ESCALONADOR DISTRIBUÍDO DE TAREFAS PARA O APACHE SPARK

João Paulo de Freitas Ramirez

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Claudio Luis de Amorim

Rio de Janeiro
Setembro de 2018

ESCALONADOR DISTRIBUÍDO DE TAREFAS PARA O APACHE SPARK

João Paulo de Freitas Ramirez

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Claudio Luis de Amorim, Ph.D.

Prof. Felipe Maia Galvão França, Ph.D.

Prof. Maria Clicia Stelling de Castro, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
SETEMBRO DE 2018

Ramirez, João Paulo de Freitas

Escalonador Distribuído de Tarefas para o Apache Spark/João Paulo de Freitas Ramirez. – Rio de Janeiro: UFRJ/COPPE, 2018.

XIV, 63 p. 29, 7cm.

Orientador: Claudio Luis de Amorim

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2018.

Referências Bibliográficas: p. 46 – 49.

1. Escalonamento. 2. Distribuído. 3. Apache Spark. I. Amorim, Claudio Luis de. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Em memória de Dadá e Cinzinha.

Agradecimentos

Agradeço a Deus por me proporcionar a oportunidade de realização de estudos. Um agradecimento especial para Eduarda que sempre esteve comigo nessa e em outras batalhas da vida. Agradeço a minha mãe por ter me dado as condições fundamentais de estudo. Agradeço muito ao meu orientador Claudio que sempre me trouxe de volta ao caminho mais certo para eu ter condições de concluir este trabalho. Agradeço a todo o Programa de Engenharia de Sistemas e Computação, ao corpo da secretaria por resolver os problemas enfrentados com paciência e compreensão. Agradeço ao Laboratório de Computação Paralela, a Universidade Federal do Rio de Janeiro e ao CNPq por me proporcionarem os recursos fundamentais para o andamento deste trabalho. Por fim, agradeço aos meus amigos e colegas que me apoiaram e trilharam comigo esse caminho, amigos das disciplinas, do programa, do trabalho e da vida.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

ESCALONADOR DISTRIBUÍDO DE TAREFAS PARA O APACHE SPARK

João Paulo de Freitas Ramirez

Setembro/2018

Orientador: Claudio Luis de Amorim

Programa: Engenharia de Sistemas e Computação

Frameworks de análises de grandes quantidades de dados possuem um papel de destaque na academia e na indústria. Eles tornam questões de paralelização e distribuição de computação transparentes a cientistas de dados, proporcionando-lhes maior produtividade no desenvolvimento e implementação de aplicações para análise de dados como algoritmos de aprendizagem de máquina e análises em *streaming* de dados.

Entretanto, prover essa camada de abstração traz grandes desafios ao envolver questões como alto desempenho no uso de diversas unidades de processamento, nos mecanismos de comunicação e nos dispositivos de armazenamento.

Um dos principais desafios está relacionado ao escalonamento de tarefas. Realizar essa ação de forma eficiente e escalável é de suma importância, visto que ela impacta significantemente o desempenho e a utilização de recursos computacionais.

Este trabalho propõe uma estratégia hierárquica de distribuir o escalonamento de tarefas em *Frameworks* de análises de grandes quantidades de dados, introduzindo o escalonador assistente, cuja função é aliviar a carga do escalonador centralizado ao se responsabilizar por uma fração da carga de escalonamento.

O Apache Spark foi o *Framework* utilizado para implementarmos uma versão do escalonador de tarefas distribuído hierárquico e para realizarmos experimentos comparativos para testar a escalabilidade da proposta.

Utilizando 32 nós computacionais, os resultados mostram que a nossa prova de conceito mantém valores para tempos de execução semelhantes aos do Apache Spark original.

Além disso, mostramos que, com o emprego dos escalonadores assistentes, conseguimos fazer melhor uso dos processadores dos nós durante a execução dos experimentos.

Por fim, expomos um gargalo existente no modelo de execução do Apache Spark devido à centralização das decisões de escalonamento.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

PROPOSAL AND EVALUATION OF A DISTRIBUTED TASK SCHEDULER FOR APACHE SPARK

João Paulo de Freitas Ramirez

September/2018

Advisor: Claudio Luis de Amorim

Department: Systems Engineering and Computer Science

Data intensive frameworks have a featured role in industry and academia. They make computation distribution and parallelization transparent to data scientists providing them greater productivity in data analytics application development like machine learning and deep learning algorithms implementations.

However, in order to provide this abstraction layer several issues emerge like obtaining good performance when using many processing units, handling communication and coping with storage devices.

One of the most relevant challenges is related with task scheduling. Perform this action in an efficient and scalable manner is very important, since it impacts significantly computing resources performance and utilization.

This work proposes an hierarchical manner of distributing task scheduling in data intensive frameworks, introducing the scheduler assistant whose role is alleviate central scheduler job as it takes responsibility of a share of the scheduling load.

We use Apache Spark for implementing a version of the hierarchical distributed task scheduler and to perform comparative experiments for testing the proposal scalability.

Using 32 computational nodes, results show our proof of concept maintains execution times values similar to those found with the original version of Apache Spark.

Moreover we show that deploying scheduler assistants the system can better utilize computational nodes processors during experiments executions.

Finally, we expose a bottleneck due the centralization of scheduling decisions at Apache Spark execution model.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Listagens

Capítulo 1

Introdução

Um vasto volume de dados é produzido em taxas jamais vistas hoje em dia. Milhões de pessoas postando mensagens, fotos e vídeos nas mais variadas plataformas e máquinas monitorando eventos e sistemas críticos através de sensores são apenas alguns exemplos de diferentes fontes geradoras de dados.

Os dados possuem um apelo comercial muito valioso. Alavancados pelos avanços em algoritmos e técnicas de análise, como aprendizagem de máquina e aprendizagem profunda, as organizações criam aplicações capazes de extrair grande valor dos dados. Essas aplicações analisam imagens, texto, linguagem natural, vídeos e muitos outros conteúdos em diversas áreas como segurança, comércio, entretenimento, medicina e diversos setores da indústria.

Dado o volume, velocidade e variabilidade em que os dados são gerados, realizar uma análise de qualidade para extrair informações relevantes é uma atividade computacionalmente complexa. Entretanto, o processamento empregado nas tarefas de análises de dados apresenta características muito favoráveis ao paralelismo. Grandes porções de dados podem ser processadas independentemente uma das outras.

A abundância de paralelismo levou os analistas a utilizarem cada vez mais recursos computacionais, abrangendo o uso de máquinas com cada vez maior quantidades de processadores e múltiplos núcleos por processador, *cluster* de computadores com milhares de máquinas e dispositivos aceleradores de computação, como *Graphical Processing Units* (GPUs), aceleradores Intel Xeon Phi e *Field Programmable Gate Arrays* (FPGAs).

Entretanto, diferentemente da computação de alto desempenho, a análise de grandes quantidades de dados pode usufruir de *hardwares* mais acessíveis (*commodity hardware*) como armazenamento em disco rígido, ao invés de tecnologias de *Solid State Disk* (SSD), e rede de interconexão Ethernet, em contrapartida a redes de alta velocidade como a rede Infiniband.

Além disso, o paradigma de computação em nuvem trouxe a facilidade e praticidade de aquisição de recursos computacionais sob demanda de forma elástica. Esse

modelo possibilita que diversas organizações realizem análises de grandes quantidades de dados, despreocupando-se com os custos de infraestrutura computacional e gastos operacionais para prestar manutenção aos recursos digitais.

1.1 Motivação

Gerenciar o uso de diversos recursos computacionais é uma atividade desafiadora. Existe, pois, grande valor em conceber um sistema que torne as questões da distribuição e paralelização da computação transparente ao usuário. Trazer esse benefício de maneira eficiente é imprescindível para a viabilidade das aplicações.

Frameworks de análise de grandes quantidades de dados foram concebidos com esse propósito. Para alcançarem seus objetivos, diferentes soluções foram aplicadas para as difíceis questões de projeto. Dentre os mais disseminados podemos citar o Hadoop [4] e o Apache Spark [5]. Ambos permitem que os cientistas de dados implementem algoritmos e desenvolvam aplicações de maneira mais produtiva e em vasto volume de dados.

Alguns dos desafios enfrentados pelos *Frameworks* incluem o balanceamento de carga nas diversas e diferentes unidades de processamento, mecanismos de comunicação para uso eficiente da rede de interconexão de computadores, gerenciamento de escrita e leitura de dados nos dispositivos de armazenamento e um modelo de programação simples e robusto. A definição de uma forma de execução direciona as soluções aplicadas nesses problemas.

Muitos *Frameworks* organizam sua execução em um modelo *dataflow* e representam a computação como um grafo direcionado acíclico (*Directed Acyclic Graph* - DAG). Esse grafo exibe porções altamente paralelizáveis que são materializadas em tarefas a serem executadas nas unidades de processamento disponíveis.

Uma distribuição ineficaz de tarefas traz uma perda de desempenho significativa para as aplicações. Ao mesmo tempo, atrasos no escalonamento também provocam consequências graves como a má utilização dos recursos. Assim, construir mecanismos de escalonamento de tarefas que sejam eficientes, robustos e escaláveis é uma questão central da comunidade científica que investiga os *Frameworks* de análise de grandes quantidades de dados e das organizações que os utilizam massivamente de forma operacional.

1.2 Objetivos e Contribuições

Reconhecendo a importância do mecanismo de escalonamento de tarefas em *Frameworks* o presente trabalho tem por objetivo propor um escalonador de tarefas distribuído e implementar uma versão deste escalonador no Apache Spark como

uma prova de conceito. Nesta dissertação, discutimos uma forma hierárquica de distribuição da carga de trabalho de um escalonador centralizado, onde inserimos a figura do escalonador assistente que se responsabiliza pelo escalonamento de um subconjunto das tarefas totais presentes no sistema em um determinado instante.

Muitos *Frameworks* utilizam uma arquitetura centralizada para o escalonamento de tarefas. A abordagem centralizada tende a apresentar problemas de escalabilidade que limitam o desempenho do sistema ao ser utilizado um número expressivo de unidades de processamento. Esta dissertação pretende tratar esses problemas e proporcionar que os *Frameworks* alcancem larga escala de maneira eficiente.

O Apache Spark é um *Framework* que ganhou relevância recentemente. Durante sua execução, ele permite que dados que são reutilizados sejam guardados em memória ao invés de gravados em disco. Isso resultou em ganhos de desempenho ao compará-lo com o Hadoop MapReduce. Além disso, ele traz consigo um conjunto de bibliotecas que facilitam a implementação de aplicações de aprendizagem de máquina, processamento de grafos, consultas interativas e processamento de *stream* de dados. O Apache Spark apresenta um escalonador de tarefas centralizado. Portanto, implementar da prova de conceito da proposta de distribuição desta dissertação tem potencial para trazer benefícios a escalabilidade de seu desempenho.

Esta dissertação apresenta resultados experimentais comparativos entre o Apache Spark original e modificado para acomodar a prova de conceito de escalonamento distribuído, a fim de avaliar comparativamente a capacidade de escalabilidade das duas versões.

Dessa maneira as principais contribuições deste trabalho são:

- Descrever alguns dos principais desafios em *Frameworks* para análise de grandes quantidades de dados. Como mencionado anteriormente nesta seção, esses sistemas possuem o objetivo de tratar diversas questões de computação paralela e distribuída transparentes ao desenvolvedor de aplicações de análise de dados;

- Discutir propostas pertencentes ao estado da arte em escalonamento de tarefas nesses ambientes. Uma vez que a abordagem centralizada de escalonamento mostra gargalos na escalabilidade desses sistemas ao serem utilizados um número maior de tarefas, surgem diversas propostas para a distribuição do escalonamento de tarefas na literatura e examinar seus aspectos positivos e negativos é importante;

- Propor um novo modelo de escalonamento de tarefas distribuído para esses *Frameworks*. A proposta consiste em decompor hierarquicamente o escalonamento de tarefas, compartilhando a carga de trabalho do nó Mestre com agentes denominados escalonadores assistentes;

- Implementar uma versão do escalonamento distribuído hierárquico com os escalonadores assistentes no Apache Spark como prova de conceito. O Apache Spark é um *Framework* que se tornou popular nos últimos anos por ter mostrado um desempenho eficiente na execução de aplicações com características iterativas e interativas;

- Realizar experimentos comparativos de desempenho de escalabilidade entre a prova de conceito da proposta de escalonamento distribuído e a versão original de escalonamento centralizada.

1.3 Estrutura da Dissertação

O restante deste dissertação está organizado desta maneira: O Capítulo 2 aborda conceitos teóricos e os trabalhos relacionados a *Frameworks* para análise de grandes quantidades de dados, analisando alguns dos principais em uso na academia e na indústria e apresentando um subconjunto dos mais importantes desafios enfrentados. O Capítulo 3 apresenta e discute a proposta em detalhes e inclui a descrição da implementação realizada no Apache Spark, relatando decisões e dificuldades encontradas. O Capítulo 4 descreve a execução dos experimentos de comparação entre a versão original do Apache Spark e a prova de conceito de nossa proposta. Finalmente, o Capítulo 5 apresenta as conclusões do trabalho juntamente com suas principais extensões e trabalhos futuros.

Capítulo 2

Aspectos Conceituais e Teóricos

Neste capítulo, elaboramos acerca dos aspectos teóricos que compõem o presente trabalho. Mais precisamente, discutimos sobre *Frameworks* de análise de grandes quantidades de dados (seção 2.1) e aqueles mais disseminados e utilizados na academia e na indústria (seção (2.2).

Além disso, descrevemos as deficiências mais relevantes relatadas na literatura (seção 2.3) e abordamos mais amplamente o assunto do escalonamento de tarefas nos *Frameworks* de análise de grandes quantidades de dados (seção 2.4).

2.1 *Frameworks* de Análise de Grandes Quantidades de Dados

A necessidade de processamento de uma quantidade imensa de dados na indústria e academia é imediato. Atualmente, estima-se que grandes empresas de tecnologia como Google e Facebook processam diariamente dados na ordem de *petabytes*.

São diversas as formas como esses dados são adquiridos, por exemplo, interação de humanos e computadores, sensores coletando dados em diversos segmentos de indústria, simulações e muito mais com os tamanhos das bases de dados crescendo de maneira exponencial.

Dada essa complexidade, o uso de *datacenters* com grandes capacidades de armazenamento e computação é a realidade deste cenário. Com isso, a adoção de modelos paralelos e distribuídos se apresenta naturalmente ao problema observando, ainda, que, de maneira geral, o processamento desse tipo de carga de trabalho é oportunamente paralelo.

Usufruir corretamente de uma vasta quantidade de recursos é uma tarefa longe de ser trivial. O desenvolvedor precisa conhecer bem a arquitetura do sistema e despender um tempo significativo na elaboração do programa para lidar com comunicação entre os agentes computacionais, tolerância a falhas, balanceamento de

carga entre outras diversas questões. Isso se traduz em um desafio prejudicial à aplicação, pois sua finalidade é cumprir suas regras de negócio e trazer valor as partes interessadas em um tempo relativamente curto .

Em virtude desse contexto, surgiram os *Frameworks* de análise de grandes quantidades de dados. Os esforços nesse tipo de trabalho ocorrem nos segmentos da indústria e da academia simultaneamente. Entretanto, o artigo publicado por colaboradores do Google sobre seu internamente bem sucedido e disseminado modelo de programação *MapReduce* [6] propagou amplamente este conhecimento às diversas áreas da Engenharia e Ciência da Computação.

A disseminação dos serviços de computação em nuvem como *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS) e *Software as a Service* (SaaS) contribuíram para a popularização do uso dos *Frameworks* de análise de grandes quantidades de dados. As grandes fornecedoras de servicços de nuvem oferecem meios especializados para criação e uso desses *Frameworks*. Podemos citar, por exemplo, o Amazon *Elastic MapReduce* (EMR) como fornecedor deste tipo de serviço. Dessa forma, uma organização não precisa mais despender gastos exacerbados com planejamento e aquisição de infraestrutura para executar aplicações de análises de dados com os *Frameworks* em questão. Todo esse trabalho pode ser realizado de forma elástica e sob demanda.

Uma característica comumente encontrada na maioria dos *Frameworks* de análise de grandes quantidades de dados é o mecanismo de execução *dataflow* baseado em um grafo direcionado acíclico. Deste modo, a execução do processamento na aplicação consiste em fases que correspondem aos nós do grafo e cada aresta corresponde a troca de dados gerados de uma fase a outra. Dentro de cada fase são executadas computações em paralelo e as fases sem arestas entre si não possuem relação de dependência e podem ser executadas simultaneamente. Os nós fonte e sorvedouro estão relacionados a entrada e saída dos dados de um processamento na aplicação .

2.2 O Estado da Arte dos *Frameworks*

Existem diversos *Frameworks* de análise de grandes quantidades de dados atualmente. Alguns dos mais populares são Hadoop [4], Dryad [7], Apache Spark [5], Apache Flink [8], Giraph [9], Apache Storm [10] e TensorFlow [11].

2.2.1 Hadoop

O Hadoop é um conjunto de softwares de código aberto voltados para a computação distribuída, confiável e escalável mantido pela Apache Software Foundation via con-

tribuição de muitos desenvolvedores. Ele foi inicialmente desenvolvido por engenheiros do Yahoo! e, posteriormente, disponibilizado em forma de código aberto. Atualmente, o Hadoop se divide em Hadoop Common, Hadoop Distributed File System (HDFS), Hadoop Yarn e Hadoop MapReduce [4].

O Hadoop foi originalmente desenvolvido por Doug Cutting como uma implementação do modelo *MapReduce* descrito pelo Google [12].

O *MapReduce* foi apresentado em [6]. É um modelo de programação inspirado pelas primitivas *map* e *reduce* presentes em diversas linguagens de programação funcional. Ele consiste em aplicar uma operação de *map* em todo registro da entrada de dados, a fim de computar um conjunto intermediário de pares em forma de chave-valor. Posteriormente, aplica-se a operação de redução em todos os pares que partilham da mesma chave com o objetivo de combinar os valores associados gerando uma saída.

Na Listagem 2.1, podemos ver um exemplo de uma operação de *map* e outra de *reduce* para um programa de contagem de palavras em documentos. A função *map* recebe o identificador, `key`, correspondendo ao nome do documento e um valor, `value`, que é o texto do documento.

A Linha 4 percorre todas as palavras do documento e emite um valor intermediário na forma de uma tupla com a palavra do documento corrente na iteração e a `string` 1. A palavra serve como identificador para a tupla emitida.

A função de *reduce* recebe a mesclagem dos valores produzidos no *map* que possuem o mesmo identificador. Esses valores são passados nos parâmetros `key`, o identificador, e `values`, a coleção dos valores mesclados para o identificador. O identificador é uma palavra do documento e os valores são todos a `string` 1.

A iteração na Linha 11 acumula as `strings` em valores e acumula a soma na variável `result`, emitindo a quantidade de vezes em que uma palavra está presente no documento na Linha 13.

A execução do modelo *MapReduce* está ilustrado na Figura 2.1. Ele pode ser dividido em seis etapas. A primeira consiste na criação dos processos de Mestre e Trabalhadores. O Mestre é encarregado de escalonar as tarefas de *map* e *reduce* entre os Trabalhadores. Isso é a segunda etapa. Na terceira etapa, ocorre a aplicação da operação de *map* em dados de entrada e o armazenamento do resultado é escrito em disco na etapa quatro. As tarefas de *reduce* são responsáveis por ler os dados gerados na etapa anterior em todos os nós na etapa cinco. A última etapa é onde as tarefas de *reduce* aplicam suas operações e escrevem o resultado em disco.

Dentro do *MapReduce* do Hadoop existe a figura do Mestre da Aplicação (*Application Master*). Ele é o responsável por coordenar, gerenciar e escalonar a execução das tarefas de *map* e *reduce* nos nós computacionais disponíveis para a aplicação [12].

```
1  map(String key, String value):
2    // key: document name
3    // value: document contents
4    for each word w in value:
5      EmitIntermediate(w, '1');
6
7  reduce(String key, Collection values):
8    // key: a word
9    // values: a list of counts
10   int result = 0;
11   for each v in values:
12     result += ParseInt(v);
13     Emit(AsString(result));
```

Listagem 2.1: Exemplo de operação de *map* e *reduce* para um programa de contagem de palavras.

2.2.2 Apache Spark

O Apache Spark é uma plataforma unificada para processamento em Big Data. Ele foi criado em 2009 na Universidade da Califórnia, Berkeley, e lançado em 2010. Atualmente, é um projeto de código aberto mantido pela Apache Software Foundation tendo contribuições de mais de mil desenvolvedores e já tendo sido utilizado por milhares de organizações [13].

Este *framework* surgiu como uma alternativa para atacar deficiências que o Hadoop apresentava em aplicações que não se adequavam ao modelo *MapReduce*. Ocaso de aplicações interativas e iterativas, pois, ao serem executadas no Hadoop, elas sofrem uma penalidade muito grande no tempo de execução devido aos frequentes acessos ao disco realizados na execução do *MapReduce* no Hadoop [5].

Esse prejuízo ocorre pela obrigatoriedade da leitura e escrita de dados no disco a cada execução *MapReduce*, mesmo que uma consecutiva execução utilize os mesmos dados ou dados gerados em uma execução anterior. Nas aplicações interativas, diversas consultas são realizadas em um mesmo conjunto de dados. Entretanto, no Hadoop, cada consulta deve ler os mesmos dados do disco toda vez em que for executada. O mesmo acontece em aplicações iterativas, onde cada iteração é uma execução *MapReduce* e utiliza o mesmo conjunto de dados.

Para resolver esse problema, o Apache Spark possibilita o *cache* de dados em memória. Isso possibilita que as aplicações reutilizem os dados em suas execuções subsequentes, favorecendo o tempo das consultas e iterações, por não haver necessidade de leitura dos dados a partir do disco. Essa mudança de paradigma levou a ganhos significativos de desempenho [5].

O Spark também propõe um modelo de programação que generaliza o *MapRe-*

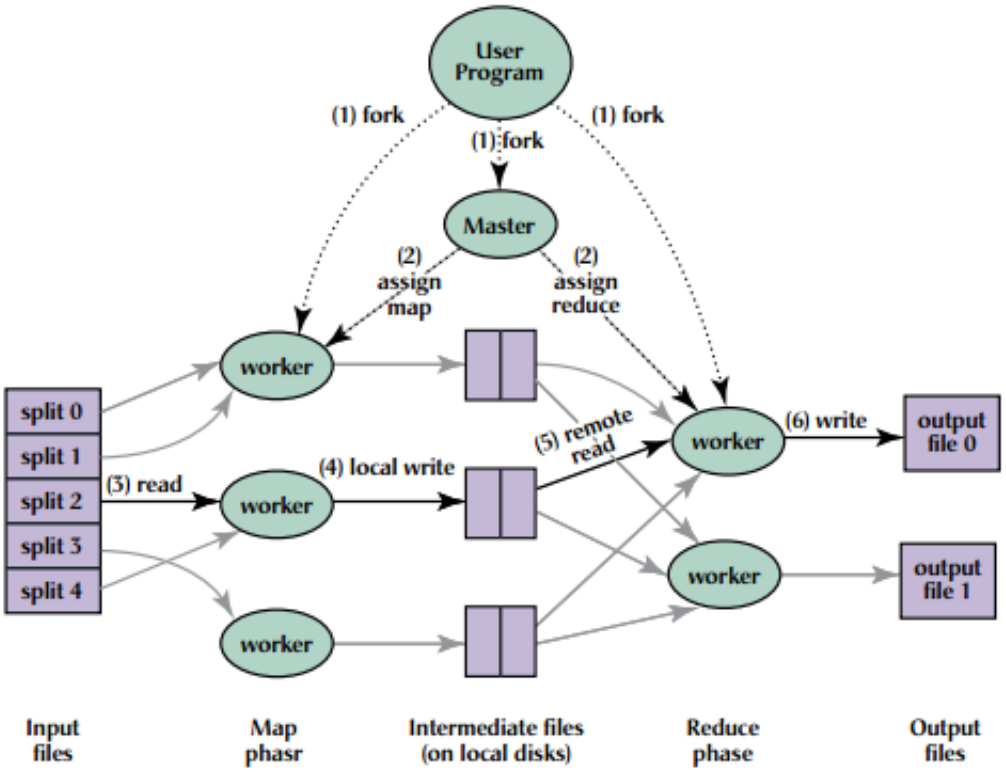


Figura 2.1: O modelo de execução *MapReduce*. Adaptado de [1]

duce através dos *Resilient Distributed Datasets* (RDDs). Um RDD é uma coleção imutável genérica de valores distribuída através dos nós do cluster de computadores que oferece uma interface para operações de grão grosso em seus elementos. Exemplos dessas operações são *map*, *filter* e *join* [5].

A execução de uma aplicação no Apache Spark segue a arquitetura de Mestre e Trabalhadores. O Mestre gerencia e escalona as tarefas da aplicação nos Trabalhadores que executam a computação designada às tarefas. Dentro de uma aplicação, o processo Mestre é denominado Driver no Spark e os processos Trabalhadores são os Executors. Os Executors são os responsáveis por realizar o *cache* dos dados em memória.

A Figura 2.2 ilustra a arquitetura do Apache Spark. O *Driver Program* é o Mestre que coordena toda a execução de uma aplicação. Ele se comunica com o *Cluster Manager* para obter os recursos necessários para a aplicação. O *Cluster Manager* instancia *Executors* nas máquinas concedidas à aplicação. Os nós Trabalhadores correspondem a essas máquinas.

O Apache Spark é capaz de se comunicar com os *Cluster Managers* Mesos [14] e YARN [15], além de possuir um *Cluster Manager* próprio.

Internamente, o Apache Spark organiza a aplicação em um conjunto de *jobs* que são submetidos para execução. Os *jobs* podem ser submetidos serialmente ou concorrentemente. A submissão concorrente só é possível quando os *jobs* são definidos

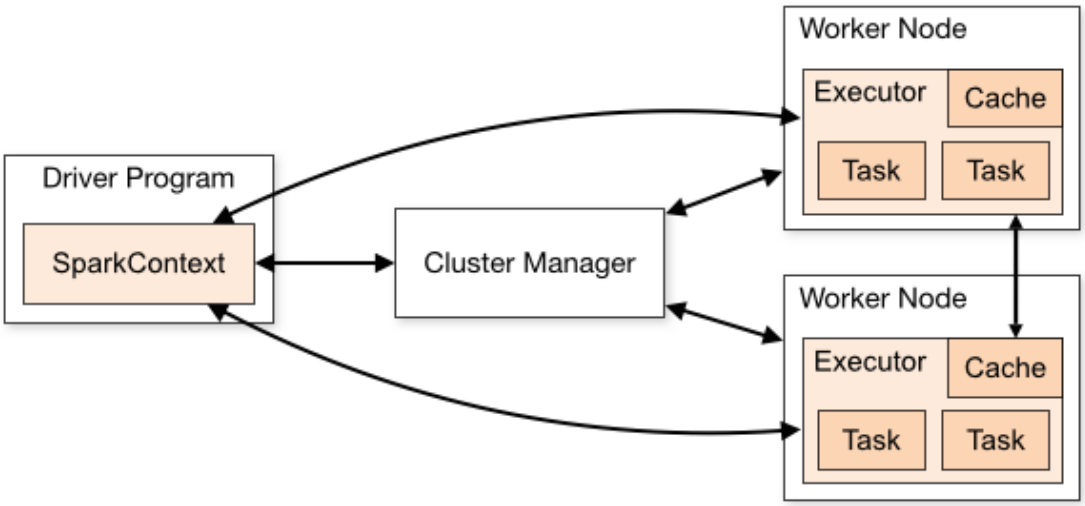


Figura 2.2: Arquitetura do Apache Spark. Adaptado de [2]

em *threads* diferentes.

Cada *job* é composto por um ou mais estágios. Um estágio consiste em um conjunto de tarefas que não se comunicam entre si e atuam independentemente uma das outras. As tarefas de um mesmo estágio realizam a mesma computação nas diferentes partições de um RDD. Os estágios de um *job* formam um Grafo Acíclico direcionado (DAG) cujos nós são os estágios e as arestas são as dependências entre os estágios e indicam a necessidade de comunicação entre os *Executors* do *cluster*.

A Figura 2.3 ilustra um DAG de estágios de um *job*. Essa representação gráfica é produzida pela interface gráfica que o Apache Spark providencia que fornece ao usuário informações sobre as aplicações executadas como *jobs*, estágios, tarefas, dados lidos e escritos, comunicação dos *Executors* entre estágios e diversas outras que facilitam o entendimento da execução das aplicações.

A composição de *jobs*, estágios e tarefas e a definição do DAG em cada *job* é realizada transparentemente ao desenvolvedor da aplicação. O desenvolvimento é todo baseado em operações nos RDDs definidos na aplicação. Entretanto, para construir uma aplicação com melhor desempenho o desenvolvedor deve estar ciente dessas questões.

2.3 Trabalhos Relacionados

Os *Frameworks* de análise de grandes quantidades de dados possuem diversos desafios e estão sendo amplamente abordados pela comunidade científica devido a seu grande apelo para a indústria e a academia.

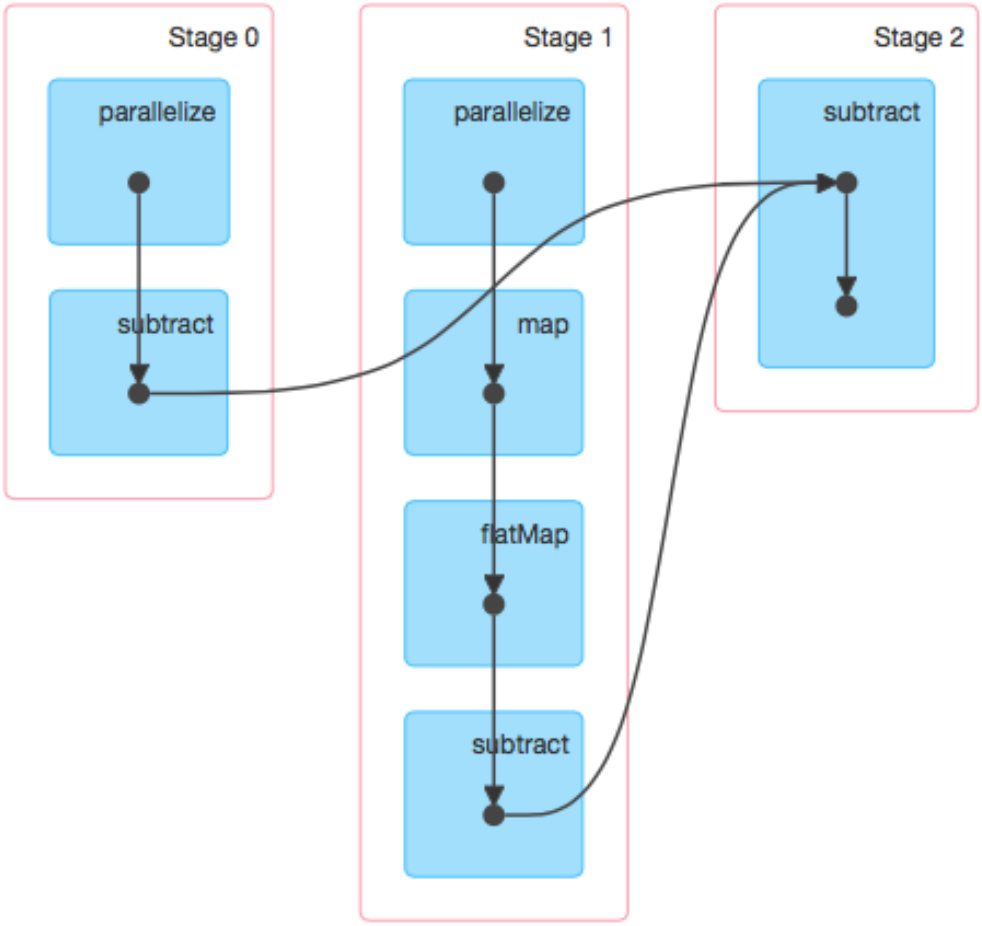


Figura 2.3: DAG de estágios de um *job* no Apache Spark. Fonte [3]

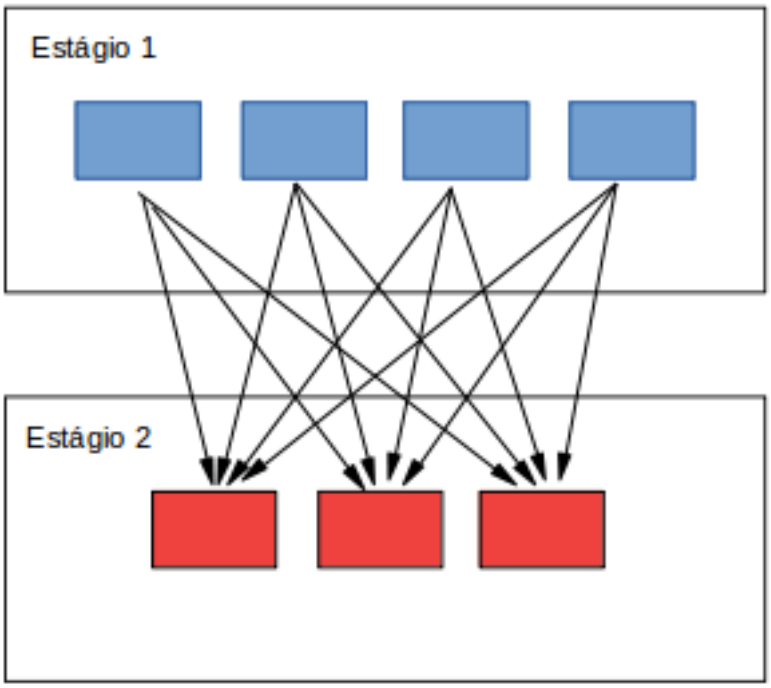


Figura 2.4: Ilustração da fase de *shuffle*

2.3.1 *Shuffle*

Shuffle é a fase da execução em que ocorre comunicação entre nós no *cluster* segundo o padrão muitos para muitos em *Frameworks* de análise de grandes quantidades de dados como o Hadoop MapReduce e o Apache Spark. A execução de uma aplicação é geralmente dividida em estágios e, entre estágios consecutivos, ocorre a fase de *shuffle*. A Figura 2.4 representa o processo de *shuffle* entre dois estágios diferentes na execução de uma aplicação.

O processo de *shuffle* é muito custoso, pois envolve escritas e leituras em disco, comunicação pela rede de interconexão e serialização/desserialização e objetos. Os autores em [16] revelam que a sobrecarga com esta fase chega a um terço em média do tempo total de duração da aplicação,ao analisarem traços de execução com uma semana de duração provenientes de um *cluster* Hadoop do Facebook.

O trabalho em [17] realiza uma otimização nessa fase. Os autores propõem um *plug-in* chamado SCache para *Frameworks* de análise de grandes quantidades de dados, cuja responsabilidade é tratar apenas do *shuffle*. Esta camada de software apresenta diversas otimizações ao processo original.

A primeira delas é desacoplar a realização do *shuffle* da execução das tarefas. Originalmente, em muitos *Frameworks* como Hadoop MapReduce e o Apache Spark, as tarefas são encarregadas de realizar o *shuffle*. Mesmo que isso seja transparente ao desenvolvedor, as tarefas precisam realizar essa atividade pesada em acesso a rede de interconexão e disco. Esse comportamento causa distorções na dinâmica das tarefas, podendo ter como consequência uma alta variância no tempo de conclusão

das tarefas. Com o desacoplamento, o SCache recebe chamadas das tarefas que o passam ou recebem os dados de *shuffle*. Com isso, as tarefas terminam logo após sua computação ou iniciam assim que atribuídas a um nó Trabalhador.

A segunda otimização é o *pre-fetch* dos dados de *shuffle*. O comportamento atual durante a leitura de dados nessa fase é fazer com que as tarefas os busquem nos nós Trabalhadores que os produziram. Isso é realizado, aproximadamente, ao mesmo tempo por todas as tarefas, podendo ocasionar uma sobrecarga na rede de interconexão. Ao ser realizado o *pre-fetch*, os recursos de comunicação tendem a ser melhor aproveitados devido a uma coordenação em sua utilização.

2.3.2 Serialização

Grande parte dos *Frameworks* de análise de grandes quantidades de dados, ao realizarem comunicação entre nós, precisam serializar objetos em uma sequência de *bytes* e desserializá-los no receptor da mensagem. Esse processo provoca uma sobrecarga significativa à execução da aplicação.

O trabalho em [18] mostra que o tempo de computação despendido no esforço de serializar e desserializar objetos no Apache Spark pode ser maior que o tempo gasto ao realizar entrada e saída de dados em disco.

Os autores de [19] relatam que o processo de serialização faz uso intenso de *reflection* uma operação de tempo de execução custosa, percorrendo toda a árvore de informação do objeto e o reformulando. Ocorrem remoção de referências armazenadas no objeto, retirada de dados de seu cabeçalho e mudanças na forma apresentada por certos metadados.

Além dessas sobrecargas, eles citam que cada objeto possui uma lógica própria para poder ser serializado corretamente e consideram, como exemplo, um objeto do tipo *HashMap*. Como sua serialização envolve toda sua estrutura de chave-valor, ela deve ser reconstruída no receptor, ou seja, todos os *hashes* de seus elementos devem ser recalculados.

Para lidar com estes problemas, esses autores propõem uma técnica na Java Virtual Machine (JVM) chamada Skyway. Essa técnica consiste em escrever, diretamente na memória do receptor, o objeto enviado pelo emissor presente em sua memória e, com isso, economizar tempo de processamento dispendido na serialização/desserialização.

2.3.3 Particionamento de Dados e Localidade

O particionamento de dados em aplicações de análise de grandes quantidades de dados é uma questão crucial para o bom desempenho. Além disso, a manutenção da localidade dos dados durante a execução do programa diminui a comunicação

entre os processos. O trabalho em [20] aborda essas questões problemáticas nos *Frameworks* de análise de grandes quantidades de dados, principalmente em aplicações de processamento de grafos propondo o *Framework* Grappa.

A solução proposta corresponde a um sistema de memória compartilhada distribuída que se beneficia do alto grau de paralelismo das aplicações, do tamanho pequeno das mensagens trocadas e da alta frequência de comunicação. Basicamente, o paralelismo permite que a latência da comunicação seja escondida e favorece o esquema de comunicação do Grappa que agrega diversas mensagens de tamanho reduzido para utilizar melhor a largura de banda da rede de interconexão disponível.

Dessa maneira, aplicações que tendem a realizar elevada transferência de dados como aquelas que sofrem de um particionamento ruim e de pouca localidade podem se beneficiar de um esquema de comunicação que se enquadre melhor nas suas necessidades.

2.3.4 Análise de Desempenho

Uma dificuldade com grande impacto nos desenvolvedores é o entendimento do desempenho de sua aplicação dentro de um *Framework*. Os *Frameworks* de análise de grandes quantidades são sistemas complexos que utilizam os diversos tipos de recursos computacionais ao longo da execução de uma aplicação. Elaborar um raciocínio acerca do desempenho de uma execução é uma tarefa árdua, porém necessária.

O investimento na aquisição de mais recursos computacionais deve ser muito bem planejado para evitar prejuízos. Aumentar a capacidade de memória, aumentar a largura de banda da rede, adquirir processadores com maior número de núcleos ou investir em uma nova tecnologia para armazenamento, como *Solid State Drivers* (SSDs) ou *Non Volatile Memories* (NVMs), são ações que precisam de informações precisas para serem tomadas, a fim de gerar uma boa análise de custo e desempenho.

A fim de prover maior informação acerca do uso de recursos, o trabalho em [21] propõe as monotarefas e a partição da execução de uma aplicação nessas monotarefas.

Uma monotarefa é uma unidade de trabalho que atua em apenas um determinado tipo de recurso como processador, rede e disco. A quebra da execução em monotarefas permite que um gargalo em um recurso computacional seja facilmente identificado além de outras vantagens para a aplicação como um escalonamento mais especializado a um dado recurso.

2.4 Escalonamento para *Frameworks* de Análise de Grandes Quantidades

O escalonamento em *Frameworks* de análise de grandes quantidades de dados é uma questão fundamental para o bom funcionamento do sistema e de suas aplicações. Em geral, podemos notar que existem dois tipos diferentes de escalonadores atuando em *Frameworks* de análise de grandes quantidades de dados.

Um deles atua no compartilhamento de recursos computacionais de um conjunto de computadores. Podemos denominá-los escalonadores de recursos ou gerenciadores de *cluster*. O outro visa distribuir da melhor maneira possível o trabalho de computação nos recursos que lhe foram concedidos, podendo ser denominados escalonadores de tarefas. Existem tanto cenários em que eles atuam separadamente em suas funções, quanto cenários em que eles devem interagir.

Ambos os tipos têm em vista, principalmente, uma elevada taxa de vazão de trabalho realizado em um *cluster*, a diminuição do tempo de conclusão de trabalho e manter a utilização dos recursos computacionais de maneira eficiente.

2.4.1 Escalonadores de Recursos

O primeiro tipo de escalonador é aquele cuja responsabilidade abrange todos os recursos disponíveis no *cluster*. Sua presença serve para lidar com os diferentes usuários dos recursos e os diferentes *Frameworks* que eles executam. Nesse contexto, desempenham papel similar aos escalonadores como o Slurm [22] e o Torque [23] presentes em ambientes de computação de alto desempenho.

Grandes organizações possuem uma gama diversificada de aplicações e serviços que devem ser executados simultaneamente. Essas aplicações são heterogêneas em diversos aspectos como carga de trabalho, tolerância a falhas, tempo de execução, demanda por recursos, restrições de disponibilidade, flexibilidade, segurança e usuários.

Mesmo que uma organização utilize apenas um *Framework*, há situações em que se deseja ter execuções experimentais, como novos algoritmos ou testar a nova versão do sistema, no mesmo *cluster* que execuções em produção.

Para todos esses cenários, é inviável possuir um conjunto de computadores dedicado para lidar com cada demanda.

Em [14], os autores propõem um escalonador de recursos, chamado Mesos, para lidar com o cenário em que um *cluster* deve ser compartilhado entre diversos *Frameworks*. Mesos visa ser escalável e resiliente. Para isso ele atua em conjunto com os sistemas clientes, enviando ofertas de recursos e recebendo aceitações ou negações desses recursos a partir dos *Frameworks*.

Embora soe contra intuitivo, o trabalho em [24] mostra que a negação de um

recurso pode ocorrer e ser benéfico para a aplicação. Nesse artigo, os autores implementaram um escalonador de tarefas no Hadoop que atrasa a atribuição de uma tarefa a um nó Trabalhador se este recurso não garantir a localidade dos dados utilizados pela tarefa. Mais ainda, apontaram uma melhora no tempo de finalização de aplicações executadas com esse mecanismo de atraso.

Ao trabalhar dessa forma, o Mesos pode distribuir com os diversos *Frameworks* o trabalho do particionamento de recursos, tornando-se escalável e minimizando o comprometimento da eficiência. Nesse sentido, o escalonador de tarefas dos *Frameworks* deve interagir com o Mesos. Para esse fim, o Mesos oferece uma *Application Programming Interface* (API) aos escalonadores de tarefas.

Apesar da dinâmica de funcionamento distribuída, o Mesos conta com uma figura central denominada Mesos Master, que coordena a oferta de recursos de *hardware* do *cluster* através de políticas, como priorização e particionamento justo. Existem políticas definidas por padrão, mas novas podem ser implementadas pelos administradores do conjunto computacional, a fim de que requisitos específicos sejam atendidos.

Diferentemente do Mesos é projetado para o compartilhamento de recursos entre diversos *Frameworks*, o Borg [25] possui a responsabilidade de gerenciar a utilização de *hardware* para diversas aplicações entre *Frameworks* e serviços de longa duração. O Borg é um projeto utilizado no Google para escalonar recursos de seu *data center*.

Esses objetivos abrangentes fazem do Borg um sistema complexo. Por esse motivo, esse gerenciador contém muitos serviços e ferramentas dentro de si como uma linguagem declarativa de configuração, para definição das execuções e tarefas que lhe são submetidas; um serviço de nomes, para que os serviços sejam encontrados por clientes externos; e monitores que verificam e avaliam o funcionamento das execuções correntes no Borg, resgatando e registrando informações de uso de recursos e disponibilidade das aplicações.

O volume de dados gerados é muito grande e expressivo. O Google divulgou o registro de um mês de execução de aplicações no Borg para a comunidade [26] e, em 2018, mais de 459 trabalhos [27] utilizaram esses registros em pesquisas para avaliar novas propostas de problemas relacionados a sistemas como em [28] e [29].

A arquitetura do Borg é definida pelo Borgmaster e os Borglets. O Borgmaster é a figura central do sistema e é composto de um processo que trata as requisições e do escalonador. Para fins de tolerância a falhas, o Borgmaster é replicado cinco vezes, elegendo-se uma das réplicas como líder e modificadora do estado dos dados utilizados para o gerenciamento do *cluster*. Os Borglets são agentes que executam nos nós do conjunto de computadores e executam as tarefas que lhe são atribuídas como processos em *containers* no Linux e reportam seu estado periodicamente ao Borgmaster.

O escalonador do Borgmaster precede por avaliar as submissões de execução considerando suas prioridades através de um esquema *round-robin*. Isso visa a a estabelecer a igualdade entre usuários e evitar o efeito de bloqueios por execuções de longa duração na frente da fila (*head-of-line*). O escalonamento é realizado em duas partes. A primeira é denominada viabilidade que localiza máquinas onde as execuções podem ocorrer e o pontuamento que seleciona em quais nós a execução ocorrerá, baseando-se em aspectos como preferências definidas pelo usuário.

2.4.2 Escalonadores de Tarefas

Aplicações exigem cada vez mais recursos para processar uma imensidão de dados. A alocação de tarefas deve tratar cenários com abundância de recursos de maneira eficiente. Essa questão não deve se tornar um gargalo para as aplicações. A ociosidade de recursos deve ser minimizada ao máximo e a sobrecarga do escalonamento também a fim de que haja *Frameworks* de análise de grandes quantidades de dados que escalem sua execução em milhares de nós.

Esses requisitos levantam questões como balanceamento de carga e tratamento de tarefas retardatárias que devem ser tratadas pelo escalonador, pois elas podem vir a atrasar a finalização das aplicações. Esses problemas estão relacionados a nós computacionais que apresentam algum mau funcionamento de disco, congestionamento na rede ou alguma outra perturbação na execução das tarefas.

Atualmente, o lançamento de tarefas especulativas é a técnica mais comumente utilizada para o tratamento de retardatários. O escalonador lança cópias especulativas de tarefas que sejam ou que possuem grandes chances de se tornarem retardatárias [30].

Além disso, as aplicações, atualmente, possuem a tendencia de segregarem o trabalho em tarefas menores com duração na ordem de centenas de milissegundos [31]. Tarefas menores implicam em maior equidade do uso dos recursos, distribuindo melhor o trabalho e evitando os casos de tarefas retardatárias por não atribuir tantas tarefas a máquinas com problemas de funcionamento.

Entretanto, um requisito fundamental para o uso de tarefas pequenas é o bom funcionamento do escalonador. Ele deve apresentar capacidade de processar grande quantidade de atribuições de tarefas em curto espaço de tempo, para não onerar o tempo de término das aplicações. Ao mesmo tempo, o escalonamento deve ser eficiente a fim de prover as melhores condições para a execução das tarefas das aplicações.

Nesse sentido, diversos trabalhos buscam uma maneira de abordar esses desafios em escalonadores de tarefas para *Frameworks* de análise de grandes quantidades de dados. Em [32], os autores propõem o Sparrow, um escalonador distribuído

sem informação do estado de uso do *cluster*, que consiste em diversos escalonadores atuando de maneira independente simultaneamente. Dessa maneira, o sistema alcança escalabilidade ao serem adicionados mais agentes de escalonamento. Além disso, cada nó Trabalhador possuí uma fila, onde são depositadas tarefas que lhe são escalonadas, mas que não podem ser executadas devido a escassez de recursos disponíveis.

Para atingir um bom nível de eficiência, os escalonadores utilizam uma versão modificada da técnica *Power of two choices*, denominado amostra em lotes. O escalonamento consiste em selecionar m tarefas de algum trabalho (*job*) e escolher o nó Trabalhador com menor carga dentre $d \times m$, com $d \geq 1$, Trabalhadores selecionados ao acaso para cada uma das m tarefas. Para realizar a escolha, o escalonador requisita o estado das filas de tarefas aos Trabalhadores selecionados.

Mais ainda, como tamanho da fila de tarefa não informa nada sobre a duração das tarefas na fila, o escalonamento em lotes pode ter um desempenho ruim em conjuntos computacionais com carga de uso elevada. Apesar de ser possível que os nós Trabalhadores informem aos escalonadores uma estimativa do tempo de duração das tarefas em sua fila, realizar essa previsão com acurácia é um desafio. Outra limitação decorre da condição de corrida na resposta e envio de novas tarefas à fila de um nó Trabalhador. Nesse caso, um Trabalhador aparenta ser o nó de menos carga a múltiplos escalonadores que lhe enviam suas tarefas concorrentemente, causando degradação do balanceamento de carga.

Para lidar com esse revés, o Sparrow emprega o uso de uma atribuição tardia, fazendo que os Trabalhadores aloquem uma reserva em sua fila tarefas e apenas responda a requisição do escalonador, quando existir possibilidade de execução. Os escalonadores enviam as tarefas aos m primeiros Trabalhadores que responderem e enviam uma negativa ao restante.

Os autores de [33] destacam que as aplicações podem ter características heterogêneas e, por isso, podem apresentar tarefas de longa duração e de curta duração. Tarefas de longa duração são muito sensíveis ao local de execução, pois em geral processam uma grande quantidade de dados e uma alocação ruim tem como consequência uma maior penalidade no acesso a esses dados. Isso se transforma em atrasos devido às latências de acessos a rede de interconexão e disco.

Esse trabalho mostra a análise de traços de carga de trabalho e constata que as tarefas de longa duração são menos frequentes, e que elas ocupam a maior parte do tempo de execução em proporção às tarefas de curta duração. Adicionalmente, os autores citam que é muito pouco provável que escalonadores distribuídos consigam achar Trabalhadores com baixa carga de tarefas ao acaso em uma situação de alta utilização do *cluster*. Por meio de simulações, eles mostram que alta carga aliada a presença de tarefas de longa duração são capazes de aumentar o tempo de espera das

tarefas de curta duração nas filas dos nós Trabalhadores, quando um escalonador distribuído com as características do Sparrow é utilizado.

Com base nessas motivações, esses autores propõe o Hawk [33], um escalonador híbrido, consistindo de um agente centralizado voltado para tarefas de longa duração e agentes distribuídos para as tarefas de curta duração. Além disso, em todos os nós Trabalhadores, existe uma fila de tarefas para acomodar aquelas que não podem ser imediatamente executadas. Para discriminar entre curto e longo, Hawk utiliza uma estimativa do tempo de execução das tarefas e baseado no histórico de execuções, utiliza um valor limite para classificação em um dos dois tipos.

Além de ser híbrido, essa proposta provê a reserva de uma porção de máquinas destinadas para execução apenas de tarefas de curta duração. Desse modo, ocorre que tarefas longas podem ser executadas em apenas um subconjuntos dos nós computacionais, ao passo que as curtas podem ser escalonadas em qualquer máquina. Essa partição tende a favorecer as tarefas pequenas, uma vez que, certamente, há filas de tarefas no conjunto de computadores sem presença de tarefas de longa duração atrasando a execução das mais curtas.

Outra medida adotada pelo Hawk é o roubo de tarefas para assegurar uma distribuição de carga equalizada entre os nós Trabalhadores. A escolha do nó a ser roubado é feita aleatoriamente. Em um ambiente com alta utilização, é muito provável que a fila de um Trabalhador escolhido ao acaso esteja sobrecarregada. O propósito maior do roubo de tarefas é tirar aquelas de curta duração que estão enfileiradas atrás daquelas de maior duração e levá-las para nós Trabalhadores da porção reservada do *cluster*. O roubo só é realizado por nós que se encontrem em estado ocioso e o nó ocioso adquire uma sequência de tarefas de curta duração que estiverem enfileiradas atrás de uma de longa duração na fila de um nó.

A adição de filas aos nós Trabalhadores é uma característica importante para evitar a ociosidade e manter os servidores sempre ocupados com trabalho útil a ser realizado. *Frameworks* de análise de grandes quantidades de dados como Hadoop e Apache Spark não implementam esse tipo de recurso, tornando as contribuições de Sparrow e Hawk bastante relevantes ao acrescentarem essa possibilidade.

Entretanto, a gerencia eficiente de filas é uma problema conhecidamente difícil. Por isso, o trabalho em [34] realiza um estudo de como técnicas de uso eficiente de filas pode contribuir no escalonamento de tarefas dos *Frameworks*. Os autores apresentam o Yaq-c e o Yaq-d, o primeiro é um escalonador centralizada e o segundo é a versão distribuída do primeiro. Vale salientar que mesmo sendo distribuído, o Yaq-d apresenta uma figura central, o Monitor de Uso, que retém informações importantes sobre o estado do *cluster* e possui papel importante na tomada de decisão dos escalonadores.

Para projetar o Yaq-c e Yaq-d, os autores experimentam adicionar filas de tarefas

em nós Trabalhadores ingenuamente com uma política FIFO de execução e medem o comportamento do tempo de término das execuções. Eles observam que não há beneficiamento para o escalonamento de tarefas nesse caso.

De fato, para alcançar um bom desempenho os autores abordam questões como o tamanho das filas, a atribuição de tarefas e o reordenamento das filas. O tamanho das filas segue duas abordagens. A primeira fixa um tamanho para as filas e analisa os possíveis valores para esse tamanho no caso do Yaq-c, baseando-se no intervalo de *heartbeat* dos nós Trabalhadores, na taxa de processamento das tarefas e na quantidade de tarefas que podem ser executadas simultaneamente. A segunda abordagem é viável apenas quando a duração das tarefas está disponível. Nesse caso, o tamanho da fila de uma máquina pode ser dinâmico, levando-se em conta qual a espera nessa fila e comparando esse valor com o tempo máximo de espera permitido para uma tarefa em fila no momento de atribuição de uma tarefa.

A atribuição de tarefas aos nós Trabalhadores ocorre também baseando-se nome tamanho das filas ou no tempo de espera das filas. Entretanto, caso existam recursos disponíveis, o escalonador escolhe o nó com recursos mais adequados para a tarefa. Para escolher uma fila para depositar uma tarefa, quando se é usado o tempo de espera das filas, o escalonador baseia-se em valores estimados calculados pelos nós Trabalhadores. Essa estimativa é realizada simulando os tempos de execução das tarefas nas filas junto com as correntemente sendo executadas.

O reordenamento das filas é um ativo importante do Yaq-c e Yaq-d. Ele substitui a política de execução FIFO por outra mais adequada, porém mantendo a execução das tarefas livre de *starvation*. As políticas implementadas são *Shortest Remaining Job First* (SRJF) que prioriza tarefas cujos *jobs* possuem o menor trabalho restante, *Least Remaining Tasks First* (LRTF) que prioriza tarefas cujos *jobs* possuem menor quantidade de tarefas restantes, *Shortest Task First* (STF) que ordena as tarefas priorizando aquelas de menor duração, *Earliest Job First*, e (EJF) que ordena as tarefas de acordo com o momento de submissão do *job* ao escalonador. Os autores indicam que SRJF foi a política que lidou melhor com o sistema proposto.

Capítulo 3

Método Proposto

Neste capítulo, descreveremos detalhadamente acerca do escalonador de tarefas distribuído para *Frameworks* de análise de grandes quantidades de dados implementado no Apache Spark.

3.1 Escalabilidade dos Escalonadores de Tarefas

A maioria dos *Frameworks* atuais utiliza uma abordagem centralizada para tratar o escalonamento de tarefas. De fato, com a centralização, diversas questões de projeto e implementação são simplificadas e há mais espaço para tratar outros problemas como a eficiência da atribuição de tarefas.

Além disso, o paralelismo nesses *Frameworks* é de grão grosso e, em geral, as próprias tarefas devem participar do processo de *shuffle* (Seção 2.3.1) que possui uma carga elevada de acesso a disco e rede de interconexão. Existe, ainda, o processo de serialização/desserialização (Seção 2.3.2) inerente às tarefas. Contabilizando todo o tempo despendido nesses processo, assumir que a sobrecarga do tempo de escalonamento de uma tarefa será ocultado é factível.

Entretanto, existe avanço nas propostas de melhorias nos *Frameworks* que diminuem os tempos dispendidos em processos ortogonais à computação das aplicações (Seção 2.3). Adicionalmente, é crescente a tendência de encurtar o tempo das tarefas, distribuindo mais a computação em cima dos dados, a fim de proporcionar uma melhoria no uso de recursos das máquinas [31]. Essas condições tornam necessário o estudo e progresso do escalonamento de tarefas nesses sistemas.

Mais ainda, é necessário acompanhar a evolução dos ambientes computacionais. Redes de interconexão mais rápidas e com maior banda e o uso de memórias não voláteis em *data centers* devem ser considerados. Outrossim, a disseminação do uso desses *Frameworks* em segmentos de computação de alto desempenho é uma realidade. É muito interessante, pois, que novas propostas de arquitetura para o escalonamento de tarefas sejam considerados, para lidar com aumento da capacidade

de processar atribuição de tarefas em recursos computacionais que favorecem um maior paralelismo como plataformas com maior número de processadores, exemplo *fat nodes*, GPUs, e Xeon Phis.

A dinâmica de execução das aplicações nos *Frameworks* é disparar as tarefas simultaneamente no *cluster* e ocupar os *slots* (núcleos de processadores) disponíveis. Em geral, o conjunto de dados é particionado de maneira balanceada entre as tarefas. Isso resulta em um tempo de execução aproximadamente igual para as tarefas, salvo uma parte de tarefas retardatárias. Com isso, o escalonador deve ser capaz de atribuir novas tarefas às unidades de processamento ociosos em alta velocidade, para manter a boa utilização dos recursos.

Um agente centralizado apresenta dificuldades para servir um grande número de requisições. Mesmo que a atividade seja paralelizada, como o escalonador deve manter o seu estado de informações sobre as execuções no conjunto de computadores consistente, um mecanismo de *lock* deve ser utilizado. Essa situação limita a escalabilidade do escalonador de tarefas.

Pode-se considerar o funcionamento do Apache Spark (Seção 2.2.2). No Apache Spark, não existe filas de tarefas nos nós Trabalhadores. Eles só recebem as tarefas cujos recursos podem ser alocados para execução.

Ao finalizar a execução de uma tarefa, o nó Trabalhador deve enviar uma mensagem ao Mestre relatando a conclusão da tarefa. Dessa maneira, o Mestre pode tomar conhecimento de que existem recursos disponíveis para a atribuição de novas tarefas. Após sair do Trabalhador a mensagem chega ao Mestre, onde ele a mantém em uma fila de mensagens provenientes dos Trabalhadores a serem processadas. Com o acréscimo de *slots* em uso, ou seja, conforme usa-se maiores quantidades de nós Trabalhadores e mais núcleos de processadores, mais a taxa de entrada de mensagens nessa fila aumenta, elevando o tempo de espera até o processamento de uma mensagem.

Somente após o processamento da mensagem enviada do nó Trabalhador, o Mestre considera atribuir uma nova tarefa no *slot* vago. Desse modo, um *slot* tende a permanecer ocioso por um período de tempo significativo, atrasando o andamento da aplicação e limitando a escalabilidade do uso de mais recursos computacionais.

Embora os escalonadores apresentados na Seção 2.4.2 proponham diversas maneiras para alcançarem alta escalabilidade, podemos fazer certas ressalvas acerca das propostas. O Sparrow quando posto em um cenário de alta utilização, pois apesar de realizar a vinculação tardia, a probabilidade de escolher nós com baixa carga é pequena. Além disso, quando as tarefas possuem restrição de localidade distintas, o Sparrow não consegue atuar em modo de amostra em lotes prejudicando seu desempenho.

O Hawk [33], o Yaq-c e Yaq-d [34] se baseiam em estimativas de tempo de

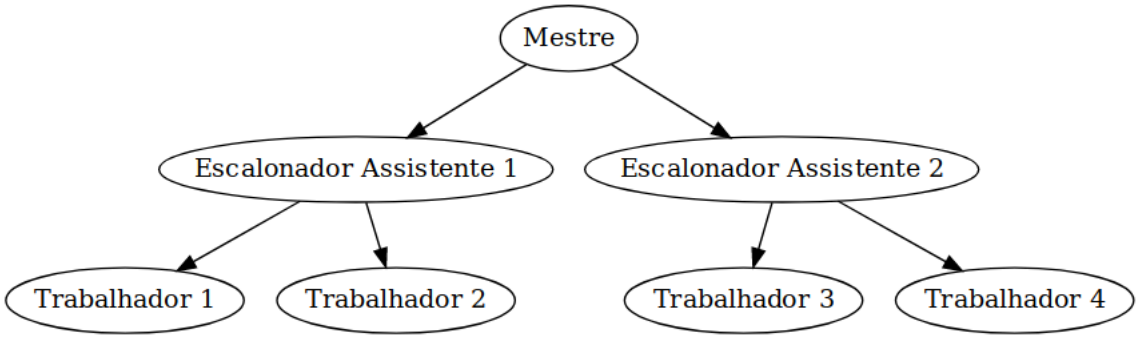


Figura 3.1: Hierarquia de escalonamento

execução das tarefas que são difíceis de se atingir em certos cenários. Essas estimativas podem resultar em valores longe dos reais em ambientes onde a carga de trabalho e a duração das tarefas sejam variados. Além disso, Yaq-c e Yaq-d apresentam alguns parâmetros que não são simples de serem ajustados, como a política de reordenamento das filas de tarefas nos nós Trabalhadores. Os *Frameworks* possuem uma gama de configurações muito grande. A dificuldade de obter a configuração mais adequada às necessidades das aplicações é tanta que existem trabalhos que tratam desse assunto [35–37].

Em [38], os autores abordam uma metodologia sistemática para ajuste de parâmetros. O trabalho investiga o impacto das parâmetros relacionados a *shuffle*, serialização e compressão no desempenho de aplicações no Apache Spark. A metodologia inclui escolha de aplicações representativas para *benchmark*, teste de valores de parâmetros, entre outros passos, e culminam em uma matriz de adjacências que correlaciona a relação dos diversos parâmetros testados e um conjunto de configurações candidatas a promoverem o melhor desempenho.

3.2 Escalonadores Assistentes

Consideramos uma forma alternativa para o escalonamento distribuído de tarefas nos *Frameworks* de análise de grandes quantidades de dados. Com essa proposta, o escalonamento ocorre hierarquicamente, onde a raiz da árvore hierárquica é o Mestre e as folhas são os nós Trabalhadores. Chamamos os nós intermediários de escalonadores assistentes. A Figura 3.1 ilustra um exemplo dessa alternativa esquematicamente.

Os escalonadores assistentes possibilitam a distribuição da carga do escalonamento de tarefas, aliviando o trabalho do Mestre. Existem diversas alternativas de utilizarmos os escalonadores assistentes. Uma delas é fazermos com que os escalonadores assistentes sejam réplicas do Mestre e gerenciem todo o escalonamento de tarefas dos nós Trabalhadores que sejam seus filhos na árvore hierárquica. Isso

inclui a atribuição e gerência do estado das tarefas. Um outro caminho é deixarmos os escalonadores assistentes tomarem conta apenas da atribuição de tarefas aos seus Trabalhadores, ficando a gerência das tarefas centralizada no Mestre.

Neste trabalho, utilizamos a segunda abordagem. Ela simplifica os escalonadores assistentes, facilitando sua implementação, ao mesmo tempo que distribui a carga de trabalho de escalonamento do Mestre. Outrossim, essa abordagem já possibilita evitarmos a ociosidade dos nós Trabalhadores, uma vez que novas tarefas podem ser buscadas nos escalonadores assistentes ao mesmo tempo que um estado de finalização da tarefa é comunicado ao nó Mestre que gerencia os estados das tarefas.

Escalonadores distribuídos como o Sparrow, o Yaq-c e o Yaq-d (Seção 2.4.2, evitam a ociosidade dos Trabalhadores ao inserirem filas de tarefas nesses nós. O Yaq utiliza vinculação precoce de tarefas em Trabalhadores e utiliza técnicas de gerenciamento de fila como reordenamento para impedir desbalanceamento de carga, ao passo que o Sparrow utiliza um mecanismo de vinculação tardia para restringir o mesmo problema, mas utiliza uma quantidade determinada de nós aleatoriamente escolhidos. Com o emprego dos escalonadores assistentes, a fila de tarefas está em um nível superior, servindo um subconjunto de nós Trabalhadores. Isso proporciona a vinculação tardia de tarefas em nós com maior afinidade para as tarefas presentes na fila.

Podemos notar que há a possibilidade de vários níveis de escalonadores assistentes. Esses níveis podem corresponder a especializações mais refinadas do algoritmo de atribuição de tarefas. Entretanto, neste momento, apenas consideramos um nível de escalonadores assistentes. Dessa maneira, podemos considerar o *cluster* particionado entre os escalonadores assistentes.

O esquema de particionamento pode levar em consideração diversos fatores que beneficiem o escalonamento das tarefas. Por exemplo, em [24], os autores mostram que tentar manter as tarefas o mais próximo possível de seus dados pode aprimorar o tempo de execução das aplicações. Desse modo, o trabalho estabelece os níveis de localidade, do mais local ao menos local, local ao nó, local ao *rack* e fora do *rack*. Esses níveis consideram que um *cluster* é composto por diversos *racks* de computadores, sendo a largura de banda da rede muito maior entre os computadores de um *rack* do que entre computadores de *racks* diferentes.

Dessa maneira podemos ter o esquema de particionamento ilustrado na Figura 3.2, onde cada *rack* possui um escalonador assistente responsável por atribuir tarefas aos nós posicionados naquele *rack*.

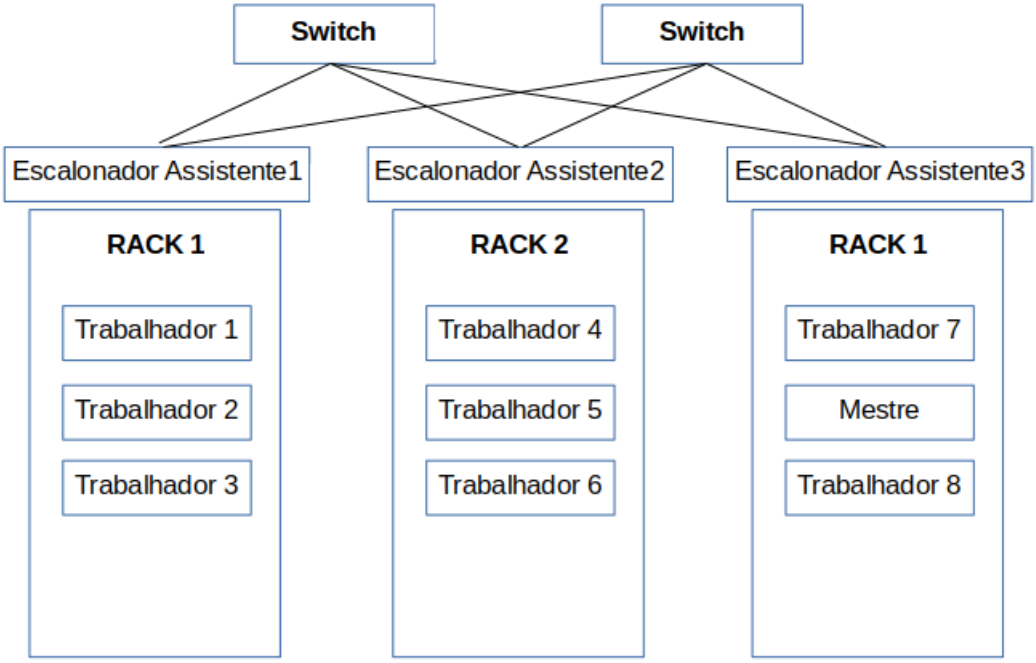


Figura 3.2: Particionamento por *rack*

3.3 Implementação no Apache Spark

Neste trabalho, implementamos os escalonadores assistentes no Apache Spark. O código do Apache Spark é escrito na linguagem Scala e está disponível em [39]. Modificamos a versão 1.6.0 para incluir a implementação dos escalonadores assistentes.

O Apache Spark é um *Framework* versátil, capaz de ser executado em escalonadores de recursos como o Mesos (2.4.1) e o YARN [15]. Além disso, ele conta com seu próprio gerenciador de *cluster*, sendo esta a maneira mais simples de por o Apache Spark em operação. Essa última opção é ilustrada na Figura 2.2.

No gerenciador de *cluster* existe um agente central denominado *Master* e o resto das máquinas possuem um agente chamado de *Worker*. No momento de execução, toda a aplicação cria uma entidade central chamada de *Driver*. O *Driver* entra em contato com o *Master* para pedir recursos nos Workers. O *Master*, então, cria os *Executors*, processos nos nós *Worker* que executam as tarefas. O *Driver* cumpre o papel de escalonador de tarefas centralizado no Apache Spark.

Existem diversas maneiras de inserirmos os escalonadores assistentes no Spark. O *Driver* poderia requisitar que alguns Workers inicializassem escalonadores assistentes ao invés de *Executors*. Outra opção seria deixar os assistentes serem parte do *cluster* do Spark junto com o *Master* e os Workers. Assim, os *Drivers* das aplicações apenas precisam tomar conhecimento que eles estão presentes para poder usá-los. Implementamos nossa prova de conceito com essa segunda opção neste trabalho. No contexto do Apache Spark, chamamos os escalonadores assistentes de *SchedulerAs-*

sistants.

Inicialização dos *SchedulerAssistants*: Como os *SchedulerAssistants* fazem parte do *cluster* do Spark, todas as aplicações que estão sendo executadas no *cluster* os compartilham. Assim, como inicializamos manualmente o *Master* e os *Workers* do *cluster* do Spark, também precisamos realizar o mesmo processo para inicializamos os *SchedulerAssistants*. Para facilitar esse processo o Spark fornece o *script* `start-all.sh`. Desse modo, tivemos que modificar esse *script* para incluir a inicialização dos *SchedulerAssistants*, acrescentando uma chamada ao *script* `start-schedulerAssistants.sh`.

A inicialização dos componentes do *cluster* do Spark é realizada através de chamadas `ssh` que criam processos em máquinas remotas, sendo o *Master* criado na máquina local em vias gerais. A especificação das máquinas onde são inicializados os *Workers* acontece através do arquivo `$SPARK_HOME/conf/slaves`. De forma similar, precisamos inserir no arquivo `$SPARK_HOME/conf/schedulerAssistants` as máquinas em que se deseja inicializar os *SchedulerAssistants*.

Ao serem inicializados, os *SchedulerAssistants* devem se apresentar ao *Master*, para que o mesmo mantenha o estado do *cluster* consistente.

Associação do *Driver* com os *SchedulerAssistants*: Toda a aplicação executada no *cluster* do Spark deve se registrar com o *Master*. O *Driver* se associa com o *Master*, registrando a aplicação. Nesse processo de associação existe uma troca de mensagens entre o *Driver* e *Master*, de maneira que o *Master* responde a mensagem de registro de aplicação com as informações sobre todos os *SchedulerAssistants* presentes no *cluster*. Com isso, o *Driver* mantém uma lista com todos eles.

Internamente, optamos por manter uma tabela *hash* indexada pelos *ids* dos *SchedulerAssistants*. Os valores armazenados correspondem a objetos com informações que visam a realização da comunicação entre o *Driver* e os *SchedulerAssistants*.

A comunicação entre os agentes distribuídos no Apache Spark ocorre por meio de *Remote Procedure Calls* (RPCs). Para enviarmos uma mensagem a um processo remoto precisamos de um objeto do tipo `RpcEndpointRef`. A criação desse objeto é custosa, pois envolve a comunicação entre as partes, realizando um *handshake*. A associação entre o *Driver* e os *SchedulerAssistants* envolve criar um `RpcEndpointRef` por *SchedulerAssistant*. Deixamos o desenvolvimento de melhores maneiras de executar esse processo para trabalhos futuros.

Associação de *Executor* com seu *SchedulerAssistant*: Cada *SchedulerAssistant* tem um identificador. Esse identificador é passado como argumento da linha de comando, quando o inicializamos. A partir desse identificador são feitas as associações entre um *Executor* e um *SchedulerAssistant*. Prover as informações para essa associação é papel do *Master*, pois é ele que lança os *Executors* no *cluster*, sendo inicializados localmente pelos *Workers*.

O *Master* envia as informações do *SchedulerAssistant* para o *Worker* na mensagem `LaunchExecutor`. De posse disso, o *Worker* inicia o processo do *Executor* passando os atributos de nome da máquina, porta e identificador do *SchedulerAssistant* como argumentos de linha de comando.

O *Executor*, uma vez iniciado, monta o um objeto chamado *SchedulerAssistantInfo* com as informações de seu *SchedulerAssistant*. A princípio o *Worker* envia para o *Master*, no momento de seu registro, qual é o *SchedulerAssistant* que lhe foi associado, pois nossa implementação faz uma associação estática entre um *Worker* e um *SchedulerAssistant* de maneira que essa associação seja utilizada nos *Executors*.

Entretanto, como o *Master* mantém o estado do *cluster* ele deve decidir a associação entre *Executors* e um *SchedulerAssistants*. Além disso, é desejável que haja uma maneira dinâmica de realizar essa associação, mas esse é um problema a ser tratado futuramente.

A associação estática corrente é realizada pelo usuário no momento de levantar o *cluster* do Spark. O usuário deve determinar identificadores para os *SchedulerAssistants* e informá-los no arquivo `$SPARK_HOME/conf/schedulerAssistants` ao lado do nome da máquina onde se deseja executar um *SchedulerAssistant*. Logo a seguir, exemplificamos o conteúdo de um arquivo `$SPARK_HOME/conf/schedulerAssistants`:

```
maquina1,schedulerAssistant1
maquina10,schedulerAssistant2
```

onde maquina1 e maquina10 são os nomes das máquinas que receberam um processo de *SchedulerAssistant* e schedulerAssistant1 e schedulerAssistant2 seus respectivos identificadores.

Do mesmo modo, a associação estática dos *Workers* com os *SchedulerAssistants* ocorre através do arquivo `$SPARK_HOME/conf/slaves`, da mesma maneira que fazemos no arquivo `$SPARK_HOME/conf/schedulerAssistants`. Devemos informar o nome da máquina a receber um processo *Worker* e o seu respectivo *SchedulerAssistant*. Logo a seguir, exemplificamos o conteúdo de um arquivo `$SPARK_HOME/conf/slaves`:

```
maquina1,schedulerAssistant1
maquina2,schedulerAssistant1
maquina10,schedulerAssistant2
maquina11,schedulerAssistant2
```

onde maquina1, maquina2, maquina10 e maquina11 são os nomes das máquinas que receberam um processo de *Worker* e schedulerAssistant1 e schedulerAssistant2 os identificadores dos seus *SchedulerAssistants* associados. Repare que temos *SchedulerAssistants* sendo executados na mesma máquina que um *Worker*.

Escalonamento de Tarefas - *Driver* para *SchedulerAssistants*: O *Driver* registra os *SchedulerAssistants* da mesma forma que registra os *Executors*. Entretanto, o *Driver* marca essa informação, indicando se tratar de um *SchedulerAssistant*.

Modificamos pouco o código que trata do escalonamento de tarefas no *Driver*. Deixamos ele escalonar as tarefas normalmente para os *Executors*, mas fazemos um filtro nos *Executors* que estão aptos para escalonamento, de maneira a apenas permitir que aqueles marcados como *SchedulerAssistants* passem no filtro. Assim, as tarefas são escalonadas apenas para *SchedulerAssistants*.

O algortimo de escalonamento de tarefas no Spark leva em consideração a localidade dos dados, definindo os níveis `PROCESS_LOCAL`, `NODE_LOCAL` `RACK_LOCAL` e `NO_PREF`. O nível `PROCESS_LOCAL` indica a localidade em *Executors*, pois o Spark permite que eles realizem o cache dos dados em memória. O `NODE_LOCAL` indica localidade na máquina e é utilizado principalmente com o Hadoop Distributed File System (HDFS) que realiza a replicação dos seus dados por diversos nós. O `RACK_LOCAL` indica que o dado está no mesmo *rack* de computadores da máquina em que a tarefa é executada. O nível `NO_PREF` indica ausência de preferência por localidade.

A preferência de execução ocorre nesta ordem: `PROCESS_LOCAL`, `NODE_LOCAL`, `RACK_LOCAL` e `NO_PREF`. Utilizamos o nível `RACK_LOCAL` para representar a preferência por *SchedulerAssistant*. Isso quer dizer que ao chegar nesse nível, a tarefa será encaminhada para o *SchedulerAssistant* do *Executor* ou máquina no qual a tarefa tem preferência de execução.

Modificamos o nível inicial que o *Driver* utiliza na realização do escalonamento para o valor `RACK_LOCAL`, pois não precisamos considerar as localidade `PROCESS_LOCAL` e `NODE_LOCAL` já que apenas *SchedulerAssistants* estão sendo utilizados no escalonamento.

Criamos também um novo tipo de mensagem para enviar tarefas do *Driver* para os *SchedulerAssistants*. Esse novo tipo contém a tarefa serializada, da mesma forma que seria enviada aos *Executors*, e a preferência de localidade da tarefa, para que os *SchedulerAssistants* decidam qual a melhor forma de escalonar a tarefa dentre seus *Executors* associados.

Ao chegar no *SchedulerAssistant*, a mensagem é desmembrada e a preferência de localidade da tarefa é analisada. O *SchedulerAssistant* possui filas de tarefas para cada executor associado, para cada nó associado, para tarefas sem preferência e uma com todas as tarefas. Assim, o *SchedulerAssistant* adiciona a tarefa na fila correspondente a essa localidade. Se a preferência for `PROCESS_LOCAL`, também adiciona na fila correspondente à máquina do *Executor* preferido. Por fim, adicionamos todas as tarefas na fila de todas as tarefas.

Escalonamento de Tarefas - *SchedulerAssistants* para *Executors*: Ao chegar uma requisição de tarefa de um *Executor*, o *SchedulerAssistant* deve verficar se há tarefas disponíveis. Caso haja, ocorre a escolha da tarefa nas filas de tarefas do *SchedulerAssistant*. Caso contrário, o *SchedulerAssistant* guarda o pedido do *Executor* e o libera assim que houver tarefas disponíveis.

Já que, na requisição de tarefas, o *Executor* deve enviar sua disponibilidade para novas tarefas, isto é, a quantidade de unidades de processamento livres disponíveis (número de núcleos disponíveis no caso do Spark), é possível o cenário em que não haja tarefas para suprir toda a disponibilidade do *Executor*. Nesse caso, o *SchedulerAssistant* também armazena o pedido e o libera ao haver tarefas disponíveis.

O escalonamento das tarefas realiza uma correspondência entre o *Executor* e a preferência de localidade de execução das tarefas. A ideia é que a requisição de um *Executor* por uma nova tarefa seja respondida por uma tarefa com preferência para esse *Executor* requisitante. Caso inexista essa opção, procura-se por uma com preferência para mesmo nó do *Executor*. A terceira opção, no caso da falta de sucesso das duas primeiras, é buscar alguma tarefa que não tenha preferência de localidade e, por fim, por alguma cuja preferência seja para um executor ou um nó que não é o requisitante corrente.

Capítulo 4

Experimentos e Resultados

Neste capítulo discutiremos sobre a realização de experimentos que visam medir o desempenho da nossa implementação de escalonamento distribuído hierárquico utilizando os escalonadores assistentes no Apache Spark.

Iniciamos com a descrição do ambiente usado nos experimentos e abordagens utilizadas para tornar o ambiente propício para os experimentos executados (Seção 4.1). Em seguida, descrevemos o planejamento e execução dos experimentos (Seção 4.2). Apresentamos e discutimos os resultados que obtivemos na Seção 4.3. Finalmente, como os resultados mostraram um comportamento diferente do que esperávamos, realizamos uma investigação mais profunda e uma análise mais detalhada da execução experimental na Seção 4.4.

4.1 Ambiente Experimental

Nossos experimentos utilizaram o supercomputador Lobo Carneiro da Universidade Federal do Rio de Janeiro para sua execução. O Lobo Carneiro é o sistema computacional mais poderoso instalado em uma instituição de ensino superior no Brasil em 2018.

Esse supercomputador conta com duzentos e cinquenta e duas máquinas de dois processadores Intel Xeon E5-2670 v3 com 64 GB de memória cada. Esses nós computacionais estão todos interligados através da rede de interconexão Infiniband FDR com largura de banda de 56 Gbs. Além disso, as máquinas estão conectadas ao sistema de arquivos de alto desempenho Lustre como armazenamento.

Os ambientes de computação de alto desempenho se tornaram uma opção para a execução de cargas de trabalho em *Frameworks* de análise de grandes quantidades de dados. Muitos dos sistemas de alto desempenho encontram-se em grandes centros de pesquisas ou em universidades e são abertos a projetos da comunidade científica.

Existem trabalhos que estudam a utilização dos *Frameworks* nesses ambientes [40, 41] . Alguns dos problemas avaliados são a interação do sistema de análise de

dados com o gerenciador de recursos do supercomputador como o PBS [23] e o Slurm [22]. Além disso, ambientes de computação de alto desempenho possuem certas características que não foram consideradas na concepção dos *Frameworks* de análise de grandes quantidades de dados. Uma das principais é a falta de armazenamento local. Uma vez que os *Frameworks* podem vir a utilizar bastante esse dispositivo, como no caso da fase de *shuffle*, ter que acessar a rede de interconexão para ir ao armazenamento pode se tornar uma desafio.

Com a finalidade da execução dos testes no Lobo Carneiro, elaboramos uma integração entre o *cluster* Spark e o gerenciador de recursos do *cluster* do Lobo Carneiro, o PBS. Sempre pedimos máquinas para uso exclusivo. O PBS no Lobo Carneiro está configurado para dividir o uso de um nó entre diversas requisições baseado em suas demandas por recursos. Utilizar uma máquina completa é a melhor maneira de termos o maior número de tarefas possível para nossa aplicação sendo executadas simultaneamente e estressando o escalonador.

Para determinarmos os nós em que inicializamos escalonadores assistentes, escrevemos um *script* em Python que, a partir da lista com os nomes das máquinas alocadas pelo PBS para nossa execução, divide as máquinas alocadas em uma quantidade de grupos de forma equilibrar o número de *Workers* associado por escalonador assistente. A quantidade de escalonadores assistentes é um parâmetro do *script*. Dentro de cada grupo, um escalonador assistente é eleito ao acaso, escrevendo o nome das máquinas eleitas no arquivo `$SPARK_HOME/conf/schedulerAssistants`.

O nó em que o *Master* é executado não participa desse procedimento e é nele que executamos o *script* `$SPARK_HOME/sbin/start-all.sh` para inicializarmos o *cluster* do Apache Spark. Um problema que ocorre nesse cenário é tentarmos executar uma aplicação logo após chamarmos esse *script*. Ao fazermos isso, corremos o risco de executarmos a aplicação sem que o *cluster* do Spark esteja completamente estabelecido.

O Spark estabelece seu *cluster* de maneira assíncrona e sequencial por comandos `ssh`. Quando o *script* `$SPARK_HOME/sbin/start-all.sh` retorna apenas se sabe que esses comandos foram realizados nas máquinas remotas. Entretanto, o *Master* só toma conhecimento dos *Workers* após receber uma mensagem deles, não podendo, portanto, permitir que sejam criados *Executors* neles antes disso. Projetamos os *SchedulerAssistants* da mesma maneira.

O principal problema é, portanto, termos a aplicação executando em menos recursos do que se planejou por um período de tempo que depende de diversos aspectos do sistema do Lobo Carneiro e de suas máquinas. Ressaltamos que a inicialização do *Master* é o único requisito do Spark para executar as aplicações, mas apenas quando houver ao menos um *Worker* é que ele pode alocar *Executors* para a aplicação. Isso quer dizer que podemos ter menos tarefas sendo executadas simultaneamente, a me-

dida que o estabelecimento do *cluster* ocorre, causando variações em medições de tempo de execução importantes para os nossos experimentos.

Para resolver esse problema, elaboramos um *script* em Python para promover a sincronização do estabelecimento do *cluster* do Apache Spark. Esse *script* retorna apenas quando se certifica que todos os componentes do *cluster* - *Master*, *Workers* e *SchedulerAssistants* - estão devidamente inicializados. Ele toma conhecimento dos *Workers* pelo arquivo `$SPARK_HOME/conf/slaves` e dos *SchedulerAssistants* pelo arquivo `$SPARK_HOME/conf/schedAssists`. Assim, o *script* espera a comunicação do *Master*, relatando quais componentes já foram completamente inicializados. Logo, o *Master* teve que ser modificado para realizar essa comunicação no momento em que recebe uma mensagem de registros dos componentes presentes no *cluster*. A comunicação é realizada através de *Named Pipes* [42], onde o *script* atua apenas como leitor e o *Master* atua apenas como escritor.

4.2 Descrição Experimental

O Apache Spark, internamente, divide uma aplicação em diversos *jobs*, que são divididos em estágios cujas fronteiras determinam a necessidade de comunicação entre os *Executors*. Os estágios são compostos pelas tarefas.

Os experimentos realizam a comparação entre a nossa proposta e o escalonamento de tarefas original do Apache Spark. Para realização dos experimentos, utilizamos um *microbenchmark* cujo objetivo é prover o maior entendimento no desempenho comparativo entre a proposta de distribuição hierárquica com escalonadores assistentes e o método original de atribuição de tarefas do Apache Spark.

O *microbenchmark* consiste no lançamento de *jobs* com tarefas de tempo fixo de duração. Os *jobs* são seriais, de maneira que apenas as tarefas de um *job* são executadas em um determinado instante pelo Apache Spark.

Desse modo, desenvolvemos o *microbenchmark* de maneira que possamos definir os parâmetros número de *jobs*, número de tarefas por *job*, e duração das tarefas. Os *jobs* desse *microbenchmark* possuem apenas um estágio. Assim, não realizamos comunicação entre os *Executors*. Isso permite excluirmos os efeitos da troca de dados pela rede de interconexão no tempo de duração dos *jobs* e da aplicação, permitindo predominante influência do escalonamento de tarefas no tempo de execução.

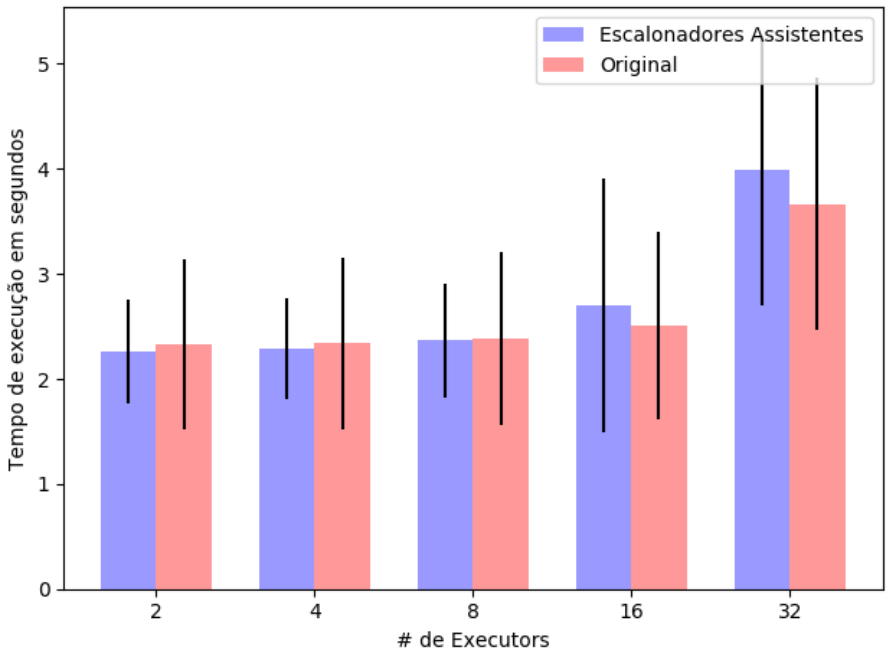


Figura 4.1: Duração das tarefas é 200 milissegundos. Média dos tempos de execução de todos os *jobs* produzidos pelo Spark para o nosso cenário de teste.

Formamos o nosso cenário de testes com a utilização do *microbenchmark* estabelecendo os seguintes parâmetros:

Cenário de teste:

- Duração das tarefas: 200 milissegundos, 1 segundo, 10 segundos.
- Número de tarefas por *job*: 5× (cinco vezes) o número de processadores disponíveis nos *Executors*.
- Número de *jobs*: 10.
- Número de nós: 2, 4, 8, 16, 32.

A escolha no número de tarefas como $k\times$ o número de processadores disponíveis reflete a sugestão dos manuais de Frameworks em se estabelecer esse parâmetro nas aplicações [43, 44].

4.3 Resultados e Discussão

As Figuras 4.1, 4.2 e 4.3 mostram a média dos tempos de execução de todos os *jobs* executados pelo Apache Spark para o nosso *microbenchmark* com o referido cenário de teste para tarefas com duração de 200 milissegundos, 1 segundo e 10 segundos respectivamente.

Podemos notar que a versão original e a versão com os Escalonadores Assistentes do Apache Spark apresentam valores semelhantes para o tempo médio de execução

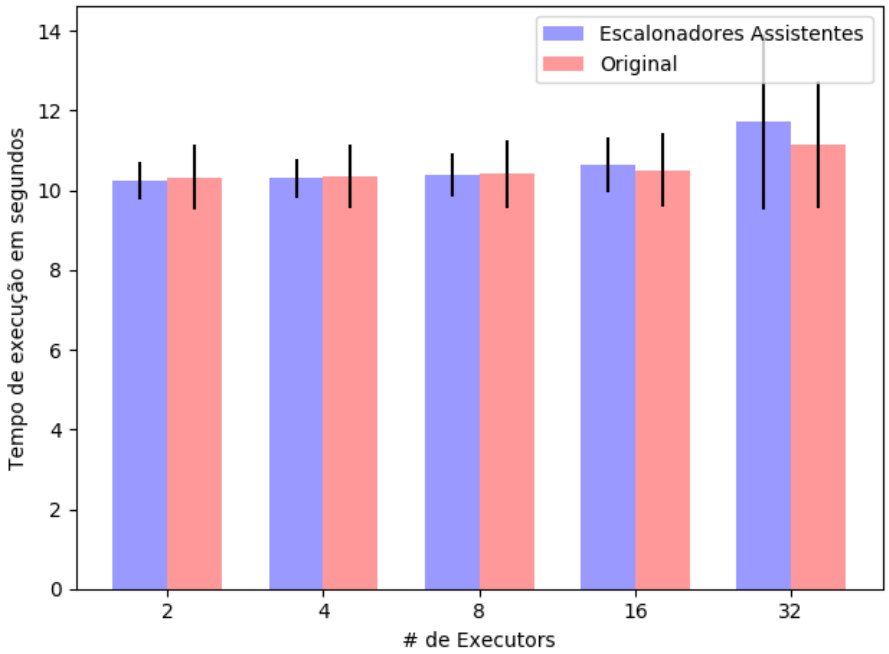


Figura 4.2: Duração das tarefas é 1 segundo. Média dos tempos de execução de todos os *jobs* produzidos pelo Spark para o nosso cenário de teste.

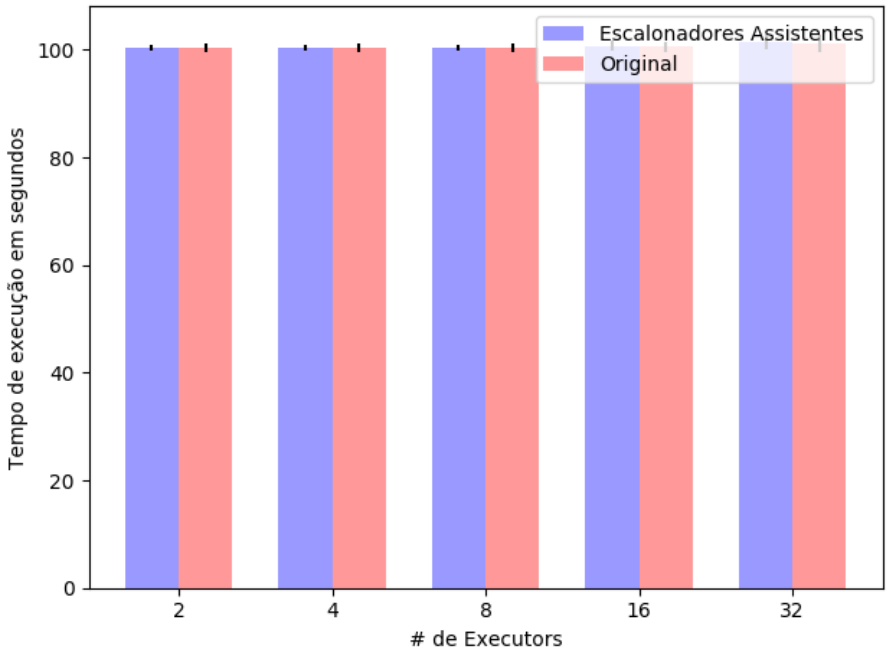


Figura 4.3: Duração das tarefas é 10 segundos. Média dos tempos de execução de todos os *jobs* produzidos pelo Spark para o nosso cenário de teste.

	Número de *Executors*				
	2	4	8	16	32
Original	2.32 ± 0.81	2.34 ± 0.81	2.38 ± 0.82	2.50 ± 0.89	3.66 ± 1.2
Escalonadores Assistentes	2.26 ± 0.5	2.29 ± 0.48	2.37 ± 0.54	2.70 ± 1.2	3.98 ± 1.3

Tabela 4.1: Média dos tempos de execução de todos os *jobs* produzidos pelo Spark para o cenário de teste com tempo de execução de 200 milissegundos para as tarefas.

	Número de *Executors*				
	2	4	8	16	32
Original	10.33 ± 0.81	10.34 ± 0.8	10.41 ± 0.84	10.51 ± 0.92	11.15 ± 1.58
Escalonadores Assistentes	10.24 ± 0.5	10.3 ± 0.48	10.38 ± 0.54	10.64 ± 1.2	11.71 ± 1.28

Tabela 4.2: Média dos tempos de execução de todos os *jobs* produzidos pelo Spark para o cenário de teste com tempo de execução de 1 segundo para as tarefas.

de um *job* em todos os casos do cenário de teste. Com relação ao aumento do número de *Executors*, as Figuras mostram também que o tempo de execução dos *jobs* apenas mostrou aumento, quando 32 *Executors* participaram da computação.

Esses resultados mostram que apesar de a implementação da nossa proposta inserir um salto intermediário no caminho das tarefas do *Driver* até os *Executors*, não se nota indícios de sobrecarga adicional ao sistema de escalonamento.

No caso de um sistema ideal, um *job* do Apache Spark deveria executar com duração de cinco segundos no cenário de teste exposto para esse *microbenchmark*, pois considerando um escalonador ótimo e tarefas executando em paralelo com duração de um segundo, lançaríamos cinco rodadas de tarefas que ocupariam todas unidades de processamento disponíveis.

As Tabelas 4.1, 4.2 e 4.3 mostram os valores representados na Figura 4.1, 4.2 e 4.3 respectivamente. Essas Tabelas mostram que com um número de nós executando tarefas reduzido, as duas versões do Apache Spark se aproximam do ótimo. Porém, a medida que mais unidades de processamento são colocadas a disposição do escalonador, um atraso na duração dos *jobs* é notada.

4.4 Investigação da Execução Experimental

Os resultados da Seção 4.3 nos levaram a investigar mais profundamente as execuções do *microbenchmark* no cenário de teste proposto com o Apache Spark original e a

	Número de *Executors*				
	2	4	8	16	32
Original	100.33 ± 0.83	100.38 ± 0.87	100.37 ± 0.79	100.54 ± 0.93	101.12 ± 1.54
Escalonadores Assistentes	100.28 ± 0.52	100.28 ± 0.48	100.35 ± 0.49	100.68 ± 0.98	101.44 ± 1.41

Tabela 4.3: Média dos tempos de execução de todos os *jobs* produzidos pelo Spark para o cenário de teste com tempo de execução de 10 segundos para as tarefas.

versão de prova de conceito com a introdução dos escalonadores assistentes.

Assim, investigamos o comportamento da taxa de ocupação das *Central Processing Units* (CPUs) dos *Executors* por tarefas durante as execuções experimentais do *microbenchmark*. Para tanto, realizamos o rastreamento da quantidade de tarefas sendo executadas sempre que um *Executor* recebe uma nova tarefa do sistema de escalonamento.

A taxa de ocupação, então, reflete a capacidade de entrega de tarefas as unidades de processamento do escalonador. Um dos principais objetivos de um escalonador é manter esta taxa elevada, resultando na baixa ociosidade do sistema.

O emprego dos Escalonadores Assistentes no Apache Spark possui como meta melhorar o desempenho desta taxa visto que a centralização do escalonamento de tarefas no processo *Driver* tende a tornar-se um gargalo na eficiência da distribuição de tarefas a medida que mais unidades de processamento são colocadas a disposição do Spark.

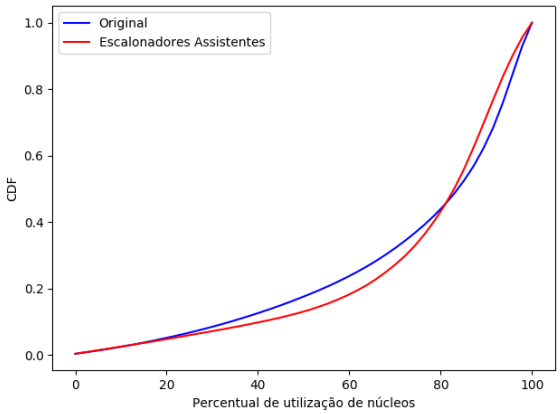
As Figuras 4.4, 4.5, 4.6, 4.7 e 4.8 mostram a distribuição do percentual de utilização das CPUs das máquinas com *Executors* nos casos com 2, 4, 8, 16 e 32 *Executors* respectivamente. Essas Figuras estão organizadas pela duração das tarefas utilizada nas execuções.

Podemos notar que as curvas da versão com Escalonadores Assistentes e da versão Original do Apache Spark, nas Figuras 4.4, 4.5 e 4.6 apresentam duas regiões de discrepância. A primeira região, aquela com valores menores que 80 no eixo horizontal, mostra que a versão com Escalonadores Assistentes conseguiu ficar uma menor porção do tempo com taxa de uso de CPU abaixo de 80 % em comparação à versão Original. Entretanto, a diferença entre as duas versões é pequena chegando, no máximo, a aproximadamente 5 %.

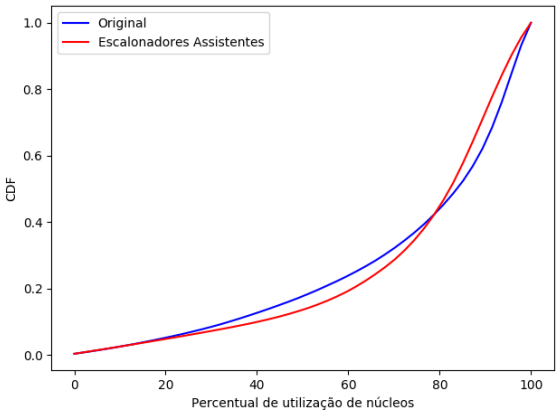
A segunda região está relacionada à sobrecarga adicional que a introdução dos Escalonadores Assistentes insere no escalonamento de tarefas no início da execução, quando os *Executors* ainda estão se associando ao *Driver* e conhecendo seus Escalonadores Assistentes. Além disso, nos momentos iniciais da execução os Escalonadores Assistentes podem não ter recebido tarefas suficientes do *Driver* para manter a taxa de uso de CPU dos *Executors* plena. A diferença entre as duas curvas nessa região apresenta valores em torno de 10 % em média no máximo.

As Figuras 4.7 e 4.8 mostram que, a medida que aumentamos o número de *Executors*, os efeitos da sobrecarga da introdução dos Escalonadores Assistentes desaparece. Mais ainda, podemos notar um aumento na diferença entre as duas curvas, chegando a um máximo de 10 % em média. Isso demonstra que o emprego dos Escalonadores Assistentes traz benefícios em relação à escalabilidade da utilização de recursos das unidades computacionais disponíveis no Apache Spark.

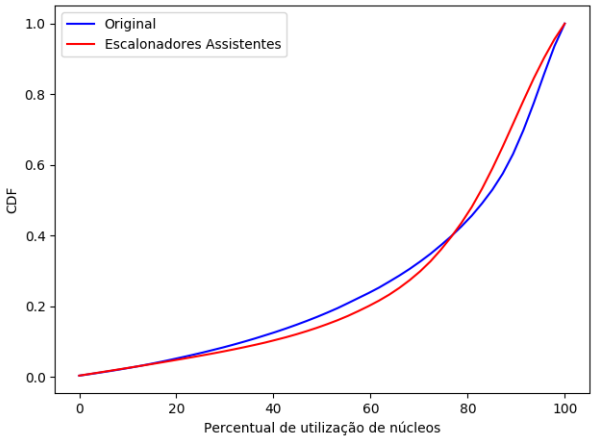
Ao constatarmos o bom desempenho dos Escalonadores Assistentes em relação à



(a) CDF do uso de CPU das máquinas com *Executors* no caso com 2 *Executors* para tarefas com duração de 200 milissegundos.



(b) CDF do uso de CPU das máquinas com *Executors* no caso com 2 *Executors* para tarefas com duração de 1 segundo.



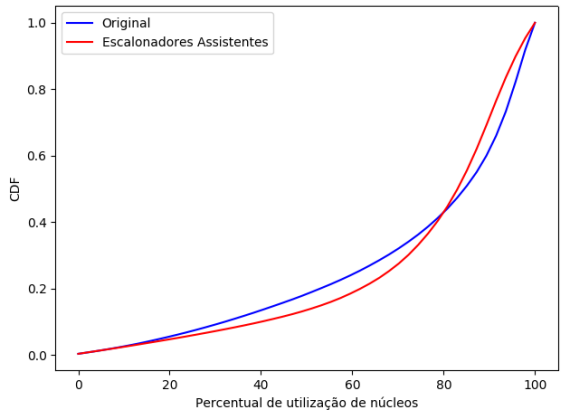
(c) CDF do uso de CPU das máquinas com *Executors* no caso com 2 *Executors* para tarefas com duração de 10 segundos.

Figura 4.4: CDF do uso de CPU das máquinas com *Executors* no caso com 2 *Executors* para tarefas com duração de 200 milissegundos, 1 segundo e 10 segundos.

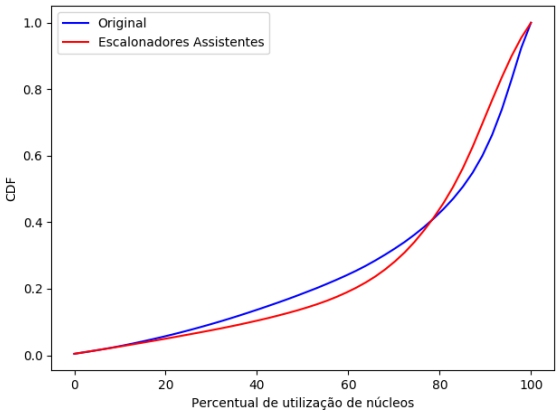
ocupação de CPUs nos nós com *EXecutors*, devemos entender o motivo de não haver ganhos no tempo de execução dos *jobs* como sugerem os resultados das Figuras 4.1, 4.2 e 4.3.

Desse modo, investigamos o comportamento dos *Executors* durante uma execução do *microbenchmark* no Apache Spark com Escalonadores Assistentes. As Figuras 4.9 e 4.10 mostram o perfil de ocupação das CPUs de máquinas com *Executors* para o caso de tarefas com duração de 1 segundo utilizando um total de 16 *Executors* ao longo do tempo decorrido desde o início até o final da execução dos 10 *jobs* da aplicação de *microbenchmark*.

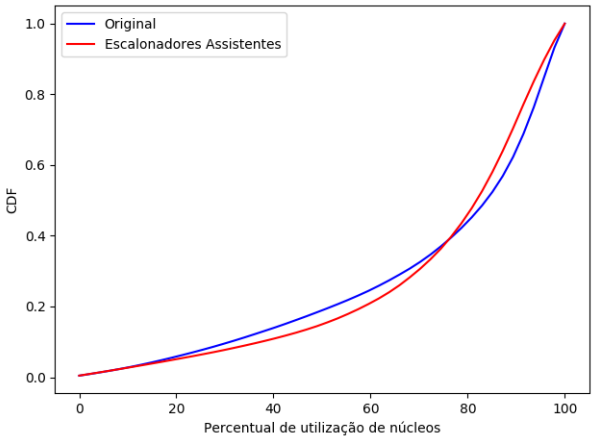
Nessas Figuras, notamos que todos os 16 *Executors* apresentam um padrão para os valores de taxa de uso de CPUs. Nesse padrão, a taxa de uso apresenta valor igual a zero, posteriormente, cresce, apresentando valores mais elevados na maior



(a) CDF do uso de CPU das máquinas com *Executors* no caso com 4 *Executors* para tarefas com duração de 200 milissegundos.



(b) CDF do uso de CPU das máquinas com *Executors* no caso com 4 *Executors* para tarefas com duração de 1 segundo.



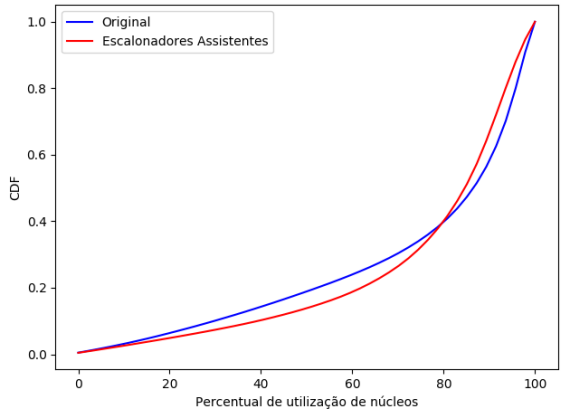
(c) CDF do uso de CPU das máquinas com *Executors* no caso com 4 *Executors* para tarefas com duração de 10 segundos.

Figura 4.5: CDF do uso de CPU das máquinas com *Executors* no caso com 4 *Executors* para tarefas com duração de 200 milissegundos, 1 segundo e 10 segundos.

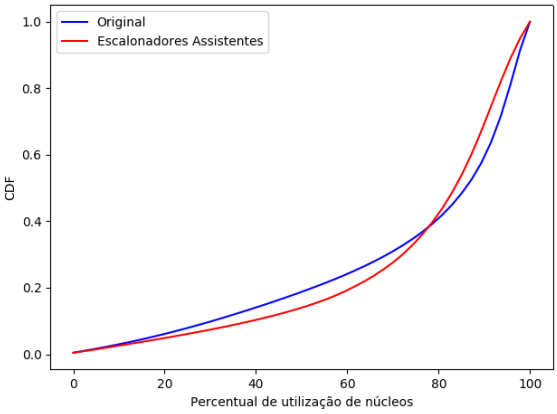
parte dos casos, e por fim, decai até o valor de zero novamente.

Os pontos nos quais a taxa de uso de CPU é nula correspondem, em geral, a trechos entre a finalização e início de dois *jobs* consecutivos da aplicação. Durante esse período, os Escalonadores Assistentes não possuem novas tarefas para atribuir aos *Executors*, pois o *microbenchmark* executa os *jobs* serialmente, como ocorre na maioria das aplicações onde os *jobs* apresentam dependências entre si. Com isso, todas as mensagens de finalizações de tarefas centralizadas no *Driver* devem ser computadas para que um novo *job* e suas tarefas sejam executadas.

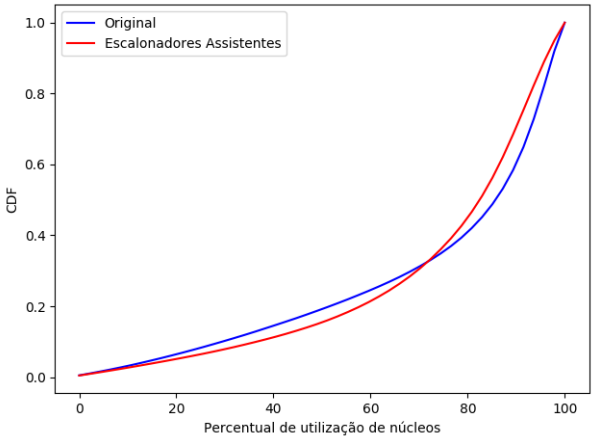
Embora os Escalonadores Assistentes aprimorem o uso de CPUs durante a execução dos *jobs*, o intervalo entre dois *jobs* demonstra ser um gargalo relevante que acomete também o Apache Spark original. Esse gargalo se reflete em perda de desempenho no tempo de execução de um *job*.



(a) CDF do uso de CPU das máquinas com *Executors* no caso com 8 *Executors* para tarefas com duração de 200 milissegundos.

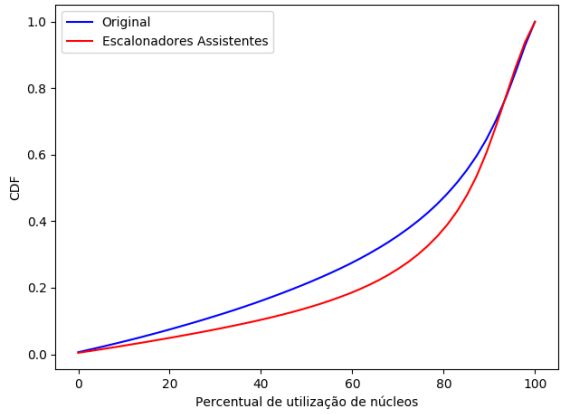


(b) CDF do uso de CPU das máquinas com *Executors* no caso com 8 *Executors* para tarefas com duração de 1 segundo.

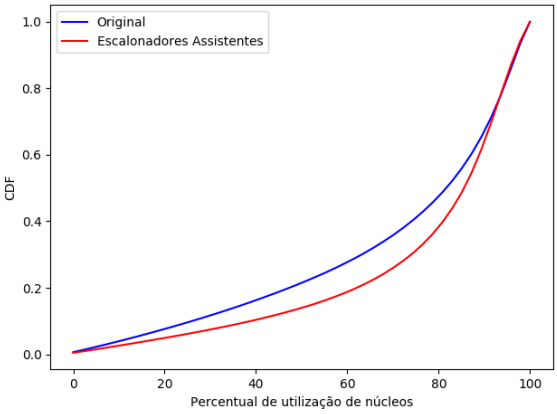


(c) CDF do uso de CPU das máquinas com *Executors* no caso com 8 *Executors* para tarefas com duração de 10 segundos.

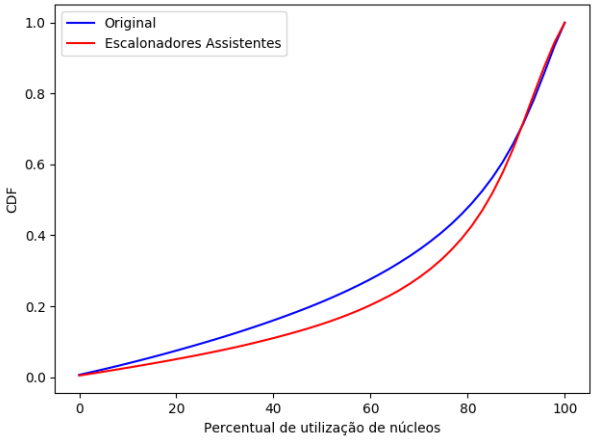
Figura 4.6: CDF do uso de CPU das máquinas com *Executors* no caso com 8 *Executors* para tarefas com duração de 200 milissegundos, 1 segundo e 10 segundos.



(a) CDF do uso de CPU das máquinas com *Executors* no caso com 16 *Executors* para tarefas com duração de 200 milissegundos.

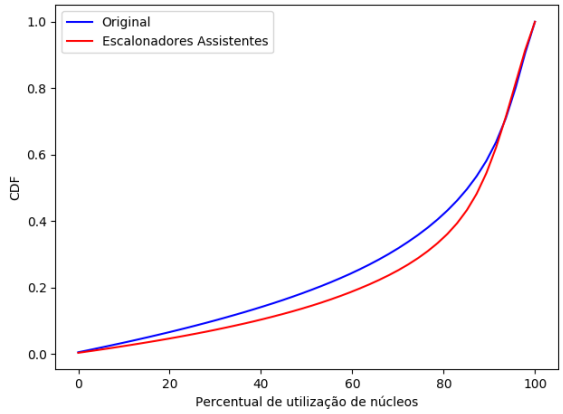


(b) CDF do uso de CPU das máquinas com *Executors* no caso com 16 *Executors* para tarefas com duração de 1 segundo.

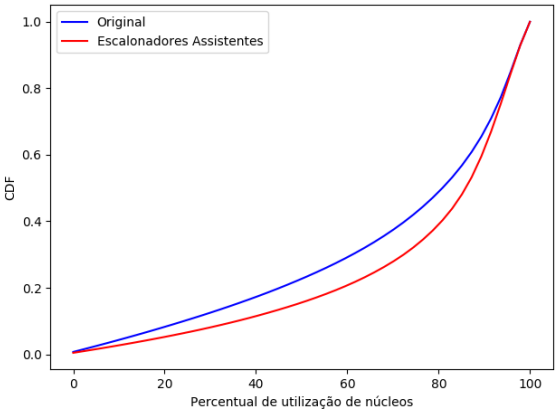


(c) CDF do uso de CPU das máquinas com *Executors* no caso com 16 *Executors* para tarefas com duração de 10 segundos.

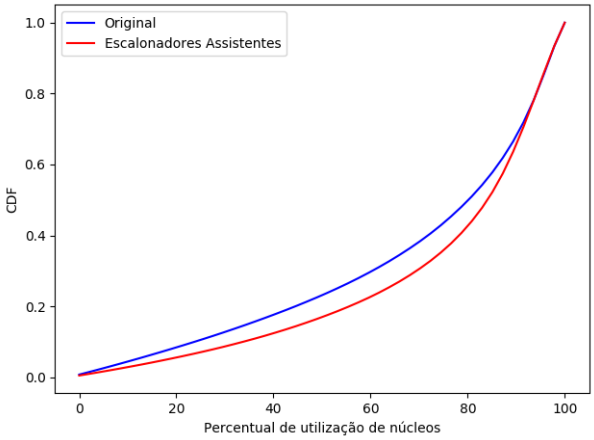
Figura 4.7: CDF do uso de CPU das máquinas com *Executors* no caso com 16 *Executors* para tarefas com duração de 200 milissegundos, 1 segundo e 10 segundos.



(a) CDF do uso de CPU das máquinas com *Executors* no caso com 32 *Executors* para tarefas com duração de 200 milissegundos.



(b) CDF do uso de CPU das máquinas com *Executors* no caso com 32 *Executors* para tarefas com duração de 1 segundo.



(c) CDF do uso de CPU das máquinas com *Executors* no caso com 32 *Executors* para tarefas com duração de 10 segundos.

Figura 4.8: CDF do uso de CPU das máquinas com *Executors* no caso com 32 *Executors* para tarefas com duração de 200 milissegundos, 1 segundo e 10 segundos.

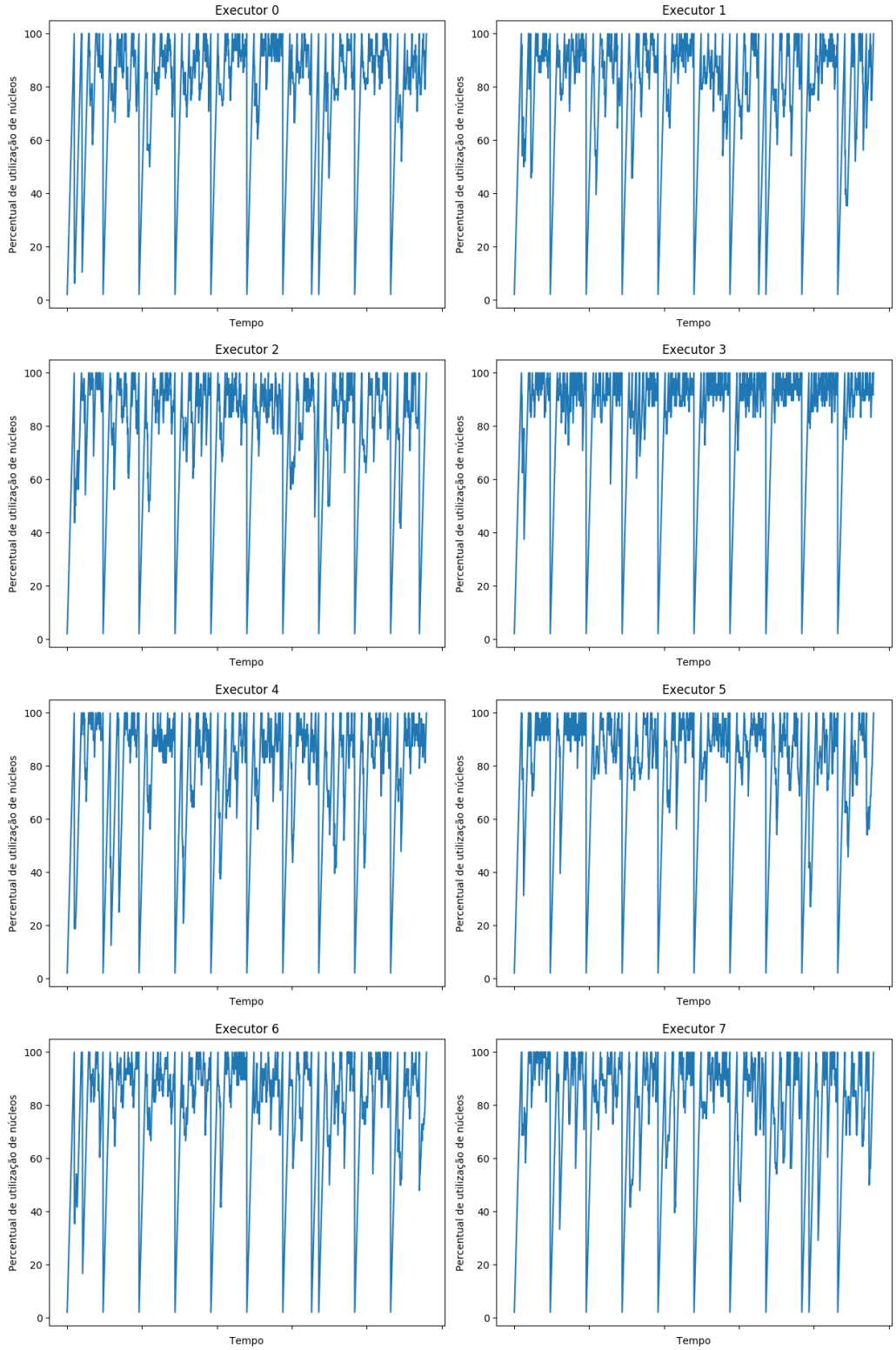


Figura 4.9: Perfil do percentual de ocupação de CPUS nos nós com *Executors* por tarefas para o caso com 16 *Executors* na execução do *microbenchmark* no Apache Spark com Escalonadores Assistentes. *Executors* de 0 a 7.

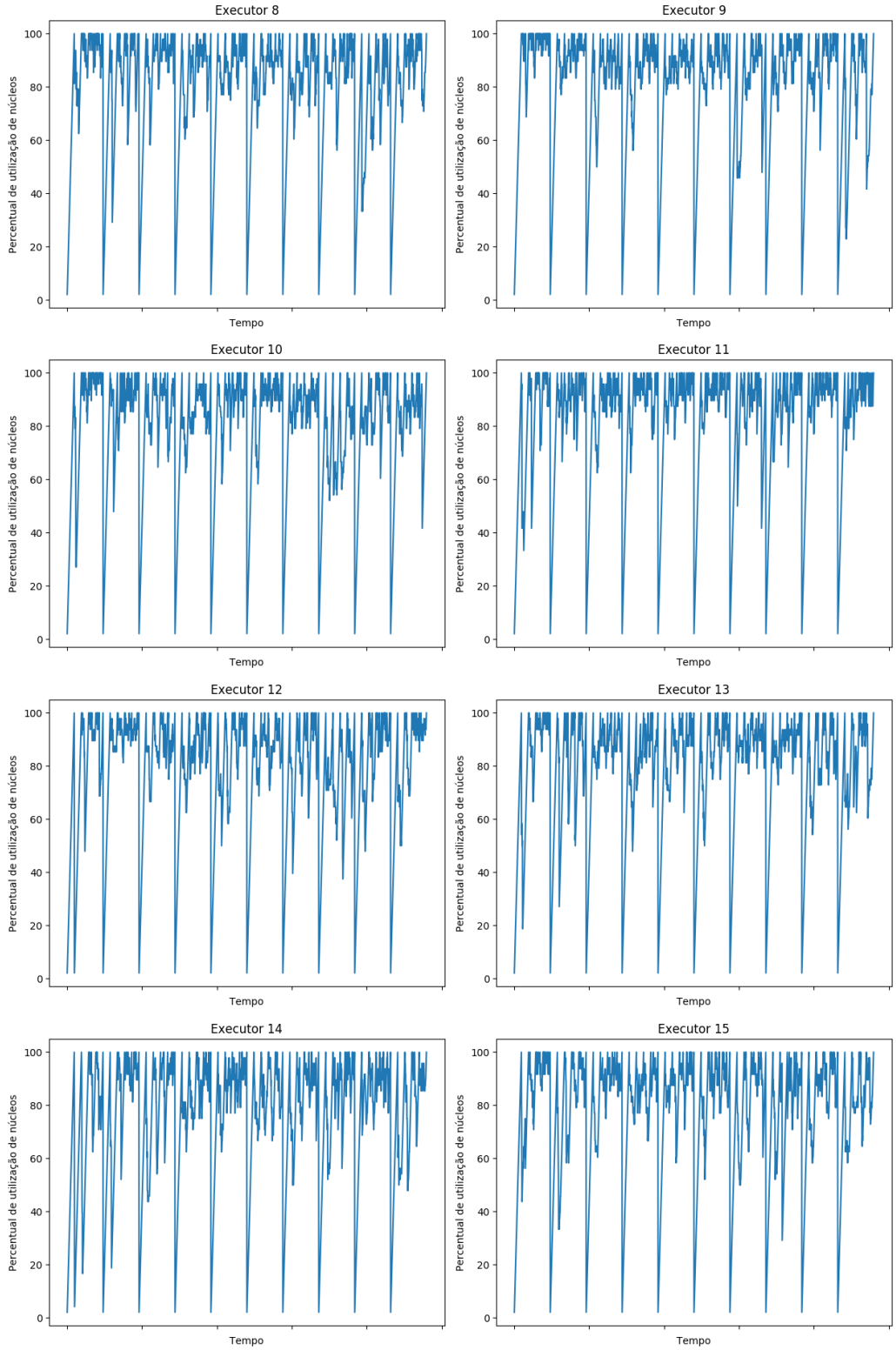


Figura 4.10: Perfil do percentual de ocupação de CPUS nos nós com *Executors* por tarefas para o caso com 16 *Executors* na execução do *microbenchmark* no Apache Spark com Escalonadores Assistentes. *Executors* de 8 a 15.

Capítulo 5

Conclusões e Trabalhos Futuros

Apresentamos a proposta de um escalonador de tarefas distribuído hierarquicamente para *Frameworks* de análise de grandes quantidades de dados. Realizamos a implementação de uma versão dessa proposta no Apache Spark, um sistema que se popularizou recentemente por apresentar um bom desempenho para aplicações iterativas e interativas em comparação ao Hadoop MapReduce.

É uma característica comum dos *Frameworks* quebrar a execução de uma aplicação em diversas tarefas que possam ser executadas em paralelo e muitos deles apresentam uma figura central responsável por promover o escalonamento das tarefas para uma grande quantidade de recursos disponíveis.

A proposta consiste no uso de escalonadores assistentes para dividir o trabalho da atribuição de tarefas aos recursos disponíveis. A divisão consiste em formar uma árvore de hierarquia onde a figura central que já existe assume o papel de nó raiz, os escalonadores assistentes são os nós intermediários e as folhas são os recursos computacionais que executam a computação das tarefas. No Apache Spark, realizamos uma prova de conceito ao implementar os *SchedulerAssistants* como um dos componentes que fazem parte do *cluster* do Spark ao lado do *Master* e dos *Workers*.

Realizamos experimentos para comparar o desempenho da nossa implementação de prova de conceito com o escalonador de tarefas original do Apache Spark. Os testes utilizaram um *microbenchmark* cujo propósito é estressar a capacidade de atribuição de tarefas dos escalonadores de maneira controlada, excluindo diversos outros aspectos do mecanismo de execução dos *Frameworks* como comunicação entre nós para troca de dados.

Os resultados mostram que os *jobs* do *microbenchmark* na versão do Apache Spark com os escalonadores assistentes apresentam valores de tempo de execução similares aos encontrados na versão original para as mesmas execuções.

Em termos de uso de CPUs pelos *Executors*, os resultados indicam que a versão do Apache Spark com escalonadores assistentes permite que os nós computacionais

mantenham o nível de carga nos processadores mais elevada do que a versão original durante a execução.

Mais ainda, a análise dos experimentos revelam que o sistema de escalonamento de tarefas sofre uma estagnação entre o término e o início de dois *jobs* serialmente consecutivos. Esse comportamento também pode ser encontrado no caso de dois estágios com dependências entre si. Nesse intervalo de tempo não tarefas para serem escalonadas.

Para um novo *job* ser iniciado é necessário que todas as tarefas do *job* anterior estejam terminadas. Tanto na versão do Apache Spark original, quanto na versão com escalonadores assistentes, a terminação de tarefas está centralizada no *Driver* que se torna um gargalo nesses sistema.

Futuramente, pretendemos permitir que os escalonadores assistentes recebam as mensagens de finalização de tarefas enviadas pelos executores. Isso distribuirá a carga de mensagens pelos escalonadores assistentes, aliviando o gargalo no *Driver*. Isso permite que o tempo entre a finalização e início de dois estágios e *jobs* consecutivos seja encurtado, favorecendo o tempo de execução das aplicações.

Além disso, vamos ampliar o espectro de testes para avaliar outros cenários e executando aplicações reais, uma vez que os *Frameworks* executam aplicações com características distintas para diversos fins.

Outrossim, para a realização dos experimentos elaboramos uma associação entre *Workers* e *SchedulerAssistants* muito simples, apenas particionando os nós em grupos e elegendo um *SchedulerAssistants* ao acaso. Experimentar outras formas de escolher os escalonadores assistentes é uma avaliação que pode trazer mais clareza no uso dessa forma de escalonamento.

Por fim, durante os experimentos avaliamos apenas a presença de dois escalonadores assistentes. Determinar como a variação do número de escalonadores assistentes influência na escalabilidade do escalonamento de tarefas é interessante considerando que a quantidade de recursos disponíveis aumenta em quantidade e tipo, e o tamanho das tarefas pode ser pequeno, levando a um estresse grande no processo de atribuição de tarefas.

Referências Bibliográficas

[1] DEAN, J., GHEMAWAT, S. "MapReduce: simplified data processing on large clusters", *Communications of the ACM*, v. 51, n. 1, pp. 107–113, 2008.

[2] SPARK, A. "Cluster Mode Overview". Acessado em 1 de setembro de 2018. Disponível em: <`https://spark.apache.org/docs/latest/cluster-overview.html`>.

[3] OVERFLOW, S. "DAG and Spark execution". Acessado em 1 de setembro de 2018. Disponível em: <`https://stackoverflow.com/questions/41638328/dag-and-spark-execution`>.

[4] HADOOP, A. "Hadoop". Acessado em 1 de setembro de 2018. Disponível em: <`http://hadoop.apache.org`>.

[5] ZAHARIA, M., CHOWDHURY, M., DAS, T., et al. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: *NSDI'12*, pp. 15–28, San Jose, CA, 2012. USENIX.

[6] DEAN, J., GHEMAWAT, S. "MapReduce: simplified data processing on large clusters". In: *OSDI'04*. Usenix, 2004.

[7] ISARD, M., BUDIU, M., YU, Y., et al. "Dryad: Distributed Data-parallel Programs from Sequential Building Blocks". In: *EuroSys'07*, pp. 59–72. ACM, 2007.

[8] CARBONE, P., KATSIFODIMOS, A., EWEN, S., et al. "Apache flink: Stream and batch processing in a single engine", *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, v. 36, n. 4, 2015.

[9] HAN, M., DAUDJEE, K. "Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems", *Proceedings of the VLDB Endowment*, v. 8, n. 9, pp. 950–961, 2015.

[10] TOSHNIWAL, A., TANEJA, S., SHUKLA, A., et al. "Storm@ twitter". In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 147–156. ACM, 2014.

[11] ABADI, M., BARHAM, P., CHEN, J., et al. "Tensorflow: a system for large-scale machine learning". In: *OSDI'16*, v. 16, pp. 265–283. Usenix, 2016.

[12] WHITE, T. *Hadoop: The definitive guide*. New York, O'Reilly Media, Inc., 2012.

[13] ZAHARIA, M., XIN, R. S., WENDELL, P., et al. "Apache spark: a unified engine for big data processing", *Communications of the ACM*, v. 59, n. 11, pp. 56–65, 2016.

[14] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., et al. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center." In: *NSDI'11*, v. 11, pp. 22–22, 2011.

[15] VAVILAPALLI, V. K., MURTHY, A. C., DOUGLAS, C., et al. "Apache hadoop yarn: Yet another resource negotiator". In: *Proceedings of the 4th annual Symposium on Cloud Computing*, p. 5. ACM, 2013.

[16] CHOWDHURY, M., ZAHARIA, M., MA, J., et al. "Managing data transfers in computer clusters with orchestra", *ACM SIGCOMM Computer Communication Review*, v. 41, n. 4, pp. 98–109, 2011.

[17] FU, Z., SONG, T., QI, Z., et al. "Efficient Shuffle Management with SCache for DAG Computing Frameworks". In: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming - PPoPP*, pp. 305–316. ACM, 2018.

[18] OUSTERHOUT, K., RASTI, R., RATNASAMY, S., et al. "Making Sense of Performance in Data Analytics Frameworks." In: *NSDI'15*, v. 15, pp. 293–307, 2015.

[19] NGUYEN, K., FANG, L., NAVASCA, C., et al. "SKYWAY: Connecting managed heaps in distributed big data systems". In: *ASPLOS'18*, pp. 56–69. ACM, 2018.

[20] NELSON, J., HOLT, B., MYERS, B., et al. "Latency-Tolerant Software Distributed Shared Memory." In: *ATC'15*, pp. 291–305. Usenix, 2015.

[21] OUSTERHOUT, K., CANEL, C., RATNASAMY, S., et al. "Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks". In: *SOSP'17*, pp. 184–200. ACM, 2017.

[22] YOO, A. B., JETTE, M. A., GRONDONA, M. "Slurm: Simple linux utility for resource management". In: *Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 44–60. Springer, 2003.

[23] STAPLES, G. "TORQUE resource manager". In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, p. 8. ACM, 2006.

[24] ZAHARIA, M., BORTHAKUR, D., SEN SARMA, J., et al. "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling". In: *Proceedings of the 5th European conference on Computer systems*, pp. 265–278. ACM, 2010.

[25] VERMA, A., PEDROSA, L., KORUPOLU, M., et al. "Large-scale cluster management at Google with Borg". In: *Proceedings of the Tenth European Conference on Computer Systems*, p. 18. ACM, 2015.

[26] GOOGLE. "Borg cluster traces from Google". Acessado em 1 de setembro de 2018. Disponível em: <`https://github.com/google/cluster-data`>.

[27] AMVROSIADIS, G., PARK, J. W., GANGER, G. R., et al. "On the diversity of cluster workloads and its impact on research results". In: *ATC'18*, pp. 533–546. USENIX, 2018.

[28] DI, S., KONDO, D., CIRNE, W. "Characterization and comparison of cloud versus grid workloads". In: *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pp. 230–238. IEEE, 2012.

[29] REISS, C., TUMANOV, A., GANGER, G. R., et al. "Heterogeneity and dynamicity of clouds at scale: Google trace analysis". In: *Proceedings of the Third ACM Symposium on Cloud Computing*, p. 7. ACM, 2012.

[30] REN, X., ANANTHANARAYANAN, G., WIERMAN, A., et al. "Hopper: Decentralized speculation-aware cluster scheduling at scale". In: *ACM SIGCOMM Computer Communication Review*, v. 45, pp. 379–392. ACM, 2015.

[31] OUSTERHOUT, K., PANDA, A., ROSEN, J., et al. "The case for tiny tasks in compute clusters". In: *Proceedings of the 14th USENIX conference on Hot Topics in Operating Systems*, pp. 14–14. USENIX Association, 2013.

[32] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M., et al. "Sparrow: distributed, low latency scheduling". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 69–84. ACM, 2013.

[33] DELGADO, P., DINU, F., KERMARREC, A.-M., et al. "Hawk: Hybrid datacenter scheduling". In: *Proceedings of the 2015 USENIX Annual Technical Conference*, pp. 499–510. USENIX, 2015.

[34] RASLEY, J., KARANASOS, K., KANDULA, S., et al. "Efficient queue management for cluster scheduling". In: *Proceedings of the Eleventh European Conference on Computer Systems*, p. 36. ACM, 2016.

[35] GOUNARIS, A., TORRES, J. "A Methodology for Spark Parameter Tuning", *Big data research*, v. 11, pp. 22–32, 2018.

[36] PETRIDIS, P. *Parameter Testing in Spark*. Tese de Mestrado, 2016.

[37] CHIBA, T., ONODERA, T. "Workload characterization and optimization of tpc-h queries on apache spark". In: *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 112–121. IEEE, 2016.

[38] GOUNARIS, A., TORRES, J. "A Methodology for Spark Parameter Tuning", *Big data research*, v. 11, pp. 22–32, 2018.

[39] SPARK, A. "Apache Spark". Acessado em 1 de setembro de 2018. Disponível em: <`https://spark.apache.org/`>.

[40] CHAIMOV, N., MALONY, A., CANON, S., et al. "Scaling Spark on HPC systems". In: *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pp. 97–110. ACM, 2016.

[41] BAER, T., PELTZ, P., YIN, J., et al. "Integrating apache spark into pbs-based hpc environments". In: *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, p. 34. ACM, 2015.

[42] GOLDT, S., VAN DER MEER, S., BURKETT, S., et al. "The Linux Programmer's Guide". The Linux Documentations Project, cap. 6, http://www.tldp.org/index.html, 1995. Disponível em: <`http://www.tldp.org/LDP/lpg/`>.

[43] HADOOP, A. "Map Reduce Tutorial". Acessado em 1 de setembro de 2018. Disponível em: <`http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Mapper`>.

[44] SPARK, A. "Spark Configuration". Acessado em 1 de setembro de 2018. Disponível em: <`http://spark.apache.org/docs/1.6.0/configuration.html`>.

Apêndice A

Códigos Fonte

Código SchedulerAssistant.scala

```
1  package org.apache.spark.deploy.schedulerAssistant
2
3  import java.text.SimpleDateFormat
4  import java.util.Date
5  import org.apache.spark.Logging
6  import org.apache.spark.SecurityManager
7  import org.apache.spark.SparkConf
8  import org.apache.spark.rpc.RpcAddress
9  import org.apache.spark.rpc.RpcEndpointRef
10 import org.apache.spark.rpc.RpcEnv
11 import org.apache.spark.util.SignalLogger
12 import org.apache.spark.util.{SerializableBuffer, Utils}
13 import scala.collection.mutable.HashMap
14 import scala.collection.mutable.Queue
15 import scala.util.Random
16 import org.apache.spark.scheduler.TaskLocation
17 import org.apache.spark.scheduler.TaskLocality._
18 import org.apache.spark.scheduler.ExecutorCacheTaskLocation
19 import scala.collection.mutable.HashSet
20 import java.util.concurrent.atomic.AtomicBoolean
21 import java.util.concurrent.atomic.AtomicLong
22 import java.util.concurrent.ConcurrentHashMap
23 import scala.collection.mutable.ArrayBuffer
24 import org.apache.spark.util.ThreadUtils
25 import org.apache.spark.scheduler.TaskDescription
26 import org.apache.spark.scheduler.TaskLocality
27 import
    ↪  org.apache.spark.scheduler.cluster.CoarseGrainedClusterMessages.LaunchTasks
28
29 private[deploy] class SchedulerAssistant(rack: String) extends Logging {
30   // rack is the id of a SchedulerAssistant since there is only one per rack
31   // rack can be anything that makes sense for using the SchedulerAssistant
```

```
32    // although the main idea is to be literally a cluster rack.
33    val id = rack
34    var endpointRefOption: Option[RpcEndpointRef] = None
35
36    // Create internal taskId. It is just for controlling
37    // the sending of tasks to executors. We do not
38    // want to send duplicate of tasks.
39    var nextInternalTaskId = new AtomicLong
40    val internalTaskIdToSerializedTaskMap = new HashMap[Long, SerializableBuffer]
41
42    val executorToTaskQueueMap = new ConcurrentHashMap[String, Queue[Long]]
43    val hostToTaskQueueMap = new ConcurrentHashMap[String, Queue[Long]]
44    val noPrefTaskQueue = new Queue[Long]
45    val allTaskQueue = new Queue[Long]
46    val alreadyLaunchedTaskSet = new HashSet[Long]
47
48    val hasTasks = new AtomicBoolean
49    val hasAwaitingTasks = new AtomicBoolean
50    val hasAwaitingOffers = new AtomicBoolean
51
52    var awaitingOffers: ArrayBuffer[WorkerOffer] = new ArrayBuffer[WorkerOffer]
53    var resourceOfferThreadPool =
      ↪  ThreadUtils.newDaemonSingleThreadExecutor("resource-offers")
54    resourceOfferThreadPool.execute(new Runnable{
55      def run = resourceOfferLoop
56    })
57
58    var awaitingTasks: ArrayBuffer[(Seq[TaskLocation], SerializableBuffer)] =
59      new ArrayBuffer[(Seq[TaskLocation], SerializableBuffer)]
60    var enqueueTaskThreadPool =
      ↪  ThreadUtils.newDaemonSingleThreadExecutor("enqueue-offers")
61    enqueueTaskThreadPool.execute(new Runnable{
62      def run = enqueueTaskLoop
63    })
64
65    def enqueueTaskLoop = while(true) {
66      if (hasAwaitingTasks.get) {
67        val currentTasks = zeroAwaitingTasksSync
68        enqueueTasks(currentTasks)
69        hasTasks.compareAndSet(false, true)
70      }
71    }
72
73    def enqueueTask(
74      preferredLocations: Seq[TaskLocation], serializedTask: SerializableBuffer
75    ) = awaitingTasks.synchronized {
76      awaitingTasks += ((preferredLocations, serializedTask))
77      hasAwaitingTasks.compareAndSet(false, true)
```

```
78    }
79
80    def zeroAwaitingTasksSync = awaitingTasks.synchronized {
81      val currentTasks = awaitingTasks
82      awaitingTasks = new ArrayBuffer[(Seq[TaskLocation], SerializableBuffer)]
83      hasAwaitingTasks.compareAndSet(true, false)
84      currentTasks
85    }
86
87    def enqueueTasks(tasks: ArrayBuffer[(Seq[TaskLocation], SerializableBuffer)]) =
       ↪  {
88      val localExecutorToTaskQueueMap = new HashMap[String, Queue[Long]]
89      val localHostToTaskQueueMap = new HashMap[String, Queue[Long]]
90      val localNoPrefTaskQueue = new Queue[Long]
91      val localAllTaskQueue = new Queue[Long]
92
93      for ((preferredLocations, serializedTask) <- tasks) {
94        val internalTaskId = nextInternalTaskId.getAndIncrement
95        internalTaskIdToSerializedTaskMap.synchronized {
96          internalTaskIdToSerializedTaskMap(internalTaskId) = serializedTask
97        }
98        for (loc <- preferredLocations) {
99          loc match {
100           case ExecutorCacheTaskLocation(host, execId) =>
101             val executorTaskQueue = localExecutorToTaskQueueMap.getOrElseUpdate(
102               execId, new Queue[Long]
103             )
104             executorTaskQueue.enqueue(internalTaskId)
105           case _ =>
106         }
107
108         val hostTaskQueue = localHostToTaskQueueMap.getOrElseUpdate(
109           loc.host, new Queue[Long]
110         )
111         hostTaskQueue.enqueue(internalTaskId)
112       }
113
114       if (preferredLocations == Nil)
115         noPrefTaskQueue.enqueue(internalTaskId)
116
117       localAllTaskQueue.enqueue(internalTaskId)
118     }
119
120     val updateExecutorToTaskQueueMap = syncTaskQueueMap(
121       localExecutorToTaskQueueMap.keys, executorToTaskQueueMap
122     )
123     val updateHostToTaskQueueMap = syncTaskQueueMap(
124       localHostToTaskQueueMap.keys, hostToTaskQueueMap
```

```
125       )
126
127       for (executor <- updateExecutorToTaskQueueMap.keys) {
128         updateQueueSync(
129           updateExecutorToTaskQueueMap(executor),
              ↪  executorToTaskQueueMap.get(executor)
130         )
131       }
132       for (host <- updateHostToTaskQueueMap.keys) {
133         updateQueueSync(
134           updateHostToTaskQueueMap(host), hostToTaskQueueMap.get(host)
135         )
136       }
137
138       updateQueueSync(localNoPrefTaskQueue, noPrefTaskQueue)
139       updateQueueSync(localAllTaskQueue, allTaskQueue)
140     }
141
142     def syncTaskQueueMap(
143       keysUpdate: Iterable[String],
144       syncQueueMap: ConcurrentHashMap[String, Queue[Long]]
145     ) = {
146       val updatesMap = new HashMap[String, Queue[Long]]
147       syncQueueMap.synchronized {
148         for (k <- keysUpdate) {
149           if (!syncQueueMap.contains(k)) {
150             syncQueueMap.put(k,new Queue[Long])
151           }
152           updatesMap += (k -> syncQueueMap.get(k))
153         }
154       }
155       updatesMap
156     }
157
158     def updateQueueSync(updates: Queue[Long], syncQueue: Queue[Long]) =
        ↪  syncQueue.synchronized {
159       syncQueue.enqueue(updates:_*)
160     }
161
162     def resourceOfferEnqueue(offers: Seq[WorkerOffer]) =
        ↪  awaitingOffers.synchronized {
163       awaitingOffers ++= offers
164       hasAwaitingOffers.compareAndSet(false, true)
165     }
166
167     /**
168      * The name of this method is intended to keep the semantic
169      * of TaskSchedulerImpl homonimous method.
```

```
170     * The similarity in both implementations explains better
171     * this reason.
172     * Currently, delay scheduling is not considered.
173     */
174    def resourceOfferLoop = {
175      var offers = new ArrayBuffer[WorkerOffer]
176      while (true) {
177        if (hasAwaitingOffers.get) {
178          offers ++= zeroAwaitingOffersSync
179        }
180
181        if (hasTasks.get) {
182          scheduleTasks(offers)
183          offers = offers.filter(_.cores > 0)
184        }
185      }
186    }
187
188    def zeroAwaitingOffersSync = awaitingOffers.synchronized {
189      val currentOffers = awaitingOffers
190      awaitingOffers = new ArrayBuffer[WorkerOffer]
191      hasAwaitingOffers.compareAndSet(true, false)
192      currentOffers
193    }
194
195
196
197    def scheduleTasks(offers: Seq[WorkerOffer]) = {
198      var tasks = new ArrayBuffer[(WorkerOffer, SerializableBuffer)]
199      val localities = List(TaskLocality.PROCESS_LOCAL,
200          TaskLocality.NODE_LOCAL, TaskLocality.NO_PREF,
201          TaskLocality.ANY)
202
203      val shuffledOffers = Random.shuffle(offers)
204      var launchedTask = false
205      for (locality <- localities) {
206        do {
207          launchedTask = assignTasksForOffers(shuffledOffers, locality, tasks)
208        } while (launchedTask)
209      }
210
211      // Use another thread to launch tasks... although we just send a message to
       ↪  endpoint here
212      launchTasks(tasks)
213    }
214
215    def assignTasksForOffers(offers: Seq[WorkerOffer], locality: TaskLocality,
216        tasks: ArrayBuffer[(WorkerOffer, SerializableBuffer)]) = {
```

```
217       var launchedTask = false
218       for (offer <- offers; if offer.cores > 0) {
219         for (tid <- getTaskForOffer(offer, locality)) {
220           launchedTask = true
221           logInfo(s"&&&&& offer.cores = ${offer.cores} &&&&&  ThreadName =
              ↪  ${Thread.currentThread.getName}")
222           offer.cores = offer.cores - 1
223
224           internalTaskIdToSerializedTaskMap.synchronized {
225             tasks += ((offer, internalTaskIdToSerializedTaskMap(tid)))
226           }
227         }
228       }
229       launchedTask
230     }
231
232     def getTaskForOffer(offer: WorkerOffer, locality: TaskLocality):
          Option[Long] = {
233       val logMsg = "Scheduling task %d for executor %s"
234
235       val executorTaskQueueOption =
236         if (executorToTaskQueueMap.contains(offer.executorId)) {
237                 Some(executorToTaskQueueMap.get(offer.executorId))
238         } else {
239                 None
240         }
241
242       for (executorTaskQueue <- executorTaskQueueOption) {
243         for (tid <- dequeueTask(executorTaskQueue)) {
244           alreadyLaunchedTaskSet.add(tid)
245           logInfo(logMsg.format(tid, offer.executorId))
246           return Some(tid)
247         }
248       }
249
250       if (TaskLocality.isAllowed(locality, TaskLocality.NODE_LOCAL) && offer.host
          ↪  != null) {
251         val hostTaskQueueOption =
252           if (hostToTaskQueueMap.contains(offer.host)) {
253             Some(hostToTaskQueueMap.get(offer.host))
254           } else {
255             None
256           }
257
258         for (hostTaskQueue <- hostTaskQueueOption) {
259           for (tid <- dequeueTask(hostTaskQueue)) {
260             alreadyLaunchedTaskSet.add(tid)
261             logInfo(logMsg.format(tid, offer.executorId))
262
```

```
263             return Some(tid)
264           }
265         }
266       }
267
268       if (TaskLocality.isAllowed(locality, TaskLocality.NO_PREF)) {
269         for (tid <- dequeueTask(noPrefTaskQueue)) {
270           alreadyLaunchedTaskSet.add(tid)
271           logInfo(logMsg.format(tid, offer.executorId))
272           return Some(tid)
273         }
274       }
275
276       if (TaskLocality.isAllowed(locality, TaskLocality.ANY)) {
277         dequeueTask(allTaskQueue) match  {
278           case Some(tid) =>
279             alreadyLaunchedTaskSet.add(tid)
280             logInfo(logMsg.format(tid, offer.executorId))
281             return Some(tid)
282           case None =>
283             hasTasks.compareAndSet(true, false)
284             logInfo(s"No more tasks to scheduler. executor = ${offer.executorId},
                ↪  cores = ${offer.cores}, ThreadName =
                ↪  ${Thread.currentThread.getName}")
285         }
286       }
287
288       return None
289     }
290
291     def dequeueTask(queue: Queue[Long]) = queue.synchronized {
292       while (!queue.isEmpty && alreadyLaunchedTaskSet.contains(queue.front)) {
293         queue.dequeue
294       }
295       if (queue.isEmpty) None
296       else Some(queue.dequeue)
297     }
298
299     def launchTasks(tasks: Seq[(WorkerOffer, SerializableBuffer)]) =
300       endpointRefOption match {
301         case Some(ref) => ref.send(LaunchTasks(tasks))
302         case None => logError(
303             "Could not call launchTask. SchedulerAssistantEndpointRef not
                ↪  available." +
304             "Tasks not lauched."
305         )
306       }
307
```

```
308   def reset = {
309     resourceOfferThreadPool.shutdownNow
310     enqueueTaskThreadPool.shutdownNow
311
312     nextInternalTaskId.set(0)
313     internalTaskIdToSerializedTaskMap.clear
314     executorToTaskQueueMap.clear
315     hostToTaskQueueMap.clear
316     noPrefTaskQueue.clear
317     allTaskQueue.clear
318     alreadyLaunchedTaskSet.clear
319     hasTasks.set(false)
320
321     awaitingOffers.clear
322     hasAwaitingOffers.set(false)
323     awaitingTasks.clear
324     hasAwaitingTasks.set(false)
325
326     resourceOfferThreadPool =
        ↪  ThreadUtils.newDaemonSingleThreadExecutor("resource-offers")
327     resourceOfferThreadPool.execute(new Runnable{
328       def run = resourceOfferLoop
329     })
330
331     enqueueTaskThreadPool =
        ↪  ThreadUtils.newDaemonSingleThreadExecutor("enqueue-offers")
332     enqueueTaskThreadPool.execute(new Runnable{
333       def run = enqueueTaskLoop
334     })
335   }
336
337   def stop = {
338     logInfo("Shutting down SchedulerAssistant")
339     resourceOfferThreadPool.shutdownNow
340     enqueueTaskThreadPool.shutdownNow
341     System.exit(1)
342   }
343 }
344
345 private[spark] object SchedulerAssistant extends Logging {
346   val SYSTEM_NAME = "sparkSchedulerAssistant"
347   val ENDPOINT_NAME = "SchedulerAssistant"
348
349   def main(args: Array[String]) = {
350     SignalLogger.register(log)
351
352     val conf = new SparkConf
353     val schedAssistArgs = new SchedulerAssistantArguments(args, conf)
```

```
354       val rpcEnv = startRpcEnv(schedAssistArgs.host, schedAssistArgs.port, conf)
355       logInfo("RpcEnv Initialized")
356
357       startSchedulerAssistantInstance(rpcEnv, schedAssistArgs.masters,
358         schedAssistArgs.rack)
359       logInfo("Scheduler Assistant Initialized")
360
361       rpcEnv.awaitTermination
362     }
363
364     def startRpcEnv(host: String, port: Int, conf: SparkConf): RpcEnv = {
365       val securityMgr = new SecurityManager(conf)
366       RpcEnv.create(SYSTEM_NAME, host, port, conf, securityMgr)
367     }
368
369     def startSchedulerAssistantInstance(rpcEnv: RpcEnv, masterUrls: Array[String],
370         rack: String) = {
371       val masterAddresses = masterUrls.map(RpcAddress.fromSparkURL(_))
372       val schedulerAssistant = new SchedulerAssistant(rack)
373       val schedAssistEndpointRef = rpcEnv.setupEndpoint(ENDPOINT_NAME, new
          ↪  SchedulerAssistantEndpoint(rpcEnv,
374           schedulerAssistant, masterAddresses))
375       schedulerAssistant.endpointRefOption = Some(schedAssistEndpointRef)
376
377     }
378
379     class SchedulerAssistantArguments(args: Array[String], conf: SparkConf) {
380       val host = Utils.localHostName
381       val port = 7080
382       var rack: String = null
383       var masters: Array[String] = null
384       val propertiesFile = Utils.loadDefaultSparkProperties(conf, null)
385
386       parse(args.toList)
387
388       def parse(args: List[String]): Unit = args match {
389         case rackValue :: masterValue :: tail =>
390           if (rack != null && masters != null) {  // Three positional arguments
              ↪  were given
391             printUsageAndExit(1)
392           }
393           rack = rackValue
394           masters = Utils.parseStandaloneMasterUrls(masterValue)
395           parse(tail)
396
397         case Nil =>
398           if (rack == null || masters == null) {  // Missing positional argument
399             printUsageAndExit(1)
```

```
400         }
401
402       case _ =>
403         printUsageAndExit(1)
404     }
405
406     def printUsageAndExit(exitCode: Int) {
407       // scalastyle:off println
408       System.err.println(
409         "Usage: Scheduler Assistant [options] <rack>  <master>\n" +
410         "\n" +
411         "rack must be a identifier for the scheduler assistant." +
412         "\n" +
413         "We refer to rack, because the main idea is to have one" +
414         "\n" +
415         "scheduler assistant per rack." +
416         "\n" +
417         "Master must be a URL of the form spark://hostname:port\n" +
418         "\n"
419   //      +
420   //      "Options:\n" +
421   //      "  -c CORES, --cores CORES  Number of cores to use\n" +
422   //      "  -m MEM, --memory MEM     Amount of memory to use (e.g. 1000M, 2G)\n"
      ↪  +
423   //      "  -d DIR, --work-dir DIR   Directory to run apps in (default:
      ↪  SPARK_HOME/work)\n" +
424   //      "  -i HOST, --ip IP         Hostname to listen on (deprecated, please
      ↪  use --host or -h)\n" +
425   //      "  -h HOST, --host HOST     Hostname to listen on\n" +
426   //      "  -p PORT, --port PORT     Port to listen on (default: random)\n" +
427   //      "  --webui-port PORT        Port for web UI (default: 8081)\n" +
428   //      "  --properties-file FILE   Path to a custom Spark properties file.\n"
      ↪  +
429   //      "                           Default is conf/spark-defaults.conf."
430       )
431       // scalastyle:on println
432       System.exit(exitCode)
433     }
434   }
435 }
```

Código SchedulerAssistantEndpoint.scala

```
1  package org.apache.spark.deploy.schedulerAssistant
2
3  import org.apache.spark.rpc.ThreadSafeRpcEndpoint
4  import org.apache.spark.Logging
```

```
5  import org.apache.spark.rpc.RpcCallContext
6  import scala.util.control.NonFatal
7  import org.apache.spark.rpc.{RpcEnv, RpcAddress}
8  import org.apache.spark.util.ThreadUtils
9  import java.util.concurrent.TimeUnit
10 import org.apache.spark.util.{SerializableBuffer, Utils}
11 import org.apache.spark.deploy.master.Master
12 import java.util.concurrent.atomic.AtomicReference
13 import java.util.concurrent.{Future => JFuture, ScheduledFuture =>
    ↪  JScheduledFuture}
14 import java.util.concurrent.atomic.AtomicBoolean
15 import java.util.concurrent.ThreadPoolExecutor
16 import org.apache.spark.deploy.DeployMessages._
17 import org.apache.spark.rpc.RpcEndpointRef
18 import org.apache.spark.scheduler.cluster.CoarseGrainedClusterMessages._
19 import scala.util.Success
20 import scala.util.Failure
21
22 private[deploy] class SchedulerAssistantEndpoint(override val rpcEnv: RpcEnv,
23     val schedAssist: SchedulerAssistant, masterRpcAddresses: Array[RpcAddress])
24     extends ThreadSafeRpcEndpoint with Logging {
25
26   private val REGISTRATION_TIMEOUT_SECONDS = 20
27   private val REGISTRATION_RETRIES = 3
28
29   // A thread pool for registering with masters. Because registering with a
      ↪  master is a blocking
30   // action, this thread pool must be able to create "masterRpcAddresses.size"
      ↪  threads at the same
31   // time so that we can register with all masters.
32   val registerMasterThreadPool = ThreadUtils.newDaemonCachedThreadPool(
33     "schedulerAssistantEndpoint-register-master-threadpool",
34     masterRpcAddresses.length // Make sure we can register with all masters at
       ↪  the same time
35   )
36   val registrationRetryThread =
      ↪  ThreadUtils.newDaemonSingleThreadScheduledExecutor(
37     "schedulerAssistantEndpoint-registration-retry-thread")
38
39   val registerMasterFutures = new AtomicReference[Array[JFuture[_]]]
40   val registrationRetryTimer = new AtomicReference[JScheduledFuture[_]]
41   val registered = new AtomicBoolean(false)
42
43   var masterAtomicEndpointRef: AtomicReference[Option[RpcEndpointRef]] =
44     new AtomicReference(None)
45
46   def sendMaster(msg: Any) = masterAtomicEndpointRef.get match {
47     case Some(ref) => ref.send(msg)
```

```
48      case None =>
49        logInfo(
50            s"Dropping \$msg because the connection to master has not yet been
              ↪  established")
51    }
52
53    override def onStart(): Unit = {
54      registerWithMaster(1)
55    }
56
57    /**
58     *  Register with all masters asynchronously and returns an array `Future`s for
        ↪  cancellation.
59     */
60    private def tryRegisterAllMasters: Array[JFuture[_]] = {
61      for (masterAddress <- masterRpcAddresses) yield {
62        registerMasterThreadPool.submit(new Runnable {
63          override def run(): Unit = try {
64            if (registered.get) {
65              return
66            }
67            logInfo("Connecting to master " + masterAddress.toSparkURL + "...")
68            val masterRef =
69              rpcEnv.setupEndpointRef(Master.SYSTEM_NAME, masterAddress,
                  ↪  Master.ENDPOINT_NAME)
70            askMasterForRegistration(masterRef)
71          } catch {
72            case ie: InterruptedException => // Cancelled
73            case NonFatal(e) => logWarning(s"Failed to connect to master
                ↪  \$masterAddress", e)
74          }
75        })
76      }
77    }
78
79    private def askMasterForRegistration(masterRef: RpcEndpointRef) = {
80      val future =  masterRef.ask[RegisterSchedulerAssistantResponse](
81          RegisterSchedulerAssistant(schedAssist.id, rpcEnv.address.host,
              ↪  rpcEnv.address.port, self))
82      future.onComplete {
83        case Success(msg) =>
84          msg match {
85            case RegisteredSchedulerAssistant(masterEndpointRef) =>
86              registered.set(true)
87              masterAtomicEndpointRef.set(Some(masterEndpointRef))
88              logInfo("Registered with master on %s:%d".format(
89                  masterEndpointRef.address.host,
90                  masterEndpointRef.address.port))
```

```
91
92            case RegisterSchedulerAssistantFailed(cause) =>
93              if (!registered.get) {
94                logError(s"Cannot register with master: ${masterRef.address}." +
95                    " Cause is $cause")
96                giveUp(s"Cannot register with master: ${masterRef.address}." +
97                    " Cause is $cause")
98              }
99
100           case MasterInStandby =>
101         }
102
103       case Failure(e) =>
104         logError(s"Cannot register with master: ${masterRef.address}", e)
105         giveUp(s"Cannot register with master: ${masterRef.address}")
106     }(ThreadUtils.sameThread)
107   }
108
109   /**
110    * Register with all masters asynchronously. It will call `registerWithMaster`
       ↪ every
111    * REGISTRATION_TIMEOUT_SECONDS seconds until exceeding REGISTRATION_RETRIES
       ↪ times.
112    * Once we connect to a master successfully, all scheduling work and Futures
       ↪ will be cancelled.
113    *
114    * nthRetry means this is the nth attempt to register with master.
115    */
116   private def registerWithMaster(nthRetry: Int) {
117     registerMasterFutures.set(tryRegisterAllMasters)
118     registrationRetryTimer.set(
119       registrationRetryThread.schedule(new Runnable {
120         override def run(): Unit = {
121           Utils.tryOrExit {
122             if (registered.get) {
123               registerMasterFutures.get.foreach(_.cancel(true))
124               registerMasterThreadPool.shutdownNow()
125             } else if (nthRetry >= REGISTRATION_RETRIES) {
126               giveUp("All masters are unresponsive! Giving up.")
127             } else {
128               registerMasterFutures.get.foreach(_.cancel(true))
129               registerWithMaster(nthRetry + 1)
130             }
131           }
132         }
133     }, REGISTRATION_TIMEOUT_SECONDS, TimeUnit.SECONDS))
134   }
135
```

```
136   override def receive: PartialFunction[Any, Unit] = {
137     case EnqueueTaskInSchedulerAssistant(preferredLocations, serializedTask) =>
138       schedAssist.enqueueTask(preferredLocations, serializedTask)
139     case RequestTaskSchedulerAssistant(executorId, executorRef, hostPort, cores)
        ↪  =>
140       val offers = (0 until cores).map(_ => new WorkerOffer(executorId, hostPort,
        ↪  executorRef, 1))
141       schedAssist.resourceOfferEnqueue(offers)
142     case LaunchTasks(tasks) => launchTasks(tasks)
143     case ResetSchedulerAssistant => resetSchedulerAssistant
144     case _ =>
145   }
146
147   override def receiveAndReply(context: RpcCallContext): PartialFunction[Any,
    ↪  Unit] = {
148     case _ =>
149   }
150
151   override def onStop(): Unit = {
152     if (registrationRetryTimer.get != null) {
153       registrationRetryTimer.get.cancel(true)
154     }
155     registerMasterThreadPool.shutdown
156     registrationRetryThread.shutdown
157     schedAssist.stop
158   }
159
160   private def giveUp(msg: String) = {
161     logInfo(msg)
162     onStop
163   }
164
165   def launchTasks(tasks: Seq[(WorkerOffer, SerializableBuffer)]) =
166     for ((offer, serializedTask) <- tasks)
167       offer.executorRef.send(LaunchTask(serializedTask))
168
169   def resetSchedulerAssistant = schedAssist.reset
170 }
```