais tarefas *map* mais tempo a execução do algoritmo leva. Cada linha representa o tempo de execução de diferentes quantidades de *mappers*, respectivamente 1,8,16,32 e 48. A taxa de crescimento do tempo de execução é a mesma para todas as quantidades de tarefas *map*, o único experimento que difere é o experimento na qual temos apenas 1 *map* executando, onde a diferença entre os tempos de execução para 1 milhão, 10 milhões e 100 milhões de pontos é mínima. Nas outras, quando dobramos o número de *maps* o tempo de execução chega a quase o dobro. Uma outra característica só se apresenta com números de pontos maiores, por exemplo, todos os algoritmos que executaram com 10 bilhões de pontos, demoraram aproximadamente 10 vezes menos do

que quando executaram com 1 bilhão de pontos. O padrão deve seguir se duplicarmos o número de *maps* e aumentarmos a quantidade de pontos em 10 vezes, pois uma única máquina Java cuidará de todos estas tarefas *maps*, mostrando assim um comportamento não paralelo.

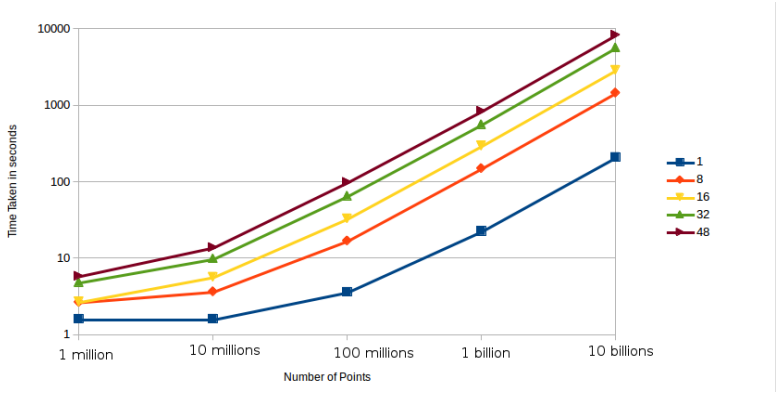

Figura 3.2 – Gráfico do Número de Pontos por Tempo Levado para execução no Hadoop 2.5.0 Standalone

Quando comparamos a execução de apenas uma tarefa *map* entre o Hadoop Standalone e o Hadoop em modo Pseudo-Distribuído (Figura 3.3), notamos que o tempo de execução com 10 bilhões de pontos é praticamente o mesmo. Mas nos outros pontos temos a diferença de aproximadamente 18 segundos. Esta diferença pode se dar por diversos fatores, como o coletor de lixo do Java tentando limpar a memória ou o tempo que o gerenciador de recursos demora para alocar as diversas estruturas de dados necessárias.

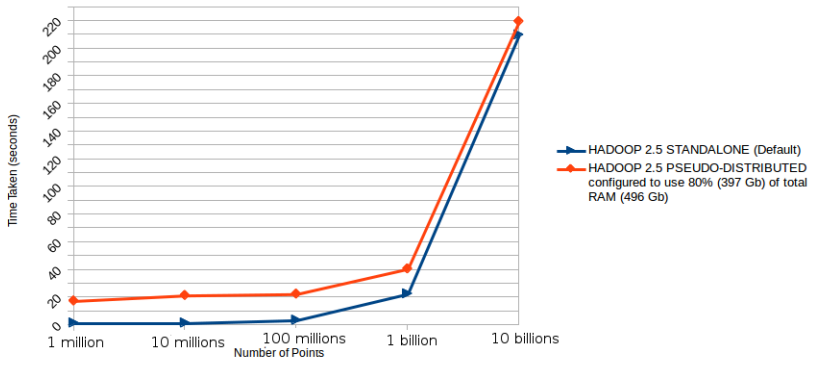

Figura 3.3 – Comparação entre um *map* executando no Hadoop Standalone e um executando no Hadoop Pseudo-Distribuído

Ao final destes dois experimentos, percebemos que para este caso, usando o algoritmo PiEstimator, o uso de uma configuração pseudo-distribuído não é favorável, visto que obteve um desempenho relativamente baixo comparado à configuração *standalone*, possivelmente gastando tempo de processamento em outras tarefas que afetaram a execução do algoritmo.

3.5 Otimização da Configuração

Após os experimentos iniciais com as duas configurações (*standalone* e pseudo-distribuído) foi necessário reavaliar a configuração imposta para o Hadoop. Levantando questionamentos sobre qual algoritmo de escalonamento (*scheduler*) o Hadoop estaria usando, sobre o número de *containers* que o Hadoop estaria alocando e como estes usam a memória e bem como outras variáveis que poderiam estar afetando o desempenho do *Apache Hadoop*.

3.5.1 Escalonadores (*Schedulers*)

O *Apache Hadoop* permite que o usuário escolha, nas configurações, qual algoritmo de escalonamento será usado para a troca de contexto entre as tarefas, chamadas de *jobs*. Dentre estes algoritmos podemos citar dois algoritmos, CapacityScheduler e FairScheduler.

3.5.1.1 CapacityScheduler

O algoritmo CapacityScheduler (Apache Software Foundation, 2014d) foi desenvolvido para ambientes onde os recursos do *cluster* estão divididos em diversas organizações, onde estas possuem recursos que satisfazem seus propósitos. O algoritmo permite que todas as organizações recebam uma capacidade mínima, mas não uma máxima, ou seja toda o recurso não usado em outra empresa pode ser alocado para outra. O conceito de filas é usado na implementação deste algoritmo, sendo possível configurar estas.

3.5.1.2 FairScheduler

Em comparação, o algoritmo FairScheduler (Apache Software Foundation, 2014e), é focado para que todos os *jobs* recebam, em média, uma parte igualitária dos recursos disponíveis no *cluster*. Sendo assim, todos os *jobs* finalizam em um tempo razoável e *jobs* com tamanho relativamente grandes não aguardam por muito tempo até serem executados.

A configuração *default* utiliza o algoritmo CapacityScheduler como escalonador. Para alterar esta configuração é necessário adicionar uma propriedade no arquivo de configuração *yarn-site.xml* para que o Hadoop defina qual escalonador irá utilizar.

Nos experimentos realizados (usando o algoritmo PiEstimator) com ambos os escalonadores podemos ver (Figura 3.4) que para experimentos realizados com até 48 *mappers* o FairScheduler apresenta um desempenho, em alguns casos, 50% menor. Mas quando tratamos

de amostras relativamente grandes, como 10 bilhões de amostras por exemplo, a diferença de tempo não passa dos 11%.

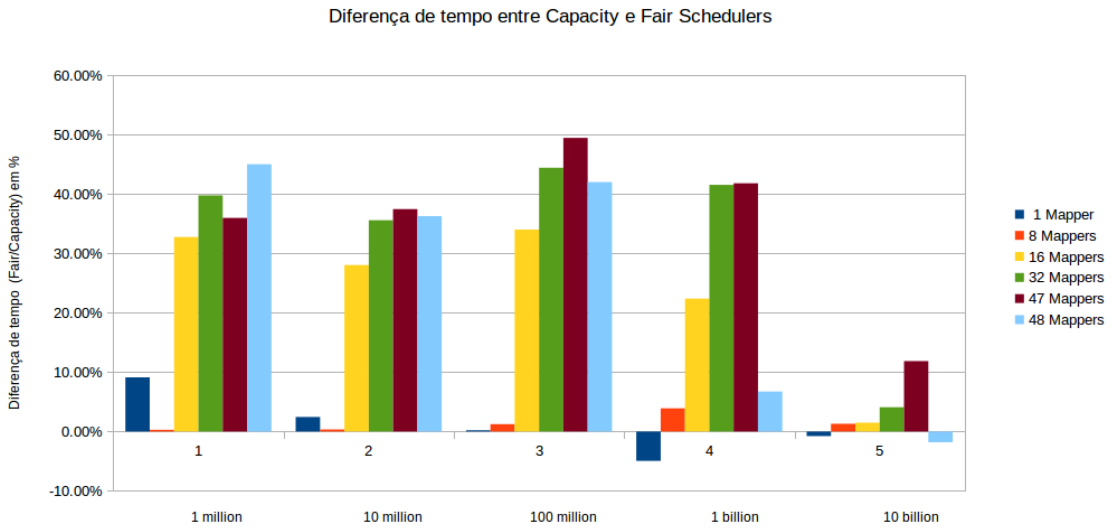

Figura 3.4 – Diferença de tempo entre os algoritmos CapacityScheduler e FairScheduler (PiEstimator com até 48 *mappers*).

Partindo para outro questionamento sobre os escalonadores, foram realizados experimentos com um número de tarefas *map* maior do que o número de processadores (48) da máquina. Foi decidido realizar estes experimentos para existir uma troca mais intensa de contexto, atingindo assim dados relativamente mais consistentes. Nestes casos (Figura 3.5) o algoritmo FairScheduler consumiu até o dobro do tempo (100%) para realizar a mesma tarefa comparado com o algoritmo CapacityScheduler. Estes dados nos mostram que para o algoritmo PiEstimator é viável utilizar o escalonador CapacityScheduler, visto que este obteve um desempenho melhor comparado com o escalonador FairScheduler.

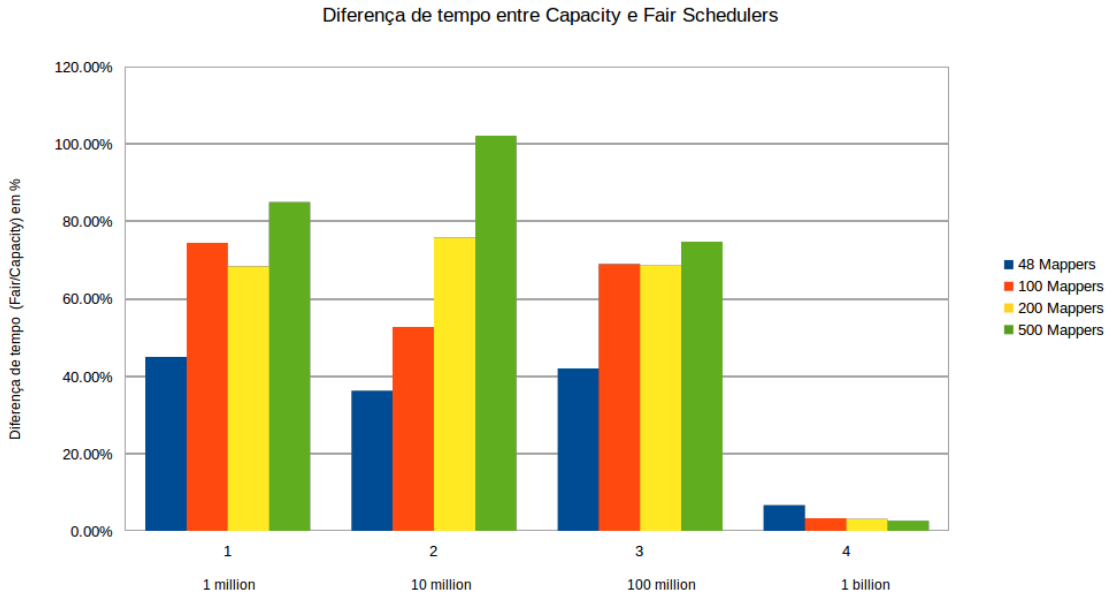

Figura 3.5 – Diferença de tempo entre os algoritmos CapacityScheduler e FairScheduler (PiEstimator com até 500 *mappers*).

3.5.2 Número de *containers* e Memória Alocada

Durante os experimentos iniciais foi constatado que o Hadoop estava alocando apenas 3 *containers*, cada um com aproximadamente 130 *gigabytes* de memória RAM e um *core* da máquina, isto devido a configuração gerada pelo *script* (Anexo B), onde o mesmo calcula o número de *containers* usando a Fórmula 3.1, que usa a função *min* entre o número de núcleos, número de discos e o total de memória RAM dividido pelo tamanho mínimo de um *container*. Como a função retorna o menor valor entre os demais valores, no caso da SGI, o resultado retornado foi 2 *containers*, visto que o número de discos na SGI é de apenas 1, vezes 1.8, adicionando o ApplicationMaster como um *container*, temos apenas 3 *containers* executando na máquina.

$$NumberofContainers = min(2{*}CORES, 1.8{*}DISKS, (TotalRAM)/(MinContainerSize)) \quad (3.1)$$

Uma configuração como esta não utiliza toda a capacidade da máquina, visto que apenas 3 processadores (um alocado para cada *container*) são utilizados. Uma possível solução para o problema encontrado foi configurar o *Apache Hadoop* para que o mesmo fizesse uso de 48 *containers*. Para isto foi necessário utilizar mais memória RAM (480 *gigabytes* ao contrário de 396

gigabytes), sendo possível assim alocar 10 *gigabytes* de memória RAM para cada *container*. As configurações alteradas e seus valores estão descritas na Tabela 3.4.

Tabela 3.4 – Configurações Otimizadas do Hadoop para gerar 48 *containers* de 10 *gigabytes*

Propriedade	Valor
yarn.scheduler.minimum-allocation-mb	10240
yarn.scheduler.maximum-allocation-mb	default
yarn.nodemanager.resource.memory-mb	491520
mapreduce.map.memory.mb	20480
mapreduce.map.java.opts	-Xmx19456m
mapreduce.reduce.memory.mb	40960
mapreduce.reduce.java.opts	-Xmx38912m
yarn.app.mapreduce.am.resource.mb	default
yarn.app.mapreduce.am.command-opts	-Xmx131072m
mapreduce.task.io.sort.mb	default

3.6 Experimentos com o Hadoop Otimizado

Após os novos parâmetros do Hadoop devidamente configurados foram realizados experimentos visando comparar os novos resultados com os obtidos a partir dos experimentos iniciais. Os resultados estão demonstrados em forma de tabelas e gráficos para melhor compreensão dos mesmos. Foram usados os dois algoritmos (PiEstimator e TeraSort) para os experimentos.

3.6.1 Discussão com Algoritmo PiEstimator

Executando o algoritmo do PiEstimator com o Hadoop otimizado e nas mesmas condições, número de *mappers* e número de pontos, dos testes iniciais, chegamos à resultados particularmente interessantes, como podem ser vistos na Tabela 3.5.

Tabela 3.5 – Resultados do algoritmo PiEstimator executando no Hadoop 2.5.0 Otimizado

Número de *mappers*	Número de Pontos	Tempo Decorrido (segundos)	Valor do PI
1	1,000,000	17.524	3.141552
8	1,000,000	18.409	3.141598
16	1,000,000	18.557	3.14159125
32	1,000,000	21.964	3.141586125
48	1,000,000	26.848	3.1415963333
1	10,000,000	17.435	3.1415844
8	10,000,000	18.412	3.14159445
16	10,000,000	18.522	3.14159155
32	10,000,000	22.678	3.141592375
48	10,000,000	26.23	3.14159285
1	100,000,000	19.481	3.14159256
8	100,000,000	20.52	3.14159287
16	100,000,000	20.586	3.1415923975
32	100,000,000	24.822	3.1415927063
48	100,000,000	28.037	3.1415928208
1	1,000,000,000	39.678	3.14159272
8	1,000,000,000	39.692	3.141592797
16	1,000,000,000	40.851	3.1415927088
32	1,000,000,000	46.005	3.1415926745
48	1,000,000,000	71.546	3.1415926717
1	10,000,000,000	230.38	3.1415926776
8	10,000,000,000	229.428	3.141592651
16	10,000,000,000	236.383	3.1415926512
32	10,000,000,000	249.814	3.1415926436
48	10,000,000,000	459.043	3.1415926451

Comparando estes resultados com os resultados encontrados com o Hadoop utilizando apenas 3 *containers*, chegamos ao gráfico na Figura 3.6 que mostra a porcentagem de aumento no desempenho do Hadoop com a nova configuração. Em quase todos os casos existiu um aumento no desempenho do Hadoop. Foi constatado ainda, uma diminuição no tempo de execução de 4932.161 segundos (aproximadamente uma hora e meia) para 249.814 segundos (aproximadamente 4 minutos), ou seja, um aumento de até 94.93% no desempenho do Hadoop otimizado quando o algoritmo do PiEstimator executou com 32 *mappers*. Possivelmente, comprovando assim a teoria de que o *Apache Hadoop* estaria usando apenas 3 *cores* com a configuração antiga, não fazendo uso dos recursos (*cores*) totais da máquina.

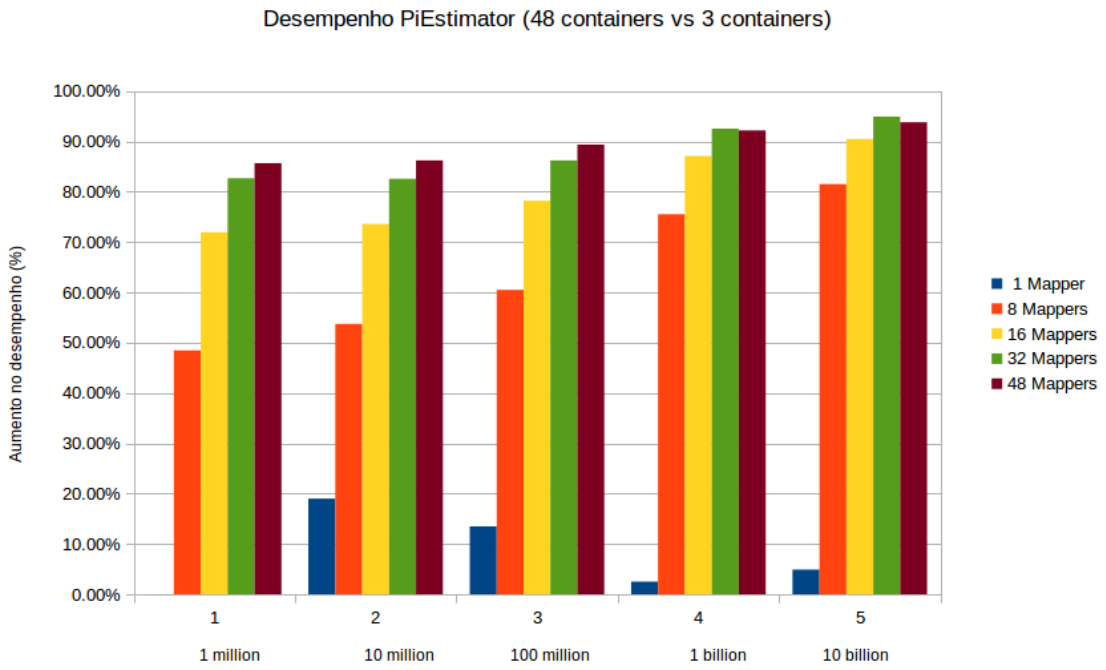

Figura 3.6 – Aumento no desempenho do algoritmo PiEstimator entre o Hadoop configurado para 48 *containers* e 3 *containers*.

3.6.2 Discussão com Algoritmo TeraSort

Com o algoritmo TeraSort foi visado realizar experimentos que combinassem a camada de MapReduce do *Apache Hadoop* e a camada do sistema de arquivo (HDFS). Como a máquina (SGI UV2000) possui 1 *terabyte* de armazenamento de disco, não podemos ordenar 1 *terabyte* de dados pois o disco já está ocupado com sistema operacional, dados pessoais e outros dados. Em vista disto, foi escolhido ordenar 5 *gigabytes* e 10 *gigabytes* para efeito de comparação, variando também o número de *mappers* e *reducers* no algoritmo. Como o algoritmo TeraSort, ao ser executado sem nenhum parâmetro adicional cria o número de *mappers* automaticamente, foi necessário adicionar um parâmetro à chamada do programa para variar o número *reducers*, sendo ele *"Dmapred.reduce.tasks"*, já o número de *mappers* é calculado em função do tamanho do *dataset* à ser processado pelo próprio algoritmo.

3.6.2.1 Resultados com ordenação de 5GB de dados

Para a ordenação dos 5 *gigabytes* de dados, o algoritmo TeraSort estipula um número de 38 *mappers* para o trabalho e número de *reducers* variou de apenas 1 *reducer* (default) até 100 *reducers*. Executando o algoritmo 4 vezes nas duas configurações, na configuração gerada pelo

script e na configuração otimizada, obtemos as Tabelas 3.6 e 3.7, respectivamente.

Tabela 3.6 – Resultados do algoritmo TeraSort Ordenando 5Gb com a Configuração gerada pelo Script (Anexo B)

Número de *reducers*	Tempo (segundos)
1	412
16	417
32	471
47	511
48	503
100	651
1	440
16	426
32	471
47	507
48	507
100	650
1	419
16	418
32	472
47	487
48	510
100	646
1	448
16	415
32	469
47	508
48	512
100	657

Tabela 3.7 – Resultados do algoritmo TeraSort Ordenando 5Gb com a Configuração Otimizada

Número de *reducers*	Tempo (segundos)
1	326
16	285
32	137
47	306
48	416
100	490
1	336
16	464
32	619
47	329
48	423
100	503
1	347
16	285
32	363
47	759
48	384
100	463
1	278
16	411
32	140
47	667
48	728
100	256

3.6.2.2 Resultados com ordenação de 10GB de dados

Para os experimentos com o *dataset* de 10 *gigabytes* foi proposto apenas 2 testes para cada configuração, visto que o tamanho dobraria e o tempo de execução possivelmente também. Mas continuando com a variação de *reducers* de apenas 1 *reducer* (default) até 100 *reducers*. O número de *mappers* para este experimento foi calculado pelo algoritmo TeraSort, sendo ele de 100 *mappers*. Os resultados encontrados estão dispostos na Tabela 3.8 para a configuração com 3 *containers* gerada pelo script e na Tabela 3.9 para a configuração otimizada.

Tabela 3.8 – Resultados do algoritmo TeraSort Ordenando 10Gb com a Configuração gerada pelo Script (Anexo B)

Número de *reducers*	Tempo (segundos)
1	882
16	900
32	903
47	989
48	972
100	1121
1	942
16	919
32	903
47	981
48	980
100	1119

Tabela 3.9 – Resultados do algoritmo TeraSort Ordenando 10Gb com a Configuração Otimizada

Número de *reducers*	Tempo (segundos)
1	477
16	785
32	942
47	835
48	432
100	471
1	716
16	556
32	872
47	949
48	986
100	449

3.6.2.3 Comparação dos Resultados

Um dos problemas encontrados ao executar o algoritmo TeraSort nos casos e nas configurações propostas foi a obtenção de resultados relativamente instáveis. Em muitos dos casos com o Hadoop otimizado o tempo de execução variou, para a mesma configuração, de 10 minutos para apenas aproximadamente 2 minutos e meio (com 32 *reducers*). Este comportamento não havia sido constatado com o Terasort executando com a configuração gerada pelo script provido pelo site HortonWorks (Anexo B), onde os tempos de execução variaram apenas alguns segundos. Podemos notar que a configuração do *Apache Hadoop* é muito sensível à fatores relacionados à aplicação, como neste caso, um uso mais extensivo de acesso à disco, com leituras e escritas.

Entretanto, ainda é possível gerar gráficos para compararmos o desempenho das duas configurações. O gráfico na Figura 3.7 representa o aumento no desempenho da configuração otimizada (48 *containers)*, chegando à 70% de melhora no caso de um *dataset* de 5 *gigabytes*. Já o gráfico na Figura 3.8 mostra que o desempenho, mesmo com a oscilação dos resultados, chega à 60% em alguns casos para a ordenação de *datasets* de 10 *gigabytes*.

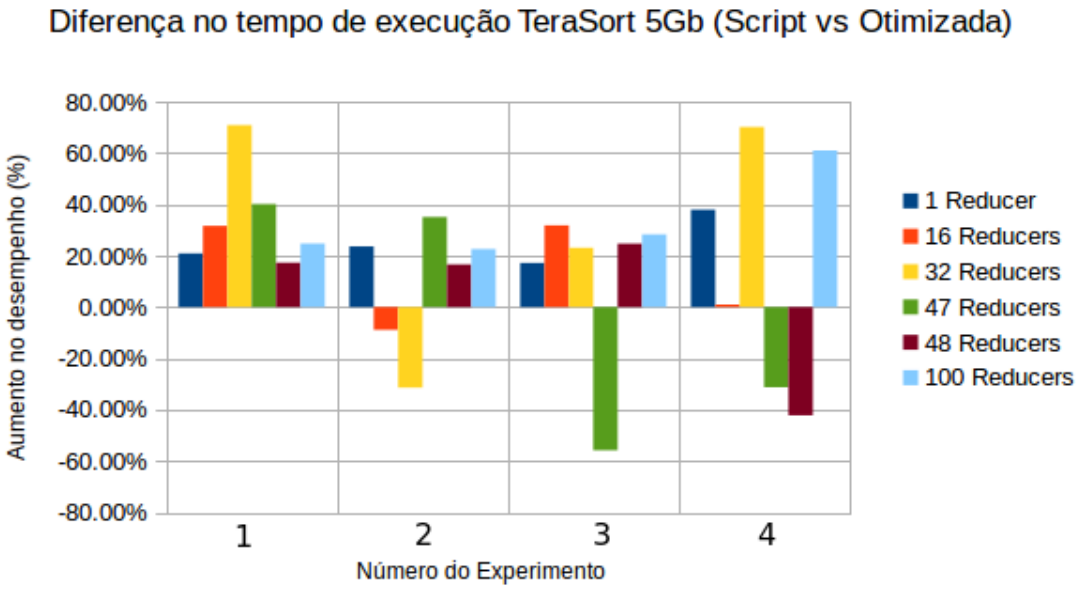

Figura 3.7 – Aumento no desempenho do algoritmo TeraSort ordenando 5Gb entre o Hadoop configurado para 48 *containers* e 3 *containers*.

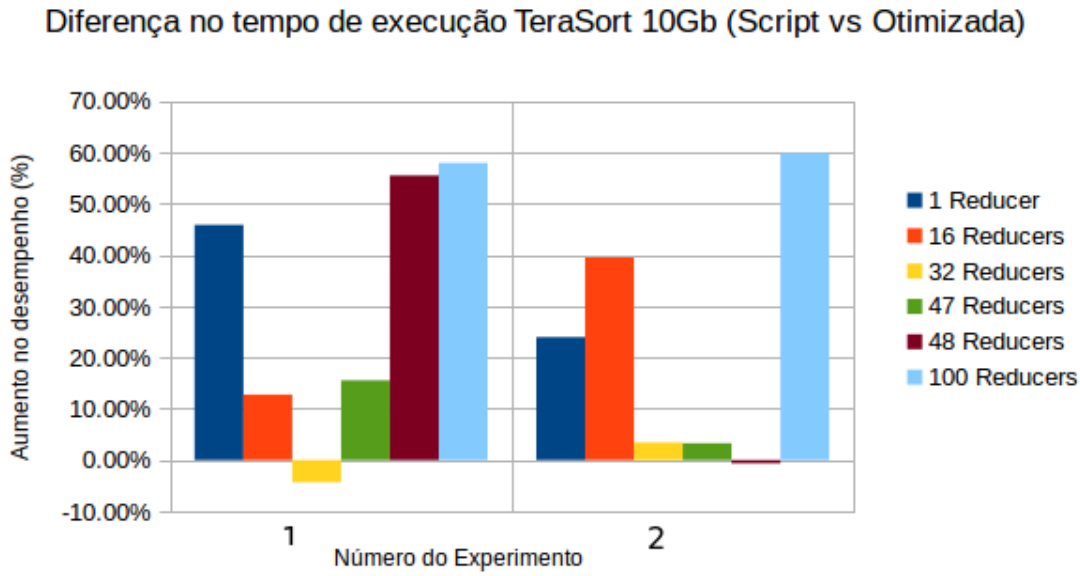

Figura 3.8 – Aumento no desempenho do algoritmo TeraSort ordenando 10Gb entre o Hadoop configurado para 48 *containers* e 3 *containers*.

Em relação as possíveis explicações quanto à instabilidade dos resultados obtidos, podemos citar o sistema de arquivos e o sistema operacional. Pois ao analisarmos os *logs* de erro do Hadoop encontramos a seguinte exceção *java.io.IOException: All datanodes 127.0.0.1:50010 are bad. Aborting...* na execução do arquivo *DFSOutputStream.java*.

Sendo que o arquivo *DFSOutputStream.java* é um componente do HDFS e como uma

das características da máquina (SGI UV2000) é apresentar um espaço em disco relativamente pequeno e só um disco apenas, este trecho do *log* nos faz acreditar que o sistema de arquivo e o sistema operacional não estão configurados adequadamente para executar com diversos *containers*, visto que nenhuma alteração foi realizada nestes dois fatores.

Como a investigação deste problema envolve uma investigação mais profunda sobre o sistema de arquivos do *Apache Hadoop* (HDFS) e sobre detalhes do sistema operacional, possivelmente, o número de arquivos abertos que o sistema pode gerenciar, não foi possível resolver o problema, visto que não existia tempo hábil para realiza-lá. Contudo, foi constatado uma melhora relativa no desempenho, em ambos os algoritmos, quando otimizamos o Hadoop, configurando o mesmo para gerar um número de *containers* igual ao número de cores da máquina.

4 CONCLUSÃO

Este trabalho teve como objetivo principal analisar o comportamento do *framework Apache Hadoop* em uma máquina (SGI UV2000) com uma arquitetura bem específica (arquitetura NUMA) e otimizar o mesmo de modo a obter um desempenho relativamente aceitável frente aos algoritmos propostos para os experimentos.

Tendo em vista os resultados encontrados durante os experimentos realizados neste trabalho, concluímos que a otimização do *framework Apache Hadoop* em máquinas de memória compartilhada, em especial na SGI UV2000, é trabalhosa devido as muitas variáveis que devem serem levadas em consideração ao configurar um *framework* deste tipo. Além disto, configurar o *Apache Hadoop* para um desempenho relativamente ótimo se demonstra uma tarefa muito sensível à diversos fatores como por exemplo, as aplicações à serem executadas, qual a configuração da máquina em questão, qual arquitetura o sistema opera e dentre outros fatores.

Ficou claro que, para algoritmos *MapReduce*, como o PiEstimator, que não dependem tanto de acesso à disco, a otimização proposta se demonstrou eficaz reduzindo o tempo de execução do algoritmo. Porém, quando algoritmos que necessitam deste acesso, no caso o algoritmo TeraSort, a configuração se mostrou dependente de fatores ligados ao sistema de arquivos distribuído do *Apache Hadoop*, mas mesmo assim, alcançando tempos menores de execução.

Como a literatura sobre o assunto ainda é pequena, este trabalho concluiu a sua função de colaborar com a comunidade científica provendo resultados dos experimentos para que outros pesquisadores da área possam vir a usufruir destes resultados agregando mais dados para a área.

REFERÊNCIAS

Apache Hadoop. **Apache™ Hadoop®**. http://hadoop.apache.org, Acesso em agosto de 2014.

Apache Nutch™. **Apache Nutch**. http://nutch.apache.org, Acesso em agosto de 2014.

Apache Software Foundation. **HDFS Architecture Guide**. http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, Acesso em setembro de 2014.

Apache Software Foundation. **Apache Hadoop NextGen MapReduce (YARN)**. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html, Acesso em setembro de 2014.

Apache Software Foundation. **Package org.apache.hadoop.examples.terasort**. https://hadoop.apache.org/docs/current/api/org/apache/hadoop/examples/terasort/package-summary.html, Acesso em outubro de 2014.

Apache Software Foundation. **Capacity Scheduler**. http://hadoop.apache.org/docs/r1.2.1/capacity_schedul
Acesso em novembro de 2014.

Apache Software Foundation. **Fair Scheduler**. http://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html, Acesso em novembro de 2014.

Bligh, Martin J. and Dobson, Matt and Hart, Darren. **Linux on NUMA Systems**. 2004. 89–102p.

Caflish, Russel E. Monte Carlo and quasi-Monte Carlo methods. **Acta Numerica**, [S.l.], p.1–49, 1998.

Carissimi, Alexandre and Dupros, Fabrice and Méhaut, Jean-François and Polanczyk, Rafael Vanoni. **Aspectos de Programação Paralela em Máquinas NUMA.** http://www.sbc.org.br/sbac/2007/cdrom/papers/wscad/minicursos/33311_1.pdf, Acesso em agosto 2014.

Chris Nyberg. **Sort Benchmark Home Page**. http://sortbenchmark.org/, Acesso em outubro de 2014.

Database Technology Group, Technische Universität Dresden. **NUMA-Architektur Support in kommerziellen Datenbankmanagementsystemen**. https://wwwdb.inf.tu-dresden.de/lectures/ws-2011-2012/kp-db-systemarchitektur-numa-unterstuetzung-in-datenbanksystemen/, Acesso em outubro de 2014.

Dean, Jeffrey and Ghemawat, Sanjay. **MapReduce**: simplified data processing on large clusters. New York, NY, USA: ACM, 2008. 107–113p. v.51, n.1.

Hortonworks, Inc. **Hortonworks Data Platform - Installing HDP Manually**. http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.0.6.0/bk_installing_manually_book/content/index.html, Acesso em setembro de 2014.

KUMAR, K. A. et al. **Hone**: "scaling down"hadoop on shared-memory systems. [S.l.]: VLDB Endowment, 2013. 1354–1357p. v.6, n.12.

Kumar, K. Ashwin and Gluck, Jonathan and Deshpande, Amol and Lin, Jimmy. **Optimization Techniques for "Scaling Down" Hadoop on Multi-Core, Shared-Memory Systems**. [S.l.]: Proceedings of the 17th International Conference on Extending Database Technology, 2014. 13–24p.

McCalpin, John D. **STREAM**: sustainable memory bandwidth in high performance computers. http://www.cs.virginia.edu/stream/, Acesso em dezembro de 2014.

McVoy, Larry and Staelin, Carl. **LMbench - Tools for performance analysis**. http://www.bitmover.com/lmbench/whatis_lmbench.html, Acesso em dezembro de 2014.

Open MPI. **Portable Hardware Locality (hwloc)**. http://www.open-mpi.org/projects/hwloc/, Acesso em setembro de 2014.

O'Malley, Owen. Terabyte sort on apache hadoop. **Yahoo, available online at: http://sortbenchmark.org/Yahoo-Hadoop. pdf,(May)**, [S.l.], p.1–3, 2008.

SAS Institute Inc. **Big Data - What it is and why it matters.** http://www.sas.com/en_us/insights/big-data/what-is-big-data.html, Acesso em agosto de 2014.

Shinnar, Avraham and Cunningham, David and Saraswat, Vijay and Herta, Benjamin. **M3R**: increased performance for in-memory hadoop jobs. [S.l.]: VLDB Endowment, 2012. 1736–1747p. v.5, n.12.

Shvachko, Konstantin et al. **The Hadoop Distributed File System**. Washington, DC, USA: IEEE Computer Society, 2010. 1–10p. (MSST '10).

Silicon Graphics International Corp. **SGI**. http://www.sgi.com/, Acesso em agosto de 2014.

Silicon Graphics International Corp. **SGI UV**: the big brain computer. https://www.sgi.com/products/servers/uv/, Acesso em agosto de 2014.

The Apache Software Foundation. **Class PiEstimator**. http://hadoop.apache.org/docs/r1.1.2/api/org/apache/hadoop/examples/PiEstimator.html, Acesso em setembro de 2014.

Vavilapalli, Vinod Kumar et al. **Apache Hadoop YARN**: yet another resource negotiator. New York, NY, USA: ACM, 2013. 5:1–5:16p. (SOCC '13).

Yahoo. **Yahoo**. www.yahoo.com, Acesso em agosto de 2014.

Yoo, Richard M. and Romano, Anthony and Kozyrakis, Christos. **Phoenix Rebirth**: scalable mapreduce on a large-scale shared-memory system. Washington, DC, USA: IEEE Computer Society, 2009. 198–207p. (IISWC '09).

Zhang, Yunming. **HJ-Hadoop**: an optimized mapreduce runtime for multi-core systems. New York, NY, USA: ACM, 2013. 111–112p. (SPLASH '13).

ANEXOS

ANEXO A – Algortimo PiEstimator

```
1  /**
2   * Licensed to the Apache Software Foundation (ASF) under one
3   * or more contributor license agreements.  See the NOTICE file
4   * distributed with this work for additional information
5   * regarding copyright ownership.  The ASF licenses this file
6   * to you under the Apache License, Version 2.0 (the
7   * "License"); you may not use this file except in compliance
8   * with the License.  You may obtain a copy of the License at
9   *
10  *      http://www.apache.org/licenses/LICENSE-2.0
11  *
12  * Unless required by applicable law or agreed to in writing, software
13  * distributed under the License is distributed on an "AS IS" BASIS,
14  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
15  * See the License for the specific language governing permissions and
16  * limitations under the License.
17  */
18
19 package org.apache.hadoop.examples;
20
21 import java.io.IOException;
22 import java.math.BigDecimal;
23 import java.math.RoundingMode;
24 import java.util.Iterator;
25
26 import org.apache.hadoop.conf.Configured;
27 import org.apache.hadoop.fs.FileSystem;
28 import org.apache.hadoop.fs.Path;
29 import org.apache.hadoop.io.BooleanWritable;
30 import org.apache.hadoop.io.LongWritable;
31 import org.apache.hadoop.io.SequenceFile;
32 import org.apache.hadoop.io.Writable;
33 import org.apache.hadoop.io.WritableComparable;
34 import org.apache.hadoop.io.SequenceFile.CompressionType;
35 import org.apache.hadoop.mapred.FileInputFormat;
36 import org.apache.hadoop.mapred.FileOutputFormat;
37 import org.apache.hadoop.mapred.JobClient;
38 import org.apache.hadoop.mapred.JobConf;
39 import org.apache.hadoop.mapred.MapReduceBase;
40 import org.apache.hadoop.mapred.Mapper;
41 import org.apache.hadoop.mapred.OutputCollector;
42 import org.apache.hadoop.mapred.Reducer;
43 import org.apache.hadoop.mapred.Reporter;
44 import org.apache.hadoop.mapred.SequenceFileInputFormat;
45 import org.apache.hadoop.mapred.SequenceFileOutputFormat;
46 import org.apache.hadoop.util.Tool;
47 import org.apache.hadoop.util.ToolRunner;
48
49 /**
50  * A Map-reduce program to estimate the value of Pi
51  * using quasi-Monte Carlo method.
52  *
53  * Mapper:
54  *   Generate points in a unit square
```

```
55  *   and then count points inside/outside of the inscribed circle of the
        square.
56  *
57  * Reducer:
58  *   Accumulate points inside/outside results from the mappers.
59  *
60  * Let numTotal = numInside + numOutside.
61  * The fraction numInside/numTotal is a rational approximation of
62  * the value (Area of the circle)/(Area of the square),
63  * where the area of the inscribed circle is Pi/4
64  * and the area of unit square is 1.
65  * Then, Pi is estimated value to be 4(numInside/numTotal).
66  */
67 public class PiEstimator extends Configured implements Tool {
68   /** tmp directory for input/output */
69   static private final Path TMP_DIR = new Path(
70       PiEstimator.class.getSimpleName() + "_TMP_3_141592654");
71
72   /** 2-dimensional Halton sequence {H(i)},
73    * where H(i) is a 2-dimensional point and i >= 1 is the index.
74    * Halton sequence is used to generate sample points for Pi estimation.
75    */
76   private static class HaltonSequence {
77     /** Bases */
78     static final int[] P = {2, 3};
79     /** Maximum number of digits allowed */
80     static final int[] K = {63, 40};
81
82     private long index;
83     private double[] x;
84     private double[][] q;
85     private int[][] d;
86
87     /** Initialize to H(startindex),
88      * so the sequence begins with H(startindex+1).
89      */
90     HaltonSequence(long startindex) {
91       index = startindex;
92       x = new double[K.length];
93       q = new double[K.length][];
94       d = new int[K.length][];
95       for(int i = 0; i < K.length; i++) {
96         q[i] = new double[K[i]];
97         d[i] = new int[K[i]];
98       }
99
100      for(int i = 0; i < K.length; i++) {
101        long k = index;
102        x[i] = 0;
103
104        for(int j = 0; j < K[i]; j++) {
105          q[i][j] = (j == 0? 1.0: q[i][j-1])/P[i];
106          d[i][j] = (int)(k % P[i]);
107          k = (k - d[i][j])/P[i];
108          x[i] += d[i][j] * q[i][j];
109        }
110      }
111    }
```

```
112
113       /** Compute next point.
114        * Assume the current point is H(index).
115        * Compute H(index+1).
116        *
117        * @return a 2-dimensional point with coordinates in [0,1)^2
118        */
119       double[] nextPoint() {
120         index++;
121         for(int i = 0; i < K.length; i++) {
122           for(int j = 0; j < K[i]; j++) {
123             d[i][j]++;
124             x[i] += q[i][j];
125             if (d[i][j] < P[i]) {
126               break;
127             }
128             d[i][j] = 0;
129             x[i] -= (j == 0? 1.0: q[i][j-1]);
130           }
131         }
132         return x;
133       }
134     }
135
136     /**
137      * Mapper class for Pi estimation.
138      * Generate points in a unit square
139      * and then count points inside/outside of the inscribed circle of the
           square.
140      */
141     public static class PiMapper extends MapReduceBase
142       implements Mapper<LongWritable, LongWritable, BooleanWritable,
            LongWritable> {
143
144       /** Map method.
145        * @param offset samples starting from the (offset+1)th sample.
146        * @param size the number of samples for this map
147        * @param out output {ture->numInside, false->numOutside}
148        * @param reporter
149        */
150       public void map(LongWritable offset,
151                       LongWritable size,
152                       OutputCollector<BooleanWritable, LongWritable> out,
153                       Reporter reporter) throws IOException {
154
155         final HaltonSequence haltonsequence = new HaltonSequence(offset.get()
              );
156         long numInside = 0L;
157         long numOutside = 0L;
158
159         for(long i = 0; i < size.get(); ) {
160           //generate points in a unit square
161           final double[] point = haltonsequence.nextPoint();
162
163           //count points inside/outside of the inscribed circle of the square
164           final double x = point[0] - 0.5;
165           final double y = point[1] - 0.5;
166           if (x*x + y*y > 0.25) {
```

```
167           numOutside++;
168         } else {
169           numInside++;
170         }
171
172         //report status
173         i++;
174         if (i % 1000 == 0) {
175           reporter.setStatus("Generated␣" + i + "␣samples.");
176         }
177       }
178
179       //output map results
180       out.collect(new BooleanWritable(true), new LongWritable(numInside));
181       out.collect(new BooleanWritable(false), new LongWritable(numOutside))
              ;
182     }
183   }
184
185   /**
186    * Reducer class for Pi estimation.
187    * Accumulate points inside/outside results from the mappers.
188    */
189   public static class PiReducer extends MapReduceBase
190     implements Reducer<BooleanWritable, LongWritable, WritableComparable
          <?>, Writable> {
191
192     private long numInside = 0;
193     private long numOutside = 0;
194     private JobConf conf; //configuration for accessing the file system
195
196     /** Store job configuration. */
197     @Override
198     public void configure(JobConf job) {
199       conf = job;
200     }
201
202     /**
203      * Accumulate number of points inside/outside results from the mappers.
204      * @param isInside Is the points inside?
205      * @param values An iterator to a list of point counts
206      * @param output dummy, not used here.
207      * @param reporter
208      */
209     public void reduce(BooleanWritable isInside,
210                        Iterator<LongWritable> values,
211                        OutputCollector<WritableComparable<?>, Writable>
                             output,
212                        Reporter reporter) throws IOException {
213       if (isInside.get()) {
214         for(; values.hasNext(); numInside += values.next().get());
215       } else {
216         for(; values.hasNext(); numOutside += values.next().get());
217       }
218     }
219
220     /**
221      * Reduce task done, write output to a file.
```

```
222       */
223       @Override
224       public void close() throws IOException {
225         //write output to a file
226         Path outDir = new Path(TMP_DIR, "out");
227         Path outFile = new Path(outDir, "reduce-out");
228         FileSystem fileSys = FileSystem.get(conf);
229         SequenceFile.Writer writer = SequenceFile.createWriter(fileSys, conf,
230             outFile, LongWritable.class, LongWritable.class,
231             CompressionType.NONE);
232         writer.append(new LongWritable(numInside), new LongWritable(
                numOutside));
233         writer.close();
234       }
235     }
236
237     /**
238      * Run a map/reduce job for estimating Pi.
239      *
240      * @return the estimated value of Pi
241      */
242     public static BigDecimal estimate(int numMaps, long numPoints, JobConf
            jobConf
243         ) throws IOException {
244       //setup job conf
245       jobConf.setJobName(PiEstimator.class.getSimpleName());
246
247       jobConf.setInputFormat(SequenceFileInputFormat.class);
248
249       jobConf.setOutputKeyClass(BooleanWritable.class);
250       jobConf.setOutputValueClass(LongWritable.class);
251       jobConf.setOutputFormat(SequenceFileOutputFormat.class);
252
253       jobConf.setMapperClass(PiMapper.class);
254       jobConf.setNumMapTasks(numMaps);
255
256       jobConf.setReducerClass(PiReducer.class);
257       jobConf.setNumReduceTasks(1);
258
259       // turn off speculative execution, because DFS doesn't handle
260       // multiple writers to the same file.
261       jobConf.setSpeculativeExecution(false);
262
263       //setup input/output directories
264       final Path inDir = new Path(TMP_DIR, "in");
265       final Path outDir = new Path(TMP_DIR, "out");
266       FileInputFormat.setInputPaths(jobConf, inDir);
267       FileOutputFormat.setOutputPath(jobConf, outDir);
268
269       final FileSystem fs = FileSystem.get(jobConf);
270       if (fs.exists(TMP_DIR)) {
271         throw new IOException("Tmp␣directory␣" + fs.makeQualified(TMP_DIR)
272             + "␣already␣exists.␣␣Please␣remove␣it␣first.");
273       }
274       if (!fs.mkdirs(inDir)) {
275         throw new IOException("Cannot␣create␣input␣directory␣" + inDir);
276       }
277
```

```
278     try {
279       //generate an input file for each map task
280       for(int i=0; i < numMaps; ++i) {
281         final Path file = new Path(inDir, "part"+i);
282         final LongWritable offset = new LongWritable(i * numPoints);
283         final LongWritable size = new LongWritable(numPoints);
284         final SequenceFile.Writer writer = SequenceFile.createWriter(
285             fs, jobConf, file,
286             LongWritable.class, LongWritable.class, CompressionType.NONE);
287         try {
288           writer.append(offset, size);
289         } finally {
290           writer.close();
291         }
292         System.out.println("Wrote input for Map #"+i);
293       }
294
295       //start a map/reduce job
296       System.out.println("Starting Job");
297       final long startTime = System.currentTimeMillis();
298       JobClient.runJob(jobConf);
299       final double duration = (System.currentTimeMillis() - startTime)
              /1000.0;
300       System.out.println("Job Finished in " + duration + " seconds");
301
302       //read outputs
303       Path inFile = new Path(outDir, "reduce-out");
304       LongWritable numInside = new LongWritable();
305       LongWritable numOutside = new LongWritable();
306       SequenceFile.Reader reader = new SequenceFile.Reader(fs, inFile,
              jobConf);
307       try {
308         reader.next(numInside, numOutside);
309       } finally {
310         reader.close();
311       }
312
313       //compute estimated value
314       final BigDecimal numTotal
315           = BigDecimal.valueOf(numMaps).multiply(BigDecimal.valueOf(
                numPoints));
316       return BigDecimal.valueOf(4).setScale(20)
317           .multiply(BigDecimal.valueOf(numInside.get()))
318           .divide(numTotal, RoundingMode.HALF_UP);
319     } finally {
320       fs.delete(TMP_DIR, true);
321     }
322   }
323
324   /**
325    * Parse arguments and then runs a map/reduce job.
326    * Print output in standard out.
327    *
328    * @return a non-zero if there is an error.  Otherwise, return 0.
329    */
330   public int run(String[] args) throws Exception {
331     if (args.length != 2) {
```

```
332         System.err.println("Usage: "+getClass().getName()+" <nMaps> <nSamples
                >");
333         ToolRunner.printGenericCommandUsage(System.err);
334         return -1;
335       }
336
337       final int nMaps = Integer.parseInt(args[0]);
338       final long nSamples = Long.parseLong(args[1]);
339
340       System.out.println("Number of Maps  = " + nMaps);
341       System.out.println("Samples per Map = " + nSamples);
342
343       final JobConf jobConf = new JobConf(getConf(), getClass());
344       System.out.println("Estimated value of Pi is "
345           + estimate(nMaps, nSamples, jobConf));
346       return 0;
347     }
348
349     /**
350      * main method for running it as a stand alone command.
351      */
352     public static void main(String[] argv) throws Exception {
353       System.exit(ToolRunner.run(null, new PiEstimator(), argv));
354     }
355   }
```

ANEXO B – Script provido pelo site HortonWorks (Hortonworks, Inc., 2013) para configuração do *Apache Hadoop*

```
#!/usr/bin/env python
'''
Licensed to the Apache Software Foundation (ASF) under one
or more contributor license agreements.  See the NOTICE file
distributed with this work for additional information
regarding copyright ownership.  The ASF licenses this file
to you under the Apache License, Version 2.0 (the
"License"); you may not use this file except in compliance
with the License.  You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
'''

import optparse
from pprint import pprint
import logging
import sys
import math
import ast

''' Reserved for OS + DN + NM,  Map: Memory => Reservation '''
reservedStack = { 4:1, 8:2, 16:2, 24:4, 48:6, 64:8, 72:8, 96:12,
                  128:24, 256:32, 512:64}
''' Reserved for HBase. Map: Memory => Reservation '''

reservedHBase = {4:1, 8:1, 16:2, 24:4, 48:8, 64:8, 72:8, 96:16,
                  128:24, 256:32, 512:64}
GB = 1024

def getMinContainerSize(memory):
  if (memory <= 4):
    return 256
  elif (memory <= 8):
    return 512
  elif (memory <= 24):
    return 1024
  else:
    return 2048
  pass

def getReservedStackMemory(memory):
  if (reservedStack.has_key(memory)):
    return reservedStack[memory]
  if (memory <= 4):
    ret = 1
  elif (memory >= 512):
    ret = 64
```

```
54    else:
55      ret = 1
56    return ret
57
58  def getReservedHBaseMem(memory):
59    if (reservedHBase.has_key(memory)):
60      return reservedHBase[memory]
61    if (memory <= 4):
62      ret = 1
63    elif (memory >= 512):
64      ret = 64
65    else:
66      ret = 2
67    return ret
68
69  def main():
70    log = logging.getLogger(__name__)
71    out_hdlr = logging.StreamHandler(sys.stdout)
72    out_hdlr.setFormatter(logging.Formatter(' %(message)s'))
73    out_hdlr.setLevel(logging.INFO)
74    log.addHandler(out_hdlr)
75    log.setLevel(logging.INFO)
76    parser = optparse.OptionParser()
77    memory = 0
78    cores = 0
79    disks = 0
80    hbaseEnabled = True
81    parser.add_option('-c', '--cores', default = 16,
82                      help = 'Number of cores on each host')
83    parser.add_option('-m', '--memory', default = 64,
84                      help = 'Amount of Memory on each host in GB')
85    parser.add_option('-d', '--disks', default = 4,
86                      help = 'Number of disks on each host')
87    parser.add_option('-k', '--hbase', default = "True",
88                      help = 'True if HBase is installed , False is not')
89    (options, args) = parser.parse_args()
90
91    cores = int (options.cores)
92    memory = int (options.memory)
93    disks = int (options.disks)
94    hbaseEnabled = ast.literal_eval(options.hbase)
95
96    log.info("Using cores=" +  str(cores) + " memory=" + str(memory) + "GB" +
97             " disks=" + str(disks) + " hbase=" + str(hbaseEnabled))
98    minContainerSize = getMinContainerSize(memory)
99    reservedStackMemory = getReservedStackMemory(memory)
100   reservedHBaseMemory = 0
101   if (hbaseEnabled):
102     reservedHBaseMemory = getReservedHBaseMem(memory)
103   reservedMem = reservedStackMemory + reservedHBaseMemory
104   usableMem = memory - reservedMem
105   memory -= (reservedMem)
106   if (memory < 2):
107     memory = 2
108     reservedMem = max(0, memory - reservedMem)
109
110   memory *= GB
111
```

```
112     containers = int (min(2 * cores,
113                           min(math.ceil(1.8 * float(disks)),
114                               memory/minContainerSize)))
115     if (containers <= 2):
116       containers = 3
117
118     log.info("Profile: cores=" + str(cores) + " memory=" + str(memory) + "MB"
119              + " reserved=" + str(reservedMem) + "GB" + " usableMem="
120              + str(usableMem) + "GB" + " disks=" + str(disks))
121
122     container_ram =  abs(memory/containers)
123     if (container_ram > GB):
124       container_ram = int(math.floor(container_ram / 512)) * 512
125     log.info("Num Container=" + str(containers))
126     log.info("Container Ram=" + str(container_ram) + "MB")
127     log.info("Used Ram=" + str(int (containers*container_ram/float(GB))) + "
          GB")
128     log.info("Unused Ram=" + str(reservedMem) + "GB")
129     log.info("yarn.scheduler.minimum-allocation-mb=" + str(container_ram))
130     log.info("yarn.scheduler.maximum-allocation-mb=" + str(containers*
          container_ram))
131     log.info("yarn.nodemanager.resource.memory-mb=" + str(containers*
          container_ram))
132     map_memory = container_ram
133     reduce_memory = 2*container_ram if (container_ram <= 2048) else
          container_ram
134     am_memory = max(map_memory, reduce_memory)
135     log.info("mapreduce.map.memory.mb=" + str(map_memory))
136     log.info("mapreduce.map.java.opts=-Xmx" + str(int(0.8 * map_memory)) +"m"
          )
137     log.info("mapreduce.reduce.memory.mb=" + str(reduce_memory))
138     log.info("mapreduce.reduce.java.opts=-Xmx" + str(int(0.8 * reduce_memory)
          ) + "m")
139     log.info("yarn.app.mapreduce.am.resource.mb=" + str(am_memory))
140     log.info("yarn.app.mapreduce.am.command-opts=-Xmx" + str(int(0.8*
          am_memory)) + "m")
141     log.info("mapreduce.task.io.sort.mb=" + str(int(0.4 * map_memory)))
142     pass
143
144 if __name__ == '__main__':
145     try:
146       main()
147     except(KeyboardInterrupt, EOFError):
148       print("\nAborting ... Keyboard Interrupt.")
149       sys.exit(1)
```