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: Profª. Drª. Andrea Schwertner Charão

Santa Maria, RS, Brasil

2016

Cassales, Guilherme Weigert

Escalonamento Adaptativo para o Apache Hadoop / por Guilherme Weigert Cassales. – 2016.
53 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.

E-mail: cassales@inf.ufsm.br

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, Profª. Drª.
(Presidente/Orientadora)

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

Patrícia Pitthan de Araújo Barcelos, Profª. Drª. (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 distribuidas. O *framework Apache Hadoop* (Apache Hadoop, 2013), é uma delas e poupa o programador as terefas 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 melhora 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 melhora 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 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

SUMÁRIO

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 (DEAN; GHEMAWAT, 2008). 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 (SU; SWART, 2012).

Sem uma configuração específica pelo administrador, o Apache Hadoop assume que um *cluster* homôgeneo está sendo utilizado para a 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 (KUMAR et al., 2012a).

Esta é uma preocupação especial quando tenta-se utilizar o Hadoop em grids pervasivos, os quais são uma alternativa aos *clusters* dedicados, uma vez que o custo de aquisição e manutenção de um *cluster* dedicado continua alto e dissuasivo para muitas organizações. De acordo com (PARASHAR; PIERSON, 2010), grids pervasivos representam a generalização extrema do conceito de grid por possuírem recursos pervasivos, ou seja, recursos computacionais ociosos que são incorporados ao ambiente 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 seu poder de processamento. 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, dimnuindo o desempenho do escalonamento de tarefas (NASCIMENTO; BOERES; REBELLO, 2008).

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 (KUMAR et al., 2012a) (ZAHARIA et al., 2008a) (RASOOLI; DOWN, 2012a) (SANDHOLM; LAI, 2010) (STEFFENEL et al., 2013). 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 disso, 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 este objetivo 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 hajam mudanças em relação à última coleta realizada. A transmissão de dados ocorre por meio dos mecanismos oferecidos pelo ZooKeeper, quando ocorre alguma alteração o escalonador é alertado e realiza uma atualização nas informações pertinentes. A solução implementada apresenta melhorias de até 40% em alguns casos testados, 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

A definição de alguns termos, técnicas e/ou ferramentas se faz necessária para o entendimento deste estudo. Por exemplo, a conceituação formal de sensibilidade ao contexto é muito importante, uma vez que se constitui da técnica de obtenção dos dados. Como complemento à sensibilidade ao contexto define-se o que é o ZooKeeper, 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 dos trabalhos já realizados neste contexto.

2.1 Apache Hadoop

O *framework* Apache Hadoop origina-se de outro projeto da Apache, o Apache Nutch (Apache Nutch, 2013), que iniciou em 2002 como um motor de buscas de código livre porém, logo 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, denominado Apache Hadoop com 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 componentes abaixo da linha pontilhada, sendo o Name Node e o Data Node os serviços mestre e escravo

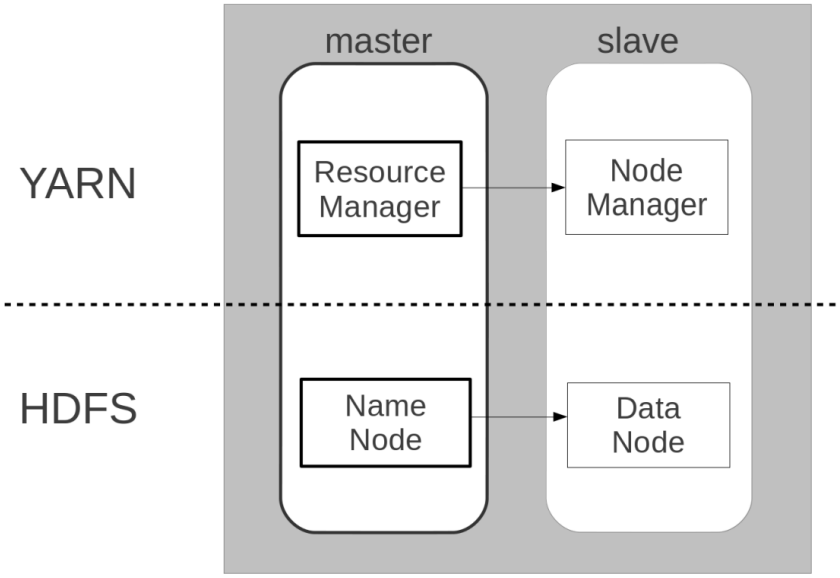

Figura 2.1 – Arquitetura geral do Apache Hadoop

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. Dessa 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 YARN é tornar a tarefa de processamento totalmente independente das tarefas de armazena-

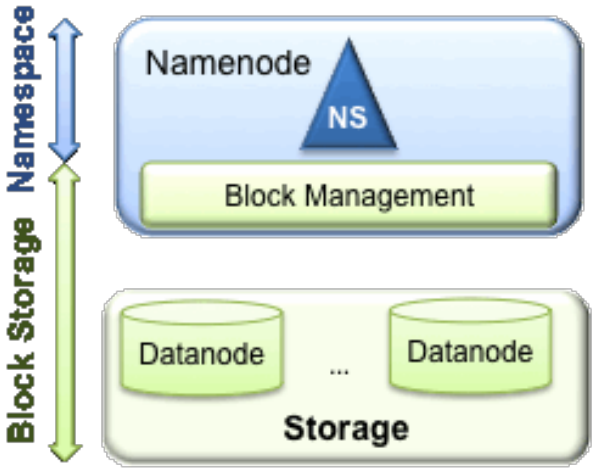

Figura 2.2 – Arquitetura geral do HDFS (HADOOP, 2013a)

mento, 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, onde é possível observar dois novos componentes do YARN.

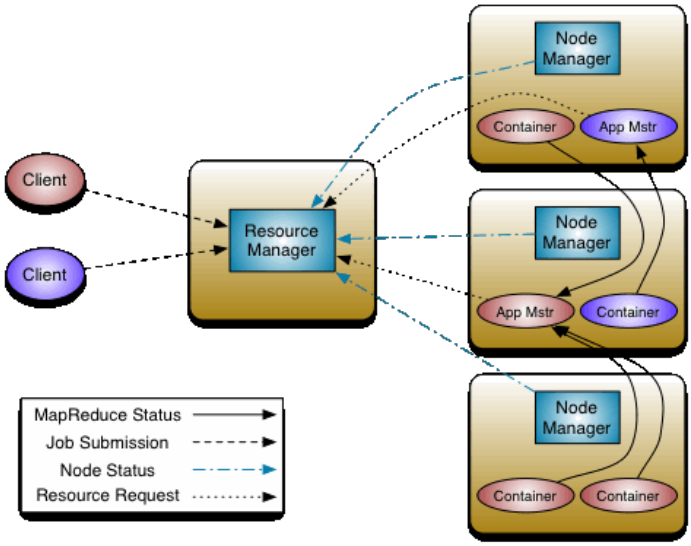

Figura 2.3 – Arquitetura geral do YARN (HADOOP, 2013b)

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 de suas instâncias.

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 configuração de recursos. 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 com-

partilhado. 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, sendo que, 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 fim, 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 igualitariamente 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 possa não ser consistente com a realidade, e monitorar quais deles estão livres e quais estão sendo utilizados pelo *framework* (HADOOP, 2013c).

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 três escalonadores já inclusos 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 estudo. Logo, buscou-se identificar quais técnicas foram mais utilizadas e quais os tipos de adaptação mais explorados nestes escalonadores.

Os autores do escalonador CASH (*Context Aware Scheduler for Hadoop* - Escalonador Sensível ao Contexto para o Hadoop) (KUMAR et al., 2012b) 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 levou 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 possuem 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 utilizaram a técnica de classificar as aplicações e máquinas de acordo com a quantidade/capacidade de E/S ou CPU. E assim como no CASH, o principal objetivo foi melhor o desempenho 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, 2012b) 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 somente a fatia mínima de recursos for utilizada e, ainda, proporcionar uma distribuição justa.

O escalonador LATE (*Longest Approximation Time to End* - Aproximação do Tempo de Término mais Longo) (ZAHARIA et al., 2008b), utiliza como informação de contexto o tempo estimado de término da tarefa com base em uma heurística que relaciona tempo decorrido e *score* – um valor que indica quanto do processamento já foi realizado. Essa informação também é utilizada para gerar um limiar que aponte quando a lentidão de uma tarefa indica sintomas de erros e, a partir desta informação iniciar uma tarefa especulativa em outra máquina possívelmente mais rápida. Este trabalho teve como objetivo 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-adaptative 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 o lançamento de uma tarefa especulativa é necessário ou não. Esta solução apresenta como diferencial o cálculo do progresso, o qual varia de acordo com informações do ambiente em que a tarefa está sendo executada. O principal objetivo do trabalho foi reduzir o 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 teve como objetivo a melhora do desempenho geral de um *cluster*, e utilizou a distribuição de recursos como informação de contexto para alcançá-lo. A contribuição do trabalho foi modificar a 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 buscou um rebalanceamento de carga foi (SANDHOLM; LAI, 2009), porém esta proposta utilizou-se de uma abordagem diferente. Neste trabalho o rebalanceamento de carga foi 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 desta proposta foi permitir um compartilhamento de recursos dinâmico baseado em preferências configuradas pelos próprios usuários.

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 estudo. 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 estudo busca melhorar a informação disponível para que o escalonamento seja adaptável ao compartilhamento do ambiente.

Nota-se que, no geral, as propostas de melhora da adaptabilidade apresentadas pelos trabalhos utilizaram, 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.

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 faz-se uma análise em profundidade da solução implementada comparando-a com o comportamento *default* do Hadoop em ambientes compartilhados.

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 (MAAMAR; BENSLIMANE; NARENDRA, 2006) 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, consequentemente, 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 (HUNT et al., 2010). 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 *ensamble*), enquanto a aplicação que o está utilizando assume o papel

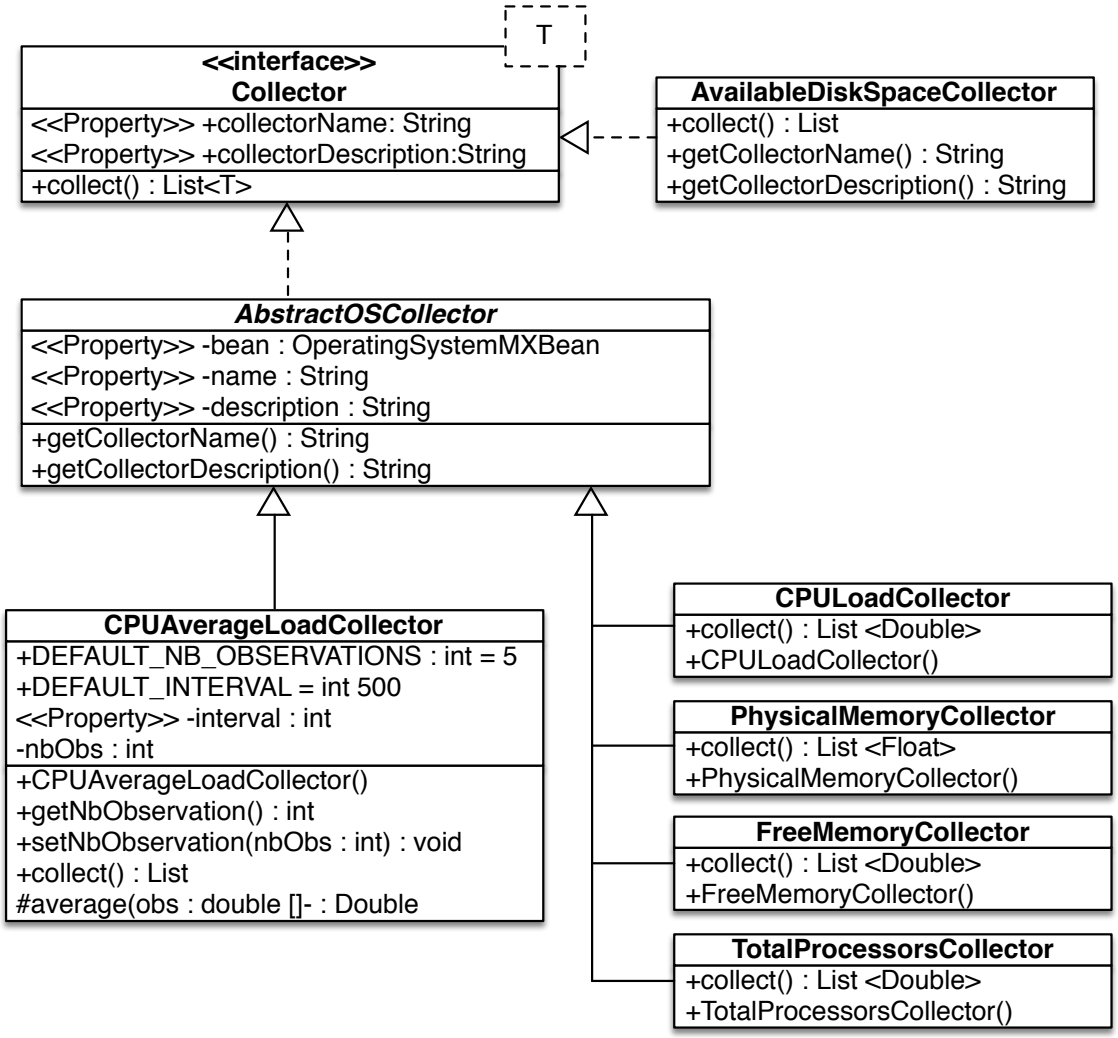

Figura 3.1 – Diagrama de classes dos coletores de contexto

de cliente. Os dados do ZooKeeper ficam armazenados em *zNodes*, abstrações que podem ser 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 *ensamble* (PHAM et al., 2014).

Um dos recursos do ZooKeeper que oferece grande utilidade é o *Watcher*, uma interface que permite aos clientes o monitoramento de 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. O disparo desta

callback é um evento único, forçando o programador a re-inserir um *Watcher* no *zNode* para continuar monitorando-o (**?**).

No contexto deste trabalho, os serviços do ZooKeeper são utilizados para monitorar as informações 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, porém existem alguns detalhes importantes no gerenciamento de *Watchers* que podem impactar na eficácia da solução. Por esta razão optou-se por utilizar um *zNode* para cada Node Manager, o que permite realizar um controle mais rígido das atualizações e possibilita uma maneira fácil e rápida tanto para a inserção das informações por parte dos Node Managers quanto para o monitoramento dos dados pelo escalonador.

Na solução adotada cada *zNode* contém informação de apenas um Node Manager e para cada *zNode* existe um Watcher, sendo que cada *Watcher* é uma thread pertencente ao escalonador. Qualquer alteração em um *zNode* dispara uma *callback* para seu *Watcher* específico, o qual lê os novos dados do *zNode* e atualiza a informação no escalonador sobre os recursos daquele nó. A utilização do *Watcher* permite que o escalonador realize operações apenas quando necessário e não desperdice tempo percorrendo os *zNodes* quando não houverem atualizações a serem feitas. A estrutura adotada no trabalho podes ser visualizada na Figura 3.2.

A solução escolhida gera dois papéis para os nós, o papel de monitoramento e o papel de atualização, os quais são explicados em detalhe a seguir.

O papel de **Monitoramento** é desempenhado pelo escalonador, o qual possui uma *thread Watcher* que monitora um *zNode*. Este *zNode* inicial servirá como *container* para os demais *zNodes*, os quais armazenarão as informações de contexto referentes aos nós escravos do *cluster*. O *Watcher* do *zNode* inicial permite ao escalonador ser notificado quando novos *zNodes* forem inseridos na estrutura. Quando a *callback* do *Watcher* é acionada, a *thread* recebe todos os *zNodes* filhos daquele que está sendo monitorado e após identificar o novo *zNode*, inicia uma nova *thread Watcher* para monitorar o novo *zNode* identificado. Desta forma, cada *zNode* será monitorado por uma *thread Watcher* única. Um possível problema da técnica utilizada é discutido, juntamente com a solução adotada para solucioná-lo, na descrição do papel de

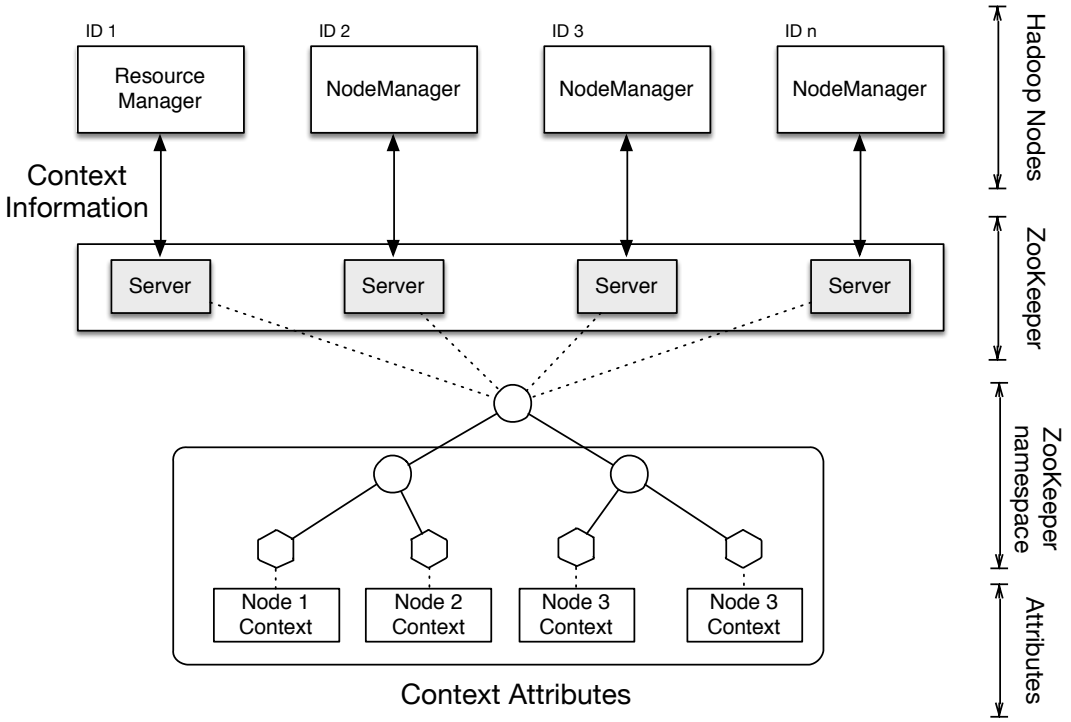

Figura 3.2 – Estrutura do ZooKeeper

Atualização.

Uma alternativa à utilização de um *zNode* para cada nó escravo, seria a utilização de uma estrutura que permitisse a inserção de todos os dados dos nós escravos, como uma Tabela Hash. Embora, em um primeiro momento, o maior impecilho para a utilização desta técnica possa parecer o limite padrão do tamanho da informação que um *zNode* comporta (1 Mb), o problema é, na verdade, relacionado com a utilização de uma única *thread Watcher*.

Como todos os nós escravos são, geralmente, inicializados ao mesmo tempo, as coletas de dado são realizadas em períodos semelhantes, estes períodos serão denominados neste estudo como fase de coleta. Durante estas fases, é possível que muitas atualizações sejam feitas em um pequeno espaço de tempo, gerando um risco à consistência das informações, uma vez que o ZooKeeper não fornece garantias de que todas as alterações notificarão a *thread Watcher*. Este comportamento ocorre devido à possibilidade de notificações serem disparadas antes da re-inserção do *Watcher* no *zNode* que contém a Tabela Hash. Esta falha só ocorre quando a segunda notificação é enviada antes que a primeira termine de ser processada, e existem duas situações que isto pode ocorrer. A primeira situação não apresenta grandes riscos à consistência das informações, pois a notificação perdida ocorre entre duas notificações normais na mesma fase de coleta. Sendo assim, a notificação seguinte fará com que o escalonador ajuste os valores de recursos dos dois nós. Já a segunda situação apresenta riscos à consistência das informações,

pois ocorre quando não há uma notificação normal na mesma fase de coleta após a notificação perdida, ou seja, a notificação perdida só terá suas informações atualizadas no escalonador na próxima fase de coleta, a qual ocorrerá em aproximadamente 30 segundos. Este período que o escalonador opera com informações inconsistentes pode causar o lançamento de *containers* que irão exceder o limite dos nós caso um novo usuário tenha iniciado a utilização do nó, gerando sobrecarga e lentidão tanto na aplicação do novo usuário como na aplicação MapReduce.

O papel de **Atualização** é realizado pelos Node Managers, os quais lançam, no momento de sua inicialização, uma *thread* responsável por fazer a coleta de dados e, caso houver alteração com relação à ultima coleta, atualizar os dados no *zNode*. A coleta dos dados é realizada a cada 30 segundos, intervalo que corresponde à média de tempo observada em *containers* executados em um *cluster* de funcionamento normal. Com objetivo de evitar o envio de dados por alterações naturais do sistema e que não impactariam no escalonamento foi implementada uma política de atualização, na qual atualizações só serão realizadas quando as variações alterarem o limite de *containers* que podem ser inicializados no nó, seja pela carga de CPU ou disponibilidade de memória. Esta política permite ao escalonador otimizar seu poder de adaptabilidade sem desperdiçar tempo com leituras desnecessárias ou transmissão de dados não diretamente relacionados com as aplicações.

Ao evitar a atualização por variações que não impactem na capacidade de *containers*, a quantidade de dados transmitidos é reduzida e, consequentemente, o número de execuções das *threads Watcher* que monitoram os *zNodes* também o são. Caso esta política não fosse utilizada, correria-se o risco de todos os nós escravos apresentarem variações em todas as leituras. Sabe-se que o sistema operacional possui alterações naturais dos recursos e que estas, geralmente, não representam uma grande quantidade de recursos. Além disso, segundo a política de alocação de *containers* do Hadoop e utilizando os valores *default*, tanto um nó com 1024 Mb livres quanto um nó com 2047 Mb livres serão capazes de inicializar somente 1 *container* (**?**). Por este motivo, a atualização de variações que não alterem a quantidade máxima de *containers* em um nó somente desperdiçariam tráfego na rede e processamento nos nós. Em um cluster de 100 nós, por exemplo, este comportamento poderia causar a execução de 100 *threads* no nó mestre a cada 30 segundos, sem levar em consideração a transmissão de dados pela rede. Contudo, uma vez que a variação representa uma alteração na capacidade de *containers* do nó, a atualização deve ser feita. Caso a variação seja positiva (mais *containers* podem ser lançados), a não atualização impactaria no desperdício de capacidade que ficaria ociosa. No

caso da variação ser negativa (menos *containers* podem ser lançados), a não atualização faria o escalonador ignorar a sobrecarga e continuar lançando novos *containers* mesmo que estes estivessem acima da capacidade do nó.

3.3 Solução Implementada

Estudos prévios indicam que o Capacity Scheduler já apresenta uma boa base para introdução de sensibilidade ao contexto e adaptatividade no Hadoop (**?**). Uma vez que o objetivo deste estudo foi melhorar a adaptabilidade à ambientes com presença de compartilhamento, um dos principais mecanismos necessários foi o de adaptação à fatores que sofrem alterações no decorrer do processamento de tarefas. Assim, foi implementada uma nova funcionalidade no escalonador que permite a ele adaptar-se às variações de recursos disponíveis nos nós durante a execução de aplicações MapReduce. A solução implementada utiliza tanto os coletores de contexto quanto a comunicação distribuída por meio do ZooKeeper.

Como já citado anteriormente, o custo de aquisição e manutenção de um *cluster* dedicado para a execução de aplicações MapReduce é alto e é possível que pequenas empresas, que não tenham recursos financeiros suficientes para a aquisição de um *cluster*, prestem serviços que geram uma grande quantidade de dados. Embora estes casos possam ser resolvidos com a utilização de *IaaS* (Infrastructure as a Service – Infra-estrutura como Serviço) como o Amazon AWS (**?**) e Amazon EC2(**?**) a custos baixos, os proprietários podem preferir não utilizar serviços em nuvem devido à outros fatores como a segurança de dados pessoais ou adaptação à nova tecnologia. Nestes casos a utilização da capacidade ociosa dos computadores na rede da empresa pode ser uma alternativa viável, uma vez que não gera custos de aquisição e a manutenção é mais simples.

O problema da utilização do Hadoop nestas condições é que, embora em menor escala, esta alternativa reproduz muitas das características presentes em *grids* pervasivos, as quais podem impactar negativamente o desempenho do Hadoop. Dentro de um contexto de disponibilidade de recursos, é possível identificar 5 situações que podem ocorrer: a saída de um nó do *cluster*, a entrada de um nó no *cluster*, um nó que estava sendo utilizado pelo Hadoop passa a ser utilizado em conjunto para outra aplicação (início do compartilhamento), um nó que estava sendo compartilhado volta a ser totalmente disponível para o Hadoop (fim do compartilhamento) e os computadores podem ter configurações diferentes entre si (nós heterogêneos).

A situação mais fácil de ser resolvida é a de nós heterogêneos, pois, mesmo sem alte-

rações no comportamento do Hadoop, bastaria alterar os arquivos XML de configuração dos nós. No caso de não alteração dos arquivos XML, alguns nós poderiam ser sobrecarregados enquanto outros poderiam ser sub-utilizados. Contudo, com a solução proposta neste estudo, a disponibilidade de recursos é coletada e enviada ao escalonador. Com isso a informação estará sempre consistente com a realidade, e haverá um ganho de desempenho através da diminuição da sobrecarga e da sub-utilização.

A saída de um nó do *cluster* já é suportada de maneira eficiente pelos procedimentos de tolerância à falhas do Hadoop. As tarefas que estavam sendo processadas pelo nó serão perdidas e terão que ser reiniciadas em outro nó. Além disso o *pool* de recursos totais é reduzido de acordo com o tamanho do Node Manager.

A entrada (registro) de um novo nó necessita de mais etapas, mas é possível de ser realizada mesmo com a distribuição *default* do Hadoop. É necessário que o *host* do novo nó seja incluído no arquivo de configuração *slaves* do mestre antes de ser inicializado o Node Manager no nó escravo. Uma vez que estas duas etapas estejam completas, o escalonador irá aumentar o *pool* de recursos totais de acordo com a capacidade do novo nó ou 8 Gb de memória e 8 *cores* na distribuição *default*.

As duas situações restantes são mais interessantes, pois não há suporte para elas na distribuição *default* do Hadoop. Esta capacidade de adaptação não seria necessária, uma vez que, desde seu projeto, todos os computadores que compõem o *cluster* são considerados dedicados para uso do Hadoop. Ainda assim, haveria uma alternativa para a utilização compartilhada dos recursos com a distribuição *default* do Hadoop. Esta alternativa seria mediante a configuração dos arquivos XML, ao configurar cada computador do *cluster* com menos recursos do que os totais que ele possui. Esta alternativa, embora simples, não é eficiente, pois implica que ou a parcela não utilizada pelo Hadoop (a diferença entre o valor no arquivo XML e o valor real dos recursos) é constante, ou que haverão momentos tanto de sobrecarga como de sub-utilização do *cluster*. Como já mencionado na Seção 3.2, outra alternativa, também ineficiente, seria inicializar o *cluster* com a capacidade ociosa de cada computador naquele momento configurada nos arquivos XML, executar uma aplicação MapReduce e em seguida terminar a execução dos serviços do Hadoop. Esta alternativa também está sujeita a alterações na disponibilidade de recursos e possível sobrecarga e/ou sub-utilização.

No caso de início de compartilhamento, o escalonador continua inicializando novos *containers* mesmo que a capacidade do nó tenha sido excedida, causando sobrecarga e lentidão

para o término das tarefas e, consequentemente, da aplicação. Com a contribuição deste estudo os recursos disponíveis para utilização pelo Hadoop no nó em questão serão reduzidos, porém nenhum *container* já inicializado será preemptado. Ainda assim, a contribuição permite ao escalonador saber que determinado nó está sobrecarregado e, com base nesta informação, não alocar novos *containers* neste nó até que a situação dos recursos esteja normalizada, ou seja, existirem recursos disponíveis capazes de alocar ao menos 1 *container*.

No caso de término do compartilhamento a distribuição *default*, quando configurada para um *cluster* dedicado, terá o mesmo comportamento da solução implementada por este estudo. Como o escalonador nunca alterou a informação do nó, ao terminar o compartilhamento este nó voltará à situação inicial, a qual todo recurso do nó está disponível para o Hadoop. Nesta situação a abordagem *default* do Hadoop, quando configurada para um *cluster* dedicado é eficiente. No caso da solução implementada neste estudo, o nó fará a coleta, atualizará a informação no *zNode* e o escalonador atualizará a capacidade do nó quando for notificado da alteração, porém haverá um atraso de no máximo 30 segundos para que as informações fiquem consistentes com relação à distribuição *default*. No caso de utilização da distribuição *default* configurada para utilizar somente uma parte de alguns nós estes recursos ficarão ociosos, piorando o desempenho por não utilizar todos os recursos disponíveis.

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 Adaptativo com atualização antes (AdaptBef): 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 (AdaptDur): 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%.

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%.

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 *Benchmarks* 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 (HAMILTON, 2008) 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* (HUANG et al., 2010), 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 e 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, enquanto 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 que desejou-se expressar com o experimento é aquela que ocorre 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	**DefDed**	**DefSha**	**MyBef**	**MyDur**
Tempo Total de Map (*s*)	149	788	348	477
Tempo Médio de Map (*s*)	39.47	222.97	38.38	68.42
# Tarefas Map	76	76	76	76
# Tarefas Especulativas	2	1	3	1

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

Caso	**DefDed**	**DefSha**	**MyBef**	**MyDur**
Tempo Total de Map (*s*)	139	444	239	364
Tempo Médio de Map (*s*)	38.95	85.01	32.20	81.62
# Tarefas Map	90	90	90	90
# Tarefas Especulativas	0	9	0	1

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

Caso	**DefDed**	**DefSha**	**MyBef**	**MyDur**
Tempo Total de Map (*s*)	155	1009	309	805
Tempo Médio de Map (*s*)	43.76	208.39	41.73	175.80
# Tarefas Map	90	90	90	90
# Tarefas Especulativas	1	15	1	10

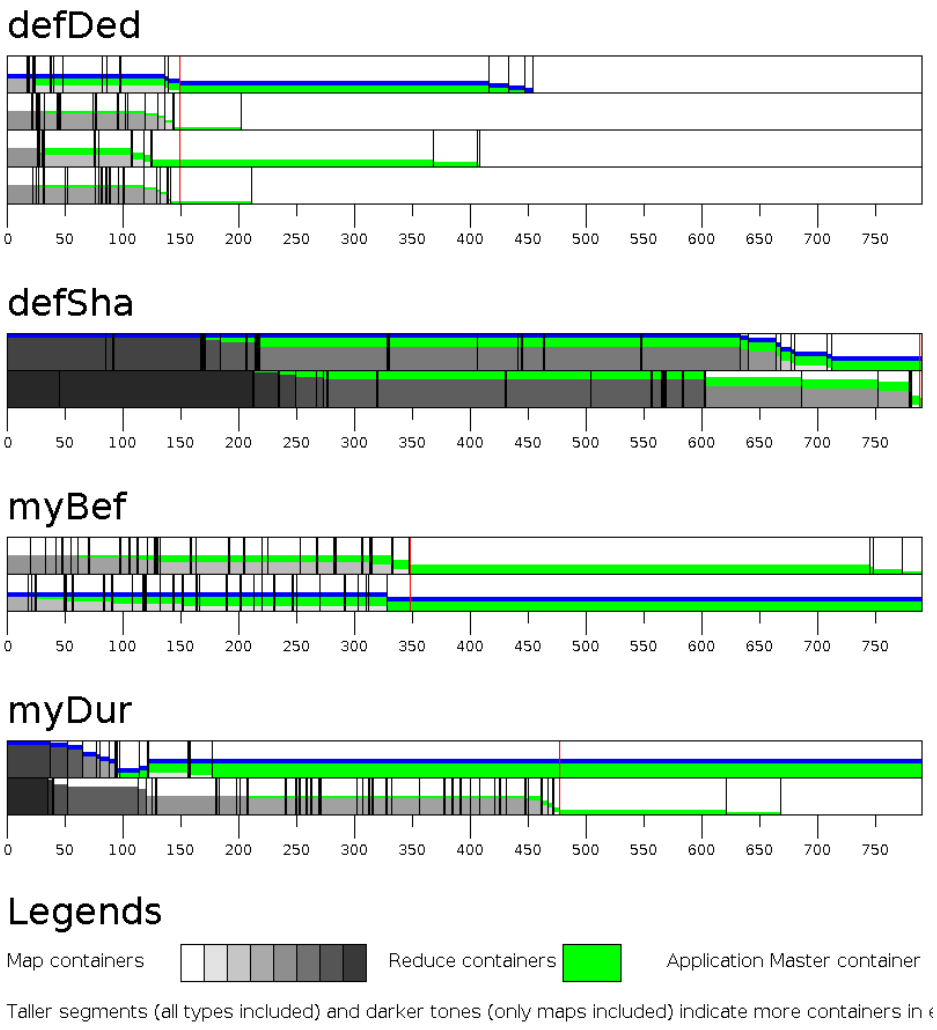

Figura 4.1 – Diagrama de Gantt para os experimentos controlados com TeraSort

Analisando as tabelas, é possível identificar alguns padrões. Todos experimentos apresentam um comportamento similar com relação ao tempo total dos *containers Map*: DefDed foi o mais rápido, seguido pelos casos MyBef, MyDur e finalmente DefSha. Ainda, os casos DefDed e MyBef possuem os menores tempos médios de *Map* e estes são muito semelhantes não importando a aplicação analizada. Este comportamento é devido à não sobrecarga dos nós, uma vez que o escalonador possuia os dados corretos com relação aos recursos dos nós durante

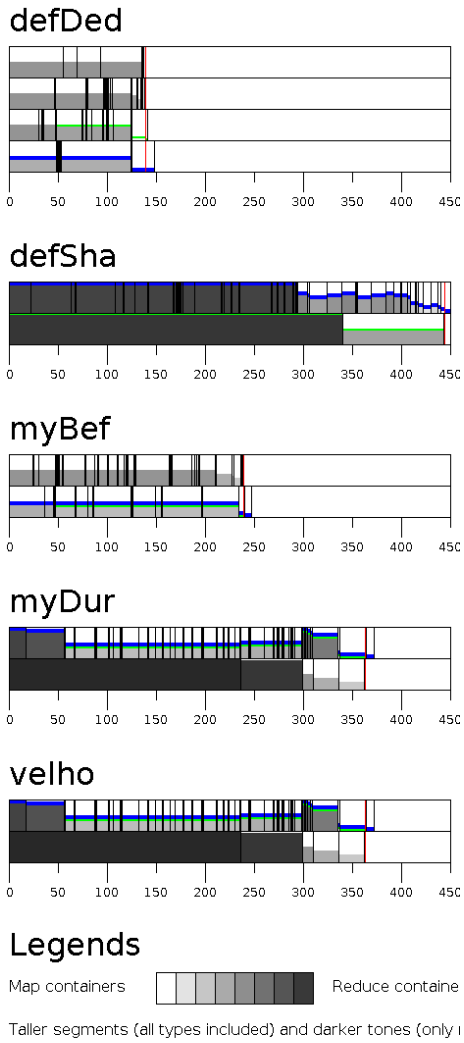

Figura 4.2 – Diagrama de Gantt para os experimentos controlados com TestDFSIO

todo experimento. Os diagramas de Gantt também apresentam esta informação uma vez que os experimentos DefDed e MyBef possuem tons mais claros e alturas menores do que os demais no início da aplicação, o que significa que há menos *containers* em execução. É possível notar também que o caso MyBef leva, em geral, o dobro do tempo do caso DefDed para terminar a etapa de Map. Este resultado é esperado, uma vez que neste experimento o caso DefDed possui o dobro de recursos disponíveis se comparado com o caso MyBef.

Outra observação interessante tem relação com a análise das tarefas especulativas iniciadas em cada aplicação. A aplicação TeraSort possui um número reduzido de tarefas iniciadas e este número é similar em todos os casos. Já o TestDFSIO e o WordCount apresentam um alto número de tarefas especulativas nos casos em que o sistema está sobrecarregado em algum

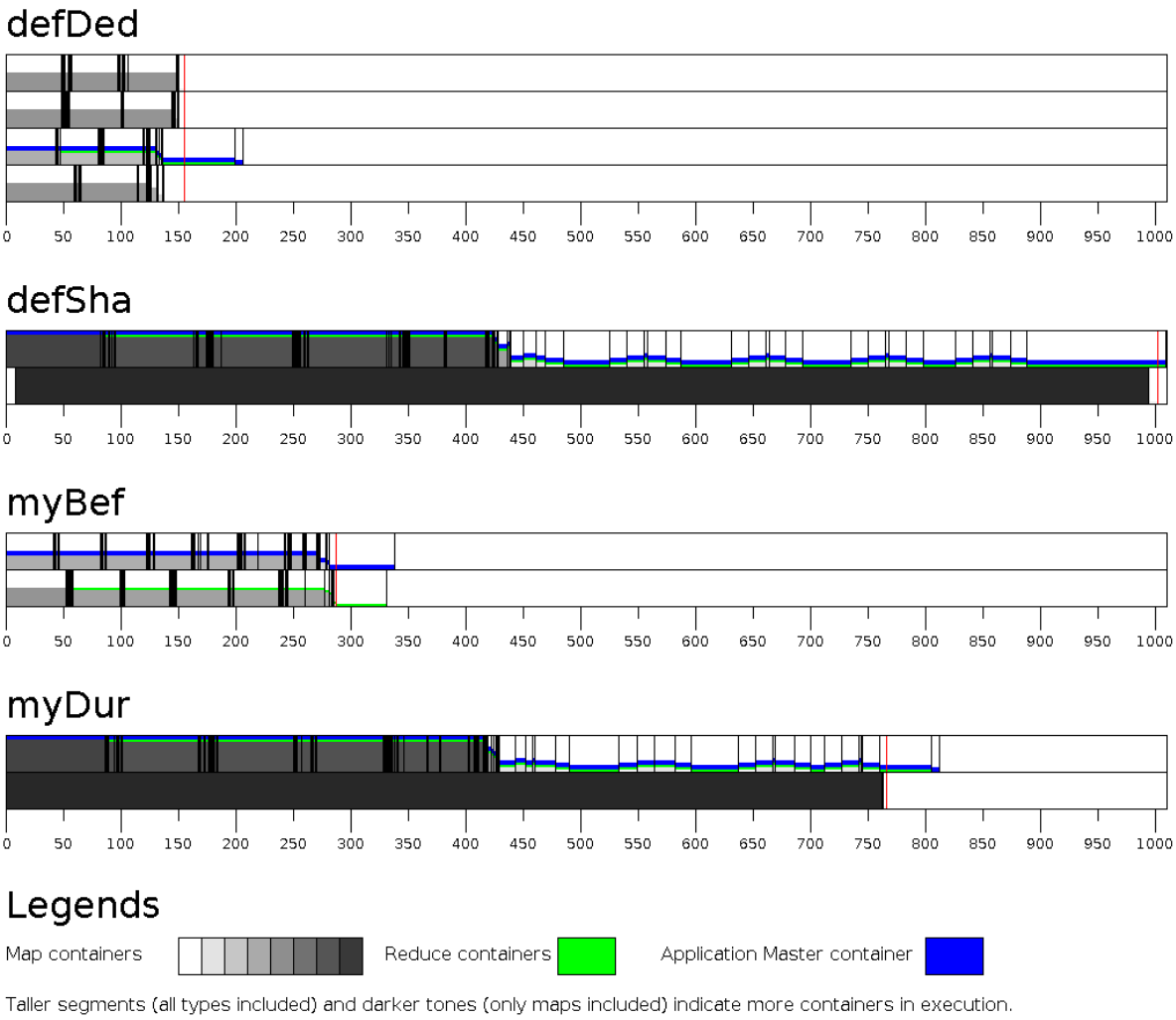

Figura 4.3 – Diagrama de Gantt para os experimentos controlados com WordCount

momento (DefSha e MyDur). Este comportamento pode ser explicado sob a análise dos fatores que determinam o lançamento ou não de uma tarefa especulativa. Sabe-se que uma tarefa especulativa é iniciada com base numa avaliação de progresso da tarefa, onde uma tarefa atrasada em relação às demais está possivelmente apresentando falhas. No caso do TeraSort, onde as tarefas demandam de maneira uniforme os recursos de memória, CPU e E/S, o tempo de execução está menos sujeito à redução de um recurso específico. Enquanto o TestDFSIO e o WordCount, demandam recursos mais específicos e, portanto, são mais sensíveis à sobrecarga destes recursos. Em todas as aplicações, a utilização de sensibilidade ao contexto como meio de melhorar a adaptabilidade no caso MyDur ajuda a reduzir o número de tarefas especulativas quando comparado com o caso DefSha que não apresenta nenhuma capacidade de adaptação à

estes ambientes.

Com relação ao fluxo de execução, como ilustrado nos diagramas, é possível notar que os casos DefSha e MyDur possuem tons mais escuros no início, o que significa que cada nó está com 16 *containers* em execução (o dobro da capacidade real do nó). Ainda, os primeiros *containers* a terminar nos casos DefDed e MyBef levam entre 20 e 50 segundos (conforme linha vertical de segmento), enquanto nos casos DefSha (e em menor grau MyDur) os primeiros *containers* levam 70 segundos ou mais para terminar, evidenciando a sobrecarga dos nós. As excessões à esta afirmação são os casos DefSha e MyDur do TestDFSIO, nos quais os primeiros *containers* a terem o processamento concluído levam aproximadamente 25 segundos para tal. Através de uma análise mais profunda, é possível notar que todos os casos do experimento TestDFSIO possuem ao menos um seguimento próximo à marca dos 25 segundos, o que significa que uma das tarefas deste conjunto de entrada foi processada rapidamente em todos os casos, e não foi afetada pela sobrecarga dos nós nos casos DefSha e MyDur.

Embora os casos DefSha e MyDur possuam as mesmas condições iniciais (recursos disponíveis de 50% e recursos informados de 100%), o caso MyDur necessita de menos tempo para concluir o processamento da aplicação em todas as aplicações. O caso MyDur possui uma melhora de 20% a 40% no tempo de execução quando comparado com o caso DefSha. A razão para este comportamento é a capacidade de adaptação inserida ao Hadoop, a qual permite que os Node Managers coletem a informação correta e à transmitam para o escalonador, que irá reorganizar a distribuição de tarefas à medida que as tarefas em execução são concluídas. Este comportamento é mais fácil de ser observado nos diagramas das aplicações TeraSort e TestDFSIO. É possível notar também que todos o casos MyDur possuem alta concentração de containers no início da aplicação mas essa concentração diminui ao longo da execução, enquanto os casos DefSha mantém a alta concentração até os últimos momentos da execução devido à ausencia de adaptabilidade. Embora o escalonador não preempte *containers* em excesso, é possível observar uma melhora de aproximadamente 40% no TeraSort e 20% no TestDFSIO e WordCount apenas pela inibição do lançamento de novos *containers* enquanto o nó está sobrecarregado.

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.

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 alguns escravos terão, em algum momento, seus recursos disponíveis reduzidos devido ao compartilhamento, ou seja, outra aplicação irá utilizar 6 cores e 6 Gb de memória em 2 dos 4 nós escravos. Para alcançar esta situação foi implementada uma aplicação em C, na qual a quantidade desejada de *threads* é inicializada e cada uma delas utiliza 1 core e 1 Gb de memória. O experimento foi realizado com as 3 aplicações para observações de possíveis alterações no comportamento com relação ao experimento inicial.

4.3.2 Resultados e Interpretações

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

Tabela 4.4 – Resumo dos resultados do TeraSort real em segundos.

Caso	**DefDed**	**DefSha**	**MyBef**	**MyDur**
Tempo Total de Map (*s*)	281	1372	740	888
Tempo Médio de Map (*s*)	38.50	198.86	39.58	62.82
# Tarefas Map	112	112	112	112
# Tarefas Especulativas	1	14	0	2

Tabela 4.5 – Resumo dos resultados do TestDFSIO real em segundos.

Caso	**DefDed**	**DefSha**	**MyBef**	**MyDur**
Tempo Total de Map (*s*)	194	674	264	431
Tempo Médio de Map (*s*)	31.24	162.47	38.56	65.24
# Tarefas Map	90	90	90	90
# Tarefas Especulativas	2	16	0	8

Através da análise das tabelas e dos diagramas, nota-se que muitos padrões foram mantidos apesar da alteração de procedimento nos experimentos. Primeiramente, é importante notar que nas aplicações anteriores a sobrecarga dos nós era de 100%, uma vez que o Hadoop estava tentando utilizar o dobro da capacidade. Na execução das aplicações deste experimento

Tabela 4.6 – Resumo dos resultados do WordCount real em segundos.

Caso	DefDed	DefSha	MyBef	MyDur
Tempo Total de Map (*s*)	153	533	361	396
Tempo Médio de Map (*s*)	34.85	105.27	37.79	57.76
# Tarefas Map	90	90	90	90
# Tarefas Especulativas	1	14	1	2

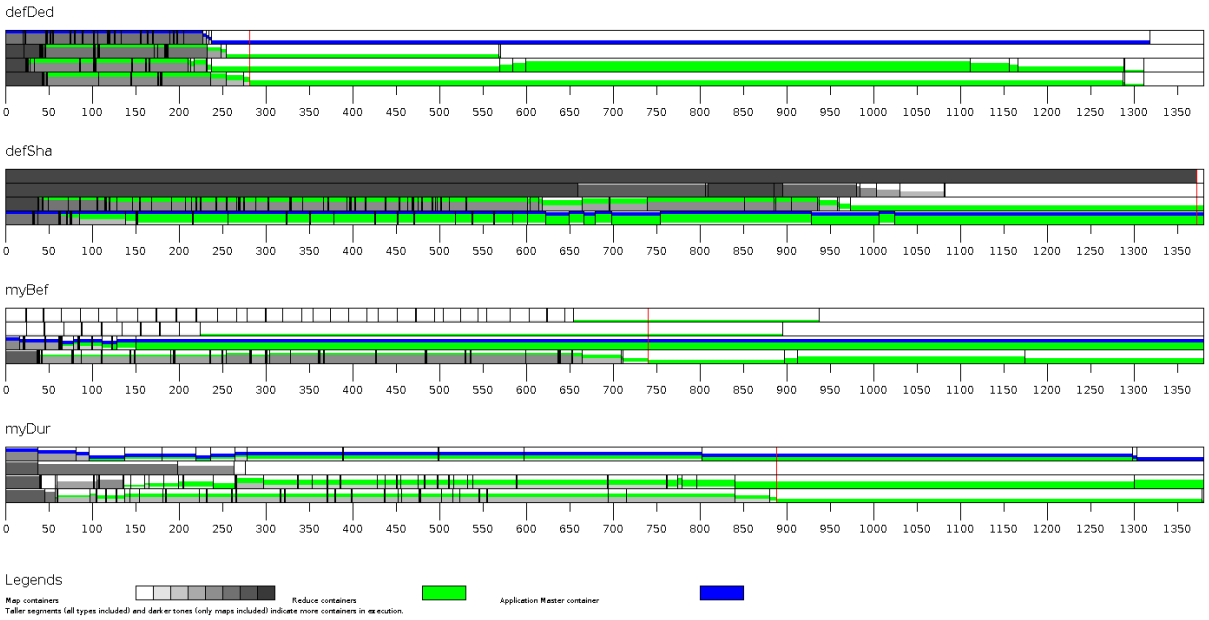

Figura 4.4 – Diagrama de Gantt para os experimentos reais com TeraSort

a sobrecarga é levemente menor, pois a informação passada ao Hadoop nos casos default não é alterada. A sobrecarga nestes experimentos é de aproximadamente 6 Gb de RAM e 6 cores, fazendo com que o nó, nos piores casos, tenha uma demanda de aproximadamente 14 Gb e 14 cores (sem considerar recursos requisitados pelo sistema operacional e serviços do Hadoop) alocada aos seus 8 Gb e 8 cores.

Novamente, todas aplicações apresentam um comportamento similar com relação ao tempo total dos *containers Map* e tempo médio destes: DefDed apresentou o menor tempo total e médio dos *containers Map*, seguido pelos casos MyBef, MyDur e finalmente DefSha. Também seguindo o padrão do experimento anterior, os tempos médio de Map nos casos DefDed e MyBef são muito semelhantes não importando a aplicação analizada. Embora alguns padrões foram mantidos, a proporção de tempo entre os casos DefDed e MyBef não apresenta uma relação tão constante quanto no experimento anterior e algumas particularidades podem ser observadas nos diagramas. Ainda assim, estes dados iniciais sugerem que uma sobrecarga

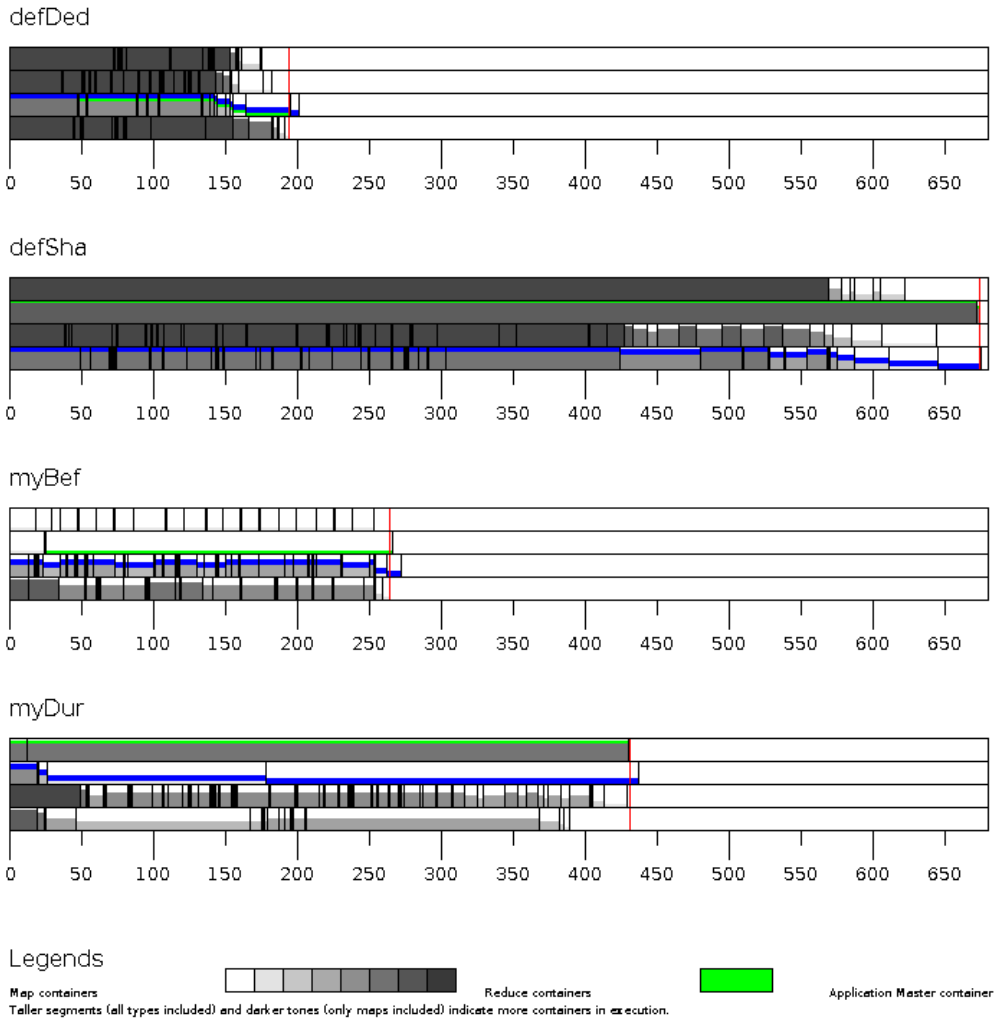

Figura 4.5 – Diagrama de Gantt para os experimentos reais com TestDFSIO

de alguns nós, mesmo em menor intensidade se comparado ao experimento anterior, deteriora significativamente o desempenho da aplicação.

Com relação às tarefas especulativas, nota-se que neste experimento o caso DefSha apresentou um número alto de tarefas especulativas nas 3 aplicações, enquanto os demais casos mantiveram valores baixos salvo o caso MyDur na aplicação TestDFSIO. Os resultados deste experimento sugerem que embora a sobrecarga, com relação à demanda total, tenha sido menor, os efeitos provocados por ela foram mais visíveis, pois os casos onde a sobrecarga esteve presente durante toda aplicação apresentam um número elevado de tarefas especulativas.

Antes de realizar a análise dos diagramas deste experimento é necessário salientar uma característica importante, todos os diagramas a gerados para este experimento possuem no má-

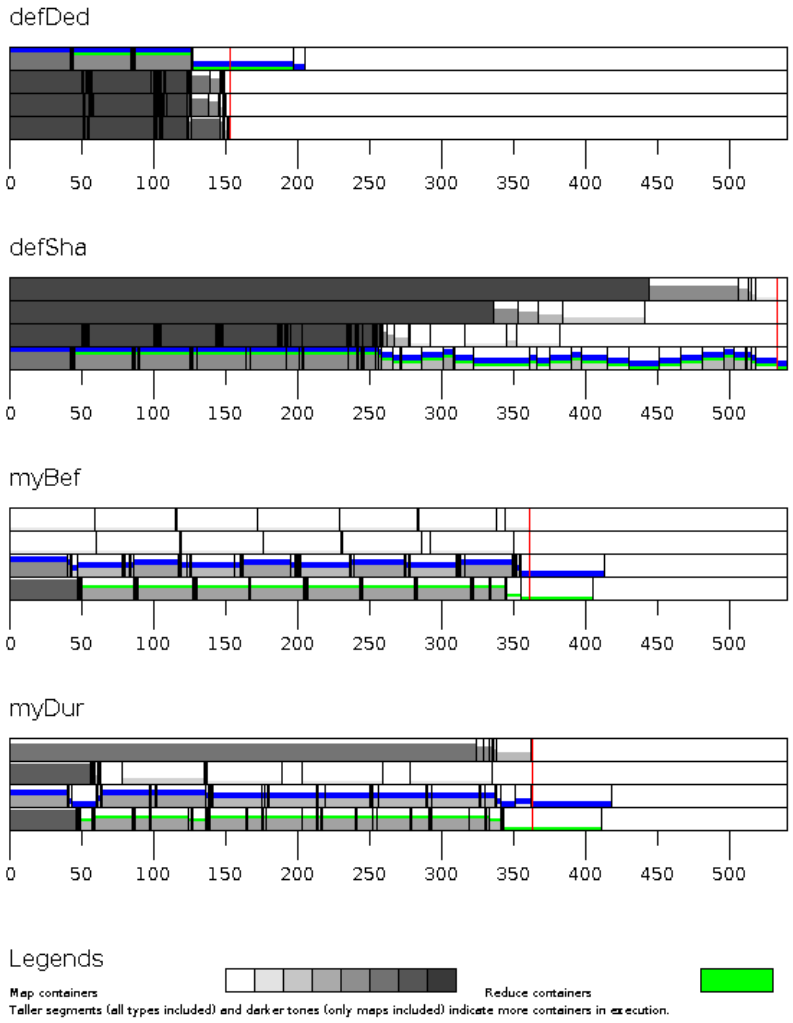

Figura 4.6 – Diagrama de Gantt para os experimentos reais com WordCount

ximo 8 *containers* em execução, enquanto no experimento anterior este valor era de 16. Além disso, os diagramas possuem características que permitem uma visualização das tarefas que originam tarefas especulativas e da reação à mudança de recursos nos nós.

Nas 3 aplicações o caso MyBef pode, em um primeiro momento, aparentar estar com 2 nós sem utilização, porém este comportamento é exatamente o esperado dos nós. O caso de teste é criado de forma que quando outro usuário utiliza o *cluster*, ele demanda 6 Gb e 6 cores de 50% dos nós, deixando apenas 2 Gb e 2 cores disponíveis para o sistema operacional e o Hadoop. Retirando os recursos do sistema operacional e dos serviços do Hadoop restam pouco mais de 1 Gb e 1 core livres, com os quais o Hadoop pode lançar apenas 1 *container* nos nós compartilhados. Com apenas 1 *container* em execução, o diagrama apresenta uma cor próxima

do branco e a altura de somente 1/8 da linha, tornando a visualização de *containers* Map difícil. No caso de um *container* Reduce em execução, como no diagrama MyBef do TestDFSIO, a cor verde facilita a visualização.

Nos diagramas gerados para os casos MyDur da aplicação TestDFSIO é possível notar que um dos nós termina uma carga de 6 a 7 containers simultaneamente e que logo após isso a aplicação toda é concluída. Uma rápida consulta à tabela fornece a informação de que nessa aplicação o número de tarefas especulativas no caso MyDur foi o mais elevado das 3 aplicações. Além disso, a análise dos logs mostra que a última tarefa especulativa termina 2 segundos antes do término de todos *containers* do primeiro nó. Estas duas observações sugerem que a sobrecarga do nó compartilhado impediu o término gradual dos *containers* lentos à medida que suas cópias especulativas finalizavam a execução. Este mesmo comportamento pode ser observado no caso DefSha.

Apesar das condições e mecanismos de compartilhamento/atualização implementados nos testes reais serem diferentes daqueles implementados nos testes controlados, os diagramas deste experimento também apresentam reação ao início do compartilhamento. É possível notar uma redução gradual dos recursos nos dois primeiros nós na maioria dos casos MyDur. Além disso, é possível notar que a linha nunca fica completamente colorida nos casos MyBef e MyDur, indicando que o coletor detectou os recursos utilizados pelo sistema operacional e excluiu os mesmos do *pool* de recursos do Hadoop.

Mais uma vez as execuções do caso MyDur apresentam uma melhora entre 36% e 24% no tempo total de Map comparado com o caso DefSha, mostrando que a detecção da sobrecarga no nó e a adaptação do processo de escalonamento para eventualmente eliminar a sobrecarga melhora o desempenho da aplicação. Ainda com relação ao tempo total de Map, nota-se que quanto maior a duração da aplicação maior é o ganho de desempenho pela utilização do escalonamento adaptativo. Os resultados também indicam que a adaptação do escalonamento em função da sobrecarga diminui a quantidade de tarefas especulativas e o tempo médio que as tarefas Map levam para concluir a execução.

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

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 32 Gb para o TeraSort. Para a escala de ambiente, utilizou-se um *cluster* com 10 escravos e entrada de 40 Gb, sendo que a proporção de recursos reduzidos/livres continuou a mesma.

4.4.2 Resultados e Interpretações

Os resultados dos experimentos em escala (de entrada e de ambiente) com a aplicação TeraSort podem ser visualizados nas Tabelas 4.7 e 4.8 e nas Figuras 4.7 e 4.8, respectivamente.

Tabela 4.7 – Resumo dos resultados do TeraSort em escala (tamanho da entrada) em segundos.

Caso	**DefDed**	**DefSha**	**MyBef**	**MyDur**
Tempo Total de Map (*s*)	1357	4627	2060	2184
Tempo Médio de Map (*s*)	43.39	299.70	45.36	65.00
# Tarefas Map	240	240	240	240
# Tarefas Especulativas	1	27	0	5

Tabela 4.8 – Resumo dos resultados do TeraSort em escala (tamanho do ambiente) em segundos.

Caso	**DefDed**	**DefSha**	**MyBef**	**MyDur**
Tempo Total de Map (*s*)	273	2138	743	1421
Tempo Médio de Map (*s*)	55.69	322.66	51.49	85.35
# Tarefas Map	298	298	298	298
# Tarefas Especulativas	2	26	1	1

No experimento realizado com o tamanho da entrada maior pode-se notar que o caso MyBef apresentou um tempo de execução mais rápido que o esperado. O tempo de execução atingido foi aproximadamente 50% maior comparado com o caso DefDed, porém o esperado era de 100%. Este valor era esperado pois no caso MyBef a configuração dos recursos alocáveis é de 2 escravos com 1Gb/core e 2 escravos com 7Gb/cores enquanto a configuração dos recursos alocáveis no caso DefDed é de 4 escravos com 8Gb/cores. Este resultado sugere

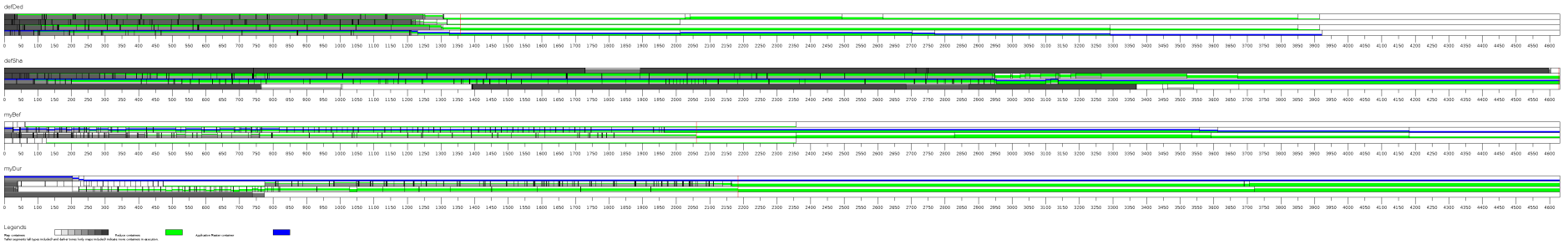

Figura 4.7 – Diagrama de Gantt para o experimento em escala (tamanho da entrada) com TeraSort

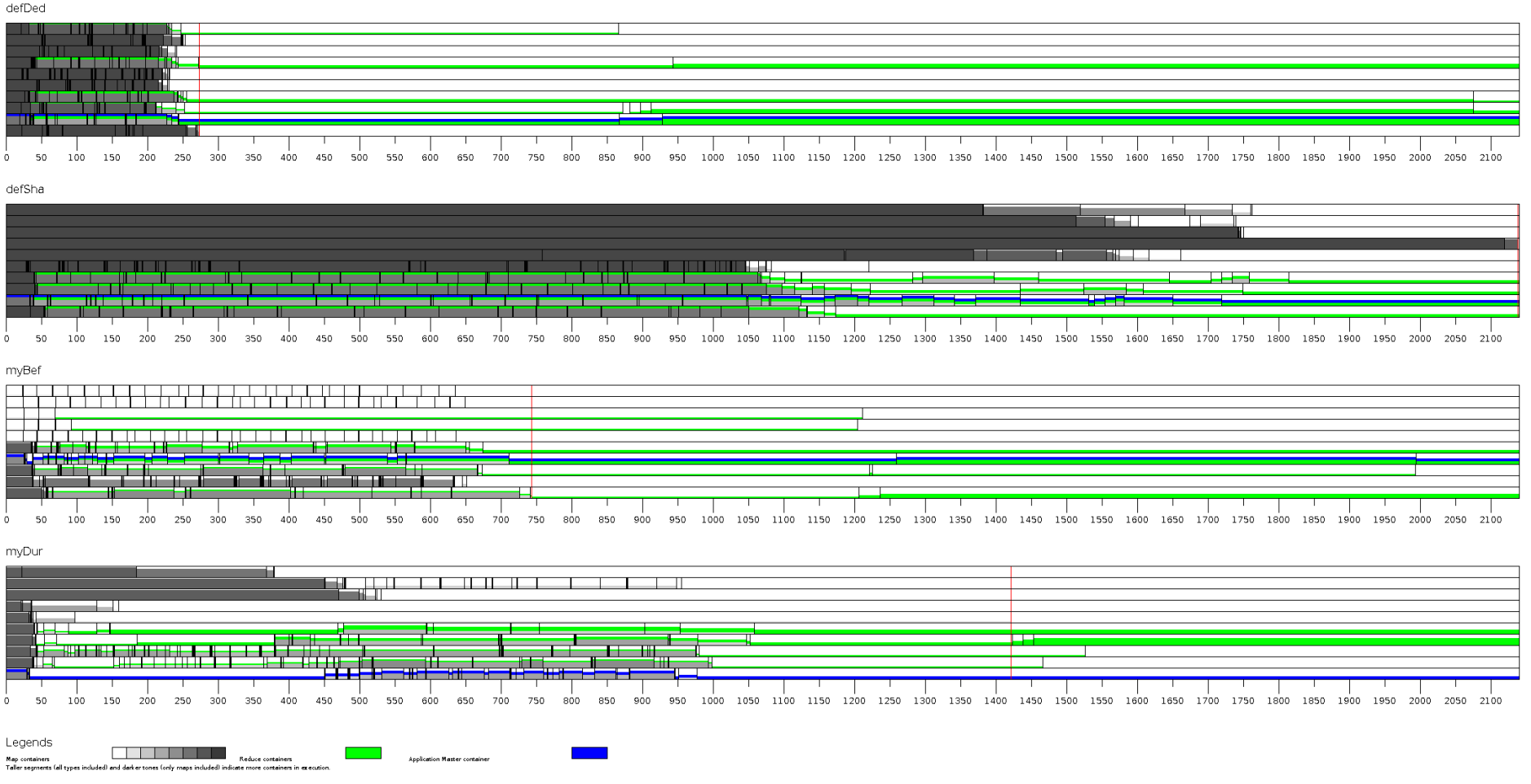

Figura 4.8 – Diagrama de Gantt para o experimento em escala (tamanho do ambiente) com TeraSort

que desconsiderar os recursos utilizados pelo sistema operacional e serviços do Hadoop já cria uma sobrecarga no nó, porém esta só é perceptível em aplicações que possuem um tempo de execução mais elevado.

Nota-se, mais uma vez, que os tempos de execução das aplicações apresenta a ordem DefDed, MyBef, MyDur e DefSha. Contudo, neste experimento a diferença de tempo entre os casos DefSha e MyBef, tanto do tempo total como do tempo médio dos *containers* é maior, sugerindo que a solução deste trabalho possui uma efetividade diretamente proporcional ao tamanho da aplicação que é executada. Ainda, o número de tarefas especulativas nos casos DefSha foi muito maior se comparado com os casos onde a solução deste estudo foi implementada.

Os diagramas de Gantt dos experimentos em escala possibilitaram a identificação de outra característica relacionada ao escalonamento, porém esta é uma falha na solução deste estudo. No caso AdaptDur, tanto na escala de entrada como de tamanho do ambiente, é possível identificar zonas onde os nós não sobrecarregados possuem poucos (0-4) *containers*. Este comportamento é causado por uma peculiaridade do Capacity Scheduler, o qual relega o lançamento de *containers* para as filas. Antes da decisão de lançar o *container* ou não, existe um teste que, ao contrário do inicialmente esperado, testa se o total de recursos utilizados **do *cluster*** é menor que o total de recursos informados. Uma vez que o teste ignora a situação dos recursos específicos de cada nó e é feito com a situação dos recursos do *cluster*, a atualização da capacidade de alguns nós fará com que os recursos utilizados sejam maiores que a capacidade total do *cluster*.

O processo de lançamento sofre influência do recurso de cada nó informado ao escalonador, porém a influência da atualização dos recursos não é local, no sentido de afetar somente o lançamento de *containers* do próprio nó que foi atualizado, mas sim global. Isso implica que a atualização dos recursos de 1 nó infuenciará em todos os *containers* que serão lançados, independentemente de serem lançado ou não no nó que teve seus recursos atualizados. Por exemplo, todos os nós terão inicialmente 7 Gbs de memória e 7 *cores* disponíveis, fazendo com que o *cluster* tenha um total de 70 Gbs e 70 cores. Assim, serão alocados 70 *containers* de 1 Gb de memória e 1 *core* cada. Após o término da alocação, 5 destes nós sofrem atualização dos recursos informados ao escalonador, deixando o *cluster* com um total de 55 Gb e 55 *cores*. Porém, existem 70 *containers* em execução, o que significa que a quantidade de recursos utilizados é de 70 Gb e 70 *cores*. Mesmo que cada um dos 5 nós não sobrecarregados utilizem apenas 2 Gb e 2 *cores* (dos seus 7 disponíveis), caso os *containers* alocados aos nós sobrecarregados não terminem, a utilização do *cluster* continuará acima de 100% (60 Gb e 60 *cores* usados, 55 Gb

e 55 *cores* informados), e portanto, nenhum novo *container* será lançado por mais que existam recursos ociosos em alguns nós.

Para seguir o objetivo inicial de não fazer grandes modificações no algoritmo de escalonamento, seria necessário um controle intermediário da quantidade de recursos disponíveis para o Hadoop e a quantidade de recursos atualmente sendo utilizada pelo Hadoop. Fazendo a redução desta informação no escalonador de forma gradual, à medida que os *containers* finalizam seu processamento. Se este controle for implementado, o problema identificado nestes diagramas de Gantt seria eliminado e não haveriam recursos ociosos em nenhum momento.

Ainda assim, os casos que utilizam a solução deste estudo apresentaram um desempenho melhor que a distribuição *default* do Hadoop.

HAHAHA

4.5 Experimento de medição do *Swap*

4.5.1 Procedimentos

4.5.2 Resultados e Interpretações

5 CONCLUSÃO

No contexto atual do processamento distribuído de dados, uma alternativa que sempre deve ser considerada é a utilização do MapReduce e seus *frameworks* de processamento. Sendo o Hadoop o mais conhecido destes, a sua utilização para o processamento de dados de maneira distribuída vem ganhando mais e mais adeptos. Embora existam serviços que disponibilizem um ambiente na nuvem para que o usuário processe seus dados, algumas pequenas empresas podem preferir outra alternativa à passar seus dados para a nuvem. Nestes casos, pode ser mais simples utilizar os recursos ociosos que estão disponíveis dentro da própria empresa.

Ainda que a utilização dos recursos ociosos da própria empresa pareça mais simples, o Hadoop não é capaz de gerenciar recursos compartilhados em um nó do *cluster*. Este estudo teve como objetivo tornar o Hadoop capaz de suportar compartilhamento de recursos através de um escalonamento adaptativo com relação aos recursos disponíveis. A solução realiza esta tarefa através da coleta e transmissão de dados, em conjunto com a utilização do escalonador Capacity Scheduler. Foram realizados experimentos que apresentavam a degradação de recursos, sendo que esta degradação poderia ocorrer tanto antes do início quanto no decorrer da aplicação de MapReduce. Estes dois cenários foram enfatizados por três fatores: (1) são os cenários que ocorrem mais comumente em ambientes compartilhados; (2) a entrada/saída de nós já é suportado, com algumas peculiaridades, pelo Hadoop; e (3) a reintegração de recursos compartilhados já têm suporte com a solução implementada, porém, comparada à degradação de recursos, não possui impacto significativo no desempenho.

A solução deste estudo foi capaz de melhorar o desempenho nos casos testados com utilização compartilhada do *cluster*. Os resultados alcançados neste estudo indicam que a utilização do Hadoop em um *cluster* compartilhado é possível, mesmo quando este for composto por computadores pertencentes à uma pequena rede empresarial que pode, a qualquer momento, ter a utilização concorrente dos recursos em determinados nós. Como resultado do desenvolvimento deste estudo, apresentou-se uma solução, simples e não intrusiva, para a utilização do Hadoop em ambientes compartilhados. Foram utilizados apenas os recursos que inicialmente são considerados pelo Hadoop (memória e *cores*) para tornar o escalonamento adaptável.

O benefício da utilização da solução deste estudo é decorrente da eliminação ou, nos piores casos, minimização da sobrecarga dos nós em virtude do compartilhamento. O simples fato de inibir o lançamento de novos *containers* quando alguns nós do *cluster* estão sobrecarregados

apresentou um ganho de desempenho de até 40% em alguns casos quando comparado com a distribuição *default* do Hadoop. Mesmo que o ganho de desempenho apresentado seja bom, foi identificada uma falha na solução. Sendo assim, é possível que o ganho de desempenho real seja ainda maior.

Utilizando este estudo como ponto de partida, diversos trabalhos futuros são possíveis. Destacando-se, é claro, um melhor controle dos recursos sobrecarregados para que a falha discutida nos experimentos em escala seja solucionada. Além disso, é possível explorar o Application Master com objetivo de compreender e tornar o gerenciamento interno de cada aplicação adaptativo.

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 OPEARTING SYSTEMS DESIGN & IMPLEMENTATION - VOLUME 6, 6., Berkeley, CA, USA. **Proceedings...** USENIX Association, 2004. p.10–10. (OSDI'04).

DEAN, J.; GHEMAWAT, S. MapReduce: simplified data processing on large clusters. **Commun. ACM**, New York, NY, USA, v.51, n.1, p.107–113, Jan. 2008.

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.

HADOOP. **Capacity Scheduler**. https://hadoop.apache.org/docs/r2.6.0/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html, Último acesso: Dezembro 2015.

HAMILTON, J. **Hadoop Wins TeraSort**. http://perspectives.mvdirona.com/2008/07/hadoop-wins-terasort/, Last access: September 2015.

HUANG, S. et al. The HiBench benchmark suite: characterization of the mapreduce-based data analysis. In: DATA ENGINEERING WORKSHOPS (ICDEW), 2010 IEEE 26TH INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2010. p.41–51.

HUNT, P. et al. ZooKeeper: wait-free coordination for internet-scale systems. In: USENIX ANNUAL TECHNICAL CONFERENCE, Boston, MA. **Proceedings...** USENIX Association, 2010. p.11–11.

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).

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).

MAAMAR, Z.; BENSLIMANE, D.; NARENDRA, N. C. What can context do for web services? **Commun. ACM**, New York, NY, USA, v.49, n.12, p.98–103, Dec. 2006.

NASCIMENTO, A. P.; BOERES, C.; REBELLO, V. E. F. Dynamic Self-scheduling for Parallel Applications with Task Dependencies. In: INTERNATIONAL WORKSHOP ON MIDDLEWARE FOR GRID COMPUTING, 6., New York, NY, USA. **Proceedings...** ACM, 2008. p.1:1–1:6. (MGC '08).

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

PARASHAR, M.; PIERSON, J.-M. Pervasive Grids: challenges and opportunities. In: LI, K. et al. (Ed.). **Handbook of Research on Scalable Computing Technologies**. [S.l.]: IGI Global, 2010. p.14–30.

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).

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).

SANDHOLM, T.; LAI, K. MapReduce Optimization Using Regulated Dynamic Prioritization. In: ELEVENTH INTERNATIONAL JOINT CONFERENCE ON MEASUREMENT AND MODELING OF COMPUTER SYSTEMS, New York, NY, USA. **Proceedings...** ACM, 2009. p.299–310. (SIGMETRICS '09).

SANDHOLM, T.; LAI, K. Dynamic Proportional Share Scheduling in Hadoop. In: INTERNATIONAL CONFERENCE ON JOB SCHEDULING STRATEGIES FOR PARALLEL PROCESSING, 15., Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2010. p.110–131. (JSSPP'10).

STEFFENEL, L. A. et al. PER-MARE: adaptive deployment of mapreduce over pervasive grids. In: EIGHTH INTERNATIONAL CONFERENCE ON P2P, PARALLEL, GRID, CLOUD AND INTERNET COMPUTING, 2013., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2013. p.17–24. (3PGCIC '13).

SU, X.; SWART, G. Oracle In-database Hadoop: when mapreduce meets rdbms. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 2012., New York, NY, USA. **Proceedings...** ACM, 2012. p.779–790. (SIGMOD '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).

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).