

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

**ESCALONAMENTO ADAPTATIVO PARA O
APACHE HADOOP**

DISSERTAÇÃO DE MESTRADO

Guilherme Weigert Cassales

Santa Maria, RS, Brasil

2016

ESCALONAMENTO ADAPTATIVO PARA O APACHE HADOOP

Guilherme Weigert Cassales

Dissertação apresentada ao Curso de Mestrado Programa de
Pós-Graduação em Informática (PPGI), Área de Concentração em
Computação, da Universidade Federal de Santa Maria (UFSM, RS),
como requisito parcial para obtenção do grau de
Mestre em Ciência da Computação

Orientadora: Prof^a. Dr^a. Andrea Schwertner Charão

Santa Maria, RS, Brasil

2016

Cassales, Guilherme Weigert

Escalonamento Adaptativo para o Apache Hadoop / por Guilherme Weigert Cassales. – 2016.

40 f.: il.; 30 cm.

Orientadora: Andrea Schwertner Charão

Dissertação (Mestrado) - Universidade Federal de Santa Maria, Centro de Tecnologia, Curso de Ciência da Computação, RS, 2016.

1. Apache Hadoop. 2. Escalonador. 3. Sensibilidade ao Contexto.
I. Charão, Andrea Schwertner. II. Título.

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

A Comissão Examinadora, abaixo assinada,
aprova a Dissertação de Mestrado

ESCALONAMENTO ADAPTATIVO PARA O APACHE HADOOP

elaborada por
Guilherme Weigert Cassales

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

COMISSÃO EXAMINADORA:

Andrea Schwertner Charão, Prof^a. Dr^a.
(Presidente/Orientadora)

Benhur de Oliveira Stein, Prof. Dr. (UFSM)

Patrícia Pitthan de Araújo Barcelos, Prof^a. Dr^a. (UFSM)

Santa Maria, 20 de Janeiro de 2016.

RESUMO

Dissertação de Mestrado
Curso de Ciência da Computação
Universidade Federal de Santa Maria

ESCALONAMENTO ADAPTATIVO PARA O APACHE HADOOP

AUTOR: GUILHERME WEIGERT CASSALES

ORIENTADORA: ANDREA SCHWERTNER CHARÃO

Local da Defesa e Data: Santa Maria, 20 de Janeiro de 2016.

Hoje em dia, o volume de dados gerados é muito maior do que a capacidade de processamento dos computadores. Como solução para esse problema, algumas tarefas podem ser paralelizadas ou distribuídas. O *framework Apache Hadoop* (Apache Hadoop, 2013), é uma delas e poupa o programador as tarefas de gerenciamento, como tolerância à falhas, particionamento dos dados entre outros. Um problema no escalonador do *Apache Hadoop* é que seu foco é em ambientes homogêneos, o que muitas vezes não é possível de se manter. O foco deste trabalho foi na melhoria de um escalonador já existente, possuindo como objetivo torná-lo sensível ao contexto, permitindo que as capacidades físicas de cada máquina sejam consideradas na hora da distribuição das tarefas submetidas. Optou-se por inserir coletores de informações de contexto (memória e cpu) no CapacityScheduler, tornando o comportamento desse sensível ao contexto. Através das mudanças feitas e de experimentos feitos usando um benchmark bem conhecido (TeraSort), foi possível demonstrar uma melhoria no escalonamento em relação ao escalonador original com a configuração padrão.

Palavras-chave: Apache Hadoop. Escalonador. Sensibilidade ao Contexto.

ABSTRACT

Master's Dissertation
Undergraduate Program in Computer Science
Federal University of Santa Maria

DEVELOPMENT OF A CONTEXT-AWARE SCHEDULER FOR APACHE HADOOP

AUTHOR: GUILHERME WEIGERT CASSALES

ADVISOR: ANDREA SCHWERTNER CHARÃO

Defense Place and Date: Santa Maria, January 20th, 2016.

Nowadays the volume of data generated by the services provided for end users, is way larger than the processing capacity of one computer alone. As a solution to this problem, some tasks can be parallelized. The Apache Hadoop framework, is one of these parallelized solutions and it spares the programmer of management tasks such as fault tolerance, data partitioning, among others. One problem on this framework is the scheduler, which is designed for homogeneous environments. It is worth to remember that maintaining a homogeneous environment is somewhat difficult today, given the fast development of new, cheaper and more powerful hardware. This work focuses on altering the Capacity Scheduler, in order to make it more context-aware towards resources on the cluster. Making it possible to consider the physical capacities of the machines when scheduling the submitted tasks. It was chosen to insert context information (memory and cpu) collectors on CapacityScheduler, making his scheduling more context-aware. Through the changes and experiments made using a common and well known benchmark (TeraSort), it was possible to notice a improvement on scheduling in relation to the original scheduler using the default configuration.

Keywords: Apache Hadoop. Scheduler. Context-aware.

LISTA DE FIGURAS

Figura 2.1 – Arquitetura geral do Apache Hadoop	12
Figura 2.2 – Arquitetura geral do HDFS (HADOOP, 2013a)	13
Figura 2.3 – Arquitetura geral do YARN (HADOOP, 2013b)	13
Figura 3.1 – Diagrama de classes dos coletores de contexto	23
Figura 3.2 – Estrutura do ZooKeeper	24
Figura 4.1 – Diagrama de Gantt para os experimentos controlados com TeraSort	31
Figura 4.2 – Diagrama de Gantt para os experimentos controlados com TestDFSIO	32
Figura 4.3 – Diagrama de Gantt para os experimentos controlados com WordCount	33
Figura 4.4 – Diagrama de Gantt para os experimentos reais com TeraSort	34
Figura 4.5 – Diagrama de Gantt para os experimentos reais com TestDFSIO	35
Figura 4.6 – Diagrama de Gantt para os experimentos reais com WordCount	36

SUMÁRIO

1 INTRODUÇÃO	9
2 FUNDAMENTOS E REVISÃO DE LITERATURA	11
2.1 Apache Hadoop	11
2.1.1 Arquitetura geral do Apache Hadoop	11
2.1.1.1 HDFS	12
2.1.1.2 YARN	12
2.1.1.3 Configuração do ambiente de execução do Hadoop	14
2.1.1.4 MapReduce	15
2.2 Escalonadores para o Hadoop	15
2.3 Trabalhos Relacionados	16
3 MÉTODOS E DESENVOLVIMENTO	20
3.1 Coletores de Contexto	20
3.1.1 Sensibilidade ao Contexto	20
3.1.2 Implementação	21
3.2 Comunicação Distribuída	22
3.2.1 ZooKeeper	22
3.2.2 Implementação	24
3.3 Solução Implementada	25
3.3.1 Integração dos Coletores de Contexto	25
3.3.2 Integração dos Clientes	25
3.3.3 Justificativas para	25
4 EXPERIMENTOS E RESULTADOS	26
4.1 Considerações Iniciais Sobre os Experimentos	26
4.1.1 Casos de teste	26
4.1.2 Aplicações de teste	28
4.1.3 Configurações de Hardware e Software	28
4.1.4 Apresentação dos resultados	29
4.2 Experimento controlado	29
4.2.1 Procedimentos	30
4.2.2 Resultados e Interpretações	30
4.3 Experimento real	31
4.3.1 Procedimentos	32
4.3.2 Resultados e Interpretações	33
4.4 Experimento de escala	33
4.4.1 Procedimentos	34
4.4.2 Resultados e Interpretações	35
5 CONCLUSION AND FUTURE WORK	37
REFERÊNCIAS	38

1 INTRODUÇÃO

Apache Hadoop é um *framework* de computação paralela e distribuída para processamento de grandes conjuntos de dados e implementa o paradigma de programação MapReduce (?). O Apache Hadoop é projetado para ser escalável de um único servidor a milhares de máquinas, cada uma oferecendo processamento e armazenamento local. Esta capacidade de utilizar grande quantidade de *hardware* barato e a crescente importância do processamento de dados não estruturados tornaram o Apache Hadoop uma ferramenta relevante no mercado (?).

Sem uma configuração específica pelo administrador, o Apache Hadoop assume que está sendo utilizado em um *cluster* homogêneo para execução de aplicações MapReduce. Uma vez que o desempenho geral do *cluster* está ligado ao escalonamento de tarefas, o desempenho do Hadoop pode diminuir significativamente quando executado em ambientes que não satisfaçam a suposição feita no projeto do *framework*, ou seja, em ambientes dinâmicos ou heterogêneos (?).

Esta é uma preocupação especial quando tenta-se utilizar o Hadoop em grids pervasivos. Grids pervasivos são uma alternativa aos *clusters* dedicados, dado que o custo de aquisição e manutenção de um *cluster* dedicado continua alto e dissuasivo para muitas organizações. De acordo com (?), grids pervasivos representam a generalização extrema do conceito de grid por possuírem recursos pervasivos, ou seja, recursos computacionais ociosos que são incorporados ao ambiente e então requisitados com objetivo de processar tarefas de maneira distribuída.

Estes grids podem ser vistos como grids formados por recursos existentes (desktops, servidores, etc.) que ocasionalmente contribuem para o poder de processamento do grid. Estes recursos são inerentemente heterogêneos e potencialmente móveis, entrando e saindo do grid dinamicamente. Sabendo disto, é possível afirmar que grids pervasivos são, em essência, ambientes heterogêneos, dinâmicos e compartilhados; tornando o gerenciamento dos recursos complexo e, quando executado de maneira ineficiente, diminuindo o desempenho do escalonamento de tarefas (?).

Muitos trabalhos propuseram-se a melhorar a adaptabilidade do *framework* Apache Hadoop em ambientes que divergem da suposição inicial, cada um possuindo sua própria proposta e objetivos (?) (?) (?) (?). Em geral os trabalhos dependem da aquisição de algum dado de contexto e subsequente tomada de decisão para maximização de algum objetivo específico.

Sabe-se que, o Apache Hadoop é baseado em configuração estática de arquivos e que

as versões correntes, apesar de possuírem tolerância a falhas, não adaptam-se a variações de recursos ao longo do tempo. Além disto, os procedimentos de instalação forçam o administrador a definir manualmente as características de cada recurso em potencial, como a memória e o número de *cores* de cada máquina, tornando a tarefa difícil e demorada em ambientes heterogêneos. Todos estes fatores impedem a utilização da capacidade total do Hadoop em ambientes voláteis, portanto, é essencial possuir sensibilidade ao contexto para tornar esta adaptação possível.

Este trabalho propõe-se a aumentar a adaptabilidade dos mecanismos de escalonamento do Apache Hadoop utilizando coleta e transmissão de dados de contexto como meio para solucionar os problemas gerados pelo compartilhamento de nós do *cluster*. Buscou-se atingir estes objetivos com o mínimo possível de intrusão e alterações nos mecanismos chave de escalonamento. A coleta de dados é feita com base no tempo médio de duração dos *containers* em execução e a informação só é transmitida caso haja mudanças em relação à última coleta realizada. A transmissão de dados ocorre por meio de uma tabela Hash distribuída, quando esta tabela é alterada o escalonador é alertado e realiza uma atualização nas informações pertinentes. A solução implementada apresenta melhorias de até 40% em alguns casos de testes, os quais utilizam aplicações com características diferentes (CPU-bound, I/O-bound, CPU e I/O-bound), provando ser uma alternativa viável para o aumento da adaptabilidade do Apache Hadoop.

2 FUNDAMENTOS E REVISÃO DE LITERATURA

É necessário definir alguns termos, técnicas e/ou ferramentas para o entendimento deste trabalho. Por exemplo, a conceituação formal de sensibilidade ao contexto é muito importante, uma vez que constitui-se da técnica de obtenção dos dados. Como complemento à sensibilidade ao contexto define-se o que é o ZooKeeper, a ferramenta utilizada para transmissão dos dados coletados. Além disso, é relevante que exista a compreensão de como o Apache Hadoop, seu escalonamento e o paradigma de programação utilizado funcionam, bem como quais trabalhos já foram feitos neste âmbito.

2.1 Apache Hadoop

O *framework* Apache Hadoop origina-se de outro projeto da Apache, o Apache Nutch (Apache Nutch, 2013). O Apache Nutch iniciou em 2002 como um motor de buscas de código livre, porém o projeto encontrou problemas devido a sua arquitetura. Quando a Google publicou um artigo em 2003 descrevendo a arquitetura utilizada no seu sistema de arquivos distribuídos, chamado GFS (*Google File System*), os desenvolvedores do Nutch notaram que uma arquitetura semelhante resolveria seus problemas de escalabilidade. A implementação da ideia foi iniciada em 2004 e o resultado foi nomeado *Nutch Distributed Filesystem* (NDFS). Contudo, a medida em que o projeto se desenvolvia o propósito original do Nutch era deixado em segundo plano, o que culminou na criação de um novo projeto em 2006. Este novo projeto foi nomeado Apache Hadoop e possui o propósito de facilitar o processamento distribuído através do paradigma MapReduce.

2.1.1 Arquitetura geral do Apache Hadoop

O *framework* Apache Hadoop é organizado em uma arquitetura de mestre-escravo e possui dois serviços principais, o serviço de armazenamento (HDFS - Hadoop Distributed File System) e o de processamento (YARN - Yet Another Resource Negotiator), os quais podem ser vistos na Figura 2.1.

Nesta figura, é possível notar a existência de 4 componentes do *framework*. Os componentes acima da linha pontilhada pertencem ao YARN, sendo o Resource Manager e o Node Manager os serviços mestre e escravo respectivamente. O HDFS é composto pelos componen-

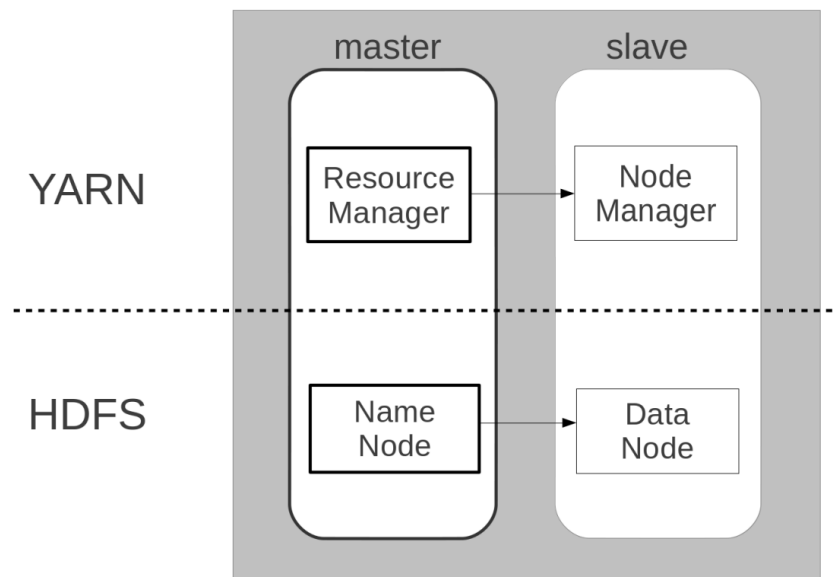


Figura 2.1 – Arquitetura geral do Apache Hadoop

tes abaixo da linha pontilhada, sendo o Name Node e o Data Node os serviços mestre e escravo respectivamente.

2.1.1.1 HDFS

Grande parte do ganho de desempenho oferecido pelo Hadoop decorre do comportamento de levar o processamento até os dados, ou seja, todo processamento é feito com dados locais. Para realizar esta tarefa, o Name Node mantém informações de quais partes de quais arquivos estão em cada Data Node, ou seja, todos os arquivos estão divididos em um grande HD distribuído e o mestre sabe exatamente quais blocos cada escravo possui. Desta forma cada nó executa tantos Maps ou Reduces quanto a quantidade de arquivos locais permitir, diminuindo a necessidade de utilizar a rede para transferência de arquivos e deixando-a disponível para ser utilizada para a transferência dos resultados que são os dados já reduzidos. Um problema dessa abordagem é que o Hadoop possui uma latência muito alta, sendo desaconselhável o uso do Hadoop em aplicações críticas ou de tempo real (?). A Figura 2.2 apresenta um esquema básico da arquitetura do HDFS.

2.1.1.2 YARN

Sendo o componente do Apache Hadoop responsável pela execução do *MapReduce*, o YARN realiza tarefas de gerenciamento e execução do processamento. Um dos objetivos do

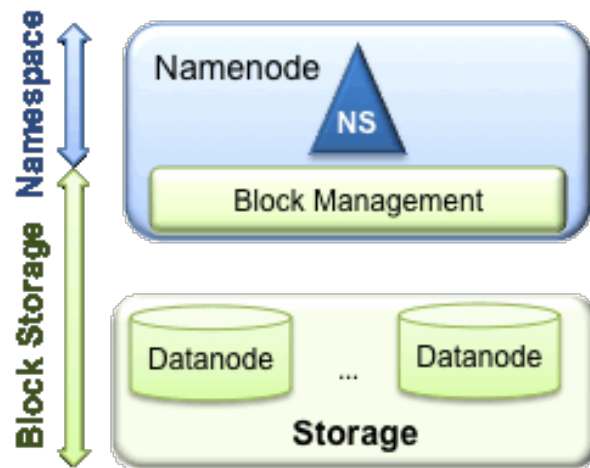


Figura 2.2 – Arquitetura geral do HDFS (HADOOP, 2013a)

YARN é tornar a tarefa de processamento totalmente independente das tarefas de armazenamento, possibilitando que o *cluster* seja utilizado em conjunto com outras ferramentas que não utilizem o paradigma MapReduce (VAVILAPALLI et al., 2013). A Figura 2.3 apresenta um esquema básico da arquitetura do YARN.

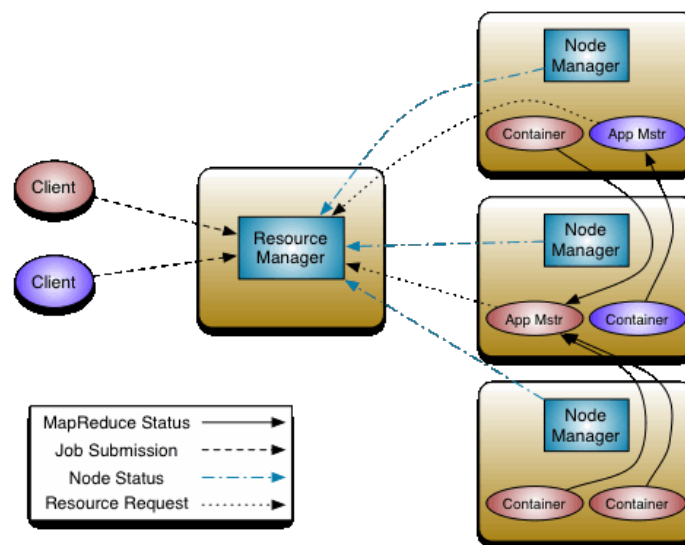


Figura 2.3 – Arquitetura geral do YARN (HADOOP, 2013b)

Na imagem é possível observar dois novos componentes do YARN. O primeiro é um componente do Node Manager, que possui o papel de um escalonador interno de cada aplicação (Application Master - apresentado na imagem como App Mstr), sendo também referenciado como escalonador de tarefas. Considera-se que tarefas sejam uma fração do processamento das aplicações, ou seja, cada tarefa de Map ou Reduce corresponde a uma das divisões que serão processadas em paralelo. É importante não confundir este componente com o escalonador de

aplicações, o qual pertence ao Resource Manager e pode ser melhor compreendido com a leitura da Seção 2.2. O outro componente presente na figura é o Container, o qual representa a alocação de recursos em um nó qualquer do *cluster*. A importância do container vem do fato de que todas as tarefas são executadas em uma instância de container.

É importante notar que existem 2 instâncias de Application Master na figura e que elas estão, assim como os clientes e os containers, coloridas de rosa ou roxo para indicar que pertencem à mesma aplicação, ou seja, o cliente rosa lançou uma aplicação que possui o Application Master e mais 3 containers de processamento. Nota-se também que o Resource Manager recebe informações tanto dos clientes quanto dos Node Managers e Application Masters, centralizando todas elas para um controle dos recursos disponíveis.

2.1.1.3 Configuração do ambiente de execução do Hadoop

Uma característica importante do Hadoop tem relação com sua configuração. O Hadoop utiliza uma série de arquivos em cada um dos nós do *cluster* para definir sua configuração. Estes arquivos de configuração são, na verdade, arquivos XML compostos por parâmetros de configuração e valores que influenciam o comportamento do *framework* no *cluster*. Para conhecimento, esses arquivos são: *core-site.xml*, *yarn-site.xml*, *mapred-site.xml* e *hdfs-site.xml*. Cada um destes arquivos possui propriedades de um serviço do Hadoop, como exemplo o arquivo *hdfs-site.xml* é responsável pela configuração do HDFS na máquina e permite a configuração de parâmetros como o tamanho dos blocos e a replicação dos dados.

É importante salientar que para a execução distribuída, ao menos algumas propriedades mais simples dos arquivos de cada nó, seja este mestre ou escravo, devem ser configuradas. No caso do administrador desejar fazer uma configuração mais específica de cada nó escravo ele deverá editar as propriedades nos arquivos de cada um dos nós. Caso o administrador não queira configurar os nós escravos, eles irão executar com base nos valores *default*, contudo esta decisão irá, provavelmente, afetar o desempenho do *cluster* devido à má configuração.

A utilização de valores *default* facilita a implementação, porém cria alguns problemas na utilização do *framework* em situações diferentes da suposição inicial de um *cluster* dedicado e homogêneo. Em um *cluster* homogêneo a configuração é simples, uma vez que todos os nós possuem a mesma. Contudo, em alguns casos é possível que o *cluster* não seja homogêneo e que a configuração torne-se mais complexa dependendo da quantidade de configurações diferentes existentes no *cluster*.

A configuração estática pode tornar-se ainda mais problemática quando o *cluster* é compartilhado. Neste caso, a informação que estava correta no arquivo XML pode, em algum momento, ficar inconsistente com a realidade devido à utilização do cluster para a execução de algo além de aplicações MapReduce.

2.1.1.4 MapReduce

O paradigma de programação MapReduce é, geralmente, associado com implementações que processam e geram grandes conjuntos de dados. Neste paradigma, todo processamento é dividido em duas etapas, a etapa de Map e a de Reduce, que são inspiradas pelas funções de mesmo nome em linguagens funcionais. Como consequência da divisão em duas etapas, o resultado da primeira é a entrada da segunda e esta transferência de dados é baseada em pares de chave e valor, o que torna possível expressar uma vasta gama de tarefas reais por meio deste paradigma (DEAN; GHEMAWAT, 2004).

O *work-flow* padrão de uma aplicação MapReduce inicia com a entrada de dados, que será dividida em n partes e cada parte será processada individualmente por uma tarefa Map. O resultado das tarefas Map será transmitido na forma de pares chave e valor, que serão utilizados como entrada para as tarefas de Reduce. Por sua vez, as tarefas de Reduce irão receber todos os pares com determinada chave e aplicar um algoritmo sobre os pares, fornecendo uma saída inteligível (?).

2.2 Escalonadores para o Hadoop

Apesar de existirem dois níveis de escalonamento no Hadoop, escalonamento de aplicações e de tarefas, este trabalho influencia apenas o escalonamento de aplicações. O escalonador de aplicações gerencia qual será a primeira aplicação a receber um *container* para seu escalonador de tarefas e quais serão os escalonadores de tarefa que receberão recursos para execução (?). O Hadoop já inclui alguns escalonadores que oferecem maneiras diferentes para a realização do escalonamento de aplicações. Para alterar o escalonador utilizado é necessário alterar uma propriedade no arquivo *yarn-site.XML* e reiniciar o Resource Manager.

Dentre os escalonadores incluídos no Hadoop, o mais simples é o Hadoop Internal Scheduler que utiliza o algoritmo FIFO e tem boa performance em *clusters* onde não existe competição por recursos. Este escalonador suporta até 5 níveis de prioridade, porém a decisão da

próxima aplicação a ser executada sempre levará em consideração a hora de submissão.

Um pouco mais complexo que o Internal Scheduler, o Fair Scheduler é utilizado principalmente para o processamento de lotes de aplicações pequenas e rápidas, opera baseado em um escalonamento de dois níveis e possui o objetivo de realizar uma divisão justa dos recursos. O primeiro nível realiza o escalonamento na forma de filas para cada usuário ativo, dividindo os recursos do cluster igualmente entre as filas. Enquanto isso, o segundo nível realiza o escalonamento dentro de cada fila da mesma forma que o Internal Scheduler (Fair Scheduler, 2013).

A terceira opção, e também o padrão do Hadoop nas últimas versões, é o Capacity Scheduler, o qual foi projetado para a utilização compartilhada do Hadoop e busca a maximização do *throughput* e da utilização do *cluster*. Seu funcionamento baseia-se em garantias mínimas de capacidade para os usuários, ou seja, qualquer usuário terá sempre uma garantia mínima de recursos para utilização. Porém quando algum usuário está com seus recursos ociosos o escalonador repassa a capacidade deste usuário para aqueles que estão utilizando o *cluster*. Esta estratégia fornece elasticidade com um bom custo benefício, uma vez que diferentes organizações possuem diferentes horários de pico para o processamento de informações. Este escalonador é capaz de rastrear os recursos registrados no Resource Manager, embora esta informação pode não ser consistente com a realidade, e monitorar quais deles estão livres e quais estão sendo utilizados pelo *framework* (?).

A existência destes escalonadores adiciona flexibilidade no gerenciamento do *framework*. Apesar disso, os escalonadores disponíveis não detectam nem reagem à dinamicidade e heterogeneidade do ambiente. Para a utilização do Hadoop em ambientes pervasivos é necessário que exista uma capacidade de adaptação neste componente.

2.3 Trabalhos Relacionados

Existem outras implementações de escalonadores além dos 3 escalonadores já incluídos no Hadoop, nas quais é possível identificar diversas propostas de adaptação. Cada proposta possui métodos e objetivos específicos, tornando um estudo sobre estas implementações interessante como ponto de partida para a proposta deste trabalho. Logo, buscou-se identificar quais técnicas foram mais utilizadas e quais os tipos de adaptação mais explorados nestes escalonadores. A seguir encontram-se os trabalhos relacionados e um breve resumo sobre a proposta, método e objetivos esperados com a adaptação.

Os autores do escalonador CASH (*Context Aware Scheduler for Hadoop* - Escalonador Sensível ao Contexto para o Hadoop) (KUMAR et al., 2012) têm o objetivo de melhorar o rendimento geral do *cluster*. O trabalho utiliza a hipótese de que grande parte das aplicações são periódicas e executadas no mesmo horário, além de possuírem características de uso de CPU, rede, disco etc. semelhantes. O trabalho ainda leva em consideração que com o passar do tempo os nós tendem a ficar mais heterogêneos. Baseados nestas hipóteses e com objetivo de melhorar o desempenho geral, foi implementado um escalonador que classifica tanto as aplicações como as máquinas com relação ao seu potencial de CPU e E/S, podendo então distribuir as aplicações para máquinas que tem uma configuração apropriada para sua natureza.

No trabalho *A Dynamic MapReduce Scheduler for Heterogeneous Workloads* (Um Escalonador de MapReduce Dinâmico para Cargas de Trabalho Heterogêneas) (TIAN et al., 2009), os autores utilizam a técnica de classificar as aplicações e máquinas de acordo com a quantidade de E/S ou CPU. E assim como no CASH, o principal objetivo é a melhora de rendimento no *cluster*. Uma das diferenças, no entanto, é que esta implementação utiliza um escalonador com três filas.

Semelhante às propostas anteriores, a proposta COSHH (*A Classification and Optimization based Scheduler for Heterogeneous Hadoop Systems* - Um Escalonador Baseado em Classificação e Otimização para Sistemas Heterogêneos do Hadoop) (RASOOLI; DOWN, 2012) utiliza a classificação das aplicações e máquinas em classes e busca por pares que possuam a mesma classe. Esta busca é feita por um algoritmo que reduz o tamanho do espaço de busca para melhorar o desempenho. O objetivo desta solução é a melhora do tempo médio em que as aplicações são completadas, além de oferecer um bom desempenho quando utilizando somente a fatia mínima de recursos e, ainda, proporcionar uma distribuição justa.

O escalonador LATE (*Longest Approximation Time to End* - Aproximação do Tempo de Término mais Longo) (ZAHARIA et al., 2008), utiliza como informação de contexto o tempo estimado de término da tarefa com base em uma heurística que faz a relação de tempo decorrido e do *score* – um valor que indica quanto do processamento já foi feito. Essa informação também é utilizada para gerar um limiar que indique quando a lentidão de uma tarefa indica sintomas de erros e a partir desta informação iniciar uma tarefa especulativa em outra máquina possivelmente mais rápida. Este trabalho possui o objetivo de reduzir o tempo de resposta em *clusters* grandes que executam muitas aplicações de pequena duração.

Outro trabalho que utiliza a ideia de mensuração do progresso de uma tarefa é o SAMR

(A *Self-adaptive MapReduce* - MapReduce Auto Adaptativo) (CHEN et al., 2010), nesta implementação a informação de contexto é referente ao cálculo do progresso de uma tarefa com objetivo de identificar se é o lançamento de uma tarefa especulativa é necessário ou não. Uma diferença desta solução é com relação ao cálculo do progresso, o qual varia de acordo com informações do ambiente em que a tarefa está executando. O principal objetivo do trabalho é a redução do tempo de execução das tarefas. As informações do ambiente utilizadas para a tomada de decisão consistem de informações históricas contidas em cada nó, e a decisão é tomada após um ajuste do peso de cada estágio do processamento.

A proposta dos autores do *Quincy* (ISARD et al., 2009) difere-se de todos os outros trabalhos, pois possui um escopo muito maior e visa tanto o *Hadoop* como outras ferramentas. O trabalho possui como objetivo a melhora do desempenho geral de um *cluster*, e utiliza a distribuição de recursos como informação de contexto para alcançá-lo. A contribuição do trabalho é uma modificação da maneira tradicional de tratamento da distribuição dos recursos. A solução proposta mapeia os recursos em um grafo de capacidades e demandas, para então calcular o escalonamento ótimo a partir de uma função global de custo.

A proposta *Improving MapReduce Performance through Data Placement in heterogeneous Hadoop Clusters* (Melhorando o Desempenho do MapReduce em Clusters Heterogêneos com Hadoop Através da Localização dos Dados) (XIE et al., 2010), busca melhorar o desempenho de aplicações CPU-bound através da melhor distribuição destes dados. Esta solução utiliza principalmente a localidade dos dados como informação para tomada de decisões. O ganho de desempenho é dado pelo rebalanceamento dos dados nos nós, deixando nós mais rápidos com mais dados. Isso diminui o custo de tarefas especulativas e de transferência de dados pela rede.

Outra proposta que busca um rebalanceamento de carga é (?), porém esta utiliza-se de uma abordagem diferente. Neste trabalho o rebalanceamento de carga é alcançado através de um sistema baseado na lei de oferta e demanda, o qual permite a cada usuário influenciar diretamente o escalonamento por meio de um parâmetro chamado taxa de gastos. Este parâmetro indica qual a prioridade da aplicação, sendo que, em horários de maior concorrência a taxa de gasto será maior. O principal objetivo deste trabalho é permitir um compartilhamento de recursos dinâmico baseado em preferências configuradas pelos próprios usuários.

Nota-se que no geral as propostas de melhora da adaptabilidade apresentadas pelos trabalhos utilizam principalmente a técnica de rebalanceamento de carga entre nós de diferentes capacidades. Ainda, é possível notar que os trabalhos, em sua maioria, seguem três opções:

1. Classificação dos nós e das aplicações de acordo com sua capacidade (CPU ou E/S) seguido de um algoritmo que delega aplicações para nós do mesmo tipo;
2. Modificação da tomada de decisão sobre o lançamento ou não de uma tarefa especulativa através de novas heurísticas;
3. Redistribuição dos dados para que eles fiquem em nós mais rápidos.

Ainda que a opção 3 assemelhe-se com a opção 1, é importante diferenciar a natureza delas. Enquanto a opção 1 classifica os nós em diversas categorias, a opção 3 apenas delega mais dados, e consequentemente mais tarefas, aos nós mais rápidos. Embora pareça uma solução mais simples ela evita transferência de dados pela rede, o que aconteceria caso a divisão dos dados para os nós fosse igualitária. No caso dos nós receberem uma quantidade igual de dados inicialmente, os nós mais rápidos terminariam o processamento primeiro e ficariam ociosos até que algum dado fosse transferido para eles.

Embora não relacionado com escalonamento, o trabalho (LI et al., 2014) propõe uma ferramenta de auto configuração que em partes assemelha-se com a proposta deste trabalho. Entretanto, a proposta de Li é baseada em aprendizagem de máquina para a auto configuração de alguns parâmetros chave do Hadoop e apresenta execuções até 10 vezes mais rápidas após a otimização dos parâmetros; enquanto a proposta deste trabalho busca melhorar a informação disponível para que o escalonamento seja adaptável ao compartilhamento do ambiente.

3 MÉTODOS E DESENVOLVIMENTO

Este capítulo descreve as etapas de desenvolvimento e as metodologias empregadas neste trabalho. Buscaram-se estratégias sem intrusão ou grandes modificações nas políticas de escalonamento já implementadas pelo *framework*. Nas Seções 3.1 e 3.2 são apresentados em maior detalhe os coletores de contexto e a ferramenta de comunicação distribuída utilizados neste trabalho, respectivamente. Finalmente, na Seção 3.3 explica-se em profundidade a solução implementada neste trabalho e as justificativas para as decisões tomadas.

3.1 Coletores de Contexto

Após um estudo aprofundado dos escalonadores do Hadoop, ficou claro que o Capacity Scheduler já está estruturado de maneira a oferecer escalabilidade e um desempenho satisfatório. Porém, este escalonador possui um ponto fraco, o qual é exposto quando utilizado em um ambiente compartilhado. O Capacity Scheduler só recebe informações sobre os Node Managers no momento da inicialização destes, e ainda, a informação é obtida de um arquivo de configuração. A importância de que as informações sobre os Node Managers estejam atualizadas decorre da dependência intrínseca do escalonamento com a disponibilidade de recursos, na qual uma informação errada pode influenciar o algoritmo de maneira prejudicial e diminuir o desempenho das tarefas. Com base nestas observações, o primeiro passo para a solução é a coleta de dados sobre os recursos dos Node managers, ou seja, a adição de sensibilidade ao contexto.

3.1.1 Sensibilidade ao Contexto

Contexto é definido por (DEY, 2001) como qualquer informação que pode ser utilizada para caracterizar a situação de uma entidade (pessoa, lugar ou objeto) considerada relevante para a interação entre usuário e aplicação. Um exemplo de utilização de contexto é quando um usuário acessa um site por meio de um dispositivo móvel e o site carrega automaticamente a versão *mobile*, a qual possui alterações que aumentam a compatibilidade com o tipo de dispositivo sendo utilizado. Outra situação semelhante ocorre quando dados de localidade são utilizados para melhorar os resultados de um motor de buscas, mostrando primeiro os resultados mais relacionados com a região ou idioma do usuário. Nota-se que nas duas situações a aplicação

utiliza dados específicos, o tipo do dispositivo e a localização do usuário, coletados no momento da execução e utiliza-os para adaptar o seu funcionamento de maneira a oferecer maior conforto ou usabilidade ao usuário.

Sendo assim, se uma aplicação é capaz de coletar informações sobre a situação do sistema no qual está sendo executada, ela é capaz de coletar informações de contexto. Porém a simples coleta não aumenta, de fato, o desempenho desta aplicação, é necessário que a aplicação seja capaz de responder às mudanças detectadas no ambiente. Esta capacidade de detecção e reação é caracterizada como sensibilidade ao contexto por (?) e vai ao encontro da definição de (BALDAUF; DUSTDAR; ROSENBERG, 2007) em que o sistema deve detectar mudanças e adaptar suas operações sem intervenção explícita do usuário, aumentando assim a usabilidade e eficácia da aplicação.

3.1.2 Implementação

Em um primeiro momento, buscou-se diminuir a dependência nos arquivos XML para a configuração dos recursos dos Node Managers com intuito de facilitar a configuração inicial dos nós e futuramente utilizar este mesmo mecanismo para a inclusão do suporte ao compartilhamento dos recursos. Para isto, fez-se necessário o desenvolvimento de um conjunto de coletores de contexto capazes de coletar de maneira eficiente os recursos do nó em questão no momento da inicialização do serviço Node Manager e, conseqüentemente, passar a informação correta sobre os recursos no momento do registro no Resource Manager.

Embora esta adição já facilite a utilização do Hadoop em um *cluster* de natureza heterogênea, ainda não é suficiente para que o Hadoop seja utilizado eficientemente em um *cluster* compartilhado. A razão para esta afirmação é que, embora a informação esteja correta no momento de inicialização, é possível que durante a execução das aplicações de *MapReduce* os recursos comecem a ser utilizados por outros usuários, diminuindo a capacidade disponível para o Hadoop. O problema causado pela utilização de alguns nós por outros usuários sem a devida atualização dos dados no escalonador é a sobrecarga destes nós, uma vez que o Hadoop irá tentar utilizar memória e processadores que já estão alocados à outros processos.

Com objetivo de solucionar o problema causado pelo compartilhamento, é necessário que a coleta de dados ocorra não apenas na inicialização do Node Manager mas ao longo da execução do serviço em intervalos periódicos. Para isso optou-se pela utilização de uma *thread* para a coleta e transmissão, se necessária, dos dados.

O coletor escolhido para a tarefa foi o coletor desenvolvido pelo projeto PER-MARE (Kirsch-Pinheiro, M., 2013), o qual utiliza a interface padrão do Java para monitoramento, `OperatingSystemMXBean` (Oracle, 2014). A implementação deste coletor de contexto é baseada em uma interface, uma classe abstrata e as classes de coleta dos recursos desejados. Devido ao seu projeto, coletores de novas informações podem ser facilmente criados, aumentando assim a quantidade de informação disponível para o escalonador.

A interface `OperatingSystemMXBean`, possibilita o acesso às informações do sistema no qual a JVM está sendo executada. Uma vez que a classe abstrata implementa esta interface, todas suas herdeiras poderão utilizá-la.

As classes utilizadas neste trabalho fazem a coleta de memória física disponível e processadores disponíveis, os recursos suportados por padrão no Hadoop. É possível visualizar o digrama de classes na figura 3.1, onde estão presentes alguns exemplos de possíveis coletores a serem utilizados.

Cada instância de Node Manager possui um conjunto de coletores (um para memória e um para processadores), os quais realizam a coleta num intervalo pré-definido. Os coletores são executados por uma *thread* independente e possuem um intervalo de 30 segundos entre as coletas para não causar sobrecarga ou interrupção no processamento das tarefas *Map* e *Reduce*.

3.2 Comunicação Distribuída

Para que a informação coletada pelos coletores de contexto possa afetar o escalonamento é necessário que exista uma maneira para os Node Managers transmitirem os dados atualizados ao escalonador, a ferramenta escolhida para esta tarefa foi o ZooKeeper.

3.2.1 ZooKeeper

O ZooKeeper é um projeto da Apache e fornece ferramentas eficientes, confiáveis e tolerantes à falha para a coordenação de sistemas distribuídos (?). Inicialmente, o ZooKeeper foi implementado como um componente do Hadoop e virou um projeto próprio conforme cresciam suas funcionalidades e sua utilização em outras aplicações.

A arquitetura utilizada no ZooKeeper é a de cliente-servidor, sendo o servidor o próprio ZooKeeper (chamado de *ensemble*), enquanto a aplicação que o está utilizando assume o papel de cliente. Os dados do ZooKeeper ficam armazenados em *zNodes*, abstrações que podem ser

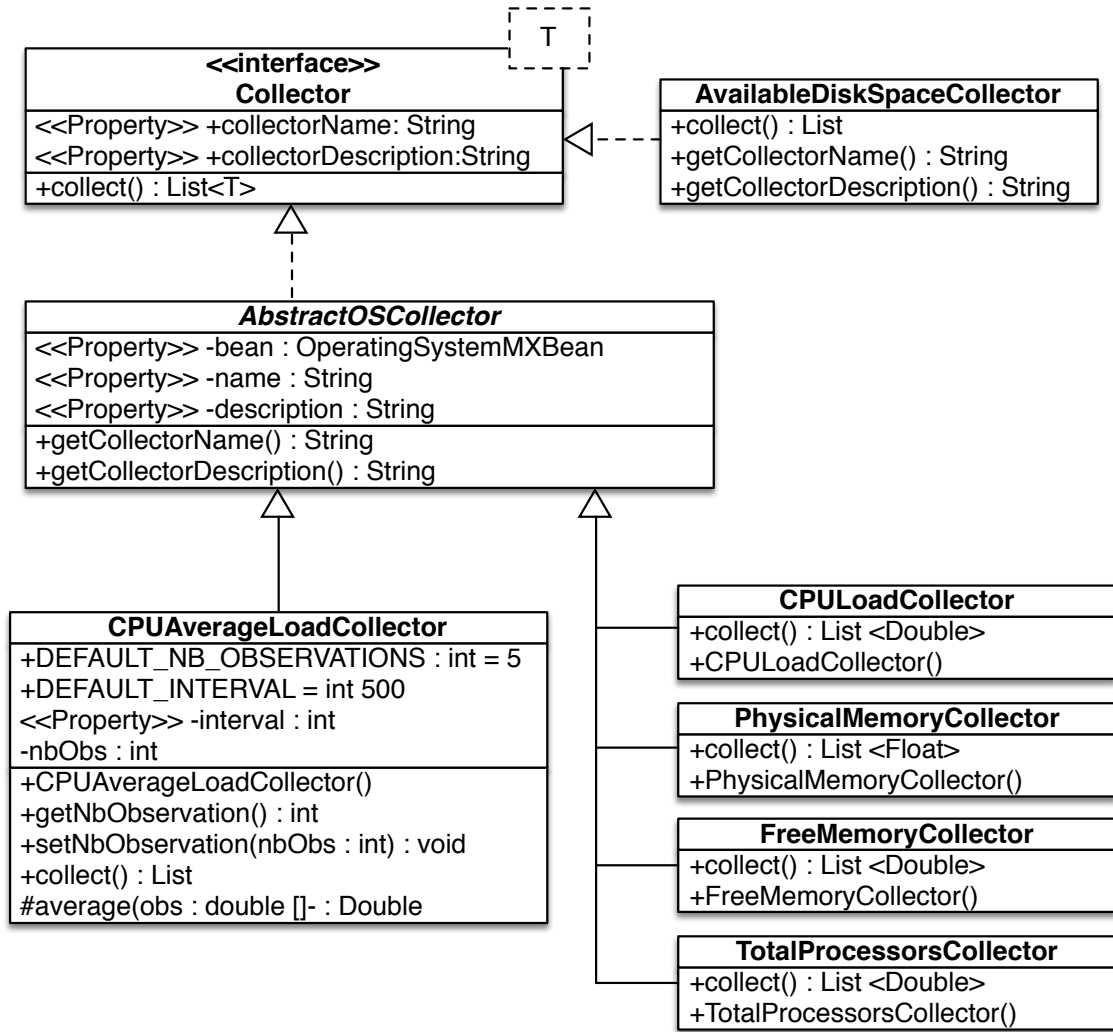


Figura 3.1 – Diagrama de classes dos coletores de contexto

tanto um *container* de dados quanto de outros *zNodes*, e formam um sistema de arquivos hierárquico que pode ser comparado à estrutura de uma árvore. Para garantir a consistência deste sistema de arquivos o ZooKeeper utiliza operações de escrita linearizáveis, as quais são obrigatoriamente processadas pelo servidor líder que é, então, encarregado de propagar as mudanças para os demais participantes do *ensemble* (PHAM et al., 2014).

Um dos principais recursos do ZooKeeper são os *Watchers*, interfaces providas pelo *framework* que permitem aos clientes monitorarem certos *zNodes*. Quando um *Watcher* é registrado como monitor de um *zNode*, ele pode ser configurado para monitorar a alteração dos dados do *zNode*, a criação/remoção de *zNodes* filhos, ou ainda para qualquer tipo de alteração no *zNode* e seus filhos. Quando um *zNode* sofre uma alteração que o *Watcher* está monitorando uma *callback* é disparada para que o cliente faça o processamento desejado (?).

No âmbito deste trabalho, os serviços do ZooKeeper são utilizados para monitorar as in-

formações de contexto coletadas nos nós escravos e transmiti-las para o escalonador. Sendo que a comunicação é feita através de processos que atualizam e monitoram o conteúdo de *zNodes*.

3.2.2 Implementação

A flexibilidade oferecida através dos *zNodes*, permite que qualquer estrutura de dado seja inserida como informação. Optou-se por utilizar uma Tabela Hash, o que permite realizar a tarefa com apenas um *zNode* e possibilita aos Node Managers a inserção das informações e ao escalonador o monitoramento de maneira simples.

Na solução adotada há apenas um Watcher, o escalonador, que monitora o *zNode* da Tabela Hash. Qualquer alteração na tabela dispara uma *callback* no escalonador, que por sua vez percorre a tabela à procura das modificações e realiza as alterações pertinentes nas informações sobre os recursos. Esta abordagem permite que o escalonador realize operações apenas quando necessário e não desperdice tempo acessando a tabela quando não houverem atualizações a fazer. A estrutura adotada no trabalho pode ser visualizada na Figura 3.2.

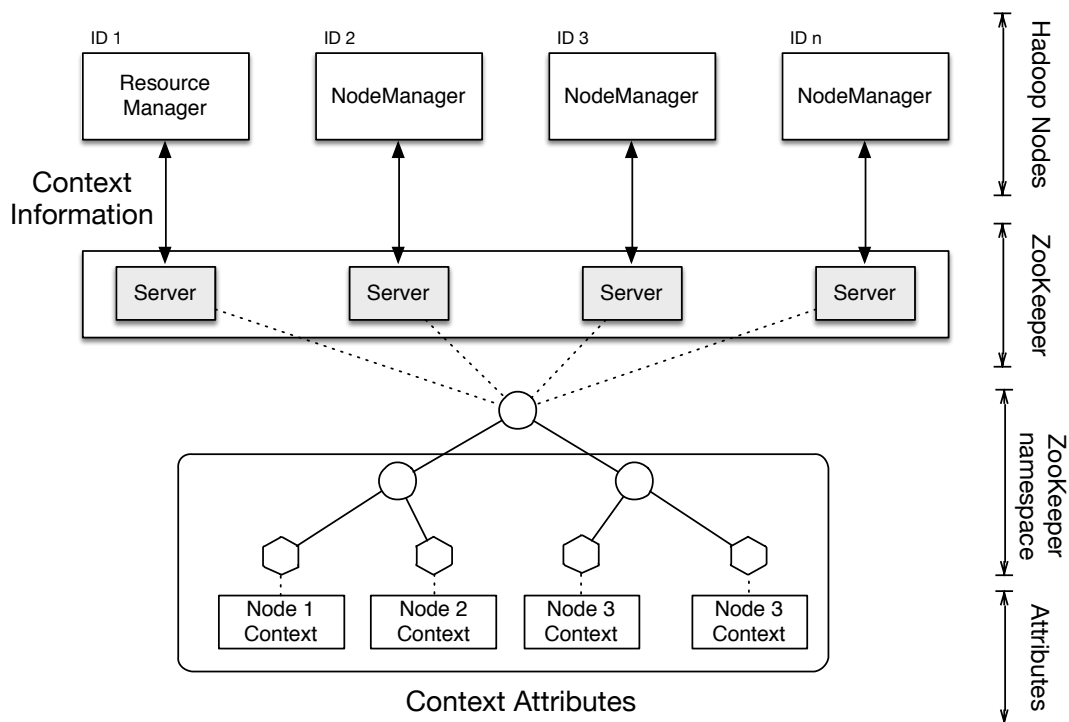


Figura 3.2 – Estrutura do ZooKeeper

Como a solução escolhida apresenta dois papéis de clientes ZooKeeper, a seguir encontra-se uma explicação aprofundada de cada um destes papéis.

- Cliente de Monitoramento: o papel de monitor é desempenhado pelo escalonador. O

escalonador implementa a interface *Watcher*, fornecida pelo ZooKeeper, e monitora o *zNode* que contém a Tabela Hash. No momento da inicialização o escalonador cria um *zNode* e insere uma Tabela Hash vazia como informação, então finalmente, inicia o monitoramento do *zNode*. Quando a *callback* é acionada, a *thread* percorre a tabela em busca de qual informação foi atualizada, atualiza os dados do escalonador e reinicia o monitoramento.

- Clientes de Atualização: o papel de atualização é realizado pelos Node Managers. Cada Node Manager lança, no momento de sua inicialização, uma *thread* que faz a coleta de dados e caso haja alteração com relação à última leitura insere os dados atualizados na tabela hash. A coleta dos dados é realizada a cada 30 segundos, média de tempo observada em *containers* executados em um *cluster* de funcionamento normal. Para evitar o envio de dados por alterações naturais do sistema e que não impactariam o escalonamento foi implementado um *threshold* mínimo de alteração de 100 Mb e/ou 1 *core*.

3.3 Solução Implementada

O Capacity Scheduler já possui adaptabilidade à entrada de novos Node Managers no cluster desde que um dos arquivos XML do Resource Manager tenha o *host* do novo manager incluído e o Node Manager seja inicializado.

3.3.1 Integração dos Coletores de Contexto

3.3.2 Integração dos Clientes

3.3.3 Justificativas para

4 EXPERIMENTOS E RESULTADOS

Este capítulo contém informações detalhadas sobre os experimentos realizados e os resultados obtidos. Embora o principal caso estudado foi o de degradação dos recursos em virtude de um compartilhamento dos nós, os experimentos realizados podem ser divididos em três categorias de acordo com seu objetivo. As três categorias são: (1) experimentos realizados em ambiente manipulado para verificação do desempenho da solução de forma simplificada; (2) experimentos realizados utilizando a implementação real para verificação do desempenho em ambiente controlado porém próximo à realidade; (3) experimentos em escala para visualização do desempenho com relação à escalabilidade da solução.

4.1 Considerações Iniciais Sobre os Experimentos

Em virtude de algumas informações sobre os experimentos serem comuns à todos eles, como os casos de teste e as aplicações utilizadas, esta Seção é destinada à apresentação destas informações. Além disso, a apresentação dos resultados utiliza diagramas de Gantt com uma estrutura modificada em relação à usualmente encontrada na literatura e será, também, explicada detalhadamente nesta Seção.

4.1.1 Casos de teste

A primeira informação relevante que refere-se à todos experimentos são os casos de teste. Todos os experimentos utilizam os mesmos casos de teste, possuindo apenas objetivos diferentes. Os casos de teste representam situações de compartilhamento, sendo que 2 casos utilizam a implementação padrão e 2 casos utilizam a solução proposta por este trabalho. Além disso, com a criação dos casos de teste buscou-se facilitar a comparação entre os resultados alcançados. A descrição dos casos de testes encontra-se a seguir.

Caso *Default Dedicado* (DefDed): utiliza a versão sem alterações do Hadoop e representa uma situação sem compartilhamento, onde o usuário possui acesso à todos os recursos do *cluster* em qualquer momento. Isto implica que os recursos informados ao escalonador **sempre** corresponderão aos recursos disponíveis para o Hadoop. Consideram-se recursos informados como os dados que o escalonador utiliza para realizar suas políticas de escalonamento, enquanto, recursos disponíveis são aqueles que estão livres e/ou sendo utilizados pelo próprio

Hadoop. Utilizando uma notação percentual, os recursos informados são de 100% e os recursos disponíveis são de 100% durante toda execução.

Caso *Default Compartilhado* (DefSha): utiliza a versão sem alterações do Hadoop e representa a situação decorrente do compartilhamento dos nós do *cluster* com outros usuários. Como consequência do compartilhamento, é possível que em, algum momento, ocorra uma inconsistência entre a quantidade de recursos informada e disponível. Este caso aplica o comportamento padrão do Hadoop, no qual os recursos são informados por meio de arquivos XML **somente** na inicialização do serviço e nunca são atualizados. Para representar a situação onde apenas alguns nós são utilizados, optou-se por deixar alguns nós inalterados e outros com redução de recursos. Nos experimentos de 4 escravos, 2 nós tiveram 6 *cores* e 6 Gb de RAM reduzidos através de um código C, restando 2Gb e 2 *cores* para o SO e Hadoop. No contexto do cluster, os recursos informados são de 100% enquanto os disponíveis são de 62,5%.

Caso Adaptativo com atualização antes (MyBef): repete as especificações do Caso DefSha, porém possui a implementação descrita no Capítulo 3. Este caso representa a situação de quando o início do compartilhamento ocorre **antes** da coleta e transmissão de dados, a qual ocorre também **antes** do lançamento de uma aplicação *MapReduce*. Em ordem cronológica, primeiramente outro usuário inicia a utilização do *cluster*, então a coleta e transmissão de dados ocorre, e finalmente uma aplicação Hadoop é lançada. O resultado desta sequência de eventos é que quando a nova aplicação *MapReduce* for submetida ao *cluster*, este já estará com os dados atualizados. Em notação percentual, os recursos informados são de 62,5% e os recursos disponíveis são de 62,5% durante toda aplicação.

Caso Adaptativo com atualização durante (MyDur): representa uma extensão do Caso MyBef (também utilizando a implementação do Capítulo 3) na qual o início do compartilhamento ocorre **antes** da coleta e transmissão dos dados e **após** a submissão de uma aplicação *MapReduce*. Em ordem cronológica, ocorre o lançamento de uma aplicação *MapReduce*, então quando esta aplicação já está em execução o compartilhamento tem início e finalmente a coleta e transmissão de dados é feita. O resultado desta sequência de eventos é que a aplicação será lançada numa situação onde o *cluster* possui a informação errada (Caso DefSha) e terá de se adaptar à nova configuração dos recursos (Caso MyBef) durante a execução. Em notação percentual, os recursos informados no início da aplicação são de 100%, enquanto os recursos disponíveis são de 62,5%. Após a coleta e transmissão de dados os recursos informados também passam a ser 62,5%.

4.1.2 Aplicações de teste

Além dos casos de teste, outra característica importante e comum à todos experimentos são as aplicações. Embora aplicações de *Big Data* geralmente possuem dependência de memória, outros fatores como a utilização de CPU e E/S podem influenciar no desempenho. Na busca de indícios de que a solução apresenta ganhos quando utilizada com aplicações de diferentes características, decidiu-se pela utilização de 3 aplicações de *benchmark*, cada uma com diferentes requisições de memória, CPU e E/S. As aplicações são as seguintes:

- TeraSort: o objetivo do TeraSort (?) é ordenar um conjunto de dados o mais rápido possível. Este *benchmark* de ordenação estressa tanto a memória como o CPU em virtude das comparações e armazenamento temporário;
- WordCount: o *benchmark* WordCount é um exemplo básico de *MapReduce*. Seu objetivo é contar o número de ocorrências de cada palavra de um texto. Como a utilização de memória e E/S é limitada nesta aplicação (tanto a etapa de processamento quanto a saída da aplicação possuem estruturas pequenas em comparação ao arquivo de entrada), o desempenho desta aplicação é determinado pelo CPU;
- TestDFSIO: o *benchmark* TestDFSIO é um teste de leitura e escrita para o HDFS. Este *benchmark* é útil para estressar o HDFS, descobrir *bottlenecks* na rede, SO e configuração do Hadoop. O objetivo é prover uma mensuração de quão rápido o *cluster* é em termos de E/S. Tanto a memória quanto o CPU são pouco utilizados.

Optou-se pela utilização das aplicações implementadas no *HiBench* (?), um conjunto de *benchmarks* para *clusters* Hadoop que foi utilizado nos trabalhos (?) (?) (?). O tamanho de entrada utilizado para cada aplicação nos testes com 4 escravos foi: um conjunto de dados de 15 Gb para o Terasort, 90 arquivos de 250 Mb para o TestDFSIO e um arquivo de 10 Gb para o WordCount.

4.1.3 Configurações de Hardware e Software

O experimento foi realizado no *cluster* genepi do Grid'5000. A configuração do *cluster* utilizado na maior parte dos experimentos foi a de 1 mestre e 4 escravos, sendo que cada um destes nós possuem a seguinte configuração: 2 CPUs Intel(R) Xeon(R) E5420 2.5GHz (totali-

zando 8 cores por nó) e 8 Gb RAM. Todos os nós do experimento possuíam o sistema operacional Ubuntu x64-12.04, com a JDK 1.8 instalada e a versão 2.6.0 do Hadoop configurada. Todas as informações foram obtidas através do sistema de *logs* do Hadoop.

4.1.4 Apresentação dos resultados

Ainda ligado aos experimentos porém com relação aos resultados, os diagramas de Gantt apresentados neste trabalho são modificados para inclusão de mais informações. A apresentação dos diagramas está agrupada por aplicação, sendo que cada aplicação possui, geralmente, 4 diagramas. Cada um dos diagramas de uma aplicação corresponde a um caso de teste e todos os diagramas de uma aplicação utilizam a mesma escala de tempo.

Cada diagrama possui 2 ou mais linhas, sendo que cada linha representa um recurso (nó do *cluster*). Estas linhas de recurso possuem diversos separadores verticais (formando diversos segmentos), os quais indicam que ao menos um *container* iniciou/terminou sua execução. Cada segmento apresenta diferentes alturas e tons de cores, os quais são utilizados para representar a carga de *containers Map* do nó. Quanto mais escuro for o tom de um segmento, mais *containers* ele possui em execução; o mesmo aplica-se para a altura do segmento, quanto mais alto mais *containers Map* em execução.

Embora as análises foram feitas principalmente com os *containers Map*, os *containers* de Reduce e do Application Master consomem recursos do *cluster* e devem ser apresentados para uma representação fiel da situação real. Por este motivo, o *container* Application Master é representado na cor azul, os *containers* de Reduce são representados pela cor verde e os *containers Map* são representados em escalas de cinza, sendo branco indicando 0 *containers* e preto indicando o máximo de *containers* em execução em algum momento naquele experimento. Para referência, diagramas da Seção 4.2 possuem máximo de 16, diagramas das demais Seções possuem máximo de 8.

4.2 Experimento controlado

Este experimento foi realizado com objetivo de obter indícios de que a solução poderia contribuir para uma melhora no desempenho do processo de escalonamento quando o Hadoop é utilizado num ambiente onde exista degradação de recursos em virtude de compartilhamento. O experimento simplifica a solução para facilitar a obtenção dos dados em menor tempo. A

situação desejada de expressar com o experimento é de quando os nós do *cluster* começam a ser utilizados por outros usuários antes/durante a aplicação *MapReduce*.

4.2.1 Procedimentos

Para que o experimento fosse totalmente controlado, decidiu-se utilizar a manipulação de informações e exclusão de nós para representar o compartilhamento. Na situação real (ver Seção 4.3) o *cluster* teria 4 escravos e ficaria com uma configuração onde metade dos seus recursos já estariam sendo utilizados por outra aplicação. Por exemplo, outra aplicação utilizaria 4 cores e 4 Gb de memória em cada nó. Para representar esta situação de maneira simples e rápida, o experimento foi realizado com apenas 2 nós (diminuindo os recursos disponíveis pela metade) que tiveram a informação da quantidade de recursos dobrada. Como resultado, o escalonador recebe informação de que os recursos disponíveis são de 16 cores e 16 Gb de memória em cada nó quando na verdade são de apenas 8 cores e 8 Gb.

Foram realizados experimentos com as 3 aplicações para que as diferenças de comportamento entre elas já pudessem ser estudadas num cenário de sobrecarga de memória e processamento.

4.2.2 Resultados e Interpretações

Os resultados dos experimentos com as aplicações TeraSort, TestDFSIO e WordCount podem ser visualizados nas Tabelas 4.1, 4.2 e 4.3 e nas Figuras 4.1, 4.2 e 4.3, respectivamente.

Tabela 4.1 – Resumo dos resultados do TeraSort controlado em segundos.

Caso	A	B	C	D
Tempo Total de Map (s)	149	788	348	477
Tempo Médio de Map (s)	39.47	222.97	38.38	68.42
Desvio Padrão	15.73	59.86	18.09	29.91
# Tarefas Map	76	76	76	76
# Tarefas Especulativas	2	1	3	1

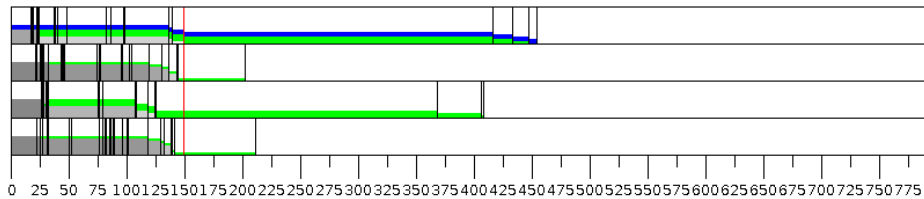
Tabela 4.2 – Resumo dos resultados do TestDFSIO controlado em segundos.

Caso	A	B	C	D
Tempo Total de Map (s)	139	444	239	364
Tempo Médio de Map (s)	38.95	85.01	32.20	81.62
Desvio Padrão	17.20	69.08	8.30	73.60
# Tarefas Map	90	90	90	90
# Tarefas Especulativas	0	9	0	1

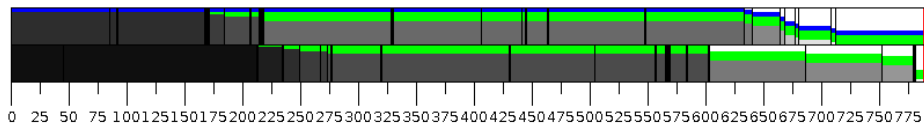
Tabela 4.3 – Resumo dos resultados do WordCount controlado em segundos.

Caso	A	B	C	D
Tempo Total de Map (s)	155	1009	309	805
Tempo Médio de Map (s)	43.76	208.39	41.73	175.80
Desvio Padrão	15.61	128.90	10.99	151.59
# Tarefas Map	90	90	90	90
# Tarefas Especulativas	1	15	1	10

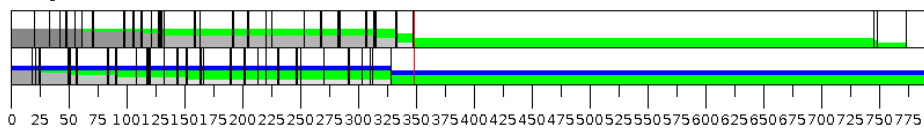
defDed



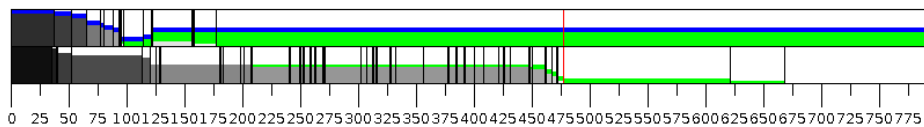
defSha



myBef



myDur



Legends

Map containers Reduce containers Application Master container

Taller segments (all types included) and darker tones (only maps included) indicate more containers in €

Figura 4.1 – Diagrama de Gantt para os experimentos controlados com TeraSort

4.3 Experimento real

Este experimento foi realizado com objetivo de obter indícios de que a solução poderia contribuir para uma melhora no desempenho do processo de escalonamento quando o Hadoop é utilizado num ambiente onde exista degradação de recursos em virtude de compartilhamento.

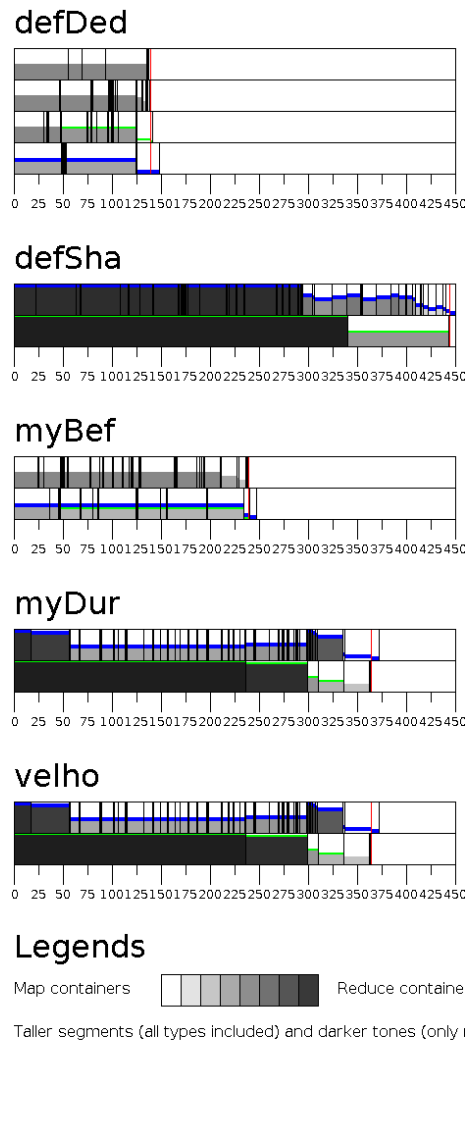


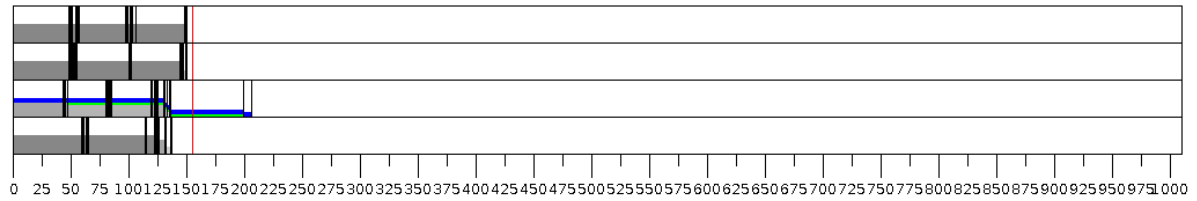
Figura 4.2 – Diagrama de Gantt para os experimentos controlados com TestDFSIO

O experimento utiliza a solução descrita no Capítulo 3. A situação expressada com este experimento é de quando os nós do *cluster* começam a ser utilizados por outros usuários antes/durante a aplicação *MapReduce*.

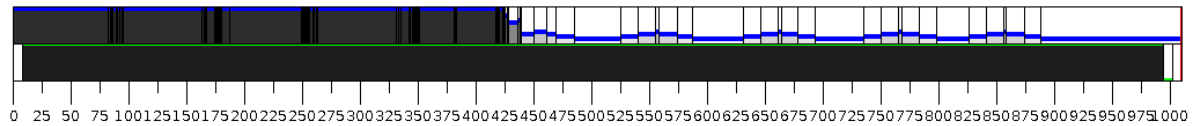
4.3.1 Procedimentos

Diferentemente do experimento anterior, neste experimento buscou-se o comportamento da solução em um ambiente realmente compartilhado. O *cluster* possui 4 escravos e todos os escravos terão, em algum momento, seus recursos disponíveis reduzidos pela metade devido ao compartilhamento, ou seja, outra aplicação irá utilizar 4 cores e 4 Gb de memória em cada nó. Para alcançar esta situação foi implementada uma aplicação em C, na qual a quantidade

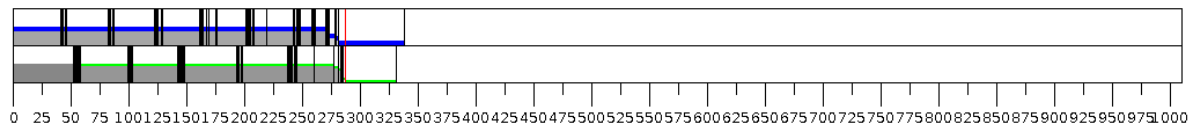
defDed



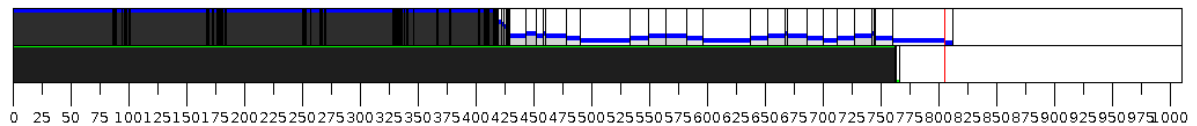
defSha



myBef



myDur



Legends




Map containers  Reduce containers  Application Master container 
 Taller segments (all types included) and darker tones (only maps included) indicate more containers in execution.

Figura 4.3 – Diagrama de Gantt para os experimentos controlados com WordCount

desejada de *threads* é inicializada e cada uma delas utiliza 1 core e 1 Gb de memória.

4.3.2 Resultados e Interpretações

Os resultados dos experimentos com as aplicações TeraSort, TestDFSIO e WordCount podem ser visualizados nas Tabelas ??, ?? e ?? e nas Figuras 4.4, 4.5 e 4.6, respectivamente.

4.4 Experimento de escala

Uma vez que os experimentos anteriores já responderam algumas questões importantes sobre a viabilidade da solução implementada, este experimento foi realizado com objetivo de analisar se as melhorias no escalonamento são mantidas quando o Hadoop é utilizado em um

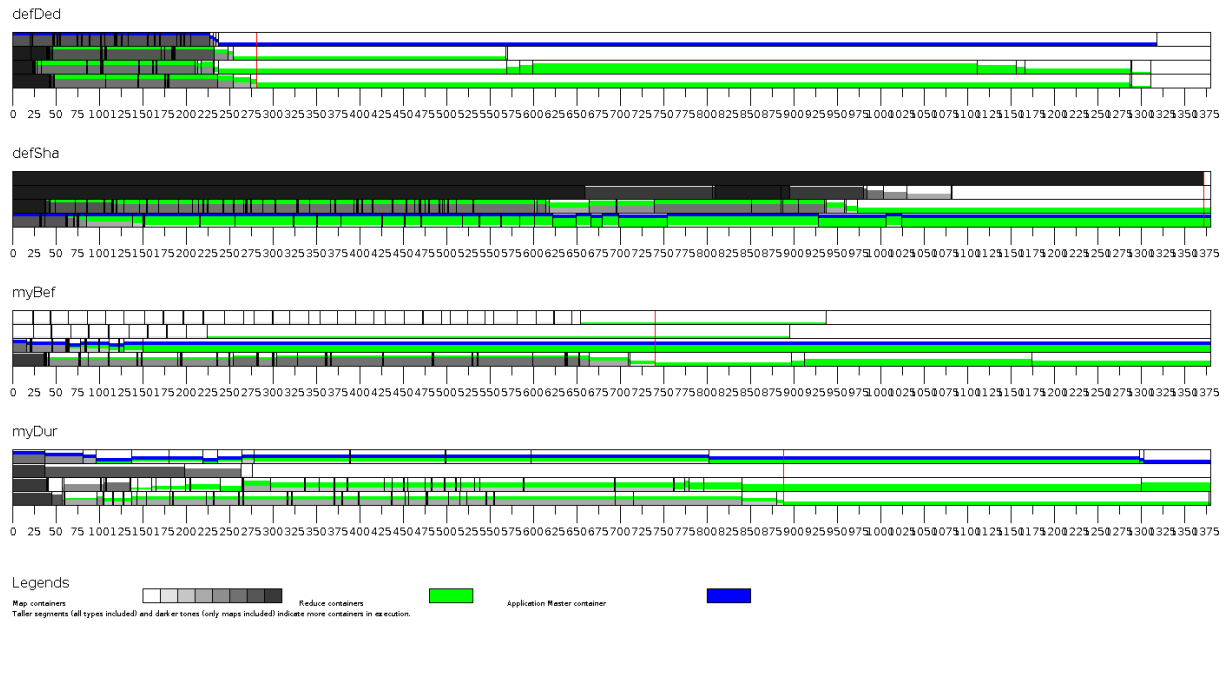


Figura 4.4 – Diagrama de Gantt para os experimentos reais com TeraSort

ambiente ou com uma entrada de dados maior. O experimento utiliza a solução descrita no Capítulo 3. A situação expressada com este experimento é de quando os nós do *cluster* começam a ser utilizados por outros usuários antes/durante a aplicação *MapReduce*.

4.4.1 Procedimentos

Neste experimento buscou-se analisar o comportamento da solução em um ambiente compartilhado de maior escala que nos experimentos anteriores. Foram feitos 2 experimentos distintos, um com escala de aplicação outro com escala de ambiente. Para a escala de aplicação a única diferença das configurações foi o tamanho da entrada, sendo utilizada uma entrada de 30 Gb para o TeraSort. Para a escala de ambiente, utilizou-se um *cluster* com X escravos, sendo que a proporção de recursos reduzidos/livres continuou a mesma.

4.4.2 Resultados e Interpretações

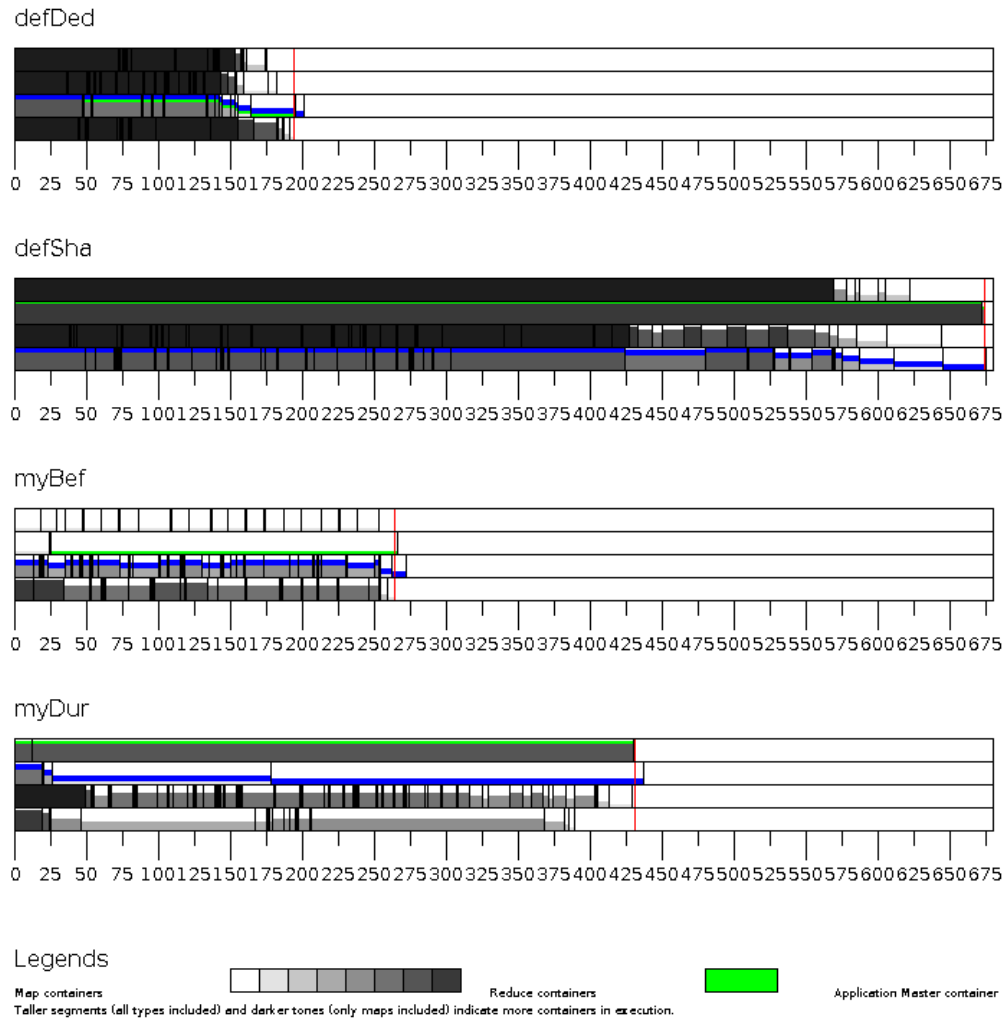


Figura 4.5 – Diagrama de Gantt para os experimentos reais com TestDFSIO

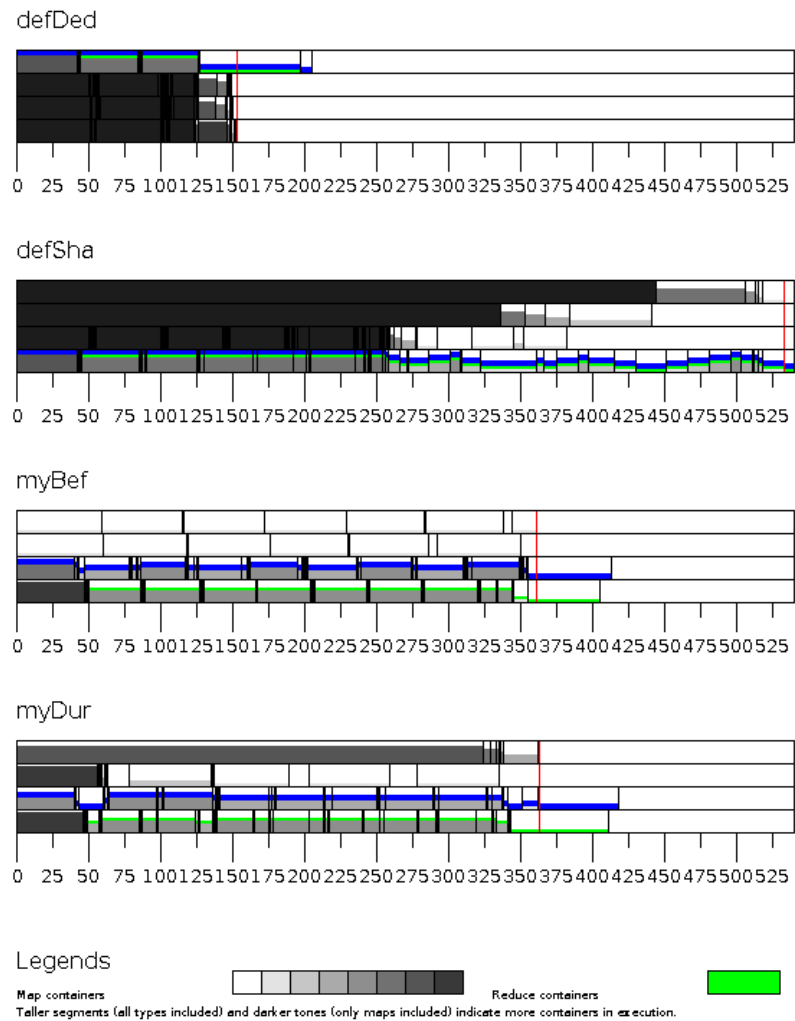


Figura 4.6 – Diagrama de Gantt para os experimentos reais com WordCount

5 CONCLUSION AND FUTURE WORK

This work had the objective of improving Hadoop scheduling, during the study process it was identified that the CapacityScheduler had already the base for a context-aware scheduling. However, this scheduler lacked some fundamental components in order to be context-aware, such as NodeManager real resource information and allocation limits scaling with the total cluster capacity.

Through development of this work changes have been made on original source code, these changes allowed Hadoop to be more aware of the context of nodes composing the cluster. The scheduling algorithm remained the same, however key limitations caused by Hadoop's default configurations were noticed. A new distribution containing a context-aware CapacityScheduler was generated in order to solve these issues.

The context-aware CapacityScheduler is capable of receiving the real capacity from each NodeManager, thanks to the collector plugged on NodeManager. This provides the cluster a better scaling potential while also using every node's full capacity. Using the context-aware CapacityScheduler, the allocations can be made to the full potential of the cluster instead of waiting for more resources when the cluster actually had almost 40GB of free memory per node.

Although the context-aware CapacityScheduler has better scaling potential and solves some problems on containers management, all contributions made are purely static and there are more ways to impact and improve Hadoop scheduling. Given Hadoop high modularity, it is possible to improve scheduling changing many areas that range from ApplicationMaster and Queues to NodeManager HeartBeat behavior.

Following there are some suggestions of future work:

- Extending the Resource class so it can track more resources like CPU load.
- Improving CapacityScheduler scheduling, taking into account other resources information.
- Modification of ApplicationMaster behavior.
- Implementation of a scheduler capable of starting containers directly on a NodeManager, and not dependant on queues.

REFERÊNCIAS

- Apache Hadoop. **Apache™ Hadoop®**. <http://hadoop.apache.org/>, Accessed August 2013.
- Apache Nutch. **Apache Nutch™**. <http://nutch.apache.org>, Acesso August 2013.
- BALDAUF, M.; DUSTDAR, S.; ROSENBERG, F. A survey on context-aware systems. **Int. J. Ad Hoc Ubiquitous Comput.**, Inderscience Publishers, Geneva, SWITZERLAND, v.2, n.4, p.263–277, June 2007.
- CHEN, Q. et al. SAMR: a self-adaptive mapreduce scheduling algorithm in heterogeneous environment. In: IEEE INTERNATIONAL CONFERENCE ON COMPUTER AND INFORMATION TECHNOLOGY, 2010., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2010. p.2736–2743. (CIT '10).
- DEAN, J.; GHEMAWAT, S. MapReduce: simplified data processing on large clusters. In: CONFERENCE ON SYMPOSIUM ON OPERATING SYSTEMS DESIGN & IMPLEMENTATION - VOLUME 6, 6., Berkeley, CA, USA. **Proceedings...** USENIX Association, 2004. p.10–10. (OSDI'04).
- DEY, A. K. Understanding and Using Context. **Personal Ubiquitous Comput.**, London, UK, UK, v.5, n.1, p.4–7, Jan. 2001.
- Fair Scheduler. **Hadoop MapReduce Next Generation - Fair Scheduler**. <http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html>, Accessed August 2013.
- HADOOP, A. **Arquitetura do HDFS**. <http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/Federation.html>, Accessed November 2013.
- HADOOP, A. **Arquitetura do YARN**. <http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html>, Accessed November 2013.

ISARD, M. et al. Quincy: fair scheduling for distributed computing clusters. In: ACM SIGOPS 22ND SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, New York, NY, USA. **Proceedings...** ACM, 2009. p.261–276. (SOSP '09).

Kirsch-Pinheiro, M. **CaptureDonnees**. <http://per-mare.googlecode.com/svn/trunk/permare-ctx/>, Accessed December 2013.

KUMAR, K. A. et al. CASH: context aware scheduler for hadoop. In: INTERNATIONAL CONFERENCE ON ADVANCES IN COMPUTING, COMMUNICATIONS AND INFORMATICS, New York, NY, USA. **Proceedings...** ACM, 2012. p.52–61. (ICACCI '12).

LI, C. et al. An Adaptive Auto-configuration Tool for Hadoop. In: INTERNATIONAL CONFERENCE ON ENGINEERING OF COMPLEX COMPUTER SYSTEMS, 2014., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2014. p.69–72. (ICECCS '14).

Oracle. **Interface OperatingSystemMXBean**. <http://docs.oracle.com/javase/7/docs/api/java/lang/management/OperatingSystemMXBean.html>, Accessed January 2014.

PHAM, C. M. et al. An Evaluation of Zookeeper for High Availability in System S. In: ACM/SPEC INTERNATIONAL CONFERENCE ON PERFORMANCE ENGINEERING, 5., New York, NY, USA. **Proceedings...** ACM, 2014. p.209–217. (ICPE '14).

RASOOLI, A.; DOWN, D. G. Coshh: a classification and optimization based scheduler for heterogeneous hadoop systems. In: SC COMPANION: HIGH PERFORMANCE COMPUTING, NETWORKING STORAGE AND ANALYSIS, 2012., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2012. p.1284–1291. (SCC '12).

TIAN, C. et al. A Dynamic MapReduce Scheduler for Heterogeneous Workloads. In: EIGHTH INTERNATIONAL CONFERENCE ON GRID AND COOPERATIVE COMPUTING, 2009., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2009. p.218–224. (GCC '09).

VAVILAPALLI, V. K. et al. Apache Hadoop YARN: yet another resource negotiator. In: ANNUAL SYMPOSIUM ON CLOUD COMPUTING, 4., New York, NY, USA. **Proceedings...** ACM, 2013. p.5:1–5:16. (SOCC '13).

XIE, J. et al. Improving MapReduce performance through data placement in heterogeneous Hadoop clusters. In: PARALLEL AND DISTRIBUTED PROCESSING, WORKSHOPS AND PHD FORUM (IPDPSW). **Anais...** IEEE International Symposium, 2010.

ZAHARIA, M. et al. Improving MapReduce performance in heterogeneous environments. In: USENIX CONFERENCE ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION, 8., Berkeley, CA, USA. **Proceedings...** USENIX Association, 2008. p.29–42. (OSDI'08).