

UNIVERSIDADE DO ESTADO DE SANTA CATARINA - UDESC
CENTRO DE CIÊNCIAS TECNOLÓGICAS - CCT
MESTRADO EM COMPUTAÇÃO APLICADA - PPGCA

LEONARDO ROSA RODRIGUES

**ESCALONADOR DE CONTÊINERES BASEADO EM ALGORITMOS
MULTICRITÉRIOS ACELERADOS POR GPU**

JOINVILLE

2020

LEONARDO ROSA RODRIGUES

**ESCALONADOR DE CONTÊINERES BASEADO EM ALGORITMOS
MULTICRITÉRIOS ACELERADOS POR GPU**

Dissertação submetida ao Programa de Pós-Graduação em Computação Aplicada do Centro de Ciências Tecnológicas da Universidade do Estado de Santa Catarina, para a obtenção do grau de Mestre em Computação Aplicada.

Orientador: Prof. Dr. Guilherme Piêgas Koslovski

Coorientador: Prof. Dr. Omir Correa Alves Junior

JOINVILLE

2020

Rodrigues, Leonardo.

Escalonador de contêineres baseado em algoritmos multicritérios acelerados por GPU/ Leonardo Rosa Rodrigues. – Joinville, 2020
91 p.

Orientador: Prof. Dr. Guilherme Piêgas Koslovski

Orientador: Prof. Dr. Omir Correa Alves Junior

Dissertação (mestrado) – Universidade do Estado de Santa Catarina, Centro de Ciências Tecnológicas, Programa de Pós-Graduação em Computação Aplicada, 2020.

1. GPU. 2. Computação em Nuvem. 3. Escalonador. 4. Redes Virtuais. 5. Métodos Multicritérios. I. Piêgas Koslovski, Guilherme. II. Correa Alves Júnior, Omir. III. Universidade do Estado de Santa Catarina, Centro de Ciências Tecnológicas, Programa de Pós-Graduação em Computação Aplicada. IV. Escalonador de Contêineres Baseado em Algoritmos Multicritérios Acelerados por GPU.

Escalonador de Contêineres Baseado em Algoritmos Multicritérios Acelerados por GPU

por

Leonardo Rosa Rodrigues


Esta dissertação foi julgada adequada para obtenção do título de

Mestre em Computação Aplicada


Área de concentração em "Ciência da Computação",
e aprovada em sua forma final pelo

**CURSO DE MESTRADO ACADÊMICO EM COMPUTAÇÃO APLICADA
DO CENTRO DE CIÊNCIAS TECNOLÓGICAS DA
UNIVERSIDADE DO ESTADO DE SANTA CATARINA.**

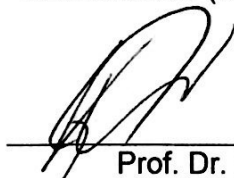
Banca Examinadora:



Prof. Dr. Guilherme Piêgas Koslovski
CCT/UDESC (Orientador/Presidente)



Prof. Dr. Maurício Aronne Pillon
CCT/UDESC



Prof. Dr. Marcelo Pasin
Université de Neuchâtel
Institut d'Informatique

Joinville, SC, 07 de fevereiro de 2020.

Dedico este trabalho aos meus familiares, amigos, colegas e professores que me acompanharam e me deram forças nessa magnífica trajetória.

“We are what we repeatedly do. Excellence, then, is not an act, but a habit.”

Will Durant

RESUMO

A utilização de contêineres passou a ser recentemente adotada como suporte para o provisionamento ágil de sistemas distribuídos. Microserviços, processamento de fluxos de dados, computação nas bordas e outros sistemas complexos podem ser concretizados sob forma de contêineres. Desta forma, o gerenciamento de rede em *Data Center* (DC) multiusuários é um fator crítico em termos de desempenho. Os usuários encapsulam suas aplicações em contêineres abstraindo os detalhes das infraestruturas hospedeiras, confiando os requisitos de *Quality-of-Service* (QoS) de comunicação e processamento ao provisionamento do *framework* de gerenciamento do DC. Devido a estes fatores somados com a heterogeneidade de configuração das requisições e a dimensionalidade dos DCs hospedeiros, o escalonamento de contêineres e redes virtuais é um problema *NP-Difícil*. Um caminho eficiente para amenizar a complexidade do escalonamento é a utilização do processamento paralelo de alto desempenho. Neste trabalho é apresentado um escalonador baseado em métodos multicritérios acelerados por *Graphics Processing Unit* (GPU), o qual realiza o escalonamento conjunto de contêineres e suas interconexões de rede através da *Augmented Forest* para realizar o mapeamento das requisições em intervalos temporais. Os experimentos apresentam que a abordagem conjunta é escalável, apresentando resultados superiores aos algoritmos encontrados nos escalonadores tradicionais utilizados pelos orquestradores de contêineres. O mapeamento do escalonador é afetado pela política de ordenação das requisições utilizada, os resultados apresentam que a melhor ordenação varia de acordo com o algoritmo de escalonamento utilizado.

Palavras-chaves: GPU, Computação em Nuvem, Escalonador, Redes Virtuais e Métodos Multicritérios.

ABSTRACT

The use of containers has recently been adopted as support for the fast provisioning of distributed systems. Micro-services, data stream processing, edge computing, and other complex systems can be achieved through the use of containers. Thus, the network management on multi-tenant container-based DC has a critical impact on performance. Tenants encapsulate applications in containers abstracting away details on hosting infrastructures, and entrust DC management framework with the provisioning of network and computational QoS requirements. Due to these facts, in addition to the DC settings heterogeneity and the dimensionality of the DC hosts, scheduling containers and virtual networks is an NP-Hard problem. An efficient way to ease the complexity of scheduling is to use high-performance parallel processing. In this work is present a scheduler based on multicriteria methods accelerated by GPU that performs the joint scheduling of containers and their network interconnections through the Augmented Forest to mapping the requests into temporal intervals. The experiments reveal that the scheduler's joint approach is scalable, presenting higher results than the algorithms found in traditional scheduler used by container orchestrators. The scheduler mapping is affected by the request sorting used, the results show that the best sorting policy changes according to the scheduling algorithm used.

Keywords: GPU, Cloud Computing, Scheduler, Virtual Networks and Multicriteria Methods.

LISTA DE ILUSTRAÇÕES

Figura 1 – Arquitetura de um contêiner implementado com <i>Linux Container</i> (LXC).	21
Figura 2 – Arquitetura de um contêiner Docker.	21
Figura 3 – Arquitetura de imagens <i>CoreOS RKT</i> (RKT) com diferentes níveis de segurança.	22
Figura 4 – Escalonamento de contêineres em DC buscando a consolidação de servidores.	27
Figura 5 – Arquitetura do Docker Swarm.	28
Figura 6 – Arquitetura do Apache Mesos.	30
Figura 7 – Arquitetura do Kubernetes.	31
Figura 8 – Augmented Tree: Composição do nó.	45
Figura 9 – Augmented Tree: Inserção de nós.	45
Figura 10 – Augmented Tree: Inserção de nós.	47
Figura 11 – Augmented Tree: Remoção de nós.	48
Figura 12 – Alocação sem agrupamento.	50
Figura 13 – Alocação com agrupamento de DC.	50
Figura 14 – Arquitetura do Escalonador.	52
Figura 15 – Funções e kernel do Escalonador.	55
Figura 16 – Tempo de processamento para algoritmos sem agrupamento em diferentes configurações da topologia <i>Fat-Tree</i>	64
Figura 17 – MCL em GPU.	65
Figura 18 – Tempo de processamento para algoritmos com agrupamento em diferentes configurações da topologia <i>Fat-Tree</i>	66
Figura 19 – Análise da fragmentação do DC resultante das variações dos cenários de pesos.	68
Figura 20 – CDF do atraso no atendimento das requisições para o Conjunto 2.	69
Figura 21 – Fragmentação da rede do DC.	71
Figura 22 – Fragmentação dos enlaces do DC.	71
Figura 23 – Métricas do escalonador com método sem e com ressubmissão.	77
Figura 24 – Métricas do escalonador com método sem e com ressubmissão utilizando <i>deadline</i>	78
Figura 25 – Métricas do escalonador com método sem e com ressubmissão utilizando <i>deadline</i> e termino antecipado.	79
Figura 26 – Métricas do escalonador através das diferentes filas de ordenação das requisições.	80

LISTA DE TABELAS

Tabela 1 – Sumário da arquitetura de escalonamento de contêineres.	27
Tabela 2 – Principais métodos MCDM.	32
Tabela 3 – Escala para julgamento de valores.	33
Tabela 4 – Resumo dos trabalhos relacionados.	40
Tabela 5 – Notação utilizada: i e j são utilizados para indexar os contêineres, enquanto u e v representam os servidores do DC.	42
Tabela 6 – Políticas de ordenação de filas.	51
Tabela 7 – Esquema de pesos para <i>Analytic Hierarchy Process</i> (AHP) e <i>Technique for Order Preference by Similarity to Ideal Solution</i> (TOPSIS). . .	53
Tabela 8 – Comparação do <i>Footprint</i> de RAM e CPU.	69
Tabela 9 – Tempo de execução do escalonador nos cenários analisados.	70
Tabela 10 – Tempo de execução, utilidades de enlaces e rede para <i>Best Fit</i> (BF), <i>Worst Fit</i> (WF), AHP e TOPSIS.	71
Tabela 11 – Composição dos contêineres.	75
Tabela 12 – Etapas da análise de Ressubmissão de Tarefas.	76
Tabela 13 – Sumário dos algoritmos AHP e Easy Backfilling com política de consolidação e disponibilidade.	80
Tabela 14 – Sumário dos algoritmos AHP e Easy Backfilling com política de consolidação e disponibilidade.	81

LISTA DE ACRÔNIMOS

ABC *Artificial Bee Colony*

ACO *Ant Colony Optimization*

AHP *Analytic Hierarchy Process*

API *Application Programming Interface*

AuFS *Advanced Multi-Layered Unification File system*

AVL *Adelson-Velsky and Landis Tree*

BF *Best Fit*

BGP *Border Gateway Protocol*

Btrfs *B-tree File System*

CPU *Central Processing Unit*

DAG *Directed Acyclic Graph*

DC *Data Center*

DE *Docker Engine*

ELECTRE *Elimination and Choice Expressing Reality*

FCFS *First Come First Served*

GPU *Graphics Processing Unit*

IaaS *Infrastructure as a Service*

IPC *Interprocess Communication*

IV *Infraestrutura Virtual*

LXC *Linux Container*

MACBETH *Measuring Attractiveness by a Categorical Based Evaluation Technique*

MAUT *Multiple Attribute Utility Theory*

MCDM *Multicriteria Decision Making*

MCL *Markov Cluster Algorithm*

MILP *Mixed Integer Linear Programming*

MTU *Maximum Transmission Unit*

NaaS *Network as a Service*

NAT *Network Address Translation*

PROMETHEE *Preference ranking organization method for enrichment evaluation*

PSO *Particle Swarm Optimization*

QoS *Quality-of-Service*

RAM *Random Access Memory*

RKT *CoreOS RKT*

SIMD *Single Instruction Multiple Data*

SDN *Software Defined Networking*

SO *Sistema Operacional*

TOPSIS *Technique for Order Preference by Similarity to Ideal Solution*

VIMAM *virtual infrastructure multicriteria allocation and migration-based broker*

MV *Máquina Virtual*

VNE *Virtual Network Embedding*

WF *Worst Fit*

SUMÁRIO

1	INTRODUÇÃO	14
1.1	OBJETIVO	16
1.2	METODOLOGIA DA PESQUISA	17
1.3	PRINCIPAIS CONTRIBUIÇÕES	18
1.4	ESTRUTURA DO TEXTO	18
2	REVISÃO DE LITERATURA	19
2.1	CONTÊINERES	19
2.1.1	Tecnologias	20
2.1.1.1	<i>Linux Container</i>	20
2.1.1.2	<i>Docker</i>	21
2.1.1.3	<i>CoreOS RKT</i>	22
2.1.2	Comunicação em Contêineres	23
2.2	ESCALONAMENTO DE CONTÊINERES	26
2.3	ORQUESTRAÇÃO DE CONTÊINERES	28
2.3.1	Docker Swarm	28
2.3.2	Apache Mesos	29
2.3.3	Google Kubernetes	30
2.4	MÉTODOS DE DECISÃO MULTICRITÉRIO	31
2.4.1	AHP	32
2.4.2	ELECTRE	34
2.4.3	MACBETH	34
2.4.4	MAUT	34
2.4.5	PROMETHEE	35
2.4.6	TOPSIS	35
2.5	TRABALHOS RELACIONADOS	36
2.6	CONSIDERAÇÕES PARCIAIS	41
3	PROPOSTA DO ESCALONADOR	42
3.1	ESCALONAMENTO TEMPORAL	43
3.1.1	Augmented Forest	44
3.1.2	Verificação de Disponibilidade	45
3.1.3	Inserção	46
3.1.4	Remoção	46
3.2	OBJETIVOS	48

3.3	RESTRIÇÕES DE CAPACIDADE	49
3.4	POLÍTICAS DE ALOCAÇÃO	49
3.5	POLÍTICAS DE CONSTRUÇÃO DE FILAS	50
3.6	ARQUITETURA DO ESCALONADOR	52
3.7	DISTRIBUIÇÃO DE PESOS	53
3.8	IMPLEMENTAÇÃO EM GPU	53
3.8.1	AHP	58
3.8.2	TOPSIS	59
3.8.3	Interconexões de Rede	60
3.9	CONSIDERAÇÕES PARCIAIS	61
4	ANÁLISE EXPERIMENTAL COM REQUISITOS DE REDE E PROCES-	
	SAMENTO	62
4.1	ANÁLISE DE ESCALABILIDADE	62
4.2	ANÁLISE DA QUALIDADE DO ESCALONAMENTO	65
4.2.1	Análise de Contêineres	67
4.2.2	Análise de Rede e Contêineres	70
4.3	CONSIDERAÇÕES PARCIAIS	72
5	ANÁLISE EXPERIMENTAL COM REQUISITOS TEMPORAIS	74
5.1	PROTOCOLO EXPERIMENTAL	74
5.2	ANÁLISE DE RESSUBMISSÃO DE TAREFAS	75
5.2.1	Etapas Padrão	76
5.2.2	Etapas de Tempo Máximo	77
5.2.3	Etapas com Tempo Máximo e Término Antecipado	77
5.2.4	Principais Considerações	78
5.3	ANÁLISE DE FILAS DE ORDENAÇÃO	79
5.4	CONSIDERAÇÕES PARCIAIS	82
6	CONCLUSÃO	83
6.1	PUBLICAÇÕES REALIZADAS	84
	REFERÊNCIAS	86

1 INTRODUÇÃO

A complexidade no desenvolvimento e manutenção de sistemas distribuídos é um desafio para desenvolvedores e pesquisadores. A virtualização baseada em contêineres oferece um mecanismo simples e escalável para hospedar e gerenciar aplicações distribuídas de larga escala para processamento de *Big Data*, *Edge Computing*, processamento de fluxo de dados, dentre outros. Os contêineres são utilizados como forma de encapsular o ambiente de suas aplicações abstraindo os detalhes do sistema operacional, versões de bibliotecas e configurações (TRIHINAS et al., 2018).

A tecnologia de contêineres virtualiza os recursos em nível de Sistema Operacional (SO), isolando grupos de processos através de funcionalidades presentes no núcleo do SO hospedeiro (e.g., *namespace* e *cgroups*). A utilização de aplicações containerizadas dispensa a presença de um hipervisor no servidor hospedeiro devido aos contêineres compartilharem o núcleo do SO hospedeiro. Ainda, a replicação de aplicações containerizadas é facilitada, devido ao contêiner possuir todos os recursos necessários para a execução da aplicação, possibilitando que os contêineres possam ser aplicados em qualquer infraestrutura (e.g., nuvens computacionais e Aglomerados) (ASSUNCAO; VEITH; BUYYA, 2018; TRIHINAS et al., 2018).

As aplicações containerizadas não degradam o desempenho da *Random Access Memory* (RAM) e *Central Processing Unit* (CPU), entretanto existe uma queda na vazão e aumento da complexidade de gerenciamento da rede quando comparado aos servidores ou Máquinas Virtuais (MVs) (SUO et al., 2018). As comunicações entre contêineres são suportadas por diversas tecnologias (e.g., comutadores virtuais, *bridges* e redes de sobreposição), porém, as necessidades de *Quality-of-Service* (QoS) (e.g., reserva de largura de banda, controle de latência, segmentação da rede) não são suportadas durante o provisionamento da rede pelos *frameworks* de gerenciamento de contêineres em sua totalidade (BURNS et al., 2016).

Ademais, os *Data Centers* (DCs) multiusuários fornecem serviços a usuários com necessidades distintas. Deste modo, o escalonador do DC deve ser apto a reconhecer e tratar diversos tipos de requisições, analisando as necessidades dos usuários em conjunto com a disponibilidade dos servidores, a carga de rede do DC e a função objetivo adotada. A decisão do escalonador é sensível à função objetivo selecionada (e.g., aumentar a confiabilidade dos contêineres, redução de custos e consolidação). Ao receber uma requisição de contêiner informando, por exemplo, uma determinada configuração mínima de CPU, RAM, entre outros, o escalonador deve selecionar no DC um hospedeiro apto a atender a demanda. Ainda, a tomada de deci-

são não é trivial. Enquanto alguns contêineres solicitam um elevado volume de processamento, outros podem solicitar majoritariamente recursos de comunicação ou armazenamento.

O problema do escalonamento de contêineres em conjunto com redes virtuais pode ser reduzido a uma série de problemas análogos, como o *multiway separator problem* (YU et al., 2008) e o *Virtual Network Embedding (VNE)* (ROST; DÖHNE; SCHMID, 2019). O problema é representado por dois grafos ponderados não direcionados. O primeiro retrata a requisição do usuário (*i.e.*, contêineres e requisitos de comunicação correspondentes), representado por um grafo $G^c = (N^c, E^c)$ (N^c denota os contêineres e E^c reflete a necessidade de enlaces pertencentes a uma requisição). Enquanto o segundo grafo descreve a infraestrutura física dos recursos do DC por um grafo $G^s = (N^s, E^s)$ (N^s denota a capacidade de recursos e E^s reflete a capacidade de enlaces existentes na infraestrutura física do DC). O mapeamento ocorre através da função $f : G^c \rightarrow G^s$ que respeite os limites de recursos do DC.

Porém, esta tarefa é complexa, uma vez que cada requisição possui diferentes necessidades de recursos (*e.g.*, o poder de processamento e a largura de banda). Ainda, os recursos da infraestrutura física são limitados, necessitando que o escalonador possua um controle de admissão de requisições, recusando ou adiando as requisições que não possam ser mapeadas, garantido que todas as requisições que foram aceitas sejam efetivamente alocadas. Por fim, o escalonador deve trabalhar com o mapeamento de diversas topologias de infraestrutura virtuais, tornando o mapeamento um problema *NP-Hard* (YU et al., 2008; CAMATI; CALSAVARA; JR, 2014).

A complexidade do problema é abordado por Zhu (ZHU; WANG; WANG, 2015), o trabalho apresenta o comportamento do escalonador do Google Borg, o qual possui 12.500 servidores (aproximadamente uma *fat-tree* $K=38$). Neste DC são recebidos em média 1.016.989 requisições por segundo e são mapeadas em média de 10.000 requisições por segundo. Ainda, as requisições que não foram mapeadas com sucesso são em sua maioria são enviadas para a fila de requisições para ressubmissão futura. A diferença entre as requisições mapeadas e a quantidade de novas requisições resultam em um aumento gradativo na fila do escalonador. Diante deste cenário, é exposto a necessidade de escalonadores que consigam mapear grandes quantidades de requisições (*i.e.*, $> 1.000.000$) em DCs com 12.500 servidores.

A diversidade de configurações das requisições, juntamente do elevado número de recursos no DC, das diferentes funções objetivos, das restrições e do considerável número de contêineres são fatores cruciais para o processamento desse problema. Recentemente, a aplicação de processamento paralelo, sobretudo a utilização de *Graphics Processing Unit* (GPU) para acelerar o escalonamento de infraestruturas virtuais em DCs de nuvens apontou um caminho para amenizar a dimensionalidade

e complexidade do problema (NESI et al., 2018a; NESI et al., 2018b). A capacidade da arquitetura *Single Instruction Multiple Data* (SIMD) presente nas GPUs permite a utilização conjunta de centenas de unidades paralelas de ponto flutuante, atingindo em alguns casos uma capacidade na casa dos teraflops. Embora promissor, o desenvolvimento de escalonadores usando GPUs é complexo e exige uma releitura de algoritmos previamente desenvolvidos com arquiteturas tradicionais.

Um método de decisão, fundamentado em métodos *Multicriteria Decision Making* (MCDM), consiste na definição das possíveis alternativas de solução do problema e dos critérios envolvidos; na avaliação das alternativas em relação aos critérios elencados; no impacto gerado por cada critério sobre o problema; e na avaliação global de cada alternativa (GARG; VERSTEEG; BUYYA, 2013). Desta forma, os métodos multicritérios podem ser utilizados para auxiliar a escolha do escalonamento dos contêineres e redes sobre os recursos do DC através da análise simultânea de um conjunto de variáveis e critérios pertinentes ao problema, considerando as necessidades da requisição de cada usuário.

Diante do exposto, o objetivo deste trabalho é especificar e desenvolver um escalonador de contêineres e redes virtuais baseado em métodos multicritérios acelerados por GPU. O escalonador é utilizado para otimizar os objetivos de maximizar a consolidação de um DC multiusuário, e maximizar a QoS das requisições (*i.e.*, alocar o máximo de recursos suportados pelo DC que satisfaçam as necessidades dos usuários). O principal diferencial do trabalho é propor um algoritmo escalável, o qual realiza a alocação de recursos em DC com configurações reais em termos de servidores e topologia de rede.

1.1 OBJETIVO

O objetivo geral do presente trabalho é desenvolver um escalonador de contêineres e redes virtuais baseado em algoritmos multicritérios acelerado por GPU. Para tanto, consideram-se os seguintes objetivos específicos:

- Especificar a arquitetura do escalonador;
- Implementar um conjunto de algoritmos multicritérios em GPU;
- Implementar a alocação conjunta de contêineres e redes virtuais;
- Analisar a escalabilidade do escalonador;
- Comparar o desempenho do escalonador com algoritmos oferecidos por gerenciadores de contêineres.

1.2 METODOLOGIA DA PESQUISA

A área da Ciência da Computação e suas subáreas são constituídas de três tipos básicos de pesquisas, a pesquisa formal, a empírica e a exploratória. A pesquisa formal possui como objetivo validar uma teoria através de provas formais, sendo o tipo de pesquisa mais complexa a ser desenvolvida. Porém, a mais difícil de ser refutada, enquanto a pesquisa empírica é baseada em comparações do artefato através de testes aceitos pela comunidade, utilizando cálculos e métodos estatísticos. Por fim, a pesquisa exploratória representa os trabalhos que não possuem comparações e validações estatísticas sobre o artefato apresentado, representado por estudos de casos em análises qualitativas (WAZLAWICK, 2017). Desse modo, a presente pesquisa é categorizada como pesquisa empírica, uma vez que utiliza ferramentas estatísticas e de análise de dados para avaliar o desempenho do artefato.

Um sistema possui diversas variáveis que impactam em seu desempenho, sendo divididas em qualitativas e quantitativas (WAZLAWICK, 2017). Entretanto, existe outra classificação que separa as variáveis em independentes, dependentes e intervenientes. As variáveis independentes são as variáveis que possuem influência sobre outra variável, utilizadas para analisar o comportamento das variáveis dependentes. Por outro lado, as variáveis dependentes são variáveis que não são alteradas durante as análises e testes, possuindo seu valor alterado pelo ambiente e pelas variáveis independentes. Por fim, as variáveis intervenientes representam os ruídos existentes durante a coleta de dados da pesquisa (RAUEN, 2012). Diante disso, a presente pesquisa possui um conjunto de variáveis independentes composto pelo número de hospedeiros físicos, número de requisições de usuários e recursos requisitados por requisição. Enquanto as variáveis dependentes definidas são a fragmentação, a disponibilidade, a QoS e o tempo de alocação.

Por fim, as fontes primárias são um conjunto de artigos, periódicos ou base de dados que são utilizados para auxiliar na definição das palavras chaves e identificar os principais artigos e autores da área pesquisada. Para realizar esta pesquisa foram elencados quatro mecanismos de busca acadêmica, os quais são o *Science Direct*, *ACM Digital Library*, *IEEEExplore* e *SpringerLink*. Visando contemplar o maior número de trabalhos pertinentes a pesquisa, foi elaborada uma chave de busca através de uma composição *booleana* de palavras chaves relacionadas ao tema, formando a chave de busca: ("data center"AND (scheduling OR orchestration) AND ((multicriteria OR MCDM) OR (container OR docker OR rkt OR lxc OR "virtual machine"OR vm)) OR network).

1.3 PRINCIPAIS CONTRIBUIÇÕES

As principais contribuições do trabalho foram: (I) desenvolver um escalonador de contêineres e redes virtuais; (II) desenvolver um escalonador capaz de considerar de forma automática múltiplos critérios, sendo independente a quantidade de variáveis consideradas, apresentando uma abordagem mista de algoritmos de agrupamento e métodos multicritérios acelerados em GPU; (III) desenvolver uma estrutura de dados denominada *Augmented Forest* para realizar o mapeamento das requisições dos usuários no DC de forma eficiente; (IV) a análise da qualidade da solução encontrada durante o escalonamento dos rastros do Google Borg em um DC com *Fat-Tree* (configurado com $k = 44$), demonstrando a aplicabilidade do escalonador em cenários com elevado número de servidores, investigando a utilização do DC e o eventual atraso total no processamento; (V) a análise de rede apresentando resultados superiores do escalonador ao comparar com os algoritmos utilizados pelos escalonadores atuais; (VI) a análise do comportamento do escalonador ao utilizar o método com e sem ressubmissão para realizar o mapeamento das requisições, utilizando 3 conjuntos de requisições, de forma a explorar o ambiente dinâmico em que o escalonador está inserido; (VII) a análise das ordenações da fila de requisições, comparando o comportamento do escalonador ao utilizar 7 variantes de ordenação dos recursos para contêineres.

1.4 ESTRUTURA DO TEXTO

O presente documento está organizado em 5 capítulos. O Capítulo 2 detalha a fundamentação teórica sobre contêineres, escalonamento de contêineres, métodos multicritérios e apresenta os principais trabalhos relacionados encontrados na literatura. A descrição da proposta do trabalho é apresentado no Capítulo 3. Os Capítulos 4 e 5 apresentam o ambiente, as métricas consideradas e os resultados obtidos a partir das análises realizadas. Por fim, o Capítulo 6 apresenta as considerações parciais.

2 REVISÃO DE LITERATURA

Os conceitos necessários para a compreensão do trabalho proposto são apresentados neste capítulo. A Seção 2.1 apresenta uma definição e contextualização dos contêineres. A Seção 2.2 discute o escalonamento de contêineres em um *Data Center* (DC). Na Seção 2.3 é descrito os orquestradores de contêineres e suas características. A Seção 2.4 apresenta os métodos multicritérios e suas definições. Enquanto na Seção 2.5 apresenta os trabalhos relacionados à esta proposta, expondo as principais pesquisas sobre escalonamento em DC. Por fim, as considerações parciais são apresentadas na Seção 2.6. Caso o leitor possua conhecimentos prévios sobre estes conceitos, a sequencia da leitura segue no Capítulo 3.

2.1 CONTÊINERES

Os contêineres são unidades autônomas e independentes que empacotam aplicações e suas dependências, provendo isolamento em nível de Sistema Operacional (SO). Similar às Máquinas Virtuais (MVs), os contêineres são uma técnica de virtualização que permitem que os recursos de um nó de computação sejam compartilhados entre múltiplos usuários e suas aplicações simultaneamente (GOLDBERG, 1974). Entretanto, diferente das MVs que utilizam virtualização em nível de *hardware*, os contêineres utilizam a virtualização em nível de SO (RODRIGUEZ; BUYYA, 2018). Independente da tecnologia utilizada e da arquitetura do sistema utilizado, os contêineres possuem 5 características intrínsecas, as quais são abordadas a seguir:

- I os contêineres que estão em execução em um mesmo servidor compartilham o núcleo do SO e são executados como processos isolados no espaço de usuário, sem a presença de um hipervisor. O isolamento realizado entre os processos não aplica sobrecarga e interferências sobre as aplicações (SHARMA et al., 2016). Embora existam interferências no desempenho das aplicações devido a competição dos processos co-allocados pelos recursos do servidor hospedeiro, este fenômeno pode ser reduzido através da configuração dos limites dos recursos que um contêiner pode utilizar;
- II os contêineres utilizam os recursos de forma elástica, aumentando ou diminuindo a capacidade de acordo com a carga da aplicação respeitando limites pré-definidos;
- III a distribuição dos contêineres é realizada por imagens, com o uso de pacotes executáveis que possuem todas as dependências para executar a aplicação. Ainda,

as imagens são construídas em camadas de sistemas de arquivos, e necessitam menos espaço que as MVs;

IV o isolamento fornecido pelos contêineres resulta em um ambiente flexível para o desenvolvimento, testes e produção para as aplicações (RODRIGUEZ; BUYYA, 2018);

V a utilização dos recursos dos nós de computação são otimizados (FELTER et al., 2015), além de possuir um tempo de provisionamento menor do que o encontrado nas MVs (PIRAGHAJ et al., 2017).

2.1.1 Tecnologias

As soluções baseadas em contêineres permitem o desenvolvimento dinâmico e o uso de microsserviços em ambientes com servidores aglomerados (THÖNES, 2015). Os padrões de desenvolvimento de microsserviços são amplamente reconhecidos para o desenvolvimento de aplicações, transformando as grandes aplicações monolíticas em diversas aplicações menores coordenadas entre si. A utilização de pequenas aplicações permite que uma funcionalidade seja isolada, desconectada ou protegida das demais partes da aplicação sem afetar o funcionamento geral do sistema (KOZHIR-BAYEV; SINNOTT, 2017). Existem diversas tecnologias que suportam microsserviços baseados em contêineres, dentre as tecnologias mais utilizadas estão o LXC (LXC, 2019), Docker (DOCKER, 2019a) e *CoreOS RKT* (RKT) (COREOS, 2019d).

2.1.1.1 *Linux Container*

O *Linux Container* (LXC) é uma tecnologia de virtualização de contêineres mantido pelo núcleo padrão do Linux que utiliza uma *Application Programming Interface* (API) flexível. Os contêineres baseados em LXC utilizam os padrões de isolamento e controle de recursos (*e.g.*, rede, *Central Processing Unit* (CPU), *Random Access Memory* (RAM)) através do *namespaces* (*i.e.*, recurso responsável por particionar e isolar os recursos de cada processo, de modo que um processo não enxerga processos de grupos vizinhos) e *cgroups* (*i.e.*, aplicado para que os contêineres possam dividir os recursos de *hardware* disponíveis e permite que limites possam ser configurados a um contêiner específico em determinado período de tempo) existentes no núcleo Linux do servidor LXC hospedeiro. A arquitetura dos contêineres baseadas em LXC é apresentada na Figura 1. O LXC utiliza um sistema *init* que gerencia vários processos, executando-os como se fossem uma única aplicação.

Desta forma o LXC atinge taxas de transferências próximas ao *bare metal*. Ainda, em um único contêiner LXC podem ser criadas múltiplas aplicações, sem a necessidade de restringir o contêiner a portar somente uma única aplicação, deste

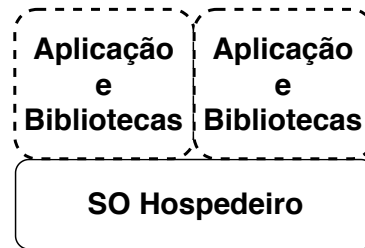


Figura 1 – Arquitetura de um contêiner implementado com LXC.

modo, o LXC permite aos desenvolvedores maiores possibilidades de utilização dos contêineres. Outro fator importante, é o sistema de arquivos utilizado pelo LXC, o *B-tree File System* (Btrfs), que permite a criação de múltiplos sistemas de contêineres clonados através de um único volume (LXC, 2019; KOZHIRBAYEV; SINNOTT, 2017).

2.1.1.2 Docker

O Docker utiliza recursos do núcleo Linux como o *cgroups*, *namespaces* e *Advanced Multi-Layered Unification File system* (AuFS) para separar e isolar processos, permitindo que contêineres executem de forma independente entre si. A tecnologia oferece um ambiente automatizado para implementação de aplicações containerizadas fornecendo controle de versionamento e distribuição das aplicações (XAVIER et al., 2013). A Figura 2 apresenta as camadas de um contêiner provisionado com Docker (BABAR; RAMSEY, 2017).

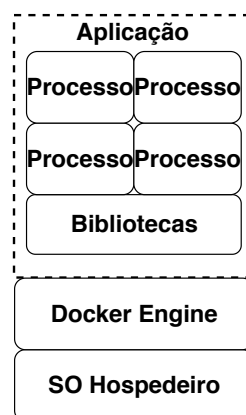


Figura 2 – Arquitetura de um contêiner Docker.

As aplicações em Docker são incentivadas a serem desenvolvidas em forma modular, segregando os processos em sub-aplicações, aumentando sua granularidade, deste modo o Docker desativa parte de uma aplicação (*e.g.*, manutenção e atualização) sem desativar a aplicação como um todo (XAVIER et al., 2013). A imagem de um contêiner Docker é composta por diversas camadas, construída com o sistema de arquivos AuFS. As camadas são combinadas em uma única imagem e uma nova camada

é criada quando há modificações na imagem do contêiner. Desta maneira, o Docker permite o versionamento de imagens de contêineres possibilitando rastrear as modificações realizadas em todas as versões de uma imagem. O qual torna possível realizar a reversão de imagens de um contêiner, possibilitando voltar um contêiner específico para versões prévias de modo ágil. Assim, é possível economizar o armazenamento e reduzir a utilização de memória com a inicialização de contêineres (XAVIER et al., 2013).

Por fim, o *Docker Engine* (DE), o principal recurso utilizado pelo Docker, é um *daemon* que executa no sistema hospedeiro. Todas as operações executadas no Docker e o gerenciamento do ciclo de vida dos contêineres são direcionadas para o *daemon* (e.g., gerenciamento de contêineres e imagens, construção de imagens, criar, iniciar, parar, deletar e pausar contêineres) (BABAR; RAMSEY, 2017).

2.1.1.3 CoreOS RKT

O RKT é uma tecnologia de contêineres mantida pelo CoreOS (COREOS, 2019d). Assim como o Docker, o RKT permite a automação do desenvolvimento de aplicações containerizadas que podem ser executadas em diferentes servidores independentemente das características de *hardware*. Além de permitir a construção de um modelo seguro de contêineres, o RKT não utiliza um Daemon, sendo integrado com os sistemas padrões de inicialização (e.g., *systemd*, *upstart*) (KOZHIRBAYEV; SINNOTT, 2017; COREOS, 2019a). A Figura 3 apresenta as diferentes possibilidades de segurança suportadas pelo RKT durante a criação de contêineres.

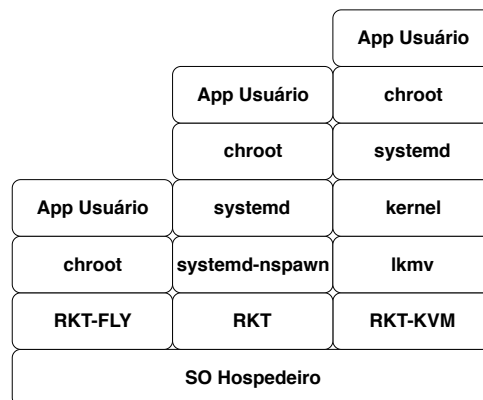


Figura 3 – Arquitetura de imagens RKT com diferentes níveis de segurança.

Segundo (COREOS, 2019a; BABAR; RAMSEY, 2017), as funcionalidades de segurança utilizadas pelo RKT, são: (I) a validação de assinatura de imagens; (II) separação de privilégios entre diferentes tarefas em um mesmo contêiner; (III) o *SELinux* automático é utilizado em sistemas que possuem esta funcionalidade habilitada em seu núcleo. Assim, é possível que cada contêiner receba *tags* apropriadas automati-

camente, fazendo o isolamento obrigatório das aplicações em execução no servidor; (IV) utiliza o *trusted platform module* presente em sistemas modernos para o registro de atividades de contêineres; e (V) o isolamento modular de contêineres é suportado pelo RKT. Apesar de utilizar o *cgroups* para prover isolamento de contêineres como padrão, a tecnologia possui suporte para diferentes soluções de isolamento, como:

- *Clean containers*/RKT-KVM: os contêineres são executados através de um hipervisor KVM, LKVM ou QEMU (COREOS, 2019b). Logo, o contêiner possui seu próprio núcleo de SO e isolamento de hipervisor. Assim sendo, o RKT contém um núcleo Linux que é executado sob o hipervisor, iniciando o *systemd* para iniciar as aplicações do contêiner. Ainda, os componentes utilizados para o funcionamento desta abordagem são semelhantes, havendo somente a troca do *systemd-nspawn*, utilizado em criações de contêineres RKT padrões, pelo hipervisor e o núcleo Linux;
- Contêineres em nível de hospedeiro/RKT-FLY: proporciona ao contêiner executar com todos os privilégios do servidor hospedeiro, mantendo os padrões de gerenciamento de imagem padrões do RKT (COREOS, 2019c). Esta abordagem reduz os componentes utilizados para a criação de um contêiner, retirando as camadas de gerenciamento de processos. Entretanto, não é possível isolar a rede, CPU e RAM do contêiner, resultando em baixo nível de isolamento e segurança.

2.1.2 Comunicação em Contêineres

As aplicações executadas em contêineres apresentam desempenhos de CPU e RAM próximos as aplicações executadas em *bare metal* e em MVs (SHARMA et al., 2016). Porém, diversas maneiras para realizar a comunicação de dados entre contêineres foram propostas. Desta forma, a comunicação dos contêineres possui um desempenho inferior se comparado a aplicações em *bare metal* e MVs. Ainda, em alguns casos, aplicações são disponibilizadas para utilização após o estabelecimento das conexões de comunicação entre os contêineres, podendo aumentar o tempo de criar e executar um contêiner, tornando a rede um fator crítico (SUO et al., 2018).

A experiência do usuário é diretamente impactada pelo desempenho da rede nas aplicações containerizadas. Logo, a escolha da técnica de comunicação de rede utilizada pelos contêineres é essencial, uma vez que existe um *trade off* entre o desempenho da comunicação e o isolamento da rede. Neste contexto, existem duas categorias principais de comunicação, a rede *single-host* e rede *multi-host* (SUO et al., 2018).

A comunicação de contêineres que estão em um mesmo servidor hospedeiro é denominada comunicação *single-host*. Neste tipo de comunicação, há um *trade off*

de desempenho e segurança, no qual o desempenho é alcançado através do compartilhamento do *net namespace* (funcionalidade responsável por proporcionar o isolamento de rede entre diferentes processos que compartilham o mesmo núcleo do SO) entre os contêineres, enquanto a segurança é obtida através do uso reforçado do isolamento dos *namespaces* (SUO et al., 2018). Esta categoria de rede contém 4 modos de comunicação, sendo:

- O *none mode* realiza a conexão do contêiner em uma rede fechada, disponibilizando somente a interface de *loopback*, não havendo conexões com outros contêineres do mesmo servidor e redes externas. Logo, este modo possui alto nível de isolamento e segurança.
- O *bridge mode* é a configuração padrão de comunicação de rede *single-host* utilizada pelo Docker. Desta maneira, todos os contêineres conectados a *bridge* pertencem a mesma subrede virtual e realizam a comunicação entre si através de endereços IP privados. Ainda, cada contêiner possui seu próprio isolamento de *net namespace* e endereço IP. Logo, através do isolamento de rede, os contêineres possuem um nível moderado de segurança. Porém, este método não fornece comunicação externa aos contêineres.
- O *container mode* compartilha um único *net namespace* entre múltiplos contêineres. Dentre os contêineres pertencentes a um grupo, um contêiner é especificado para funcionar como *proxy* e configurado em modo de *bridge*. Desta forma, os contêineres do grupo acessam a rede externa através da interface Ethernet virtual do *proxy*. Como consequência do compartilhamento do *namespace* pelos contêineres, somente um endereço de IP é fornecido ao grupo e cada contêiner é identificado através do IP do grupo mais o número da porta do contêiner. Logo, a comunicação entre os contêineres do grupo é realizada pelo *Interprocess Communication* (IPC). Esta forma de comunicação aplica uma menor sobrecarga na rede, se comparado com o modo *bridge*, além de possuir um nível intermediário de segurança.
- O *host mode* permite que todos os contêineres de um servidor hospedeiro compartilhem o *net namespace* do SO. Logo, todos os contêineres estão visíveis uns aos outros e a sua comunicação é baseada em IPC. Ainda, todos os usuários e contêineres dividem o mesmo endereço IP. Este modo de comunicação apresenta o menor nível de segurança entre todos os modos de comunicação *single-host*.

A comunicação de contêineres que estão em servidores hospedeiros diferentes é denominada comunicação *multi-host*. Neste tipo de comunicação, cada modelo

de rede apresenta um *tradeoff* entre o desempenho, flexibilidade e segurança do contêiner (SUO et al., 2018). Esta categoria de rede contém 4 modos de comunicação, os quais são:

- O modo hospedeiro faz com que a comunicação entre dois contêineres que estão em servidores distintos possam se comunicar através dos endereços de IP dos respectivos servidores hospedeiros. Este modo realiza a comunicação entre apenas dois contêineres. Ainda, não é aplicado isolamento de rede entre os contêineres do mesmo servidor.
- o modo *Network Address Translation* (NAT) configura a rede através do mapeamento do endereço IP privado do contêiner para um porta na tabela do NAT. Desta forma, o endereço de um contêiner é descrito como o IP público do servidor mais a porta do NAT que identifica o contêiner. Embora não necessite de diversos endereços públicos de IP para configurar a comunicação entre aplicações que possuem muitos contêineres, o NAT possui uma sobrecarga no desempenho da rede, pois necessita fazer a tradução de endereço de rede a cada pacote de entrada e saída. Ainda, há limitações no desenvolvimento de soluções que utilizem redes dinâmicas de contêiner.
- A rede de sobreposição configura enlaces virtuais customizados entre os contêineres através de uma rede subjacente. Os contêineres salvam o mapeamento entre seu endereço IP privado e de seu respectivo servidor hospedeiro em um par de chaves-valores, o qual é visível entre todos os servidores da rede de sobreposição. Deste modo, para realizar a comunicação, a rede de sobreposição adiciona uma camada na pilha de rede do servidor.

Consequentemente, quando um pacote é enviado por um contêiner, a rede de sobreposição procura o IP do servidor de destino contido nos pares de chave-valores utilizando o IP privado do contêiner de destino no pacote original, este processo é denominado encapsulamento de pacote. A rede de sobreposição proporciona isolamento de endereços e permite que os contêineres se comuniquem através de IPs privados. Ainda, dentre os modos de comunicação *multi-host*, as redes de sobreposição facilitam realizar alterações e gerenciar a topologia da rede. Entretanto, encapsulamento e desencapsulamento de pacotes são operações custosas, e o possível aumento da quantidade de pacotes transmitidos devido a limitações de *Maximum Transmission Unit* (MTU) na rede subjacente geram uma alta sobrecarga na rede.

- O modo de roteamento implementa um roteador virtual no núcleo do hospedeiro e utiliza o *Border Gateway Protocol* (BGP) para realizar o roteamento dos

dados. A sobrecarga aplicada na rede é menor quando comparada aos modelos de redes de sobreposição e NAT. Entretanto, esta abordagem suporta poucos protocolos de rede, apresentando limitações em sua aplicação em diversos cenários. Ainda, a escalabilidade da rede de comunicação é limitada pelo tamanho da tabela de roteamento.

2.2 ESCALONAMENTO DE CONTÊINERES

Os escalonadores são uma parte crucial para o funcionamento e operação do DC. No provisionamento de contêineres em DCs, a tarefa do escalonador é selecionar hospedeiros apropriados para atender as requisições. O DC deve selecionar a arquitetura de escalonador adequada as características de usuários e necessidades do DC. Os escalonadores de contêineres são divididos em três tipos de arquitetura, as quais são: monolítica, em dois níveis e de estado compartilhado (RODRIGUEZ; BUYYA, 2018). A Tabela 1 sumariza os detalhes das arquiteturas.

Os escalonadores monolíticos são compostos por um único agente que orquestra a alocação de todas as requisições dos usuários, aplicando um único algoritmo de alocação para cada tipo de requisição. Este tipo de escalonador possui uma visão global do ambiente em que está inserido. Porém, a arquitetura apresenta o problema de ponto único de falha e de escalabilidade (RODRIGUEZ; BUYYA, 2018).

Nos escalonadores em dois níveis a alocação da requisição é separada em dois módulos, o coordenador e agente. O coordenador central é responsável por coordenar dinamicamente a quantidade de requisições que cada agente de escalonamento recebe, realizando um controle pessimista das requisições (*i.e.*, oferecer uma requisição a somente um agente). Por sua vez, os agentes de escalonamento efetuam a alocação dos contêineres de acordo com as políticas do DC e cada agente de escalonamento pode executar um algoritmo de escalonamento diferente.

Ainda, para evitar conflitos entre os agentes, o controlador central fornece somente uma requisição para cada agente por vez. Em suma, a arquitetura apresenta uma estratégia menos propensa a erros, mas devido ao escalonador possuir uma única unidade de controle das requisições, o desempenho do escalonador é inferior ao controle otimista (*i.e.*, oferecer uma requisição a múltiplos agentes) (HINDMAN et al., 2011; RODRIGUEZ; BUYYA, 2018).

A arquitetura de estado compartilhado é uma variação da arquitetura em dois níveis. Nesta arquitetura, o escalonador possui agentes de escalonamento executando paralelamente, porém não existe um controlador geral para gerenciar a carga de trabalho dos agentes. Para mediar a competição e falhas durante a execução dos agentes, o estado de cada agente é compartilhado com todos os demais e o controle de concor-

rência otimista é utilizado para atualizar o estado de todos os agentes e as operações de atualização de estado são atômicas. Desta forma o escalonador consegue utilizar diferentes algoritmos de escalonamento em paralelo, cada um podendo executar suas próprias políticas de escalonamento e cargas de trabalho (VERMA et al., 2015).

Arquitetura	Recursos	Controle	Alocação
Monolítico	requisições globais	não possui	controlador central
Dois níveis	particionamento dinâmico	controle pessimista	controlador central
Estado compartilhado	requisições globais	controle otimista	definido por agente

Tabela 1 – Sumário da arquitetura de escalonamento de contêineres.

Para exemplificar a atuação do escalonador de contêineres independente da sua arquitetura, a Figura 4 apresenta 10 requisições, com configurações distintas de CPU e memória, que devem ser escalonadas em 6 servidores homogêneos. O escalonador realiza a alocação das requisições mantendo o menor número de servidores ativos (*i.e.*, reduzindo a fragmentação do DC) enquanto seleciona o servidor que possui a maior capacidade de recursos disponível. O escalonador somente ativará um servidor inativo se os equipamentos previamente ativos não forem capazes de comportar as demandas.

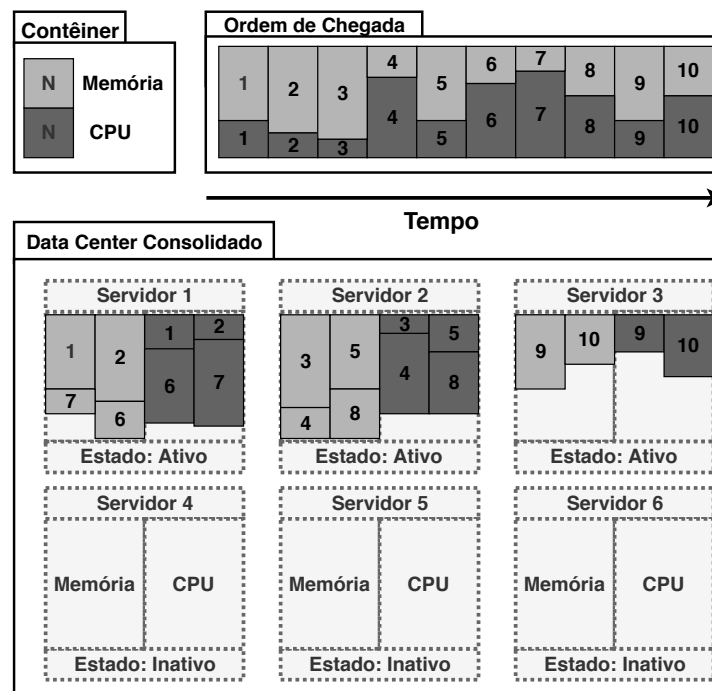


Figura 4 – Escalonamento de contêineres em DC buscando a consolidação de servidores.

Como observado na Figura 4, a decisão do escalonador é sensível à função objetivo selecionada (*e.g.*, aumentar a confiabilidade dos contêineres, redução de custos

e consolidação). Ao receber uma requisição de contêiner informando, por exemplo, uma determinada configuração mínima de CPU, RAM, entre outros parâmetros, o escalonador deve selecionar no DC um hospedeiro apto a oferecer a demanda. Ainda, a tomada de decisão não é trivial. Enquanto alguns contêineres solicitam um elevado volume de processamento, outros podem solicitar majoritariamente recursos de comunicação ou armazenamento.

2.3 ORQUESTRAÇÃO DE CONTÊINERES

Os orquestradores de contêineres oferecem aos usuários a capacidade de criar, alterar, deletar contêineres de forma ágil possibilitando o controle da infraestrutura da aplicação. Um orquestrador é uma coletânea de diversas ferramentas, tornando-se responsável por escalonar contêineres, cuidar do balanceamento de carga, redimensionar e gerenciar o ciclo de vida de contêineres. Ainda, a arquitetura do escalonador utilizado pelos orquestradores varia de acordo com a tecnologia utilizada. Os principais orquestradores revisados no presente trabalho são o Docker Swarm, Apache Mesos e Kubernetes.

2.3.1 Docker Swarm

O Docker Swarm é um orquestrador desenvolvido pelo Docker e sua arquitetura é composta por dois elementos, o gerenciador Swarm e os agentes Docker (DOCKER, 2019a), conforme apresentado na Figura 5.

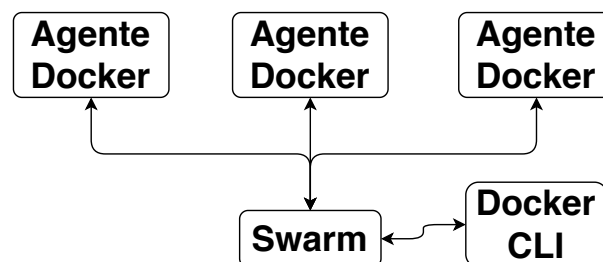


Figura 5 – Arquitetura do Docker Swarm.

O Swarm utiliza múltiplos agentes Docker, denominados *nodes*. Um agente Docker é definido como um servidor que possui uma API remota do Docker disponível para configuração pelo gerenciador Swarm. Deste modo, cada agente pode configurar e executar diferentes imagens Docker dependendo do contêiner que foi designado pelo gerenciador. Visando facilitar o mapeamento dos contêineres entre os agentes, cada agente pode configurar rótulos com informações que são utilizadas pelo gerenciador Swarm para filtrar os *nodes* do DC para executar um novo contêiner (DOCKER, 2019b).

O escalonamento de contêineres pode ser alterado entre três estratégias presentes no Swarm, são elas: (I) o espalhamento, método padrão utilizado no Swarm, escolhe o servidor que possui a menor quantidade de contêineres em execução, independente do seu estado; (II) o *binpack* seleciona o servidor com menor quantidade de CPU e RAM disponível; e (III) o randômico realiza a seleção do servidor aleatoriamente. Caso dois ou mais *nodes* sejam selecionados pela estratégia de mapeamento, o escalonador seleciona aleatoriamente um destes *nodes* (DOCKER, 2019a).

O Swarm utiliza filtros para auxiliar o escalonador na tomada de decisão da seleção do servidor. Para isto, existem duas categorias de filtros, os filtros de nó (restrição e integridade), que operam sobre as características do Docker *host* ou do Docker *daemon*, e os filtros de configuração de contêiner (afinidade, dependência e porta) atuam sobre as características do contêiner requisitado (DOCKER, 2019a). O filtro de restrição associa pares de chave e valor para *nodes* específicos para restringir o escalonamento dos contêineres a somente os servidores que possuam o mesmo rótulo que foi especificado durante a criação do contêiner. Caso nenhum servidor com o rótulo do contêiner possua recursos disponíveis para suportar suas demandas, então o contêiner não será inicializado (DOCKER, 2019c). Enquanto isso, o filtro de integridade evita que contêineres sejam mapeados a *nodes* indisponíveis ou com falhas (DOCKER, 2019c).

O filtro de afinidade é utilizado para criar um agrupamento durante a execução de um novo contêiner. Este filtro possui 3 tipos de agrupamentos, de contêiner, imagem e rótulo. O agrupamento de contêiner cria um elo entre contêineres, fazendo com que o mapeamento destes contêineres seja próximo, e caso um contêiner termine sua execução, todos os demais contêineres que possuam elo também são encerrados. Ainda, o agrupamento de imagem realiza o escalonamento de contêineres somente nos servidores que possuam a imagem específica já está disponível para utilização. Por fim, o agrupamento por rótulo trabalha com o rótulo de contêineres (DOCKER, 2019c).

Por outro lado, o filtro de dependência pode ser utilizado para executar um contêiner que seja dependente de outro contêiner. Tornar um contêiner dependente significa possuir um volume compartilhado com outro contêiner necessitando que exista um elo ou que ambos contêineres estejam na mesma rede. Por fim, o filtro de porta faz com que um contêiner seja mapeado e executado em um *node* caso o servidor possua a porta específica de rede disponível (DOCKER, 2019c).

2.3.2 Apache Mesos

Com o intuito de promover escalabilidade e eficiência, o Apache Mesos aplica uma interface de compartilhamento em um sistema apto para suportar diversos *fra-*

meworks. O Mesos é baseado em uma arquitetura de dois níveis, e seu funcionamento consiste em 3 entidades, o Mestre, os agentes e *Frameworks* (MESOS, 2019), detalhados na Figura 6.

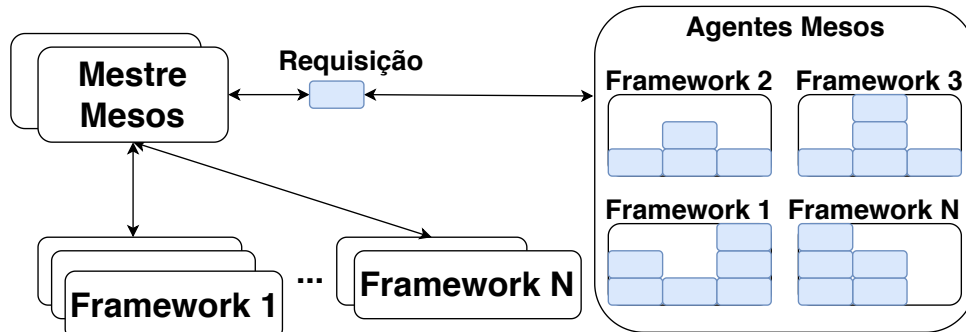


Figura 6 – Arquitetura do Apache Mesos.

Cada agente Mesos, é uma instância que executa no servidor do DC, e tem a função de informar ao mestre a quantidade de recursos disponíveis no servidor. Já o mestre é responsável por gerenciar os agentes que estão em execução, e coordenar a interface de compartilhamento entre os *frameworks*. A interface de compartilhamento é coordenada pelo mestre através de ofertas de recursos (*i.e.*, uma oferta de recursos é definida como uma lista dos recursos disponíveis de múltiplos agentes) (HINDMAN et al., 2011). Desta forma, o mestre decide quantos recursos serão ofertados a cada *framework* de acordo com as políticas definidas pelo DC (*e.g.*, compartilhamento justo e prioridade). Para suportar diversas políticas de alocação, o Mesos permite a utilização de módulos externos via *plugins* (HINDMAN et al., 2011).

2.3.3 Google Kubernetes

O Google Kubernetes é um sistema de orquestração para contêineres Docker usando os conceitos de rótulos e conjuntos para agrupar contêineres em uma mesma unidade lógica, denominada pods. Os contêineres configurados a um mesmo pod são escalonados e executados juntos. Esta abordagem simplifica o gerenciamento do DC e da aplicação de filtros e restrições (GOOGLE, 2019b). A Figura 7 apresenta a arquitetura do Kubernetes.

O mestre é responsável pelo gerenciamento e manutenção do DC, composto por 3 aplicações que executam em um mesmo servidor. As aplicações são: (I) o *kube-apiserver* é uma API Rest, incumbida de validar e configurar os dados para a API de objetos (*e.g.*, pods, serviços, volumes, replicações); (II) o *kube-api-controller-manager* é um *daemon* que incorpora as principais etapas de controle do Kubernetes; e (III) o *kube-scheduler* é um escalonador que considera diversas políticas de alocação, sensível a topologia do DC considerando os requisitos de recursos e de *Quality-of-Service*

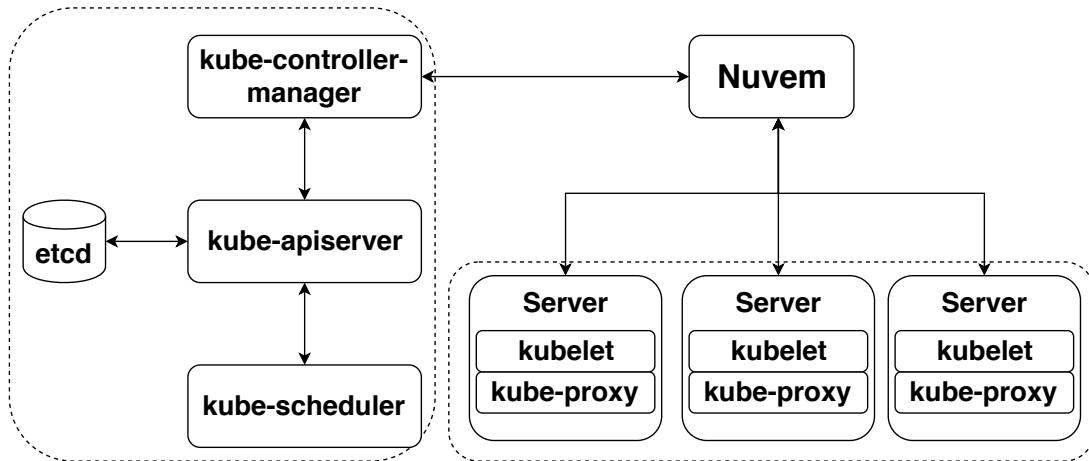


Figura 7 – Arquitetura do Kubernetes.

(QoS) individuais de cada servidor e do DC como um todo. Ainda, o Kubernetes utiliza predicados para auxiliar o escalonador na tomada de decisão. As configurações padrões podem ser sobrescritas, permitindo a configuração de novas políticas de escalonamento. Cada agente é designado como um servidor do DC que possui uma instância do Kubernetes, o qual executa os pods designados pelo mestre (GOOGLE, 2019a).

Os predicados são regras utilizadas para escalonar um novo pod no DC. Caso nenhum servidor satisfaça os predicados estabelecidos pelo pod, então seu estado é marcado como pendente até obter um servidor que contemple os predicados. Existem um total de 5 predicados, os quais são: (I) o *PodFitPorts* examina se o servidor apresenta suporte aos requisitos de rede do pod, sem gerar conflitos; (II) o *PodFitsResources* averigua se o servidor possui recursos suficientes que satisfaçam as requisições do pod; (III) o *NoDiskConflict* verifica se o servidor possui espaço suficiente para comportar o pod e os volumes requisitados; (IV) o *MatchNodeSelector* corresponde ao parâmetro de consulta do seletor definido na descrição do conjunto ; e (V) o *HostName* define o nome do servidor como parâmetro na descrição do pod (GOOGLE, 2019a).

2.4 MÉTODOS DE DECISÃO MULTICRITÉRIO

O escalonamento conjunto de contêineres e suas redes de comunicação requerem uma decisão baseada em vários critérios, analisando variáveis e objetivos usualmente conflitantes. Desta forma, o escalonamento pode ser modelado como um problema de tomada de decisão.

A tomada de decisão é um processo de seleção de uma opção entre diversas alternativas viáveis, utilizada para avaliação de um problema de forma consciente,

com o objetivo de maximizar os resultados de acordo com um critério específico. Antes do desenvolvimento dos métodos multicritérios, os métodos de tomadas de decisão eram baseados em equações matemáticas que solucionavam apenas uma função objetivo, não sendo possível analisar todos os critérios em conjunto para obter uma única resposta (MOREIRA, 2007).

Os métodos de *Multicriteria Decision Making* (MCDM) incorporam as incertezas, subjetividades e ambiguidades nos problemas modelados, aproximando o modelo ao problema real. É necessário que os métodos possuam análises de coerência, para identificar e tratar ambiguidades e conflitos que possam surgir no processo decisório (COSTA, 2002) e são aplicáveis a diversos problemas (*e.g.*, na área de agro-negócios, finanças, planejamento civil, medicina e telecomunicações) (RODRIGUEZ; COSTA; CARMO, 2013).

Os métodos de MCDM, segundo (RODRIGUEZ; COSTA; CARMO, 2013) são classificados em: (I) métodos de sobreposição, os quais criam relações não compensatórias entre as alternativas do problema. As relações não são caracterizadas pela criação de uma única função que envolva todos os critérios; e (II) métodos de teoria da utilidade multiatributo, a qual definem uma função que engloba diferentes funções objetivos em uma única função (*i.e.*, criar uma única função que englobe todos os critérios). Os métodos são individualmente descritos na Tabela 2, apresentando suas respectivas classificações.

Método	Classificação
<i>Analytic Hierarchy Process</i> (AHP)	Teoria da utilidade multiatributo
<i>Elimination and Choice Expressing Reality</i> (ELECTRE)	Método de sobreposição
<i>Measuring Attractiveness by a Categorical Based Evaluation Technique</i> (MACBETH)	Teoria da utilidade multiatributo
<i>Multiple Attribute Utility Theory</i> (MAUT)	Teoria da utilidade multiatributo
<i>Preference ranking organization method for enrichment evaluation</i> (PROMETHEE)	Método de sobreposição
<i>Technique for Order Preference by Similarity to Ideal Solution</i> (TOPSIS)	Teoria de utilidade multiatributo

Tabela 2 – Principais métodos MCDM.

2.4.1 AHP

O processo do AHP trata os problemas de forma hierárquica, possibilitando o reconhecimento e tratamento da subjetividade dos problemas de decisão através dos julgamentos de valor, tratando a subjetividade de forma matemática (SAATY, 1990; COSTA, 2002). Os benefícios do AHP são: (I) realizar a comparação par a par dos elementos da hierarquia, através da etapa de julgamentos de valor; (II) analisar a con-

sistência dos pesos atribuídos através do cálculo do índice de inconsistência; e (III) representar prioridades em intervalos numéricos (BERZINS, 2009).

O AHP é baseado em três princípios analíticos (COSTA, 2002), sendo: (I) a construção de hierarquias, especificando e avaliando de forma eficiente os critérios do problema, identificando os principais elementos para a tomada de decisão, agrupando-os em conjuntos e os estruturando em camadas; (II) a definição de prioridades, realizando a obtenção dos julgamentos e a comparação paritária dos elementos da hierarquia; (III) a consistência lógica, analisando o modelo de prioridades quanto à consistência dos julgamentos elencados no item .

A partir dos princípios do AHP, 4 etapas são elencadas por (SAATY, 1990), que são aplicadas a todos os modelos fundamentados no método AHP, a concepção de hierarquia, a aquisição de dados, a síntese de dados e a análise da consistência do julgamento. A etapa de concepção da hierarquia consiste em identificar o foco principal do problema a ser resolvido, os critérios envolvidos e alternativas viáveis, formando assim uma estrutura hierárquica (COSTA, 2002). O foco principal é o objetivo central do problema, sendo o ponto de partida para realizar a modelagem de problemas de decisão. O conjunto de alternativas viáveis é formado por todas as alternativas A_i que satisfaçam as condições elencadas no foco principal (*i.e.*, $A = \{A_1, A_2, A_3, ..., A_n\}$). Por fim, os critérios são as variáveis que influenciam o resultado final da seleção das alternativas (SAATY, 1990).

A etapa de aquisição de dados descreve o que deve ser julgado dentro da hierarquia e como deve ser feita a análise par a par dos elementos (COSTA, 2002). Com a hierarquia definida, o avaliador deve comparar cada elemento de uma camada hierárquica com cada um dos elementos conectados da camada superior. Para padronizar os valores emitidos durante a comparação dos elementos, a Tabela 3 apresenta a escala que auxilia o avaliador a emitir seus julgamentos de uma forma simples (SAATY, 2000).

Escala Verbal	Escala numérica
Igual preferência (importância)	1
Preferência (importância) moderada	3
preferência (importância) forte	5
preferência (importância) muito forte	7
preferência (importância) extrema	9

Tabela 3 – Escala para julgamento de valores.

A síntese de dados é a etapa responsável por calcular a prioridade de cada alternativa em relação ao foco principal e de cada critério em relação aos demais. Esta etapa é decomposta na obtenção das matrizes de julgamentos, das matrizes de julgamentos normalizados, dos vetores de prioridades médias locais e de prioridades

médias globais (COSTA, 2002). Por fim, a etapa de análise de consistência do julgamento analisa através do índice de inconsistência (IC) e da razão de consistência (RC) o quanto o sistema de classificação mantém as propriedades de transitividade e consistência numérica das matrizes desenvolvidas na etapa de síntese de dados.

2.4.2 ELECTRE

Os métodos da família ELECTRE utilizam a modelagem de preferências baseadas nas relações de sobreclassificação entre pares de ações. O ELECTRE foi desenvolvido na década de 1960 por Bernard Roy e desde então foram desenvolvidas diversas versões (ELECTRE I, ELECTRE II, ELECTRE III, ELECTRE IV, ELECTRE IS e ELECTRE Tree). Todas as versões do ELECTRE mantêm os mesmos fundamentos, possuindo diferenças em termos operacionais e no tipo de problema a ser resolvido (INFANTE; MENDONÇA; VALLE, 2014). O ELECTRE possui o diferencial de ser um método não compensatório (*i.e.*, bons resultados de um critério não compensam resultados ruins de outros critérios) (WHAIDUZZAMAN et al., 2014).

Este método possui dois conjuntos de parâmetros, os quais são: a importância dos coeficientes, e o limiar de veto. Permitindo a manipulação de critérios discretos, podendo ser qualitativos ou quantitativos. As alternativas dependem dos índices de discordância, valores de limiar e gráficos de relacionamentos (utilizados em um processo iterativo para obter o ranqueamento das alternativas) (WHAIDUZZAMAN et al., 2014; MOTA; ALMEIDA, 2007). Uma desvantagem do método ELECTRE é a arbitrariedade com que são determinados os limites de preferência e indiferença utilizados na avaliação do desempenho das alternativas (BRAZ, 2011).

2.4.3 MACBETH

O MACBETH é baseado em uma abordagem multicritério que quantifica a atratividade ou valor das alternativas através da comparação qualitativa dos pares (ANDRADE, 2015; KARANDE; CHAKRABORTY, 2014). Com a comparação das alternativas de um problema não só entre si, mas também com referências, o MACBETH se torna uma ferramenta ideal para a construção dos ranqueamentos. A principal desvantagem deste método é a subjetividade dos problemas, a qual pode ser induzida pelos questionários presentes no MACBETH, fazendo com que ela seja reduzida e não eliminada (BRAZ, 2011; COSTA; CORTE; VANSNICK, 2011).

2.4.4 MAUT

O MAUT é um método que possui uma base teórica sólida, baseado em cálculos matemáticos, facilitando a identificação da violação da coerência e independência entre atributos e alternativas (BRAZ, 2011). O método utiliza uma função denominada

utilidade, que visa substituir o valor associado a um critério para um nível de satisfação do decisor. Os valores associados são calculados diretamente, onde os valores atribuídos a um critério pertencem ao domínio entre 0 e 1. O vetor de decisão é obtido pela multiplicação da matriz de decisão pelo vetor de pesos dos critérios. A síntese de resultados do MAUT acontece através da utilização de análise de sensibilidade, para acessar os impactos de um critério específico na decisão gerada (FREITAS et al., 2013).

2.4.5 PROMETHEE

A família de métodos PROMETHEE foi desenvolvida na década de 1980 por Brans e Vincke, como uma forma melhorada do método ELECTRE, apresentando diferenças no estágio de comparação dos critérios (WHAIDUZZAMAN et al., 2014). As principais características do PROMETHEE são a simplicidade, clareza e estabilidade. Devido ao método ser estruturado em uma modelagem matemática robusta e transparente, apresentando um encadeamento lógico e racional (ARAÚJO; ALMEIDA, 2009).

Durante a etapa de análise de alternativas, o PROMETHEE apresenta funcionamento parecido com o ELECTRE, com um adicional na consideração do grau da melhor alternativa, usando esta informação para eliminar as alternativas dominadas e identificar as alternativas não dominadas, facilitando assim o ranqueamento das alternativas (WHAIDUZZAMAN et al., 2014; SILVA; SCHRAMB; CARVALHO, 2014).

2.4.6 TOPSIS

O método TOPSIS foi proposto inicialmente por Hwang e Yoon (1981) e é utilizado para ranquear alternativas por ordem de preferência. O princípio básico do TOPSIS consiste em escolher uma alternativa que esteja tão próxima quanto possível da solução ideal positiva (*i.e.*, solução construída com o maior valor de cada critério) e o mais distante quanto possível da solução ideal negativa (*i.e.*, solução que possui o menor valor de cada critério) (HWANG; YOON, 1981).

A solução ideal é formada tomando-se os melhores valores alcançados pelas alternativas durante a avaliação em relação a cada critério de decisão, enquanto a solução ideal negativa é composta de forma similar, tomando-se os piores valores (HWANG; YOON, 1981). O algoritmo possui 3 benefícios: (I) consegue tratar um grande número de critérios e alternativas; (II) requer um pequeno número de entradas qualitativas quando comparado ao AHP; e (III) é um método compensatório, permitindo a análise do *tradeoff* entre os critérios modelados.

2.5 TRABALHOS RELACIONADOS

A orquestração de recursos em DCs virtualizados é um tema amplamente pesquisado. Entretanto, o estado da arte sobre orquestração de contêineres ainda está em construção. Assim, esta seção apresenta trabalhos que enfatizam o escalonamento de contêineres e propostas relacionadas com alocação de MVs. Embora os problemas compartilhem fundamentos de formulação, o cenário baseado em contêineres é mais sensível ao tempo de resposta do escalonador (GUERRERO; LERA; JUIZ, 2018). A Tabela 4 resume os trabalhos discutidos, indicando o método multicritério utilizado e o foco do trabalho.

Os autores Souza *et al.* apresentam uma breve contextualização sobre computação em nuvem e do impacto que a alocação de Infraestrutura Virtual (IV) representa dentro das nuvens computacionais. Com o conceito de *Software Defined Networking* (SDN), os equipamentos de rede se tornaram mais flexíveis, gerando diversos benefícios para as IVs, porém a implementação de ambos conceitos dentro das nuvens é uma tarefa complexa. Deste modo, a proposta dos autores era desenvolver um mecanismo de alocação de IVs em nuvens baseadas em SDN (SOUZA et al., 2017; SOUZA et al., 2019). Para representar o problema, foi utilizado o *Mixed Integer Linear Programming* (MILP) e as restrições inteiras foram relaxadas (e.g., latência, largura de banda e necessidades de máquinas virtuais). Os resultados apontam que a latência das IVs podem ser reduzidas enquanto as demais restrições de QoS são garantidas. O artigo apresenta como pontos positivos: (I) a formulação da alocação de IVs em nuvens baseadas em SDN através do MILP; (II) elencar os aspectos positivos e negativos ao se considerar a utilização de SDN para a alocação das IV; (III) considerar uma requisição combinada de *Infrastructure as a Service* (IaaS) e *Network as a Service* (NaaS); e (IV) a seleção dos servidores baseada na latência fim a fim (SOUZA et al., 2017).

As técnicas de MILP oferecem soluções ótimas e geralmente são utilizadas como linha de base para comparações. Porém, devido a complexidade do problema e do espaço de busca as abordagens exatas se tornam computacionalmente inviáveis. Neste contexto, as soluções baseadas em heurísticas se tornam uma alternativa viável (ROST; DÖHNE; SCHMID, 2019). Um escalonador de microsserviços hospedados em contêineres foi proposto buscando o balanceamento de carga de um DC (GUERRERO; LERA; JUIZ, 2018). Como atributos para tomada de decisão, o escalonador considera a carga de trabalho do contêiner e a sobrecarga de comunicação. Para avaliar o desempenho, os resultados obtidos pelos autores foram comparados com o Kubernetes em um DC heterogêneo contendo 400 servidores. As análises indicam que o escalonador apresenta resultados superiores aos obtidos pela política originalmente presente no Kubernetes. Entretanto, é visível a limitação de escalabilidade atingindo um conjunto restrito de servidores.

Guo e seus colegas propuseram um escalonador que utiliza o algoritmo de divisão de vizinhança de contêineres, desempenhando função de equilibrar o balanceamento de carga e reduzir as distâncias entre os contêineres no DC (GUO; YAO, 2018). A análise experimental foi realizada em um DC homogêneo com uma topologia *fat-tree* configurada com $k = 18$, (*i.e.*, 1.458 servidores), comparando o escalonador proposto, com dois outros, onde o primeiro algoritmo utiliza a meta-heurística *Particle Swarm Optimization* (PSO) (SHEIKHAN; MOHAMMADI, 2013) e o segundo a técnica de espalhamento.

Havet *et al.* propuseram um *framework* de escalonamento e monitoramento de contêineres, denominado GenPack (HAVET et al., 2017). O *framework* utiliza os princípios do *generational garbage collection* e monitoramento ativo com o objetivo de reduzir o consumo energético do DC. Para avaliar o desempenho, a proposta foi comparada com o escalonador do Docker Swarm (KAEWKASI; CHUENMUNEEWONG, 2017) em um DC com 13 servidores, utilizando uma amostra de 1% do arquivo de rastros de execuções do Google Borg, considerando as métricas de uso de CPU, de alocações de memória e de consumo de energia. As análises revelam resultados superiores aos obtidos pela política de escalonamento do Docker Swarm, em termos de eficiência energética.

Um escalonador baseado em uma abordagem conjunta de fila de prioridade, AHP e PSO foi proposto por Alla *et al.* (ALLA et al., 2016). O escalonador foi desenvolvido para ser adaptável as características dinâmicas dos ambientes de nuvens computacionais. Para tanto, o escalonador possui 3 fases que executam em *pipeline*. (I) todas as requisições de entrada são armazenadas em uma fila de prioridade dinâmica, para serem ordenadas de acordo com seu nível de prioridade e tempo de espera; (II) as requisições são enviadas para o AHP para serem ranqueadas e enviadas ao PSO; e (III) o PSO realiza a busca pelo servidor mais adequado e então o escalonador realiza a alocação da requisição no DC.

O ambiente utilizado foi um DC contendo 60 servidores e 30 requisições de MVs. Para analisar o desempenho do escalonador, foi utilizado as métricas de makespan e realizada comparações com os algoritmos PSO e o *First Come First Served* (FCFS). Os resultados apontam que o escalonador conjunto reduz o makespan em até 50% e 15% quando comparado aos FCFS e PSO, respectivamente (ALLA et al., 2016).

Paswar (PANWAR et al., 2018) propôs um escalonador baseado nos algoritmos PSO e TOPSIS para escalonar MVs. O escalonador é executado em 2 fases, a primeira utiliza o TOPSIS para calcular o ranqueamento de cada servidor do DC; enquanto a segunda fase, utiliza os resultados do TOPSIS como entrada para o PSO realizar a busca pelo servidor mais adequado e realizar a alocação. O escalonador é comparado com PSO, PSO dinâmico, *Artificial Bee Colony* (ABC) aperfeiçoado e ABC através de 4 mé-

tricas: makespan, tempo de transmissão, custo e utilização de recursos. Através das análises, é possível observar que o escalonador obteve um desempenho melhor em todas as métricas, obtendo resultados de até 75% de melhora quando comparado aos outros escalonadores.

Um destaque é apontado para Nesi e seus colegas (NESI et al., 2018a) que propuseram um *framework* acelerado em *Graphics Processing Unit* (GPU) para realizar a alocação de MVs demonstrando que é possível escalar o problema através do uso de programação paralela. O *framework* é composto por algoritmos de agrupamento (*Markov Cluster Algorithm* (MCL) e K-MEANS), de verificação do menor caminho (Dijkstra e R-Kleene) e a proposta de alocação de grafos, todos implementados em GPU. Ainda, o *framework* possui suporte para as topologias de DC Fat-Tree, Bcube e Dcell. Deste modo, foram realizadas comparações entre o desempenho da proposta desenvolvida e os algoritmos *Best Fit* (BF) e *Worst Fit* (WF), também implementados em GPU. Diferentes ambientes de teste foram considerados, alterando o tamanho e a topologia do DC. Através das análises os autores obtiveram *speedups* de até 6234x em algoritmos específicos (NESI et al., 2018a).

Um estudo baseado em diferentes políticas de ordenação de fila utilizando o easy backfilling foi apresentado por Carstan *et al.* (CARASTAN-SANTOS et al., 2019). O estudo realiza comparações entre cinco base de dados de computação de alto desempenho, utilizando uma amostra de dados acumulados de 180 semanas. Ainda, o objetivo do escalonador é reduzir o tempo de espera e o *slowdown* das requisições. Foram consideradas 4 filas de ordenação durante as análises, sendo: (I) FCFS; (II) Menor estimativa de tempo; (III) Menor quantidade de recursos primeiro; e (IV) Menor área estimada primeiro. Através dos testes, os autores demonstram que um único algoritmo de escalonamento apresenta diferenças no seu comportamento ao trocar a política de ordenação da fila de requisições. Com a utilização da ordenação por menor área estimada foi possível melhorar o desempenho do escalonador em até 80% na métrica de *slowdown* enquanto garante que não ocorrerá *starvation* nas requisições.

Devido a grande quantidade de variáveis e objetivos que devem ser considerados durante o escalonamento de contêineres, Mahmoud e seus colegas desenvolveram um escalonador baseado em algoritmos genéticos denominado Mongas. O escalonador é baseado no algoritmo NSGA-III (DEB; JAIN, 2013), possuindo 2 objetivos para serem otimizados simultaneamente, sendo: (I) aumento da distribuição igualitária de tarefas; (II) redução do consumo energético. Ainda, o escalonador foi desenvolvido para ser tolerante a falhas, não parando sua execução em caso de falha de um servidor do DC. O ambiente utilizado contempla um DC com 400 servidores e utilizadas 1.200 *requisiçõesdeusuários*. Durante os testes, o desempenho do escalonador é comparado ao *Ant Colony Optimization* (ACO), apresentando um aumento no de-

sempenho de 50% na distribuição igualitária de tarefas entre os nós do DC e uma redução no consumo energético de 7% (IMDOUKH; AHMAD; ALFAILAKAWI, 2019)

Seguindo a mesma linha de raciocínio, os autores Hu *et al.* (HU et al., 2020) desenvolveram um escalonador de contêineres, modelando o problema do mapeamento de recursos na forma do problema do fluxo de custo mínimo. Através da utilização de duas heurísticas, a primeira denominada de *dot-product* prioriza os servidores em ordem decrescente, de acordo com o produto das capacidades do contêiner e do servidor. Enquanto a segunda heurística, denominada de *most-loaded* prioriza os servidores em ordem decrescente, de acordo com a utilização de seus recursos. As análises foram realizadas em um DC com 30 servidores heterogêneos e submetidas 2500 requisições, os resultados obtidos do escalonador foram comparadas com o escalonador do Kubernetes e do Swarm. Os resultados apontam que o escalonador reduz o *slowdown* em até 30% enquanto consegue aumentar a utilização dos recursos no DC.

Zhu (ZHU; TANG, 2019) propôs um escalonador adaptável a diferentes objetivos e métodos de ordenação. O escalonador analisado pelos autores busca otimizar a restrição de *deadline* das requisições, utilizando *Directed Acyclic Graph* (DAG) para modelar o problema. O ambiente de testes é composto por 5 servidores e 1000 requisições, comparando a proposta com 4 diferentes escalonadores, (I) guloso; (II) ICPCP; (III) PSO; e (IV) CGA. Os resultados demonstram que a proposta cumpre a premissa de ser adaptável, além de apresentar resultados superiores aos escalonadores comparados ao otimizar a restrição de *deadline*.

Os escalonadores atuais, desenvolvidos para executar em CPU, apresentam limitações de escalabilidade devido a complexidade do problema, realizando análises em ambientes reduzidos de até 1.500 servidores. Ainda, a literatura especializada apresenta uma vasta quantidade de trabalhos que tratam o escalonamento de MVs e suas interconexões de rede. Entretanto, existe uma carência de ferramentas e pesquisas que abordem o escalonamento conjunto de contêineres e suas interconexões de rede. Neste contexto, o trabalho proposto apresenta uma quebra de paradigma nos escalonadores, através da implementação de um escalonador escalável que considera o escalonamento conjunto de contêineres e suas interconexões de rede. O escalonador extrapola os limites encontrados na literatura, considerando DCs com mais de 24.000 servidores (conforme analisado no Capítulo 4). Outro diferencial da proposta é a utilização de métodos MCDM acelerados por GPU para realizar a seleção do servidor mais adequado a comportar uma requisição. Permitindo a análise extensiva dos servidores do DC em relação a um conjunto dinâmico de critérios, suas restrições, considerando o *tradeoff* existente na otimização de diferentes critérios.

Trabalho	Mecanismo do Algoritmo	Foco do trabalho	Aborda Rede	Qtd. Servidores
(ALLA et al., 2016)	abordagem conjunta de fila de prioridades, AHP e PSO	MVs	não	60 servidores
(GUERRERO; LERA; JUIZ, 2018)	algoritmos genéticos	contêiner sobre MV	sim	400 servidores
(GUO; YAO, 2018)	divisão de vizinhança	contêiner	sim	1.458 servidores
(HAVET et al., 2017)	<i>generational garbage collection</i> e monitoramento ativo	contêiner	não	13 servidores
(PANWAR et al., 2018)	abordagem conjunta do TOPSIS e PSO	MVs	não	10 servidores
(SOUZA et al., 2017)	MILP	MVs	sim	64 servidores
(NESI et al., 2018a)	algoritmos em GPU	MVs	sim	16.000 servidores
(CARASTAN-SANTOS et al., 2019)	easy backfilling e filas de ordenação	MVs	não	144 servidores
(IMDOUKH; AHMAD; ALFAILAKAWI, 2019)	NSGA-III	contêiner	não	400 servidores
(HU et al., 2020)	problema do fluxo de custo mínimo	contêiner	sim	30 servidores
(ZHU; TANG, 2019)	Filas de prioridade e DAG	MVs	não	5 servidores

Tabela 4 – Resumo dos trabalhos relacionados.

2.6 CONSIDERAÇÕES PARCIAIS

Os contêineres são uma tendência para clientes e provedores de infraestrutura que possuem a otimização de seus processos como objetivo. Esta solução de virtualização apresenta ganhos de desempenho de RAM, CPU e armazenamento quando comparado a virtualização de MV. Porém, os requisitos de rede são desconsiderados ou parcialmente atendidos pela maioria dos escalonadores revisados. Até mesmo os orquestradores conhecidos (*e.g.*, Kubernetes) consideram a rede como parâmetro secundário e não crítico. Os contêineres são utilizados para modelar aplicativos distribuídos em grande escala e é evidente que a alocação de rede pode afetar o desempenho dos aplicativos (SUO et al., 2018).

As abordagens de análise multicritério podem ser uma solução, uma vez que analisam diversos critérios para tomar uma decisão. Cada método multicritério apresenta sua peculiaridade e forma de tratar um problema, sendo necessário o conhecimento das características do problema para aplicar um método que melhor se adapte. Embora muitas soluções multicritério sejam utilizadas na literatura especializada, não foi encontrado escalonadores baseados em métodos MCDM que considerem a alocação de contêineres e suas redes virtuais. Por fim, dentre os trabalhos revisados, não foram encontrados trabalhos que utilizam escalonadores de contêineres e rede em conjunto com GPUs. Por isso, tal fato revela-se um aspecto original no trabalho aqui apresentado. O Capítulo 3 apresenta em maiores detalhes a especificação do escalonador desenvolvido.

3 PROPOSTA DO ESCALONADOR

O problema abordado apresenta um *tradeoff* entre a proximidade da alocação ótima das requisições no *Data Center* (DC) e a escalabilidade dos escalonadores (CHEN; BATSON; DANG, 2010). A Tabela 5 sintetiza a notação utilizada neste capítulo. A utilização de algoritmos acelerados por *Graphics Processing Unit* (GPU) é uma alternativa viável para contornar a limitação da escalabilidade nos escalonadores atuais (NESI et al., 2018a). Desta forma, foram desenvolvidos dois algoritmos de escalonamento, sem ressubmissão e com ressubmissão (*i.e.*, reinserir uma requisição do usuário para realizar mapeamento futuro), os quais utilizam métodos multicritérios acelerados por GPU para acelerar o escalonamento conjunto de contêineres e redes considerando os requisitos de *Quality-of-Service* (QoS), o *Analytic Hierarchy Process* (AHP) e *Technique for Order Preference by Similarity to Ideal Solution* (TOPSIS).

Notação	Descrição
$G^s(N^s, E^s)$	grafo do DC composto por N^s servidores e E^s enlaces.
$c_u^s[r]$	Vetor de capacidades de recursos do servidor $u \in N^s$.
bw_{uv}^s	Capacidade da largura de banda entre servidores u e v , $uv \in E^s$.
$Req(N^c, E^c)$	Requisição composta por N^c contêineres e E^c enlaces.
$c_i^{min}[r], c_i^{max}[r]$	Capacidades de recursos mínimos e máximos para o contêiner $i \in N^c$.
$bw_{ij}^{min}, bw_{ij}^{max}$	Requisitos mínimos e máximos de largura de banda entre contêineres i e j , $ij \in E^c$.
$c_{iu}^a[r]$	Vetor de capacidades de recursos ($r \in R$) alocado a um contêiner $i \in N^c$ no servidor $u \in N^s$.
bw_{ijuv}^a	Capacidade de largura de banda alocada na comunicação $ij \in E^c$ no enlace $uv \in E^s$.
f_u	O servidor $u \in N^s$ hospeda ao menos um contêiner.
$\phi(t)$	A política de ordenação no tempo t (Eventos).
$s(t)$	Tempo atual de processamento do escalonador (Eventos).
sub_I	Tempo de submissão da requisição (Eventos).
$proc_I$	Tempo estimado de processamento da requisição (Eventos).
$wait_I$	Tempo de espera aplicado na requisição (Eventos).
$deadline_I$	Tempo máximo permitido para a execução da requisição (Eventos).

Tabela 5 – Notação utilizada: i e j são utilizados para indexar os contêineres, enquanto u e v representam os servidores do DC.

Os principais métodos *Multicriteria Decision Making* (MCDM) apresentados na Seção 2.4 possuem diferentes formulações matemáticas e etapas para realizar o ranqueamento das alternativas. Os métodos AHP e TOPSIS foram selecionados pois são os mais favoráveis para implementação na arquitetura *Single Instruction Multiple Data* (SIMD) da GPU. Além de possuírem a capacidade de realizar uma análise multidimensional das alternativas, permitindo a comparação simultânea de diversos servidores de um DC. Proporcionando um método estruturado para decompor o problema e considerar o *tradeoff* dos critérios. Desta forma, ambos algoritmos analisam o mesmo conjunto de critérios c_u^s para um dado servidor u . Além disso, a soma de toda a capacidade de largura de banda bw_{uv}^s com origem em u (dado por bw_u^s) e a fragmentação atual do servidor (f_u) são contabilizados e incluídos na capacidade do vetor c_u^s .

Para isto, os recursos e enlaces do DC são representados por um grafo ponderado não direcionado. A infraestrutura física dos recursos do DC é descrita por um $G^s = (N^s, E^s)$ (N^s denota a capacidade de recursos e E^s reflete a capacidade de enlaces existentes na infraestrutura física do DC). Os mapeamentos dos servidores ($\mathcal{M}^s(i)$) e dos enlaces ($\mathcal{M}^e(i, j)$) ocorrem através da alocação da requisição do usuário no DC que respeite os limites de recursos do DC.

Neste contexto, uma requisição é definida como $Req(N^c, E^c)$ (*i.e.*, um conjunto de contêineres e seus enlaces de comunicação), não possuindo diferenças entre os métodos com e sem ressubmissão. Ainda, R representa todos os recursos considerados nos contêineres. As especificações de recursos e enlaces de comunicação são representadas em intervalos contendo os valores de mínimo e máximo determinados pelo usuário, $\forall r \in R [c^{min}[r], c^{max}[r]]$ e $\forall e \in E^c [bw^{min}[e], bw^{max}[e]]$, respectivamente.

3.1 ESCALONAMENTO TEMPORAL

O escalonador realiza uma análise temporal para realizar o mapeamento das requisições do usuário, desta forma é possível estimar o $wait_I$ da requisição no momento em que ela foi submetida no tempo $sub_I = s(t)$. Deste modo, foram desenvolvidos dois métodos para realizar as análises temporais, a análise com ressubmissão e a sem ressubmissão. Ainda, a estrutura de dados utilizada para representar o intervalo temporal do consumo de recursos das requisições no DC é o mesmo para ambos os métodos, esta estrutura de dados foi denominada de Augmented Forest, descrito na Seção 3.1.1.

Para o funcionamento da análise temporal, cada requisição possui 3 variáveis que fornecem os parâmetros necessários para validar se a requisição pode executar entre os limites temporais estabelecidos. Durante a submissão de uma requisição, o usuário possui um parâmetro $deadline_I$ como opcional. Esta variável representa o

tempo máximo, em eventos, permitido para a tarefa ser executada e encerrada. Caso o usuário não forneça um valor para $deadline_I$ ele será configurado como infinito, para simular que a requisição não possui um tempo limite para sua execução. Desta forma, uma requisição é válida para mapeamento se $sub_I + wait_I + proc_I \leq deadline_I$.

Os métodos sem e com ressubmissão devem ser selecionados pelo provedor nas configurações do escalonador, não podendo ser alterados durante o mapeamento das requisições. Ainda, Os métodos possuem diferenças quanto a verificação da requisição, se deve ser aceita ou rejeitada. O método sem ressubmissão busca realizar o mapeamento da requisição entre o intervalo de tempo $[sub_I, deadline_I - proc_I]$. Caso a requisição não possa ser mapeada neste intervalo, então a requisição é rejeitada. Entretanto, no método com ressubmissão, o mapeamento da requisição é realizado entre o intervalo de tempo $[sub_I + wait_I, deadline_I - proc_I]$. Deste modo, caso a requisição não possa ser mapeada no tempo $sub_I < s(t) < deadline_I - proc_I$, então será adicionado um atraso nesta requisição, através da variável $wait_I$ e então ressubmetendo-a na fila de requisições para tentativa futura de mapeamento.

3.1.1 Augmented Forest

A Augmented Tree é uma estrutura otimizada para trabalhar com intervalos de tempos, baseando-se em árvores binárias balanceadas, especificamente na *Adelson-Velsky and Landis Tree* (AVL) (ADEL'SON-VEL'SKII; LANDIS, 1962). Para manter a estrutura em seu estado ideal (*i.e.*, todos os nós balanceados) as funções de inserção, remoção e balanceamento dos nós possuem os mesmos princípios. A árvore armazena as informações temporais de somente um recurso, fazendo com que cada servidor do DC possua um conjunto de árvores, cada uma responsável por um recurso específico. Diferente da AVL, a Augmented Tree utiliza o intervalo temporal como chave e permite chaves duplicadas. O conjunto de árvores utilizadas por um servidor é denominado **Augmented Forest**. Desta forma, a estrutura auxilia o escalonador a manter as informações da quantidade de recursos que estão sendo consumidos pelas requisições, de forma detalhada, em cada um dos servidores do DC.

Um nó na estrutura é composto por uma tupla contendo o intervalo de tempo em que a tarefa do usuário estará ativa, representada por $[sub_I + wait_I, sub_I + wait_I + proc_I]$, a quantidade de recursos consumida pelo contêiner e os nós da esquerda e direita, conforme apresentado na Figura 8.

As operações de inserção, remoção e balanceamento são similares as operações encontradas em árvores AVL, possuindo a única divergência quanto a chave utilizada, a qual é uma tupla. Durante a execução das funções, o método verifica disponibilidade é utilizado, este método é responsável por verificar se as capacidades dos nós existentes na estrutura no instante $s(t)$ ultrapassam a capacidade máxima de recursos



Figura 8 – Augmented Tree: Composição do nó.

da árvore.

3.1.2 Verificação de Disponibilidade

A Augmented Tree verifica a disponibilidade de capacidade no instante $s(t)$ através de filtragem dos nós da árvores em busca de nós de sobreposição. A filtragem da árvore capta todos os nós que possuam uma sobreposição nos valores de intervalo mínimo e máximo fornecidos e retorna a soma de suas respectivas capacidades. Desta forma, são considerados 8 casos de sobreposição entre nós que possuam as chaves $[min1, max1]$ e $[min2, max2]$, onde: (I) $min1 < min2$ e $max2 < max1$; (II) $min1 < min2$ e $max1 < max2$; (III) $min2 < min1$ e $max2 < max1$; (IV) $min2 < min1$ e $max1 < max2$; (V) $min1 = min2$ e $max2 < max1$; (VI) $min1 = min2$ e $max1 < max2$; (VII) $min1 < min2$ e $max1 = max2$; e (VIII) $min2 < min1$ e $max1 = max2$. Os casos de sobreposição são ilustrados na Figura 9.

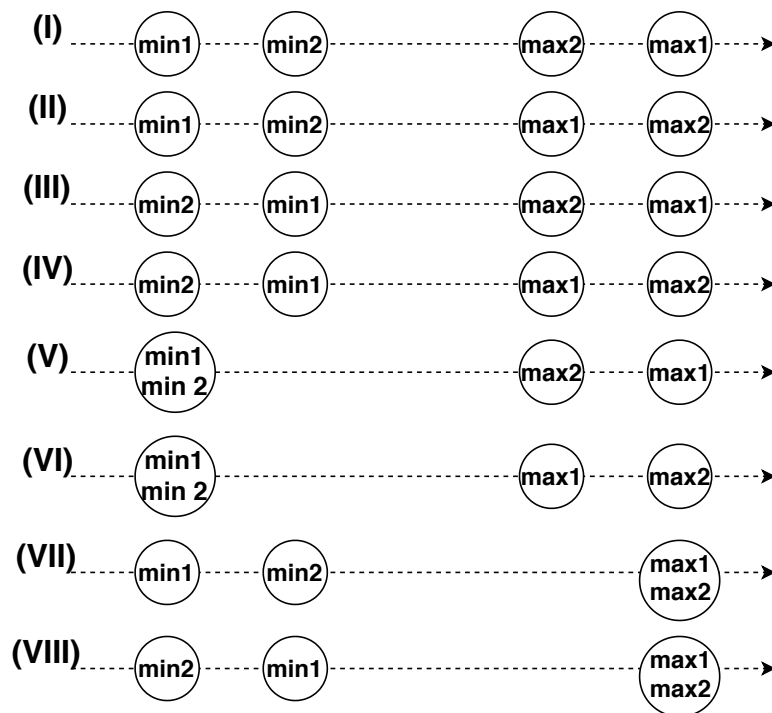


Figura 9 – Augmented Tree: Inserção de nós.

3.1.3 Inserção

A inserção de um nó é realizada no momento em que a requisição é mapeada em um servidor específico. A função de inserção é semelhante a encontrada nas árvores AVL, porém há diferenças no método para selecionar a posição do nó. Adicionalmente, a cada inserção é realizada a verificação se os nós inseridos na estrutura ultrapassam a capacidade que o servidor possui, em caso afirmativo o nó é rejeitado, caso contrário é inserido.

Ao utilizar uma tupla como chave, faz-se necessário realizar uma busca cruzada de valores, primeiro é verificado a posição da árvore em relação ao primeiro elemento da tupla, caso a posição seja encontrada e existam outros nós com o mesmo valor, então a busca continua sua execução utilizando seu segundo elemento. Ainda, caso o novo nó a ser inserido possua uma chave que existe na árvore, então o atributo consumido é somado.

A Figura 10 apresenta a inserção de 6 nós na estrutura. No decorrer da inserção dos 3 nós iniciais, o comportamento da árvore é similar ao encontrado nas árvores AVL. Porém, durante inserção do 4 nó (*i.e.*, intervalo: [7,9] consumindo 15 unidades de recurso) a função de inserção não pode ser completada, pois a árvore não possui recursos suficientes no intervalo de tempo determinado. Como o intervalo temporal do 5 nó já existe na estrutura, os valores da quantidade consumida são somadas e a função encerra com sucesso. Por fim, a inserção do 6 nó ocorre através de duas etapas, primeiro, encontrado a posição relativa do nó na estrutura utilizando o tempo mínimo. Ao encontrar sua posição relativa, a inserção ocorre utilizando através do intervalo máximo.

3.1.4 Remoção

A função de remoção recebe 3 parâmetros, o intervalo mínimo, máximo e a capacidade a ser reduzida na árvore. Primeiro é realizada a busca por um nó que possua a chave igual ao intervalo mínimo e máximo fornecido. Após o nó ser encontrado, a quantidade consumida é reduzida pela capacidade informada, se a capacidade for igual a 0 o nó é removido da árvore. A Figura 11 apresenta a remoção de 2 nós na árvore. A remoção do primeiro nó é realizada de forma similar a encontrada em árvores AVL, devido a quantidade consumida de recursos do nó a ser removido e a quantidade consumida no nó da estrutura serem iguais. Porém, a remoção do segundo nó resulta na subtração da quantidade consumida pelo nó inserido na árvore, pois a quantidade consumida pelo nó é maior do que a quantidade a ser removida.

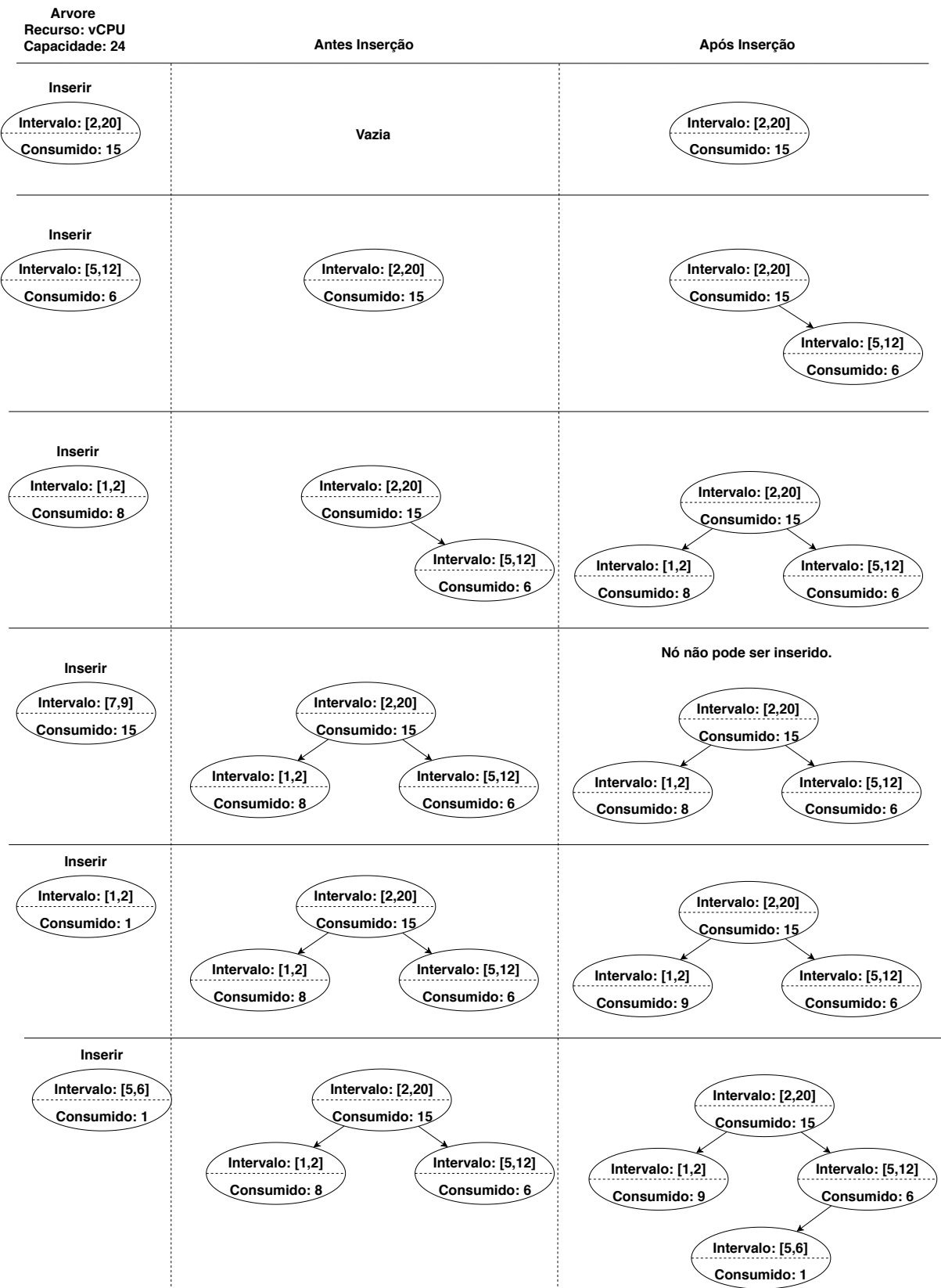


Figura 10 – Augmented Tree: Inserção de nós.

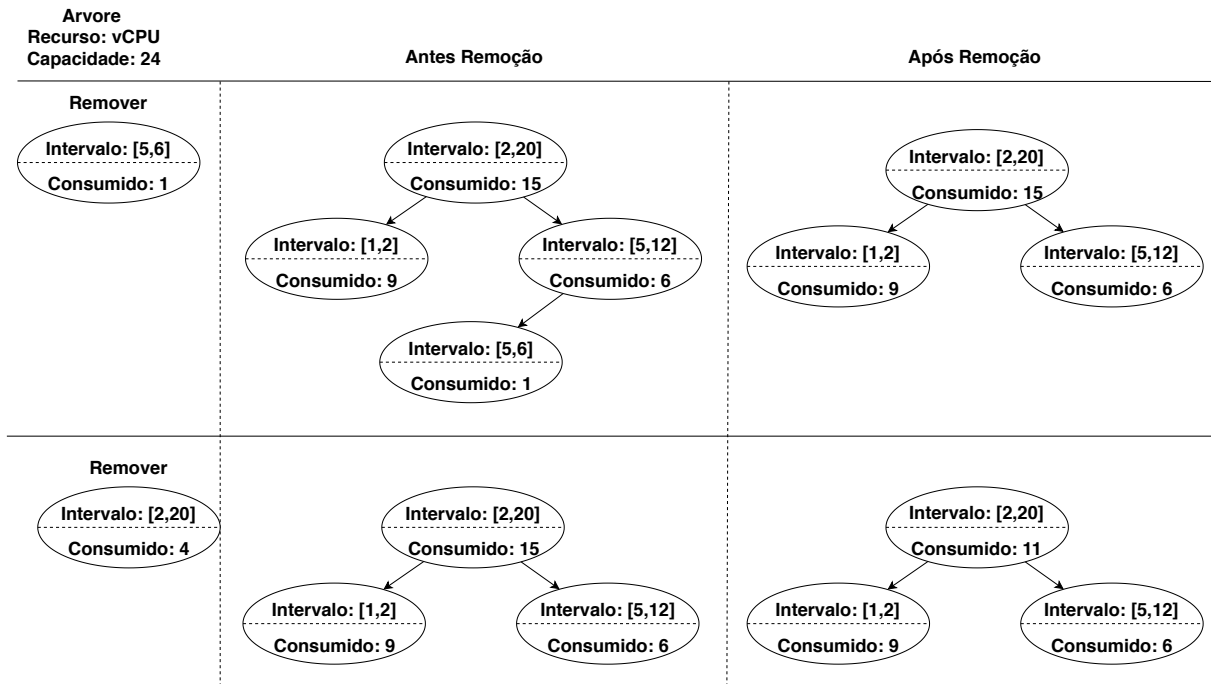


Figura 11 – Augmented Tree: Remoção de nós.

3.2 OBJETIVOS

O resultado de um escalonador é sensível ao conjunto de objetivos a serem otimizados. Isto posto, foram elencados os objetivos de consolidação e QoS para serem otimizados em conjunto. Desta forma, o escalonador deve $\max(\text{QoS})$ e $\min(\text{consolidação})$.

Para reduzir o consumo de energia, os contêineres são agrupados na menor quantidade de servidores possíveis, permitindo o desligamento dos servidores que estão inativos. Esta técnica é denominada consolidação e é alcançada através da minimização da fragmentação do DC, definida como a razão entre o número de servidores ativos $N^{s'}$ (e.g., servidores que hospedam ao menos 1 contêiner) pelo total de servidores no DC. A fragmentação do DC é dada pela Equação 3.1, enquanto o mesmo princípio é aplicado aos enlaces, denotado pela Equação 3.2, onde $E^{s'}$ representa a quantidade de enlaces ativos no DC.

Um contêiner pode ser executado com sucesso com qualquer configuração de recursos que esteja entre os limites máximos e mínimos especificados na requisição. Entretanto, o desempenho ótimo somente é atingido quando os valores máximos são alocados. Neste contexto, as funções de utilidade podem ser aplicadas para representar a melhora nas configurações de QoS do contêiner. O indicador de desempenho QoS é calculado pela razão entre a quantidade de recursos requisitados pelo usuário efetivamente alocados dividido pela quantidade máxima de recursos requisitados.

A otimização da QoS pode ser obtida através de maximizar as Equações de utilidade de requisição $\mathcal{U}(i)$ (Equação 3.3) e utilidade de rede $\mathcal{U}(ij)$ (Equação 3.4) para cada contêiner $i \in N^c$, onde $c_{iu}^a[r]$ e bw_{ijuv}^a representam a capacidade efetiva de nós e enlaces, respectivamente.

$$F(N^s) = \frac{|N^{s'}|}{|N^s|} \quad (3.1)$$

$$F(E^s) = \frac{|E^{s'}|}{|E^s|} \quad (3.2)$$

$$\mathcal{U}(i) = \frac{\sum_{r \in R} \frac{c_{iu}^a[r]}{c_i^{max}[r]}}{|R|}; u = \mathcal{M}^s(i) \quad (3.3)$$

$$\mathcal{U}(ij) = \frac{\sum_{uv \in \mathcal{M}^e(ij)} bw_{ijuv}^a}{bw_{ij}^{max}} \quad (3.4)$$

3.3 RESTRIÇÕES DE CAPACIDADE

Para realizar a alocação combinada de contêineres e rede, um conjunto de restrições binárias, de capacidade e QoS devem ser respeitadas. Um servidor do DC $u \in N^s$ deve ser capaz de comportar todos os contêineres hospedados. Para isto, o somatório dos recursos dos contêineres deve ser menor ou igual a quantidade de recursos que o servidor possui. Enquanto as comunicações de contêineres devem ser contidas nos enlaces do DC, de modo que o somatório da largura de banda de comunicação não exceda a capacidade do enlace. A alocação das capacidades dos recursos entre o intervalo máximo e mínimo para um contêiner $i \in N^c$, aplicando o mesmo raciocínio é aplicado para as comunicações dos contêineres. Finalmente, os contêineres são opcionalmente organizados em pods. Para garantir a integridade das especificações dos pods, todos os contêineres agrupados em um pod devem ser hospedados em um mesmo servidor.

3.4 POLÍTICAS DE ALOCAÇÃO

Para ser adaptável a vários provedores com diferentes configurações de DC, duas estratégias para escalonamento das requisições são propostas. A seleção da estratégia apropriada ocorrerá dinamicamente (*i.e.*, o escalonador obterá o tamanho do DC e selecionará a estratégia através de um conjunto de políticas), conforme apresentado pelas Figuras 12 e 13.

A primeira estratégia, denominada de Alocação Sem Agrupamento é aplicável a pequenos DCs. Nesta abordagem, o método de seleção multicritério é aplicado

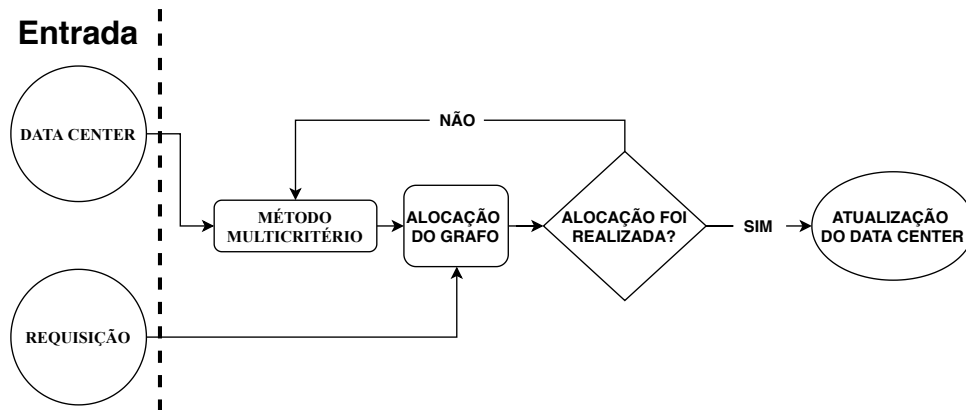


Figura 12 – Alocação sem agrupamento.

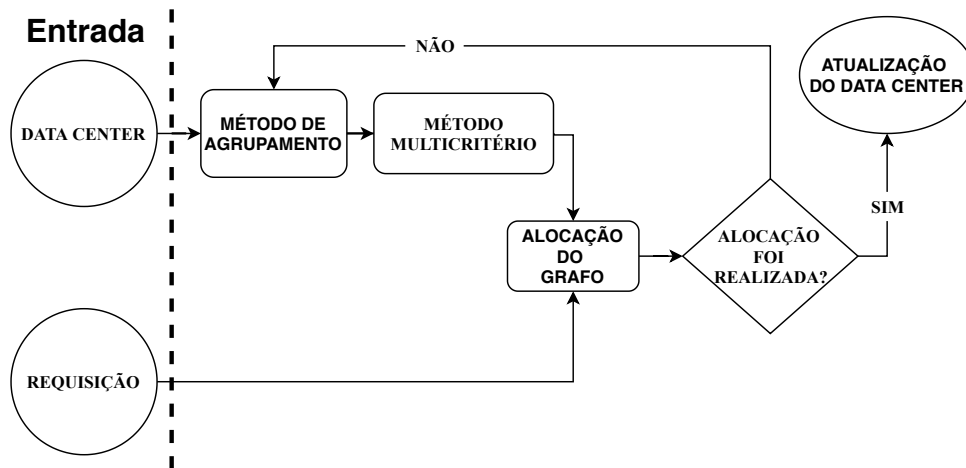


Figura 13 – Alocação com agrupamento de DC.

para todos os candidatos, e após a obtenção do ranqueamento dos servidores e rede, a alocação é realizada. Enquanto, a segunda estratégia denominada Alocação Com Agrupamento de DC é utilizada para agrupar servidores e, em seguida, aplicar o método multicritério para obtenção do grupo mais adequado. A abordagem será recursivamente aplicada até a identificação de um limiar (*i.e.*, número de servidores > 1024) aceitável para o tamanho dos grupos. Se o limiar for atingido, o método multicritério expandirá o grupo em questão ranqueando os servidores e identificando os recursos hospedeiros.

3.5 POLÍTICAS DE CONSTRUÇÃO DE FILAS

A política de ordenação das requisições ϕ adotada pelo escalonador impacta o desempenho do mapeamento das requisições dos usuários no DC (*e.g.*, no tempo de processamento, no *footprint* de RAM e CPU). Nenhuma política de prevenção de *starvation* foi aplicada, uma vez que o escalonador busca realizar o mapeamento de

todas as requisições que possuem o $sub_I(t)$ igual ao $s(t)$, fazendo com que nenhuma requisição no tempo t não seja analisada. Deste modo, foram desenvolvidas 7 políticas que consideram as peculiaridades das requisições de contêineres, descritas na Tabela 6.

Nome	Descrição	Função
FCFS	First-Come-First-Served	$\phi(t) = c_i$
SPF	Smallest Estimated Processing Time First	$\phi(t) = proc_I$
SQF Min	Smallest Lower Resource Requirement First	$\phi(t) = c_i^{min}$
SQF Max	Smallest Upper Resource Requirement First	$\phi(t) = c_i^{max}$
SAF Min	Small Estimated Lower "Area" First	$\phi(t) = proc_t \cdot c_i^{min}$
SAF Max	Small Estimated Upper "Area" First	$\phi(t) = proc_t \cdot c_i^{max}$
SDAF	Small Estimated "Area" Difference First	$\phi(t) = proc_t \cdot (c_i^{max} - c_i^{min})$

Tabela 6 – Políticas de ordenação de filas.

A política **FCFS** considera somente o tempo de chegada da requisição do usuário no escalonador, fazendo com que a primeira requisição a chegar seja a primeira a ser tratada. Enquanto isso, a **SPF** ordena as requisições através da estimativa de tempo de execução das requisições, fazendo com que as requisições que possuem um menor tempo de execução sejam mapeadas antes das requisições que demandam muito tempo computacional. Por outro lado, a política SQF considera o somatório dos recursos mínimos e máximos requisitados pelas tarefas, especificados em **SQF Min** e **SQF Max**, respectivamente.

Porém a estimativa de tempo realizada pelo SPF e de recursos por SQF Min e Max não são justas com todos os usuários de um DC, deste modo as políticas de SAF Min e SAF Max, assim como SDAF foram desenvolvidas. A política SAF tem como objetivo ser equalizar a importância fornecida entre os diferentes tipos de requisições de usuários, realiza a ordenação através de uma estimativa de área entre o tempo de processamento e os recursos mínimos e máximos requisitados pela tarefa, definindo assim o **SAF Min** e **SAF Max**, respectivamente. A fim de explorar a diferença entre os limites de recursos requisitados pelos usuários, foi definida a política **SDAF**, a qual calcula a área entre o tempo de processamento e a diferença entre o limite máximo e mínimo da requisição do usuário.

3.6 ARQUITETURA DO ESCALONADOR

A arquitetura do escalonador em GPU combina métodos multicritérios e de agrupamento, implementando os algoritmos da Figura 14. O *kernel*¹ do escalonador é adaptável às configurações especificadas pelo gerenciador.

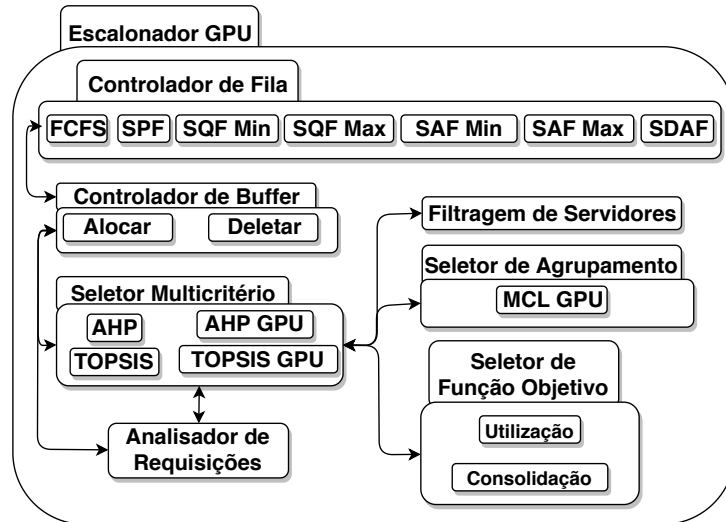


Figura 14 – Arquitetura do Escalonador.

A arquitetura do escalonador é composta por 6 submódulos. O Controlador de fila é responsável por conter os métodos de ordenação descritos na Seção 3.4. O Controlador de *Buffer* é responsável pela orquestração do fluxo de requisições, organizando-as em filas de prioridade, utilizando os métodos contidos no controlador de fila. Este módulo utiliza o Controlador de Fila para obter qual método de ordenação será aplicado nos buffers das requisições. O Seletor Multicritério alterna o método multicritério (*e.g.*, AHP, TOPSIS) a ser executado, de acordo com a política do provedor. Por sua vez, o Seletor de Agrupamento utiliza o *Markov Cluster Algorithm* (MCL) para reduzir o espaço de busca do escalonador. O algoritmo é baseado em fluxos de dados e cadeia de Markov (DONGEN, 2001) e é responsável por realizar o agrupamento dos dados organizados em grafos. A literatura indica que o MCL produz agrupamentos com maior eficiência em topologias de DC quando comparado com outros algoritmos (NESI et al., 2018a; NESI et al., 2018b). O Seletor de Funções Objetivo adequa o escalonador com as necessidades do provedor. Por fim, o Analisador de Requisições recebe os servidores ranqueados e realiza chamadas para o controlador de *buffer*. Após, o analisador retorna o servidor mais adequado, obtido pelo seletor multicritérios, para a requisição em questão.

¹ Nomenclatura usada para identificar funções executadas em GPU.

3.7 DISTRIBUIÇÃO DE PESOS

Os algoritmos AHP e TOPSIS são guiados por um vetor de pesos W que define a importância de cada critério, onde W representa os pesos definidos a cada elemento pertencente a R ($W = \alpha_0, \alpha_1, \alpha_{|R|-1}$) e o somatório dos valores de W deve ser igual a 1. É importante ressaltar que os métodos multicritérios respeitam todas as restrições descritas na Seção 3.3. A Tabela 7 apresenta diferentes composições de W para diferentes cenários.

Cenário	CPU	RAM	Fragmentação	Largura de Banda
Flat	0,25	0,25	0,25	0,25
Agrupamento	0,17	0,17	0,5	0,16
Rede	0,17	0,17	0,16	0,5

Tabela 7 – Esquema de pesos para AHP e TOPSIS.

A análise multicritério com a configuração de agrupamento otimiza o problema visando a consolidação do DC, através da definição de alta importância do critério de fragmentação (50%), enquanto os demais critérios dividem os 50% restantes. Não obstante, a execução com a configuração de rede utiliza o critério de largura de banda como de alta importância (50%) enquanto os demais critérios dividem igualmente os 50% restantes. Esta configuração faz o escalonador selecionar os servidores que possuem a maior largura de banda residual. Finalmente, a configuração Flat define a mesma importância para todos os critérios.

3.8 IMPLEMENTAÇÃO EM GPU

O escalonador utiliza o algoritmo MCL para realizar a construção de grupos dentro da topologia do DC. Para isto, foi utilizado um *framework* que fornece uma interface para a utilização do algoritmo MCL em GPU (NESI et al., 2018a). O agrupamento é utilizado para reduzir o espaço de busca do escalonador. Na topologia Fat-Tree, entende-se por grupo um conjunto de servidores físicos do DC. Deste modo, o algoritmo gera agrupamentos de servidores que possuem quantidades de recursos aproximadas considerando ainda as arestas da topologia. A eficiência do algoritmo é representada pelo tempo computacional necessário para realizar sua tarefa, quando comparado ao algoritmo K-Means (NESI et al., 2018a).

O escalonamento combinado de contêineres e suas redes de comunicação é dividido em três funções principais, a função que representa o controle do fluxo da alocação de uma requisição, a função que realiza o ranqueamento dos servidores do DC e a função que configura o mapeamento da comunicação entre os contêineres da requisição. A Figura 15 apresenta o fluxo do escalonador, detalhando as etapas pertencen-

centes a cada função principal do escalonador e os *kernels* dos métodos AHP, TOPSIS e *widest path*, guiando a explicação do texto desta seção. Enquanto as funções principais são descritas no Algoritmos 1, 2 e 3.

O Algoritmo 1 inicia verificando qual é o método multicritério definido pelo DC (linha 2). Para captar os possíveis grupos formados pelo método *MCL*, o algoritmo utiliza a função *obtemGrupos()* (linha 3), responsável por verificar a política de alocação configurada e se foram formados grupos no DC, em caso positivo o agrupamento é armazenado na variável *grupos*, caso contrário é retornado um conjunto vazio. Seguindo da criação de uma variável denominada *alocado* configurada como *true*, utilizada para fazer o controle da alocação das requisições. Posteriormente, para cada pod pertencente a uma requisição é aplicado a função de escalonamento de contêineres (linha 5). Ao fim do escalonamento do contêiner, é verificado se o mapeamento dos servidores \mathcal{M}^s e dos recursos \mathcal{M}^r não foi realizado com sucesso (linha 6). Em caso afirmativo, a requisição de contêiner é postergada para a próxima execução do escalonador, ou seja, a requisição retorna para a fila e aguardará o encerramento de um provisionamento para escalonar (linha 7) e a variável de controle *alocado* é configurado como *false*.

Após a aplicação do escalonamento é verificado se a requisição de contêiner foi mapeada com sucesso no DC através da variável *alocado* (linha 12). Caso a variável esteja configurada como *true*, o mapeamento foi realizado com sucesso e então o escalonador busca realizar o mapeamento das interconexões de rede da requisição (linhas 13 – 19). Em caso negativo, o escalonador retorna o estado atual do DC sem alocar a requisição no DC (linha 21). A função de escalonar a rede de uma requisição retorna o mapeamento dos enlaces \mathcal{M}^e e da largura de banda \mathcal{M}^{bw} necessária para configurar a intercomunicação entre os contêineres (linha 13). Se o mapeamento foi realizado com sucesso, o DC é atualizado com a nova requisição (linhas 14 – 16), caso negativo a requisição é postergada (linhas 17 – 19). Por fim, o estado do DC é retornado (linha 21).

O Algoritmo 2 inicia com a verificação da existência de grupos no DC (linha 2). Em caso afirmativo, um número reduzido de candidatos deve ser futuramente comparado pelo método multicritério selecionado pelo DC. Ainda, como o Escalonador deve respeitar os objetivos elencados (*i.e.*, consolidação e QoS), é necessário encontrar o grupo que melhor comporta a requisição, desta forma, o método multicritério é executado (linha 3).

Posteriormente, os grupos são percorridos de acordo com a classificação multicritério efetuada. Para cada grupo, os servidores são novamente classificados para identificar a melhor opção de acordo com os objetivos (linhas 4 – 6). O servidor com maior valor de classificação é selecionado (linha 7), e a função *verifica_min_max*

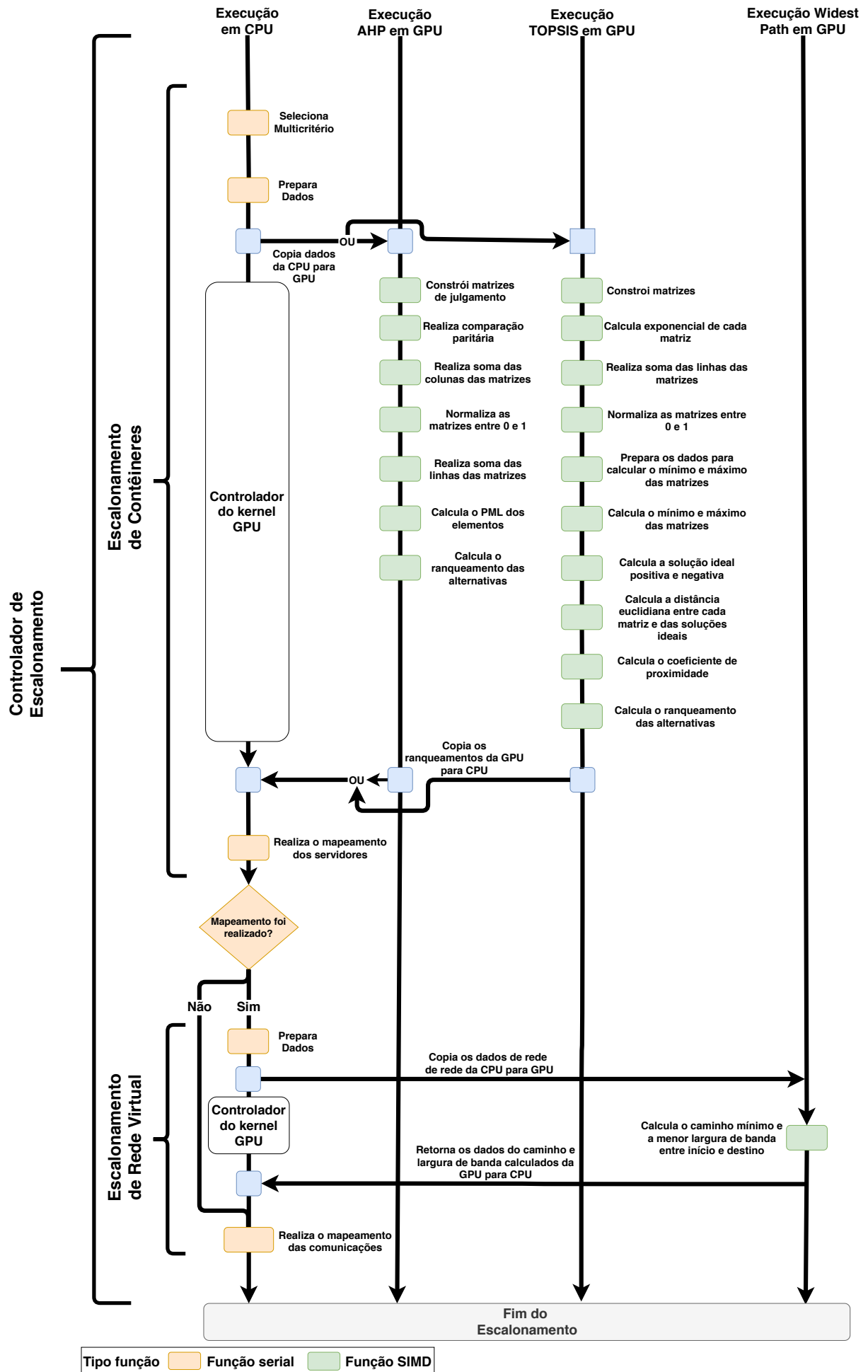


Figura 15 – Funções e kernel do Escalonador.

Algoritmo 1: ESCALONADOR GPU: pseudocódigo do controle de escalonamento.

Entrada: $G^s(N^s, E^s), Req(N^c, E^c)$
Saída: $G^s(N^s, E^s)$

```

1  início
2    Multicriterio ← selecionaMulticriterio()
3    grupos ← obtemGrupos()
4    alocado ← true
5    para  $\forall pod_g \in Req(N^c, E^c)$  faça
6       $(\mathcal{M}^s, \mathcal{M}^r) \leftarrow escalona\_contêineres(N^s, pod_g, grupos)$ 
7      se  $(\mathcal{M}^s = \emptyset) \text{ and } (\mathcal{M}^r \neq \emptyset)$  então
8        posterga_escalonamento( $Req(N^c, E^c)$ )
9        alocado = false
10       quebra_loop
11     fim
12   fim
13   se alocado = true então
14      $(\mathcal{M}^e, \mathcal{M}^{bw}) \leftarrow escalona\_rede(G(N^s, E^s), Req^c, \mathcal{M}^s)$ 
15     se  $(\mathcal{M}^e \neq \emptyset) \text{ and } (\mathcal{M}^{bw} \neq \emptyset)$  então
16       atualiza_dc( $G^s(N^s, E^s), \mathcal{M}^s, \mathcal{M}^r, \mathcal{M}^e, \mathcal{M}^{bw}$ )
17     fim
18     senão
19       posterga_escalonamento( $Req(N^c, E^c)$ )
20     fim
21   fim
22   retorna  $(G^s(N^s, E^s))$ 
23 fim
  
```

(h, pod_i) verifica se o servidor h realmente suporta a configuração do pod i (linhas 8 e 18).

Diferente da análise tradicional de alocação em ambientes virtualizados, o escalonamento de contêineres permite a especificação de valores mínimos e máximos para cada atributo. Assim, a função *verifica_min_max* analisa todos os contêineres pertencentes ao pod, buscando o provisionamento do valor máximo solicitado para cada atributo (CPU e RAM). Se necessário, os valores são gradativamente reduzidos em 10% da diferença entre o valor máximo e mínimo estipulado, respeitando o limite mínimo (*i.e.*, é realizado uma combinação entre os valores máximo e mínimos de cada contêiner). Caso o contêiner não possa ser comportado pelo servidor, o mapeamento \mathcal{M}^r é configurado como nulo (\emptyset). Entretanto, caso todos os contêineres consigam ser mapeados no servidor, implica na que o pod foi mapeado com sucesso. Então é realizada a verificação se o pod é comportado pelo servidor (linhas 9 e 19). Em caso afirmativo, o mapeamento do servidor é armazenado (linhas 10 e 20) para futuramente ser efetuada. Consequentemente, os mapeamentos de recursos e servidores

são informados encerrando a execução (linhas 11 e 21). Caso contrário, o próximo servidor é selecionado (linhas 7–13), e se necessário, outros grupos são analisados (linhas 4–14). Se o DC não possuir grupos formados, o mesmo raciocínio das linhas 7–13 é aplicado as linhas 17–23. Se todos os servidores e grupos forem analisados e nenhuma solução for encontrada, o algoritmo retorna um mapeamento vazio (linha 25).

Algoritmo 2: ESCALONADOR GPU: pseudocódigo do escalonamento de contêineres.

```

Entrada:  $N^s, pod_i, grupos$ 
Saída:  $(\mathcal{M}^s, \mathcal{M}^r)$ 

1  início
2      se ( $grupos \neq \emptyset$ ) então
3          grupos_classificados = Multicriterio(grupos)
4          para  $\forall g \in grupos\_classificados$  faça
5              servidores =  $\{ \forall h \mid h \in g \}$ 
6              servidores_classificados = Multicriterio(servidores)
7              para  $\forall h \in servidores\_classificados$  faça
8                   $\mathcal{M}^r \leftarrow verifica\_min\_max(h, pod_i)$ 
9                  se  $\mathcal{M}^r \neq \emptyset$  então
10                      $\mathcal{M}^s[i] = h$ 
11                     retorna  $(\mathcal{M}^s, \mathcal{M}^r)$ 
12                 fim
13             fim
14         fim
15     fim
16     senão
17         para  $\forall h \mid h \in N^s$  faça
18              $\mathcal{M}^r \leftarrow verifica\_min\_max(h, pod_i)$ 
19             se  $\mathcal{M}^r \neq \emptyset$  então
20                  $\mathcal{M}^s[i] = h$ 
21                 retorna  $(\mathcal{M}^s, \mathcal{M}^r)$ 
22             fim
23         fim
24     fim
25     retorna  $\emptyset$ 
26 fim

```

Após o mapeamento dos pods nos servidores do DC, os enlaces virtuais de comunicação entre os contêineres devem ser configurados, conforme descrito no Algoritmo 3. O algoritmo inicia com a iteração dos contêineres pertencentes a requisição analisada, verificando as comunicações entre os contêineres. Então é analisado se o algoritmo que configura o caminho da comunicação na topologia entre dois contêineres foi realizado com sucesso (linha 6), realizando a verificação do intervalo de largura de banda máximo e mínimo através da variável P_{ij}^e . Se a configuração da interconexão foi realizada então o mapeamento dos enlaces e o mapeamento da maior

largura de banda disponível são atualizados (linhas 7 e 8). Caso contrário é retornado um mapeamento vazio (linha 11). Por fim, caso todas as interconexões de redes sejam escalonadas com sucesso, é tornado a tupla com o mapeamento dos enlaces e largura de banda.

Algoritmo 3: ESCALONADOR GPU: pseudocódigo do escalonamento de rede virtual.

Entrada: $G(N^s, E^s), Req^c, \mathcal{M}^s$
Saída: $(\mathcal{M}^e, \mathcal{M}^{bw})$

```

1  início
2      para  $\forall container_i \in Req^c$  faça
3          para  $\forall container_j \in comunicação(container_i)$  faça
4               $P^e \leftarrow \emptyset$ 
5               $bw_{selecionado} \leftarrow 0$ 
6              se  $widest\_path(\mathcal{M}^s[container_i], \mathcal{M}^e[container_j],$ 
7                   $bw_{ij}^{min}, bw_{ij}^{max}, \&P_{ij}^e, \&bw_{selecionado}) = true$  então
8                   $\mathcal{M}^e[i] \leftarrow P_{ij}^e$ 
9                   $\mathcal{M}^{bw}[i] \leftarrow bw_{selecionado}$ 
10             fim
11         senão
12             retorna  $\emptyset$ 
13     fim
14 fim
15 retorna  $(\mathcal{M}^e, \mathcal{M}^{bw})$ 
16 fim

```

Os algoritmos utilizados para realizar o escalonamento de contêineres e as redes de comunicação foram modelados para executar em GPU. A utilização da GPU proporciona um alto grau de paralelismo e taxa de memória. Porém, desenvolver algoritmos de grafos para a GPU não é trivial, uma vez que estes utilizam técnicas que possuem baixo desempenho na arquitetura SIMD da GPU. Logo, para conseguir extrair o máximo de desempenho em GPU é necessário reestruturar os algoritmos e suas estruturas de dados, segregando-os em subfunções (*kernel*²) para executarem em grupos de *threads*.

3.8.1 AHP

O algoritmo AHP otimizado para a GPU foi reestruturado utilizando somente vetores (*i.e.*, todas as matrizes $N \times N$ são convertidas para vetores N^2) em sua estrutura de dados e técnicas de memória compartilhada entre as *threads* e execução em *pi-*

² Nomenclatura utilizada para definir funções executadas em GPU.

peline dos *kernels*. Deste modo, adaptações na indexação e formulação matemática foram necessárias para o funcionamento adequado do método, descritas a seguir.

Primeiro, dois vetores (M_1 e M_2) são construídos através da combinação dos critérios e alternativas (*i.e.*, segundo e terceiro nível da hierarquia do AHP) aplicando os pesos definidos na Tabela 7. Em outras palavras, $M_1[v] = W[v]; \forall v \in R$ enquanto $M_2[v \times |N^s| + u] = c_u^s[v]; \forall u \in N^s; \forall v \in R$.

Posteriormente, a comparação paritária é aplicada para todos os elementos da hierarquia. Se $M_1[v \times |R| + u] > 0$, o valor $M_1[v \times |R| + u] - M_1[i \times |R| + u]$ é atribuído; Além disso, se o valor for < 0 , é aplicado o valor $\frac{1}{M_1[v \times |R| + u] - M_1[i \times |R| + u]}$; e 1, caso contrário. O mesmo raciocínio é aplicado para M_2 , indexado por $v \times |N^s|^2 + i \times |N^s| + u$. Em seguida, ambos os vetores são normalizados.

Sucessivamente, o algoritmo calcula o ranqueamento local de cada elemento na hierarquia (L_1 e L_2), como descritos nas Equações (3.5) and (3.6), $\forall u, v \in R; \forall i, j \in N^s$. Por fim, a prioridade global (PG) das alternativas é calculado para guiar a seleção dos servidores, descrita por $PG[v] = \sum_{x \in N^s} P_1[v] \times P_2[v \times |N^s| + x]$.

$$L_1[v \times |R| + u] = \frac{\sum_{x \in R} M_1[v \times |R| + x]}{|R|} \quad (3.5)$$

$$L_2[v \times |N^s| + j] = \frac{\sum_{x \in N^s} M_2[v \times |N^s|^2 + i \times |N^s| + x]}{|N^s|} \quad (3.6)$$

Frente a arquitetura da GPU e buscando otimizar o desempenho do método, foram desenvolvidos 5 *kernels* responsáveis por uma atividade específica, evitando conflitos de memória e comunicação entre as *threads* da GPU. Ainda, o algoritmo AHP possui um algoritmo que executa em *Central Processing Unit* (CPU) controlando o fluxo dos *kernels* em execução na GPU.

3.8.2 TOPSIS

O algoritmo TOPSIS segue os princípios de otimizações e estrutura de dados do método AHP (*i.e.*, utilizando vetores de tamanho N^2). Ainda, devido ao funcionamento do TOPSIS, foi possível utilizar técnicas de otimização de memória compartilhada entre as *threads* da GPU e redução paralela com evasão de conflitos de memória. Devido as adaptações necessárias para o funcionamento do método, a seguir são descritos as etapas e formulação matemática do TOPSIS em GPU.

O TOPSIS realiza o ranqueamento dos servidores candidatos do DC é realizado em 5 fases. Inicialmente, o vetor de avaliação M correlaciona os recursos do DC (N^s) e os critérios (R), sendo $M[v \times |N^s| + u] = c_u^s[v]; \forall u \in N^s; \forall v \in R$, o qual é norma-

lizado posteriormente. A próxima etapa é a aplicação do esquema de pesos sobre os valores de M , da forma $M[v \times |N^s| + u] = M[v \times |N^s| + u] \times W[v]; \forall u \in N^s; \forall v \in R$.

Baseado em M , dois vetores são compostos com os valores máximos e mínimos de cada critério, representado por A^+ (*i.e.*, o limite superior) e A^- (*i.e.*, o limite inferior). O TOPSIS requer o cálculo da distância euclidiana entre M e os limite superiores e inferiores, compondo Ed^+ e Ed^- . Finalmente, o vetor de coeficiente de proximidade é calculado para todos os servidores do DC, obtido por $Rank[u] = \frac{Ed^- [u]}{Ed^+ [u] + Ed^- [u]}$; $\forall u = N^s$, e posteriormente o vetor resultado é ordenado em ordem decrescente, indicando os candidatos selecionados (CHEN, 2000).

O método desenvolvido possui um total de 11 *kernels* que realizam todas as operações e configurações necessárias de quantidade de memória compartilhada de forma dinâmica. Ainda, um algoritmo em CPU foi desenvolvido para controlar o fluxo e realizar as chamadas dos *kernels*.

3.8.3 Interconexões de Rede

A configuração dos caminhos de interconexão de rede entre os contêineres de uma requisição do usuário é realizada pelo algoritmo do caminho mais amplo. Este algoritmo é uma modificação do algoritmo de *Dijkstra*, o qual realiza a busca dos servidores hospedeiros entre dois contêineres (*i.e.*, $\mathcal{M}^s[contêiner_i]$ e $\mathcal{M}^s[contêiner_j]$). O método realiza a busca gulosa em todos os caminhos possíveis na topologia do DC. O objetivo deste algoritmo é selecionar um caminho entre dois servidores, maximizando a largura de banda da aresta que possui a menor largura de banda do caminho. Após o caminho que ótimo ser gerado, é verificado se o enlace com a menor vazão do caminho satisfaz as necessidades de comunicação dos contêineres. Se necessário, os valores são gradativamente reduzidos em 10% da diferença entre o valor máximo e mínimo estipulado, respeitando o limite mínimo.

Para extrair o máximo de desempenho da GPU o algoritmo deve ser segregado em funções, de modo que cada *kernel* realize a mesma operação entre as *threads* em execução. Deste modo, o algoritmo de caminho mais amplo apresenta problemas de otimização na GPU, uma vez que existe muita comunicação entre as *threads* em execução, acarretando em perda de desempenho.

Visando escalar o algoritmo e reduzir o tempo computacional necessário para realizar o mapeamento de comunicação de uma requisição, o algoritmo foi portado para a GPU em um único *kernel*. Ao utilizar o algoritmo em uma única *thread*, o *kernel* executa diversas *threads* em paralelo, porém com início e destinos da busca diferentes, realizando N buscas simultaneamente. Então, os valores obtidos através da execução paralela são armazenadas para aproveitamento futuro, reduzindo a quantidade

de execuções sequenciais do algoritmo.

O algoritmo foi estruturado usando uma estrutura de dados diferente da presente nos métodos AHP e TOPSIS. Para reduzir o espaço de armazenamento necessário na GPU, uma configuração especial de vetor é utilizada. O princípio utilizado na construção do vetor considera que entre os servidores de origem u e destino v , onde $u < v$ os enlaces $u \rightarrow v$ e $v \rightarrow u$ são iguais. Desta forma, o algoritmo necessita de $\frac{N \times (N - 1)}{2}$ elementos.

3.9 CONSIDERAÇÕES PARCIAIS

Neste capítulo foi apresentado o escalonador conjunto de contêineres e suas interconexões de redes, detalhando seus objetivos, restrições, políticas de alocação, arquitetura e implementação. Os métodos para selecionar o servidor mais adequado a comportar uma requisição foram desenvolvidos em GPU, visando tornar o escalonador escalável.

As políticas de alocação são um fator crucial para o desempenho computacional do escalonador. A escolha da política de alocação utilizada é realizada de forma automática, através da verificação do tamanho do DC. Esta verificação se faz necessária devido a complexidade da construção de grupos. Caso o DC possua menos de 1024 servidores, a aplicação direta dos métodos multicritérios necessita de um tempo de computação menor do que através da abordagem que utiliza os métodos de agrupamento. Entretanto, caso a quantidade de servidores seja superior a 1024, a utilização dos métodos de agrupamento reduz drasticamente o tempo computacional necessário para realizar a alocação das requisições no DC.

Devido as diferenças de arquitetura utilizadas pela CPU e GPU, torna-se necessário a remodelagem dos algoritmos e estrutura de dados utilizadas. Os algoritmos em GPU necessitam segregar o algoritmo em diversas subetapas, portando cada uma a um *kernel* específico. Além de utilizar técnicas de divisão e conquista, de memória compartilhada e de evitar conflito de acesso de memória, são essenciais para otimizar os algoritmos e prover escalabilidade ao escalonador.

O Capítulo 4 discute a análise de escalabilidade e desempenho do escalonador e realiza uma comparação de qualidade da alocação entre o escalonador e os algoritmos utilizados nos orquestradores atuais. Ainda, no Capítulo 5 são detalhadas as análises do escalonador referente aos métodos sem e com ressubmissão e das diferentes ordenações das filas de requisições.

4 ANÁLISE EXPERIMENTAL COM REQUISITOS DE REDE E PROCESSAMENTO

Neste capítulo são apresentadas as análises do escalonador proposto. A fim de analisar o escalonador em diferentes perspectivas, foram realizados 3 experimentos, a análise de escalabilidade, de qualidade do escalonamento e de rede. A análise de escalabilidade tem o objetivo de apresentar o tempo computacional necessário para o escalonador alocar uma determinada quantidade de requisições em diferentes tamanhos de *Data Center* (DC), detalhada na Seção 4.1. Além de ser escalável, o escalonador deve ser sensível aos cenários de pesos para otimizar diferentes funções objetivos, desse modo foi realizada a análise de qualidade do escalonamento. Nesse experimento, é exposto o comportamento do escalonador ao utilizar diferentes cenários de pesos através de duas etapas, descrito na Seção 4.2.

O escalonador e um simulador de eventos discretos foram desenvolvidos na linguagem de programação C++, utilizando o compilador GCC *v* 8.3 e CUDA *V* 10.2. Ainda, o ambiente utilizado para realizar as análises consiste em uma GPU NVIDIA RTX 2080 *TI* (11GB) hospedada por uma máquina com um processador Intel *i7-8700K* com 32GB de RAM DDR4 (3200MHz) executando GNU/Linux Archlinux 5.1.7.

4.1 ANÁLISE DE ESCALABILIDADE

A análise de escalabilidade utiliza a métrica de tempo de execução para indicar o tempo de resposta do processamento completo para o escalonador alocar um conjunto de requisições no DC, contabilizando a representação dos dados e a execução dos algoritmos em CPU e *Graphics Processing Unit* (GPU).

Para quantificar a escalabilidade do escalonador, duas dimensões de parâmetros foram variadas. A primeira dimensão é o tamanho da topologia do DC, seguindo o padrão de topologia *Fat-Tree*, sendo avaliada para valores entre $k = 4$ e $k = 46$ (*i.e.*, de 16 a 24.334 servidores). A segunda dimensão representa o conjunto de contêineres em uma requisição, variando entre 1 até 262.144 contêineres. O limite da configuração de *Fat-Tree* foi definido de acordo com a limitação de memória da GPU utilizada (neste caso, 11GB). Além disso, as requisições são constituídas por contêineres com configurações homogêneas e que solicitam poucos recursos. Desta forma, nenhuma requisição deixará de ser atendida, pois existem mais recursos físicos que o somatório de recursos virtuais requisitados. Uma combinação completa de possibilidades indica a fronteira aceitável do tempo de processamento para escalonar as requisições em diferentes configurações de topologia.

Os resultados apresentados nas Figuras 16 e 18 utilizam uma escala de cores

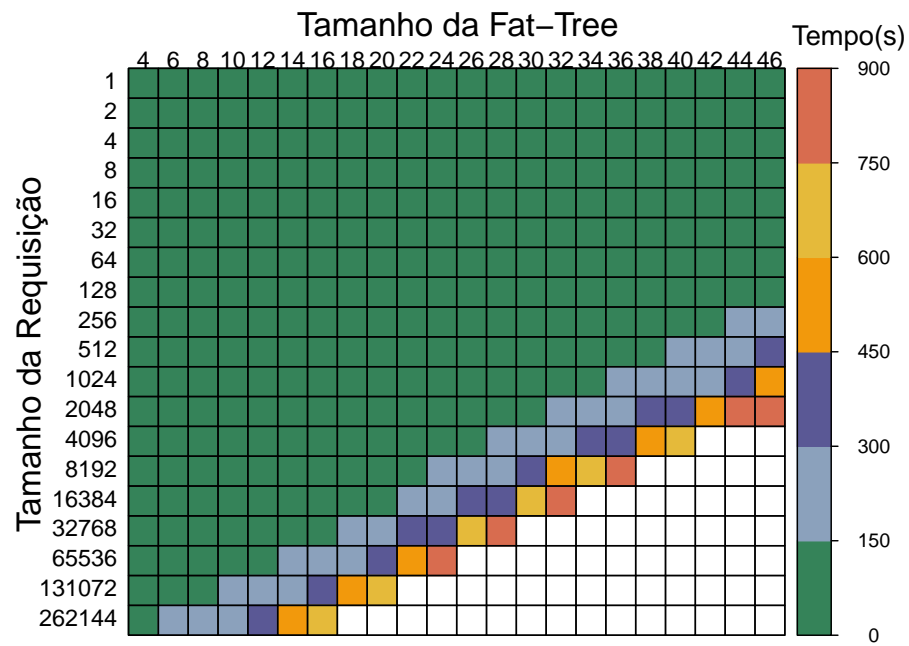
para representar o tempo de processamento do escalonador. Para caracterização do cenário, o teto de processamento foi arbitrariamente configurado como 900 segundos, conforme indicado pela escala de cores. As células na cor branca indicam que o processamento não foi concluído no tempo configurado. O eixo X representa o tamanho da *Fat-Tree* adotada, enquanto o eixo Y representa a quantidade de requisições de contêineres. As análises dos gráficos são realizadas através da leitura dos valores máximos obtidos nos eixos Y e X . A qualidade dos valores é interpretada através da maior quantidade de células preenchida no gráfico, assim como o tempo necessário para realizar a alocação (*i.e.*, quanto menor a coloração na escala de cores, melhor o resultado).

Inicialmente, de acordo com as Figuras 16a e 16b, a utilização dos métodos multicritérios são alternativas viáveis para realizar o escalonamento de contêineres em grandes topologias de DC. Os algoritmos *Analytic Hierarchy Process* (AHP) e *Technique for Order Preference by Similarity to Ideal Solution* (TOPSIS) fornecem a escalabilidade máxima de *Fat-Tree* $k = 46$, tratando requisições compostas por até 2.048 e 16.384 contêineres, respectivamente.

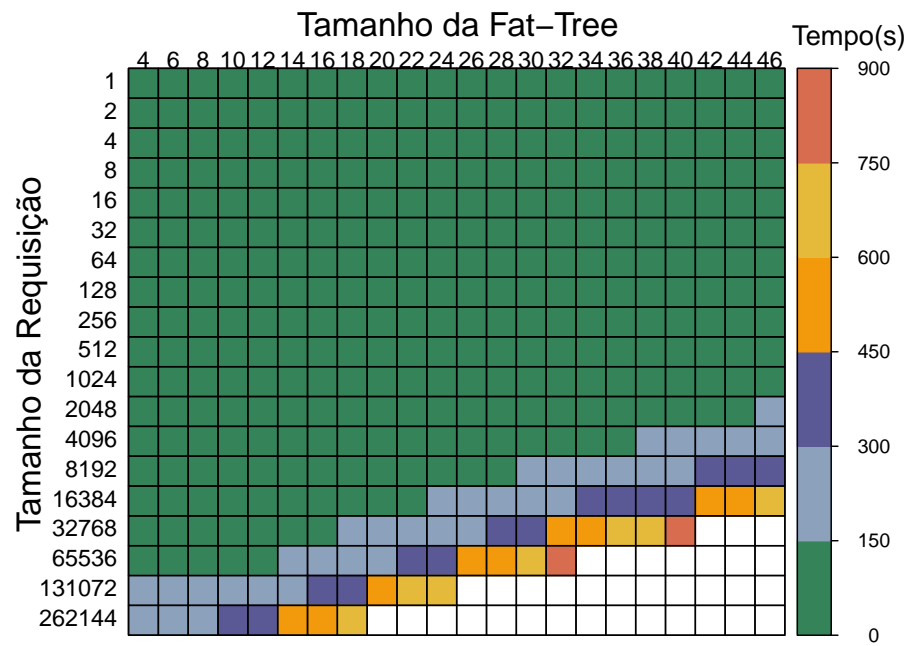
Visando aumentar a escalabilidade do escalonador, o método *Markov Cluster Algorithm* (MCL) é utilizado para formar agrupamentos no DC. A Figura 17 apresenta o tempo necessário para o escalonador realizar o agrupamento do DC, não considerando os tempos de classificação e alocação entre contêineres. Deste modo é observado que o MCL é favorável para utilização conjunta com os métodos de agrupamento, uma vez que consegue formar os agrupamentos do DC em todo o espaço de busca proposto. Porém, realizar a execução do agrupamento do DC a cada nova requisição é inadequado e potencialmente desnecessário, ou seja, um agrupamento deve ser utilizado para escalonar diversos contêineres. Ainda, é importante ressaltar que o Algoritmo MCL em CPU não é viável para ser utilizado, devido ao tempo computacional para realizar os agrupamentos superar as restrições de tempo existentes no problema abordado (NESI et al., 2018a).

Em contraponto, a Figura 18 apresenta a abordagem híbrida de método de agrupamento e multicritério. O MCL é executado para realizar o agrupamento do DC uma única vez durante a execução do escalonador, e a cada nova requisição processando somente o método AHP ou TOPSIS em GPU. Deste modo, a abordagem torna-se escalável, permitindo os algoritmos AHP e TOPSIS tratarem um DC com topologia $k = 46$ com até 131.072 e 65.536 contêineres, respectivamente. Obtendo um aumento de 1.339% e de 673% na quantidade de servidores, e um aumento na alocação de até 6.400% e de 400% mais a quantidade de requisições quando comparado com as abordagens em GPU.

Ainda, o método AHP ao utilizar os agrupamentos apresenta uma redução na



(a) AHP em GPU.



(b) TOPSIS em GPU.

Figura 16 – Tempo de processamento para algoritmos sem agrupamento em diferentes configurações da topologia *Fat-Tree*.

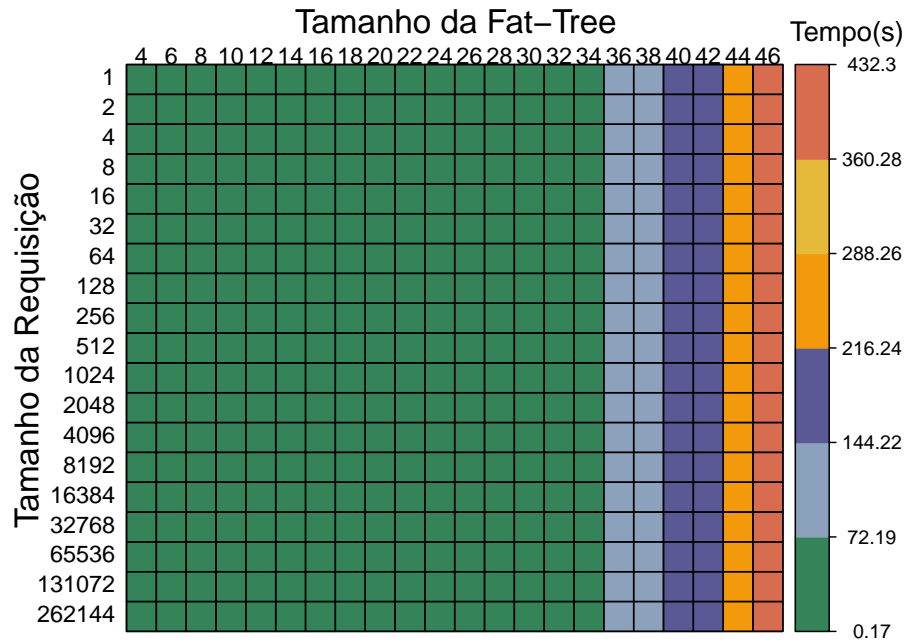
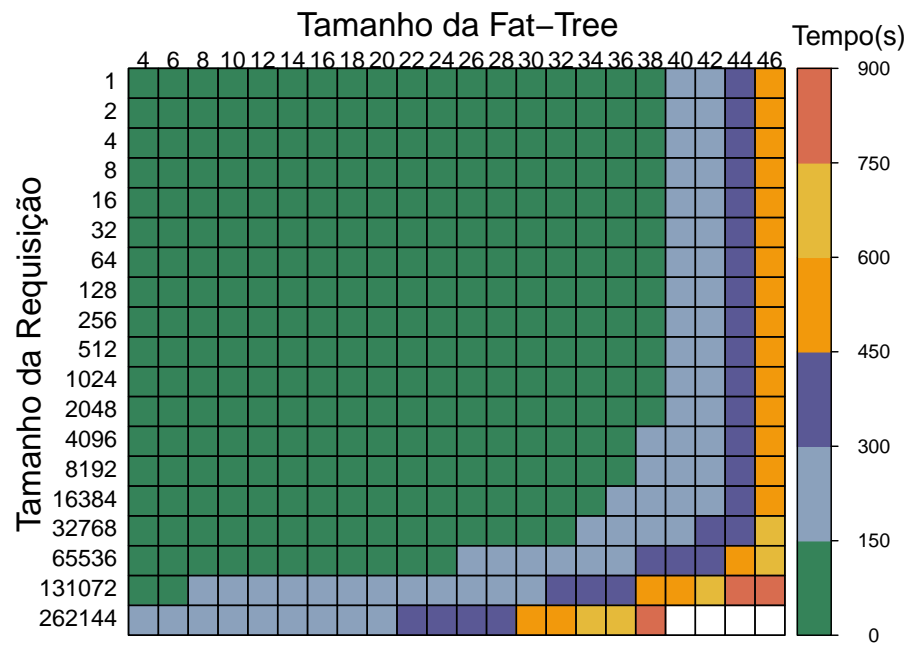


Figura 17 – MCL em GPU.

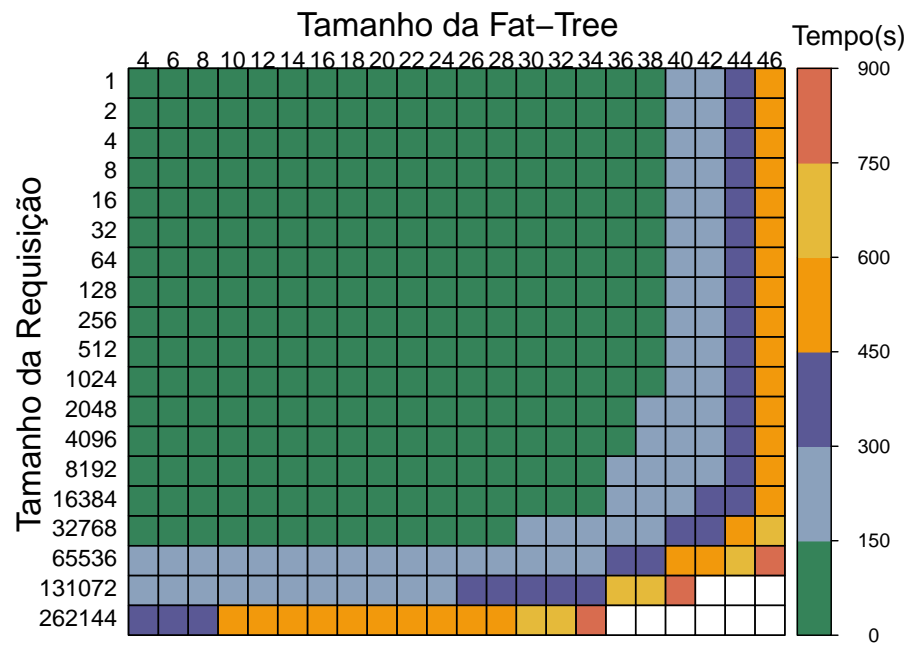
quantidade de operações realizadas durante a comparação paritária. Para uma topologia *fat-tree* $k = 40$ são de 48.000 operações para cada *thread* para o AHP em GPU e de 1.824 operações para a abordagem híbrida. Deste modo, a abordagem híbrida reduz em 26x a quantidade de operações realizadas na comparação paritária quando comparado com o AHP GPU. Este comportamento ocorre devido ao método de agrupamento gerados em torno de 43 grupos, onde cada grupo contém em média 372 servidores. Por fim, os resultados revelam que o escalonador supera as dimensões previamente consideradas para escalonamento de contêineres (Seção 2.5), permitindo o processamento de DCs e requisições em larga escala.

4.2 ANÁLISE DA QUALIDADE DO ESCALONAMENTO

A análise de qualidade do escalonamento possui o objetivo de complementar a análise de escalabilidade (Seção 4.1), investigando o comportamento do escalonador quando aplicado a um conjunto heterogêneo de requisições de contêineres e sob diferentes cenários de pesos. Desta forma, a análise é realizada em duas etapas, primeiro é observado o comportamento do escalonador ao considerar somente contêineres durante a alocação das requisições (Seção 4.2.1), seguido da análise conjunta de contêineres e interconexões de comunicação (Seção 4.2.2).



(a) AHP com agrupamento em GPU.



(b) TOPSIS com agrupamento em GPU.

Figura 18 – Tempo de processamento para algoritmos com agrupamento em diferentes configurações da topologia *Fat-Tree*.

4.2.1 Análise de Contêineres

A análise de contêineres possui o objetivo de investigar o comportamento do escalonador quando aplicado a um conjunto heterogêneo de requisições de contêineres e sob diferentes cenários de pesos. Para realizar a análise, foi definido um conjunto de 3 métricas a serem coletadas e avaliadas, a fragmentação e *footprint* do DC e o atraso das requisições. A fragmentação e o *footprint* analisam a visão do DC, enquanto o atraso na execução de uma requisição quantifica a perspectiva dos usuários, indicando o impacto da decisão do método multicritério no tempo total de execução das requisições.

O cenário utilizado durante a realização da análise conta com requisições oriundas do rastro de execução do Google Borg (REISS; WILKES; HELLERSTEIN, 2011). Especificamente, as requisições correspondem a 1 hora do rastro, selecionado entre 6480s e 10080s do arquivo original, totalizando 1.313.596 requisições. Originalmente, o rastro do Google Borg informa somente dados abstratos para representar o uso de CPU e RAM. Assim, foi realizada uma normalização adaptando as requisições às capacidades disponíveis nos servidores, resultando em dois cenários. Arbitrariamente, a normalização ocorreu considerando 10% da capacidade máxima do servidor (Conjunto 1) e considerando a capacidade máxima de um servidor (Conjunto 2).

Neste contexto, as requisições são mapeadas em um DC homogêneo, constituído por servidores compostos por 24 *Central Processing Units* (CPUs) com 256GB de *Random Access Memory* (RAM), dispostas em duas configurações de *Fat-Tree*, com tamanho de $k = 20$ e $k = 44$. O limite superior ($k = 44$) é ligeiramente diferente do aplicado na Seção 4.1 devido ao aumento dos dados manipulados (deve respeitar o limite da GPU, 11GB). Deste modo, duas distribuições de pesos similares as descritas na Seção 3.7 foram utilizadas, porém o critério de largura de banda não é considerado, uma vez que a análise não considera os requisitos de rede. A primeira representa a busca pela consolidação ao indicar um maior peso para o critério de fragmentação possuindo peso 0,5, enquanto os critérios CPU, RAM possuem peso 0,25, enquanto a segunda distribuição, denominada padrão, considera todos os critérios com o mesmo peso.

Os valores da média e desvio padrão para a métrica de fragmentação são apresentados na Figura 19. O comportamento dos valores observados advém dos dados de entrada utilizados, o qual submete as requisições em rajadas ao escalonador. Ao verificar o primeiro conjunto de requisições no qual a relação da quantidade de recursos requisitada pela capacidade de um servidor do DC é $\leq 0,1$, é possível observar que o escalonador com a configuração padrão, tende a espalhar as requisições entre os servidores, ativando todos os servidores do DC. Por outro lado, alterando a distribuição de pesos, o escalonador considera a fragmentação como um critério crítico.

Assim, o comportamento do algoritmo é alterado, e passa a concentrar as requisições alocadas nos servidores já ativos, obtendo porcentagem de fragmentação de 61,7% e 5,79% para as topologias *Fat-Tree* $k = 20$ e $k = 44$, respectivamente. Neste conjunto de requisições nenhuma requisição foi postergada, uma vez que todas as requisições conseguiram ser mapeadas no DC no momento de sua submissão. Porém, ao analisar o segundo conjunto de requisições, no qual a relação da quantidade de recursos requisitados pela quantidade de recursos de um servidor do DC é $\geq 0,25$, a fragmentação máxima é obtida em todos os casos. Neste conjunto, todos os servidores estão ativos, porém há pouco ou nenhum recurso ocioso, havendo requisições que não conseguiram ser mapeadas no DC no momento de sua submissão, necessitando postergar seu escalonamento.

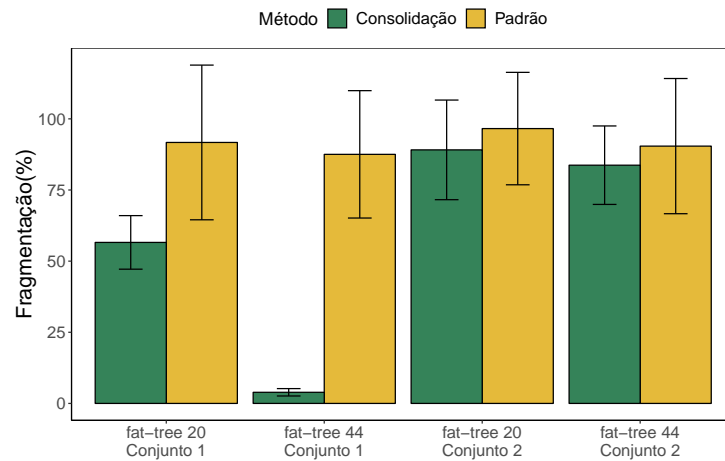


Figura 19 – Análise da fragmentação do DC resultante das variações dos cenários de pesos.

Entretanto, a Figura 20 apresenta a distribuição acumulada do atraso aplicado pelo escalonador, considerando o segundo cenário de requisições. Durante o processo de alocação, o tamanho do DC, a aplicação das requisições em rajadas e seu tempo de duração, foram variáveis que formaram um ambiente que gerasse uma sobrecarga no DC. Deste modo, impossibilitando o mapeamento das requisições de todas as requisições submetidas no tempo específico. A quantidade de requisições que foram mapeadas é inferior a quantidade de requisições que foram postergadas, acarretando no surgimento de um segundo gargalo no sistema, fazendo com que a quantidade de requisições postergadas crescesse rapidamente.

Complementando a análise, os valores de *footprint* da memória e de CPU dos servidores são sintetizados na Tabela 8. Através da análise do *footprint* é possível observar que as requisições submetidas ao escalonador ocupam uma proporção maior de CPU frente aos recursos de memória. A alteração da distribuição de pesos não afeta o *footprint* observado durante a aplicação do primeiro conjunto (Conjunto 1) de requisições, uma vez que todas as requisições são mapeadas na sua submissão, não ha-

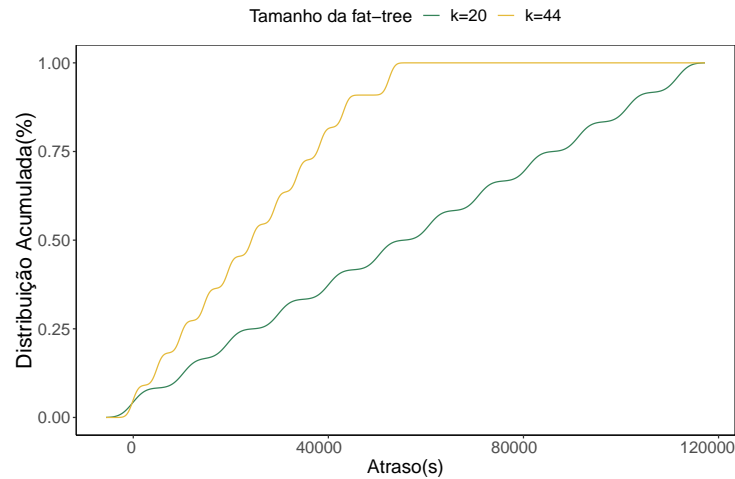


Figura 20 – CDF do atraso no atendimento das requisições para o Conjunto 2.

vendo nenhuma iteração na qual o DC ficasse sobrecarregado. Porém, quando o DC fica sobrecarregado (Conjunto 2), é necessário a postergação de algumas requisições, impactando alterações no *footprint* ao se alterar a distribuição de pesos. Ao considerar a função objetivo de consolidação, é constatado que a aplicação do escalonador utilizando um peso maior para fragmentação consegue diminuir o número de servidores ativos do DC enquanto busca reduzir a quantidade de recursos ociosos nos servidores já ativos (indicado pelo *footprint*).

Requisições	<i>fat-tree</i>	Configuração AHP	Footprint CPU	<i>Footprint</i> RAM
Conjunto 1	$k = 20$	Padrão	53,82%	20,18%
		Consolidação	53,82%	20,18%
	$k = 44$	Padrão	5,05%	1,90%
		Consolidação	5,05%	1,90%
Conjunto 2	$k = 20$	Padrão	97,37%	35,54%
		Consolidação	98,71%	36,03%
	$k = 44$	Padrão	98,57%	36,03%
		Consolidação	98,89%	36,17%

Tabela 8 – Comparação do *Footprint* de RAM e CPU.

Por fim, o tempo de computação necessário para alocar um conjunto de requisições é sensível a ocupação do DC, representado pelas métricas de *footprint* e fragmentação. Uma vez que o DC esteja com o *footprint* próximo de 100%, o escalonador pode não conseguir mapear todas requisições nos servidores, fazendo com que algumas requisições sejam postergadas. Este fato é apresentado através da Tabela 9, a qual apresenta a comparação do tempo médio necessário para escalonar os dois conjuntos de requisições definidos, no qual o conjunto de requisições que sobrecarregam o DC (Conjunto 2), necessita de um tempo de computação superior para a mesma topologia quando comparado ao tempo para alocar o conjunto de requisições que não sobrecarregam o DC (Conjunto 1).

Requisições	<i>fat-tree</i>	Tempo Total	Tempo Individual
Conjunto 1	$k = 20$	300min	0,01s
	$k = 44$	1250min	0,05s
Conjunto 2	$k = 20$	2550min	0,17s
	$k = 44$	5400min	0,25s

Tabela 9 – Tempo de execução do escalonador nos cenários analisados.

4.2.2 Análise de Rede e Contêineres

A análise de rede possui o objetivo de investigar o comportamento do escalonador ao realizar a alocação conjunta de contêineres e suas redes virtuais, otimizando os objetivos descritos na Seção 3.2. Para isto, são utilizadas as métricas de utilidade de requisição $\mathcal{U}(i)$, de utilidade de rede $\mathcal{U}(i, j)$, atraso das requisições, fragmentação dos enlaces e da rede. A avaliação considera um DC homogêneo composto por servidores com 24 núcleos, 256 GB RAM e interconectados por uma topologia *Fat-Tree* de tamanho $k = 20$ e todos os enlaces possuem 1Gbps de largura de banda. Um conjunto de 6.000 requisições foram submetidas ao escalonador, cada uma sendo composta por 4 contêineres executando até 250 eventos. Para compor uma requisição, até 50% dos contêineres de uma única requisição são agrupados em pods, enquanto os requisitos de largura de banda entre um par de contêineres é configurado em até 50Mbps, representando uma aplicação com alto requisito de rede.

Para analisar a qualidade do escalonamento foram realizadas comparações com os algoritmos *Best Fit* (BF) e *Worst Fit* (WF) em cada um dos cenários de pesos modelados na Seção 3.7. Os resultados são sumarizados na Tabela 10 e nas Figuras 21 e 22, apresentando os dados de tempo de execução, utilidade de requisições e rede, aplicação de atraso em relação a fragmentação do DC e a fragmentação do DC, respectivamente. A Figura 21 representa no eixo X a quantidade de fragmentação de rede do DC (*i.e.*, fragmentação da largura de banda do DC), enquanto o eixo Y exibe o atraso aplicado em cada requisição submetida ao escalonador. Os resultados apresentam que os algoritmos multicritérios possuem uma pequena variação no atraso da requisição, agrupando os dados em altos percentuais de fragmentação. Por sua vez, o algoritmo BF impõe maiores atrasos as requisições, resultando em um pequeno percentual de fragmentação, abaixo de 30% da fragmentação de rede. O WF e BF geram longas filas de requisições, impactando diretamente no tempo computacional total necessário para realizar o escalonamento das requisições dos usuários.

Entretanto, a análise isolada da Figura 21 é insuficiente para apresentar com precisão o comportamento do escalonador, desta forma a qualidade dos valores deve ser interpretada através de uma análise cruzada entre a Tabela 10 e as Figuras 21 e 22. Os melhores resultados são aqueles que apresentam o menor tempo de alocação médio mantendo os valores de fragmentação da rede e dos links do DC baixos. Conse-

Algoritmo	Cenário	# Eventos	Tempo de Alocação Médio (s)	$\mathcal{U}(ij)$	$\mathcal{U}(i)$
BF	-	2.462	79,38	100%	96,89%
WF	-	1.007	47,80	100%	99,41%
AHP	Flat	949	9,45	100%	98,22%
	Agrupamento	936	7,51	100%	99,10%
	Rede	928	6,90	100%	98,41%
Topsis	Flat	894	3,67	100%	98,85%
	Agrupamento	916	3,84	100%	99,01%
	Rede	892	3,48	100%	98,94%

Tabela 10 – Tempo de execução, utilidades de enlaces e rede para BF, WF, AHP e TOPSIS.

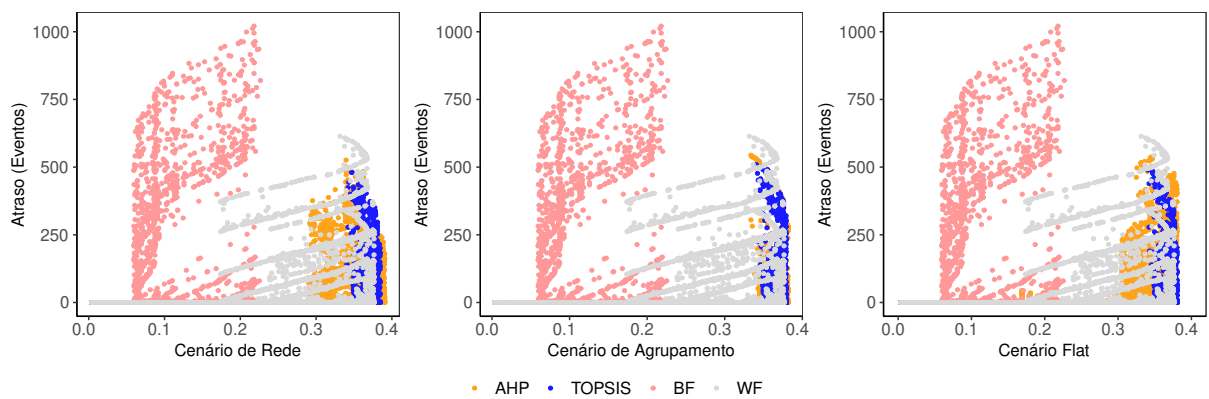


Figura 21 – Fragmentação da rede do DC.

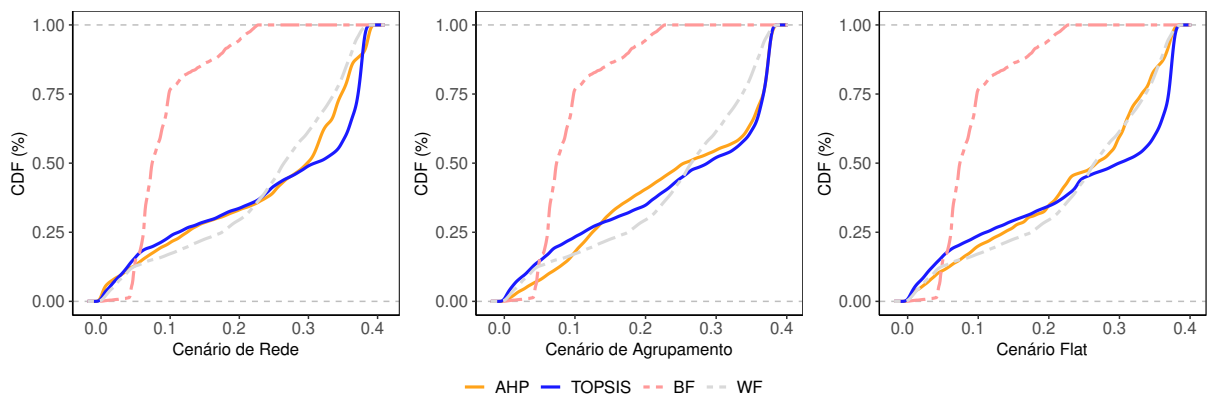


Figura 22 – Fragmentação dos enlaces do DC.

quentemente, é possível observar que apesar dos métodos AHP e TOPSIS possuírem uma fragmentação superior ao encontrado nos métodos BF e WF, os métodos multicritérios apresentam tempos de alocação médio das requisições inferior a 10 segundos, apresentando a eficiência destes algoritmos em realizar o escalonamento reduzindo o tempo de escalonamento em até 22 vezes, enquanto obtém um aumento na fragmentação de no máximo 2 vezes.

Ao analisar a utilidade da requisição (Tabela 10), os algoritmos multicritérios realizam o escalonamento das requisições priorizando a combinação entre os requisitos máximos e mínimos dos contêineres, resultando no aumento do número de requisições alocadas no DC. Em contraponto, o WF tende a alocar o valor máximo das requisições, enquanto o BF fornece os valores mínimos requisitados. Enquanto os algoritmos multicritérios aumentam o número de contêineres no DC reduzindo o atraso total das requisições, a fragmentação de rede apresenta um comportamento similar com o algoritmo WF, exibido na Figura 22. Enquanto isso, o BF mantém a fragmentação da rede pequena devido aos longos atrasos aplicados nas requisições. Desta forma é possível observar que os algoritmos multicritérios apresentam melhores resultados de consolidação ao comparar com o WF e BF, devido a sua capacidade de alocar mais recursos no DC mantendo a fragmentação similar ao WF. Ainda, ao analisar o comportamento do escalonador ao aplicar diferentes cenários de pesos, é possível concluir que o cenário de rede apresenta melhores resultados. Pois ambos os métodos multicritérios apresentam percentuais de fragmentação de rede próximos aos obtidos pelo cenário de agrupamento, e possui uma média de alocação menor que os demais cenários.

4.3 CONSIDERAÇÕES PARCIAIS

Através das análises apresentada neste capítulo, foi possível constatar o desempenho computacional do escalonador e comparar a qualidade da alocação das requisições com os algoritmos BF e WF. Através da análise de escalabilidade é possível concluir que o desempenho do escalonador depende diretamente do tamanho do DC e do conjunto de requisições. Ainda, é constatado o limite de memória que a GPU possui, fazendo com que seja necessário aplicar técnicas para redução da quantidade de dados na memória (*e.g.*, trocar a precisão de *double* para *float*), além da reutilização de vetores e estruturas. A partir dos resultados obtidos, foi constatado que o desempenho do escalonador é afetado não somente pelo tamanho do conjunto de requisições, mas também pelo comportamento em que estas são submetidas.

Ao comparar o comportamento do escalonador para realizar o mapeamento dos contêineres e redes virtuais e somente o mapeamento de contêineres, é possível concluir que ao considerar as redes virtuais na tomada de decisão do escalonador,

uma sobrecarga é aplicada, afetando não somente o tempo de execução do escalonador e o atraso aplicado às requisições, mas também a utilização do DC, aumentando a quantidade de recursos ociosos. Logo, é notável a necessidade de ferramentas que utilizem os dados de rede para realizar o escalonamento conjunto de contêineres e redes virtuais.

5 ANÁLISE EXPERIMENTAL COM REQUISITOS TEMPORAIS

Neste capítulo são apresentadas as análises temporais do escalonador proposto. A fim de analisar o comportamento do escalonador em diferentes perspectivas sobre o escalonamento sem e com ressubmissão, foram realizados 2 experimentos, a análise de qualidade do escalonamento temporal com termino antecipado e limite de execução (*deadline*), e a análise de qualidade do comportamento das ordenações das requisições dos usuários.

5.1 PROTOCOLO EXPERIMENTAL

O *Easy Backfilling* é um algoritmo de escalonamento amplamente utilizado em escalonadores (e.g., IBM, CTC SP2, HPC2N, KTH, LLNL Thunder) devido a simplicidade de implementação e garantir que todas as requisições sejam verificadas (i.e., evita *starvation* das requisições) enquanto busca reduzir a quantidade de recursos ociosos do *Data Center* (DC) (CARASTAN-SANTOS et al., 2019). Deste modo, as análises apresentadas neste capítulo comparam os resultados obtidos pelo escalonador com o algoritmo *Easy Backfilling*, utilizando duas políticas distintas de escalonamento, de consolidação e de disponibilidade.

Ainda, as análises otimizam os objetivos descritos na Seção 3.2, utilizando as métricas de utilidade de requisição $\mathcal{U}(i)$, de utilidade de rede $\mathcal{U}(i, j)$, atraso das requisições, fragmentação dos enlaces e da rede. As avaliações consideram um DC homogêneo composto por servidores com 24 núcleos, 256 GB RAM e interconectados por uma topologia *Fat-Tree* de tamanho $k = 12$ e todos os enlaces possuem 1Gbps de largura de banda.

Com o objetivo de abstrair as necessidades dos usuários do DC, o artigo (MARCONDES et al., 2016) apresenta a configuração de um conjunto de aplicações que possuem diferentes requisitos de largura de banda, enquanto os autores (IMDOUKH; AHMAD; ALFAILAKAWI, 2019) apresentam as configurações de vCPU e RAM de aplicações amplamente utilizadas em DCs. Com o objetivo de modelar um conjunto de requisições fiel ao utilizado pelos usuários, as configurações de vCPU, RAM e largura de banda foram sumarizados na Tabela 11. Deste modo, foi modelado um conjunto de 35.000 requisições que foram submetidas ao escalonador, cada requisição é composta por 4 contêineres, executando até 200 eventos. Ainda, os contêineres foram gerados através de uma distribuição uniforme de acordo com as configurações apresentadas na Tabela 11.

A análise de qualidade tem o objetivo de apresentar as diferenças entre uti-

vCPU	RAM	Largura de Banda
[0, 1; 1]	[25Mb, 1.000Mb]	0,16 Mbps
[0, 1; 1]	[25Mb, 1.000Mb]	5,09 Mbps
[0, 1; 1]	[25Mb, 1.000Mb]	0 Mbps
[0, 1; 1]	[25Mb, 1.000Mb]	0,71 Mbps
[0, 1; 1]	[25Mb, 1.000Mb]	4,08 Mbps
[0, 1; 1]	[25Mb, 1.000Mb]	2,95 Mbps

Tabela 11 – Composição dos contêineres.

lizar o método com e sem ressubmissão, a Seção 5.2 demonstra o impacto sobre a utilização e fragmentação nos recursos do DC e no *slowdown* das requisições. Além da presença de diferentes métodos de escalonamento, é imprescindível o escalonador ser capaz de realizar diferentes ordenações das requisições de usuário de acordo com a necessidade do DC. Para tanto, o experimento descrito na Seção 5.3 compara o comportamento do escalonador através de 7 filas de ordenação, conforme apresentado na Seção 3.5.

O escalonador e um simulador de eventos discretos foram desenvolvidos na linguagem de programação C++, utilizando o compilador GCC *v* 8.3 e CUDA *V* 10.2. Ainda, o ambiente utilizado para realizar as análises consiste em uma GPU NVIDIA RTX 2080 *TI* (11GB) hospedada por uma máquina com um processador Intel *i7-8700K* com 32GB de RAM DDR4 (3200MHz) executando GNU/Linux Archlinux 5.1.7.

5.2 ANÁLISE DE RESSUBMISSÃO DE TAREFAS

A análise de ressubmissão possui o objetivo de analisar o comportamento do escalonador ao alterar o método de ressubmissão utilizado pelo escalonador, sendo realizada em 3 etapas apresentadas na Tabela 12. Todas as etapas utilizam a configuração de requisições apresentada no Capítulo 5, porém atributos adicionais em cada requisição são adicionados, simulando os diferentes comportamentos dos usuários. As figuras apresentas nesta análise representam a distribuição acumulada dos dados obtidos, o eixo *X* representa o valor em percentual da métrica analisada, enquanto o eixo *Y* representa o percentual acumulado.

Etapa	Descrição
Padrão	Requisições simples, sem parâmetros adicionais.
Tempo máximo	As requisições abstraem a necessidade do usuário possuir um tempo máximo para executar uma aplicação.
Tempo máximo e término antecipado	As requisições abstraem a intervenção do usuário durante a execução de uma aplicação, permitindo assim que sua execução seja interrompida, além de possuírem um tempo máximo para executar suas atividades.

Tabela 12 – Etapas da análise de Ressubmissão de Tarefas.

5.2.1 Etapa Padrão

A etapa padrão realiza a comparação dos métodos sem e com ressubmissão utilizando a configuração padrão de requisições sem utilizar atributos adicionais. Esta etapa tem o objetivo de apresentar as diferenças entre os métodos em um cenário, no qual o usuário não realiza ativamente ações sobre as requisições e não possui um tempo limite para a execução de sua tarefa. A Figura 23 apresenta o comportamento dos algoritmos através de 4 métricas, *footprint* da largura de banda (Figura 23a), a fragmentação de enlace (Figura 23b), *footprint* de vCPU (Figura 23c) e *slowdown* (Figura 23d).

Os dados demonstram que todos algoritmos apresentam uma diferença significativa nos resultados ao alterar o método de escalonamento utilizado, no qual o método com ressubmissão apresenta melhores resultados. Este comportamento acontece devido a tentativa de remapeamento que é realizada por cada requisição, permitindo com que as requisições utilizem o máximo de recursos possíveis, reduzindo assim, a quantidade de recursos ociosos. Em contrapartida, ao utilizar o método sem ressubmissão, não há um limite temporal estabelecido para o mapeamento das requisições, fazendo com que todas as requisições sejam mapeadas no DC.

Ao analisar o desempenho entre os algoritmos, o método AHP apresenta resultados superiores ao Easy Backfilling em todas as métricas. Devido a característica de realizar os mapeamento considerando diversos objetivos a serem otimizados, o método AHP reduz também a diferença entre os resultados obtidos entre os métodos sem e com ressubmissão. O Easy Backfilling com política de consolidação (*i.e.*, reduzir a quantidade de servidores ativos no DC) apresenta resultados inferiores ao Easy Backfilling com política de disponibilidade (*i.e.*, distribuir as requisições entre os servidores do DC).

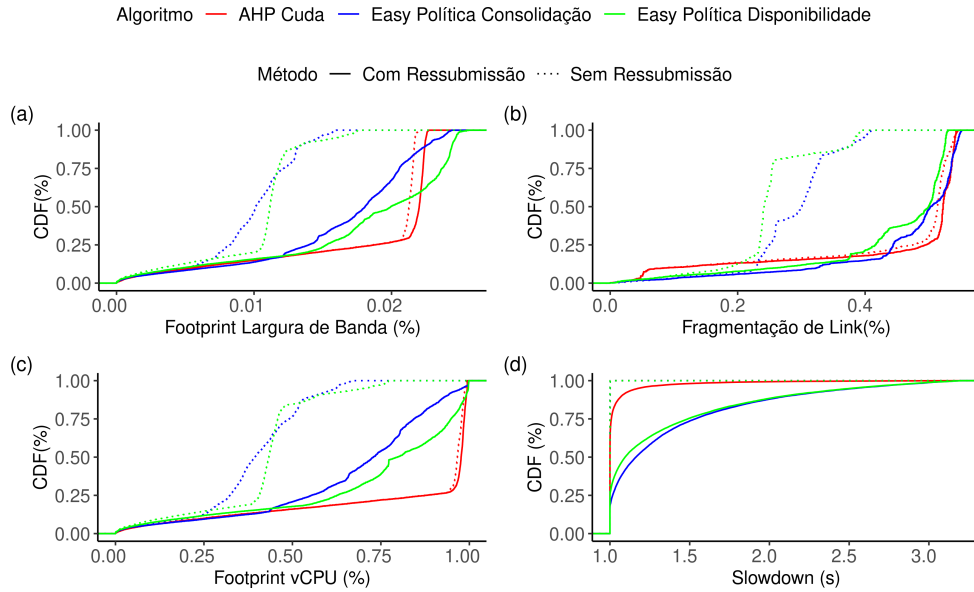


Figura 23 – Métricas do escalonador com método sem e com ressubmissão.

5.2.2 Etapa de Tempo Máximo

A etapa de tempo máximo contempla os usuários que possuem tempo limite para a conclusão de sua tarefa, este comportamento é representado pelo atributo *deadline*. Para isto, este atributo foi adicionado ao conjunto de requisições, sendo configurado para cada requisição de forma aleatória entre o intervalo $[0, 7 \times proc_I, proc_I]$. O comportamento do escalonador é apresentado nas Figuras 24a, 24b, 24c, 24d.

Os resultados apresentam uma redução na diferença entre os métodos com e sem ressubmissão. Este comportamento acontece devido a existência do tempo máximo para o mapeamento ser realizado em cada requisição, reduzindo o intervalo de busca temporal do escalonador no intervalo $[s(t), deadline_I]$. Ao analisar o desempenho dos algoritmos, o método *Analytic Hierarchy Process* (AHP) se destaca no mapeamento das requisições. O algoritmo Easy Backfilling com política de consolidação obteve melhores resultados ao se comparar com a primeira etapa da análise, conseguindo obter um resultado próximo ao AHP na métrica de fragmentação de enlace e superior a métrica de *slowdown*.

5.2.3 Etapa com Tempo Máximo e Término Antecipado

Por fim, a etapa com tempo máximo e término antecipado considera o cenário apresentado na segunda etapa em conjunto com a possibilidade do usuário interromper a execução da requisição. Deste modo, o conjunto de requisições presente na segunda etapa foi modificado, incluindo o atributo *early*, responsável por abstrair o comportamento do usuário. Os resultados dos dados analisados são apresentadas nas Figuras 25a, 25b, 25c, 25d.

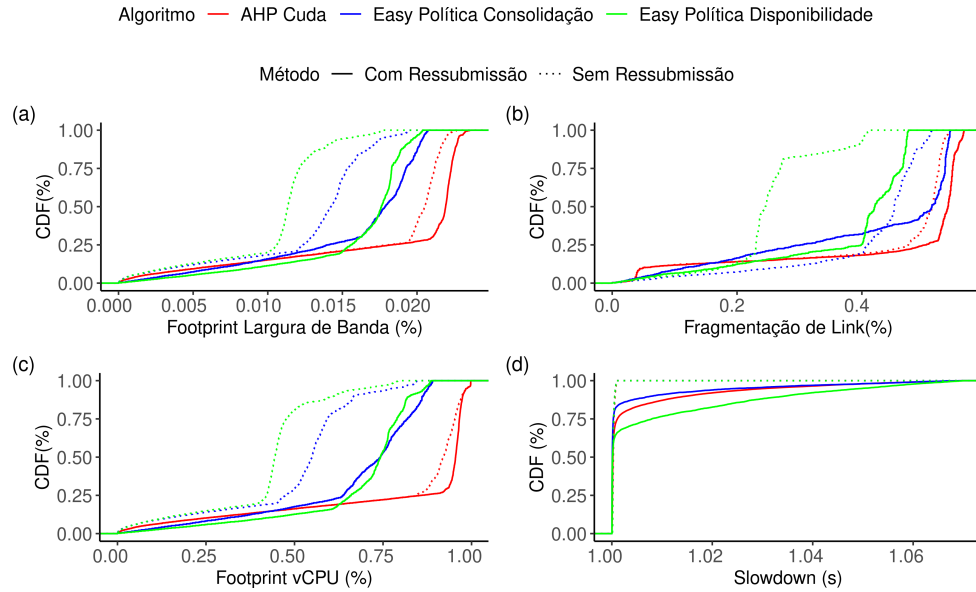


Figura 24 – Métricas do escalonador com método sem e com resubmissão utilizando *deadline*.

Apesar da inclusão do novo atributos nas requisições, o escalonador manteve o comportamento encontrado na primeira e segunda etapas frente ao método com e sem resubmissão, além apresentar resultados similares do método AHP. Porém os resultados apresentam uma inversão no comportamento do Easy Backfilling com política de disponibilidade e com política de consolidação, onde o uso da política de disponibilidade obtém resultados próximos ao método AHP na utilização dos recursos (*i.e.*, vCPU e largura de banda), porém torna-se o algoritmo que apresenta o maior *slowdown* aplicado nas requisições. Entretanto, a métrica de fragmentação de enlace apresenta uma proximidade dos resultados dos 3 algoritmos ao utilizar o método sem resubmissão.

5.2.4 Principais Considerações

Os dados obtidos demonstram que o método AHP apresenta resultados superiores ao algoritmo Easy Backfilling com política de disponibilidade e consolidação, independente da configuração de usuário (*i.e.*, de acordo apresentado na Tabela 12). Ao realizar a seleção dos servidores considerando múltiplos recursos, o método AHP consegue reduzir as diferenças nas métricas entre o método com e sem resubmissão, demonstrando a eficiência do método na tomada de decisão. Entretanto, independente da política de alocação utilizada pelo algoritmo Easy Backfilling, é possível notar que os resultados obtidos através dos métodos com e sem resubmissão divergem em proporção maior do que no método AHP, este comportamento acontece devido a seleção dos servidores considerar somente 1 recurso do DC.

Para complementar a análise dos métodos com e sem resubmissão é impres-

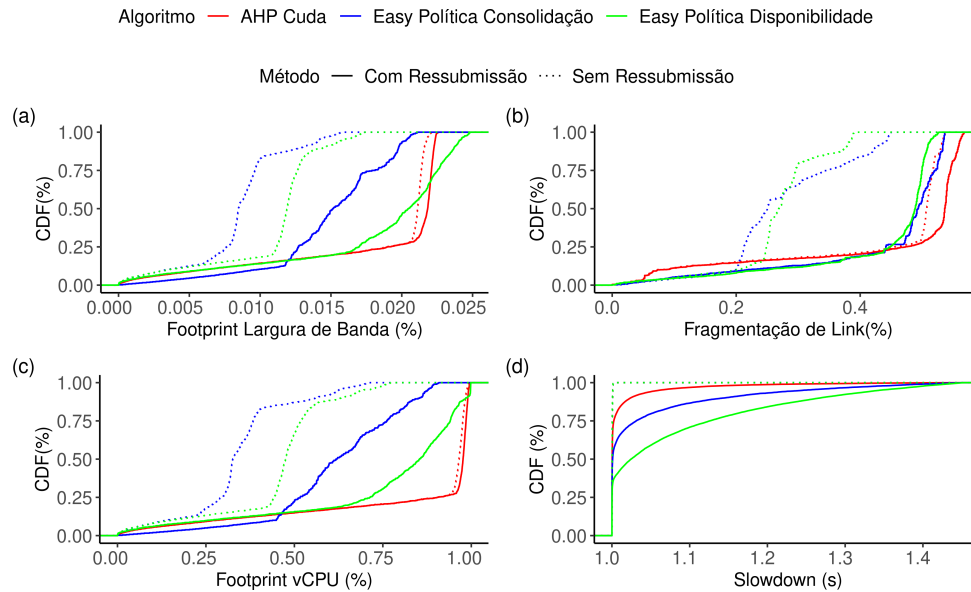


Figura 25 – Métricas do escalonador com método sem e com ressubmissão utilizando *deadline* e termino antecipado.

cindível analisar o total de eventos executado pelo escalonador para realizar o mapeamento das requisições, a quantidade de requisições do conjunto que foram mapeadas com sucesso no DC e o tempo de alocação médio das requisições. Deste modo, os resultados são sumarizados na Tabela 13. Além do escalonador apresentar diferenças na utilidade dos recursos do DC ao alterar o método de escalonado, é possível observar a diferença entre as requisições mapeadas com sucesso. A diferença existente entre os métodos é maior ao comparar o método Easy Backfilling com a política de disponibilidade. O algoritmo AHP consegue na maioria dos casos alocar todo o conjunto de requisições ao utilizar o método com ressubmissão, possuindo uma perda de somente 2,57% na quantidade de requisições aceitas ao se utilizar o método sem ressubmissão, demonstrando a adaptabilidade do método a cenários dinâmicos de escalonamento.

5.3 ANÁLISE DE FILAS DE ORDENAÇÃO

A análise de filas de ordenação possui o objetivo de analisar o comportamento do escalonador ao se alterar a forma com que as requisições são submetidas ao mapeamento utilizando o método sem ressubmissão de requisições, sendo consideradas 7 tipos diferentes de ordenação, conforme apresentado na Seção 3.5. A análise utiliza o conjunto de requisições de acordo com o apresentado no Capítulo 5. O comportamento do escalonador é apresentado na Figura 26a, 26b, 26c, 26d.

As métricas demonstram que o algoritmo Easy Backfilling apresenta uma diferença significativa nos resultados ao alterar o método de ordenação utilizado, enquanto o método AHP apresenta pequenas alterações nos resultados ao modificar a

Etapa	Ressubmissão	Algoritmo	# Eventos	#Requisições Aceitas
Padrão	Sem	Easy Backfilling Disponibilidade	1.198	52,40%
		Easy Backfilling Consolidação	1.198	55,14%
		AHP	1.198	97,05%
	Com	Easy Backfilling Disponibilidade	1.359	100%
		Easy Backfilling Consolidação	1.245	100%
		AHP	1.200	100%
Tempo máximo	Sem	Easy Backfilling Disponibilidade	1.201	69,58%
		Easy Backfilling Consolidação	1.201	56,49%
		AHP	1.201	98,68%
	Com	Easy Backfilling Disponibilidade	1.201	20,55%
		Easy Backfilling Consolidação	1.201	29,23%
		AHP	1.201	99,79%
Tempo máximo e término antecipado	Sem	Easy Backfilling Disponibilidade	1.169	45,69%
		Easy Backfilling Consolidação	1.169	59,15%
		AHP	1.172	97,43%
	Com	Easy Backfilling Disponibilidade	1.201	32,1%
		Easy Backfilling Consolidação	1.195	100%
		AHP	1.172	100%

Tabela 13 – Sumário dos algoritmos AHP e Easy Backfilling com política de consolidação e disponibilidade.

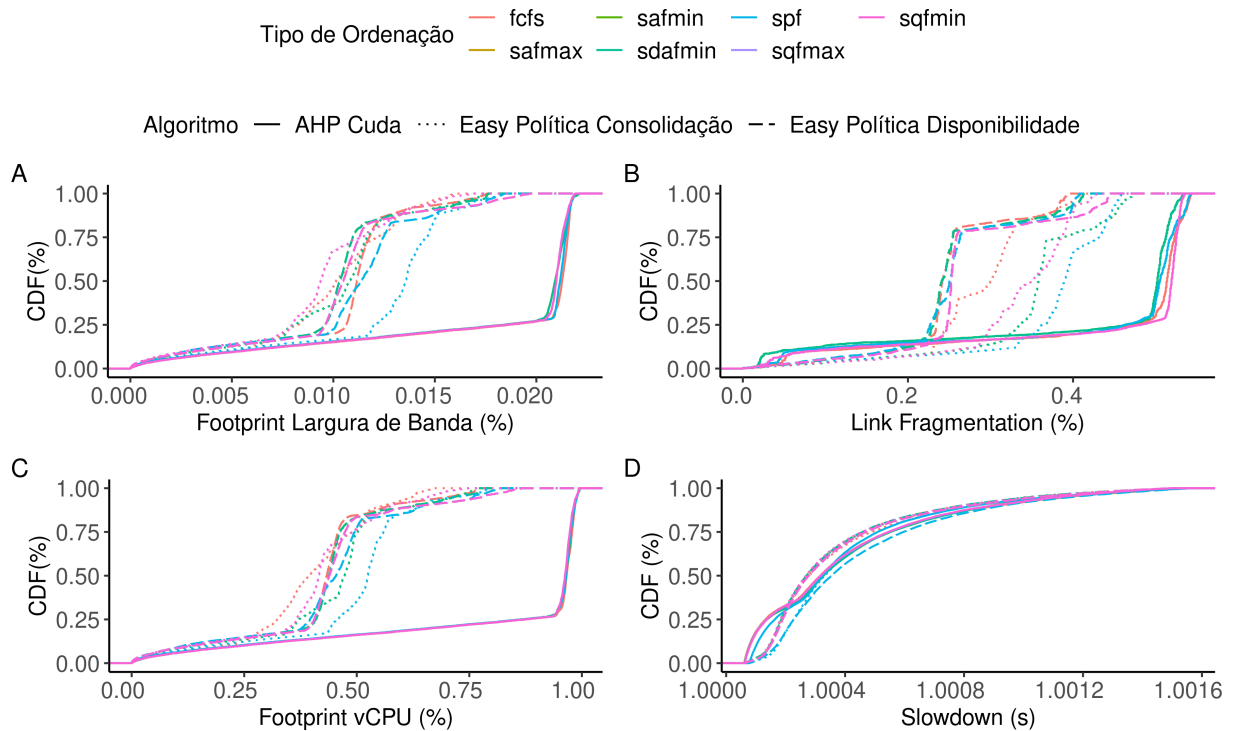


Figura 26 – Métricas do escalonador através das diferentes filas de ordenação das requisições.

ordenação utilizada. É importante ressaltar que o *slowdown* das requisições é pouco afetado pela alteração da ordenação, uma vez que o método utilizado de escalonamento é o sem ressubmissão, independente do algoritmo utilizado. Os resultados obtidos pelas variantes de recursos mínimos utilizados nas ordenações são bem próximos aos obtidos pelas variantes de recursos máximos (*i.e.*, $sqf\ min = sqf\ max$ e $saf\ min = saf\ max$).

A quantidade de eventos necessários para mapear as requisições e a quantidade de requisições mapeadas com sucesso são apresentadas na Tabela 14. Através das análises realizadas, não é possível definir a melhor forma de ordenação para o escalonador, pois os resultados apontam que cada algoritmo apresenta uma ordenação ótima distinta. A quantidade de eventos necessários para realizar o mapeamento das requisições não é alterado pela alteração das ordenações da fila, porém o total de requisições mapeadas com sucesso é influenciado de forma direta. O algoritmo Easy Backfilling independente da ordenação política utilizada, apresenta a ordenação *spf* como melhor alternativa para ser utilizadas na fila de requisições. Entretanto, o método AHP possui baixa variação ao método de ordenação utilizado ao analisar as métricas do DC, porém obtendo o método *sqf min* como melhor alternativa devido ao aumento no número de requisições mapeadas com sucesso.

Ordenação	Algoritmo	# Eventos	#Requisições Aceitas
FCFS	Easy Backfilling Disponibilidade	1.198	52,4%
	Easy Backfilling Consolidação	1.198	52,4%
	AHP	1.198	97,05%
SPF	Easy Backfilling Disponibilidade	1.196	76,03%
	Easy Backfilling Consolidação	1.196	67,53%
	AHP	1.198	96,25%
SAF Min	Easy Backfilling Disponibilidade	1.198	60,33%
	Easy Backfilling Consolidação	1.197	55,77%
	AHP	1.198	96,25%
SAF Max	Easy Backfilling Disponibilidade	1.198	60,33%
	Easy Backfilling Consolidação	1.197	55,77%
	AHP	1.198	96,25%
SQF Min	Easy Backfilling Disponibilidade	1.198	56,02%
	Easy Backfilling Consolidação	1.198	57,61%
	AHP	1.198	97,15%
SQF Max	Easy Backfilling Disponibilidade	1.198	56,02%
	Easy Backfilling Consolidação	1.198	57,63%
	AHP	1.198	97,15%
SDAF Min	Easy Backfilling Disponibilidade	1.198	60,33%
	Easy Backfilling Consolidação	1.197	55,77%
	AHP	1.198	96,25%

Tabela 14 – Sumário dos algoritmos AHP e Easy Backfilling com política de consolidação e disponibilidade.

5.4 CONSIDERAÇÕES PARCIAIS

Através das análises apresentada neste capítulo, foi possível constatar o comportamento do escalonador e comparar a qualidade da alocação das requisições com o algoritmo Easy Backfilling com as políticas de consolidação e disponibilidade. Com a análise de ressubmissão é possível concluir que o comportamento escalonador depende diretamente dos parâmetros comportamentais do usuários (*i.e.*, presença de tempo limite de execução, encerrar a tarefa antecipadamente, etc), onde a cada novo parâmetro adicionado a requisição, o comportamento do escalonador pode mudar completamente.

A partir dos resultados, foi evidenciado que o método AHP consegue se adaptar ao cenário dinâmico do escalonamento de requisições. Entretanto, o algoritmo Easy Backfilling começa a perder sua precisão no escalonamento a medida em que a quantidade de atributos são inseridos nas requisições. Ainda, através da análise das ordenações da fila de requisições, é possível concluir que o escalonador é sensível a ordem em que as requisições são mapeadas no DC, afetando a utilidade dos recursos do DC, enquanto o impacto no *slowdown* das requisições é menor. A escolha do método de ordenação utilizado pelo escalonador deve ser configurado conforme o DC, caso DC não possuam suas necessidades e requisitos bem definidos, é recomendado a utilização o método de ordenação FCFS, pois este método apresenta resultados médios em todos os cenários analisados. Entretanto, caso o DC tenha definido suas necessidades, se faz necessário um estudo específico sobre comportamento do algoritmo de escalonamento utilizado com diferentes filas de ordenação.

6 CONCLUSÃO

Os escalonadores de contêineres atuais utilizam algoritmos simples, analisando poucos critérios de alocação e usualmente otimizando somente uma função objetivo, devido a complexidade do problema (*NP-Difícil*). Existe um *trade off* entre o tempo de resposta para realizar a alocação e a aproximação da alocação ótima. Ainda, os requisitos de rede não são considerados, havendo uma falta de ferramentas que realizem o escalonamento conjunto de contêineres e redes virtuais. Neste contexto, este trabalho abordou aspectos relevantes através da discussão da utilização da abordagem multicritério acelerada por *Graphics Processing Unit* (GPU) para realizar o escalonamento de contêineres em conjunto com redes virtuais.

Apesar da complexidade computacional do problema estudado, o escalonador apresenta uma quebra do paradigma encontrado na literatura especializada. Através da utilização de algoritmos acelerados por GPU, o escalonador pode ser aplicado em DCs com mais de 20.000 servidores. As análises experimentais demonstraram a sensibilidade que os métodos multicritérios possuem frente ao problema analisado, realizando alocações de requisições através da ponderação de diversos critérios, conseguindo otimizar os objetivos elencados. Também é apresentado a diferença de complexidade necessária entre realizar o escalonamento de contêineres e o escalonamento conjunto de contêineres e redes virtuais. Por meio da abordagem híbrida entre métodos multicritérios, algoritmos de agrupamento acelerador por GPU e considerando os recursos de rede, foi possível desenvolver um escalonador escalável que apresenta resultados superiores aos obtidos pelos algoritmos *Best Fit* (BF), *Worst Fit* (WF), Easy Backfilling com política de disponibilidade e consolidação.

Em suma, as principais contribuições do trabalho foram: (I) desenvolver um escalonador de contêineres e redes virtuais; (II) desenvolver um escalonador capaz de considerar de forma automática múltiplos critérios, sendo independente a quantidade de variáveis consideradas, apresentando uma abordagem mista de algoritmos de agrupamento e métodos multicritérios acelerados em GPU; (III) desenvolver uma estrutura de dados denominada *Augmented Forest* para realizar o mapeamento das requisições dos usuários no DC de forma eficiente; (IV) a análise da qualidade da solução encontrada durante o escalonamento dos rastros do Google Borg em um DC com *Fat-Tree* (configurado com $k = 44$), demonstrando a aplicabilidade do escalonador em cenários com elevado número de servidores, investigando a utilização do DC e o eventual atraso total no processamento; (V) a análise de rede apresentando resultados superiores do escalonador ao comparar com os algoritmos utilizados pelos escalonadores atuais; (VI) a análise do comportamento do escalonador ao utilizar o

método com e sem ressubmissão para realizar o mapeamento das requisições, utilizando 3 conjuntos de requisições, de forma a explorar o ambiente dinâmico em que o escalonador está inserido; (VII) a análise das ordenações da fila de requisições, comparando o comportamento do escalonador ao utilizar 7 variantes de ordenação dos recursos para contêineres.

Recomenda-se como trabalhos futuros a implementação de um módulo de balanceamento de carga, permitindo otimizar os recursos já alocados no DC no decorrer do tempo. Sugere-se também a implementação de mecanismos de tolerância a falhas, permitindo que o escalonador se recupere de falhas graves. Ainda, expandir a implementação escalonador para realizar o mapeamento das requisições em múltiplas GPUs.

6.1 PUBLICAÇÕES REALIZADAS

Durante o desenvolvimento da proposta, foram realizadas 5 publicações em revistas e periódicos. O artigo (RODRIGUES; KOSLOVSKI; ALVES JUNIOR, 2018) apresenta um estudo sobre a seleção do provedor de nuvem computacional mais adequado para a demanda de um usuário. Desta forma, através do desenvolvimento de um sistema baseado no método AHP foram realizadas análises comparando 4 provedores de nuvem, considerando a *Quality-of-Service* (QoS) e dos preços praticados dos provedores. Os resultados indicam que há uma alteração na frequência de seleção entre os provedores ao alterar a seleção baseada somente em custos para a análise conjunta com QoS.

Com o foco em nuvens OpenStack, a proposta do artigo (RODRIGUES et al., 2019b) aborda o desenvolvimento de um escalonador multicritério acelerado por GPU. O estudo propõem mudanças no fluxo de comunicação do OpenStack, tendo como objetivo reduzir a comunicação inter-módulos, além de apresentar a arquitetura e estratégias de escalonamento. Já, o artigo (RODRIGUES et al., 2019a) aborda o desenvolvimento de um escalonador de contêineres em GPU utilizando o algoritmo AHP. Foram realizadas análises 2 análises do escalonador, a análise de desempenho e de qualidade. Deste modo, foi apresentado que o escalonador em GPU apresenta uma quebra no paradigma encontrado na literatura especializada, permitindo que o escalonador possa ser aplicado em DCs com mais de 2.000 servidores.

A pesquisa realizada em (RODRIGUES et al., 2019a) apresenta o *virtual infrastructure multicriteria allocation and migration-based broker* (VIMAM). O *broker* possui a função de verificar qual é o provedor de nuvem mais adequado para as necessidades do usuário, utilizando o método AHP. Após o provedor ser selecionado, é realizado a migração da infraestrutura virtual entre o provedor atual e o provedor

selecionado. O VIMAM considera a migração de Máquinas Virtuais (MVs), contêineres, *switches* e outros elementos de topologia. Foram realizadas 2 análises, a primeira apresenta a validação dos diferentes cenários de qualidade para a seleção do provedor mais adequado, enquanto a segunda análise demonstra a eficiência do módulo de migração do *broker*. O protótipo desenvolvido é baseado em nuvens OpenStack e Docker, permitindo a migração entre dois provedores de nuvens IaaS.

Com foco no escalonamento conjunto de redes virtuais e contêineres, o artigo (RODRIGUES et al., 2019b) apresenta um escalonador de contêineres para DCs multiusuários. O escalonador desenvolvido utiliza métodos multicritérios acelerados por GPU para realizar o mapeamento das requisições no DC e o algoritmo *widest path* para realizar as configurações de comunicação dos contêineres. Foram realizadas análises comparando o escalonador com os algoritmos encontrados nos orquestradores Kubernetes e Docker Swarm. Através das análises, foi demonstrado que realizar o escalonamento, considerando os requisitos de rede das requisições dos usuários, é fundamental para melhorar o desempenho do escalonador, reduzindo a quantidade de recursos ociosos do DC.

REFERÊNCIAS

- ADEL'SON-VEL'SKII, G.; LANDIS, E. An algorithm for the organization of information. **Dokl. Akad. Nauk SSSR**, Soviet, v. 146, 1962.
- ALLA, H. B. et al. An efficient dynamic priority-queue algorithm based on ahp and pso for task scheduling in cloud computing. In: **HIS**. Marrakech, Marrocos: Springer, 2016. p. 134–143.
- ANDRADE, G. N. de. Proposta de integração do método multicritério MACBETH para restrições aos pesos em análise envoltória de dados. **SBPO**, 2015.
- ARAÚJO, A. G. d.; ALMEIDA, A. T. d. Apoio à decisão na seleção de investimentos em petróleo e gás: uma aplicação utilizando o método PROMETHEE. **Gestão & produção**, SciELO Brasil, v. 16, n. 4, p. 534–543, 2009.
- ASSUNCAO, M. D. d.; VEITH, A. da S.; BUYYA, R. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. **Journal of Network and Computer Applications**, Elsevier, v. 103, p. 1–17, February 2018. ISSN 1084-8045. Disponível em: <<https://www.sciencedirect.com/science/article/pii/S1084804517303971>>.
- BABAR, M. A.; RAMSEY, B. **Understanding Container Isolation Mechanisms for Building Security-Sensitive Private Cloud**. Adelaide, Australia, 2017.
- BERZINS, L. Avaliação de desempenho pelo AHP através do superdecisions: Caso inmetro. **Dissertação de Mestrado**. IBMEC, Rio de Janeiro, 2009.
- BRAZ, J. M. B. P. **O MACBETH como ferramenta MCDA para o benchmarking de aeroportos**. Tese (Doutorado) — Universidade da Beira Interior, 2011.
- BURNS, B. et al. Borg, omega, and kubernetes. **Queue**, ACM, New York, NY, USA, v. 14, n. 1, p. 10:70–10:93, 2016. ISSN 1542-7730.
- CAMATI, R. S.; CALSAVARA, A.; JR, L. L. Solving the virtual machine placement problem as a multiple multidimensional knapsack problem. **ICN 2014**, p. 264, 2014.
- CARASTAN-SANTOS, D. et al. One can only gain by replacing easy backfilling: A simple scheduling policies case study. In: . [S.l.: s.n.], 2019.
- CHEN, C.-T. Extensions of the topsis for group decision-making under fuzzy environment. **Fuzzy sets and systems**, Elsevier, v. 114, n. 1, p. 1–9, 2000.
- CHEN, D.-S.; BATSON, R. G.; DANG, Y. **Applied integer programming: modeling and solution**. USA: John Wiley & Sons, 2010.
- COREOS. **CoreOS is building a container runtime, rkt**. 2019. Disponível em: <<https://www.coreos.com/blog/rocket.html>>.
- COREOS. **CoreOS is building a container runtime, rkt**. 2019. Disponível em: <<https://coreos.com/rkt/docs/latest/running-kvm-stage1.html>>.

COREOS. **CoreOS is building a container runtime, rkt**. 2019. Disponível em: <<https://coreos.com/rkt/docs/latest/running-fly-stage1.html>>.

COREOS. **RKT A security-minded, standards-based container engine**. 2019. Disponível em: <<https://www.coreos.com/rkt/>>.

COSTA, C. A. Bana e; CORTE, J.-M.; VANSNICK, J.-C. MACBETH (measuring attractiveness by a categorical based evaluation technique). **Wiley Encyclopedia of Operations Research and Management Science**, Wiley Online Library, 2011.

COSTA, H. G. Introdução ao método de análise hierárquica: análise multicritério no auxílio à decisão. **Niterói: UFF**, 2002.

DEB, K.; JAIN, H. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: solving problems with box constraints. **IEEE Transactions on Evolutionary Computation**, IEEE, v. 18, n. 4, p. 577–601, 2013.

DOCKER. **Enterprise Container Platform for High-Velocity Innovation**. 2019. Disponível em: <<http://www.docker.com>>.

DOCKER. **Enterprise Container Platform for High-Velocity Innovation**. 2019. Disponível em: <<https://docs.docker.com/engine/swarm/key-concepts/>>.

DOCKER. **Enterprise Container Platform for High-Velocity Innovation**. 2019. Disponível em: <<https://docs.docker.com/swarm/scheduler/filter>>.

DONGEN, S. M. V. **Graph clustering by flow simulation**. Tese (Doutorado) — University of Utrecht, Utrecht, Holanda, 2001.

FELTER, W. et al. An updated performance comparison of virtual machines and linux containers. In: IEEE. **2015 IEEE international symposium on performance analysis of systems and software (ISPASS)**. Philadelphia, PA, USA, 2015. p. 171–172.

FREITAS, L. V. de et al. Decision-making with multiple criteria using AHP and MAUT: An industrial application. **European International Journal of Science and Technology**, v. 2, 2013.

GARG, S. K.; VERSTEEG, S.; BUYYA, R. A framework for ranking of cloud computing services. **Future Generation Computer Systems**, Elsevier, v. 29, n. 4, p. 1012–1023, 2013.

GOLDBERG, R. P. Survey of virtual machine research. **Computer**, IEEE, v. 7, n. 6, p. 34–45, 1974.

GOOGLE. **Concepts**. Google, 2019. Acessado em: 26/05/2019. Disponível em: <<https://kubernetes.io/docs/concepts/>>.

GOOGLE. **Production-Grade Container Orchestration**. Google, 2019. Acessado em: 26/05/2019. Disponível em: <<http://kubernetes.io>>.

GUERRERO, C.; LERA, I.; JUIZ, C. Genetic algorithm for multi-objective optimization of container allocation in cloud architecture. **Journal of Grid Computing**, Springer, 2018.

GUO, Y.; YAO, W. A container scheduling strategy based on neighborhood division in micro service. In: **IEEE. NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symp.** Taipei, Taiwan, 2018. p. 1–6.

HAVET, A. et al. Genpack: A generational scheduler for cloud data centers. In: **IEEE Int. Conf. on Cloud Engineering (IC2E)**. Vancouver, BC, Canada: IEEE, 2017. p. 95–104.

HINDMAN, B. et al. Mesos: A platform for fine-grained resource sharing in the data center. In: **NSDI**. Berkeley, California: University of California, 2011. v. 11, n. 2011, p. 22–22.

HU, Y. et al. Concurrent container scheduling on heterogeneous clusters with multi-resource constraints. **Future Generation Computer Systems**, Elsevier, v. 102, p. 562–573, 2020.

HWANG, C.-L.; YOON, K. **Multiple Attribute Decision Making**. New York: Lecture Notes in Economics and Mathematical Systems, Springer, 1981.

IMDOUKH, M.; AHMAD, I.; ALFAILAKAWI, M. Optimizing scheduling decisions of container management tool using many-objective genetic algorithm. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, p. e5536, 2019.

INFANTE, C.; MENDONÇA, F. M. d.; VALLE, R. d. A. B. d. Análise de robustez com o método ELECTRE ii: O caso da região de campo das vertentes em minas gerais. são carlos-sp. **Revista Gestão e Produção**, v. 21, n. 2, p. 245–255, 2014.

KAEWKASI, C.; CHUENMUNEEWONG, K. Improvement of container scheduling for docker using ant colony optimization. In: **IEEE. Knowledge and Smart Technology (KST), 2017 9th Int. Conf. on**. Chonburi, Thailand, 2017. p. 254–259.

KARANDE, P.; CHAKRABORTY, S. A facility layout selection model using MACBETH method. In: **Proceedings of the 2014 International Conference on Industrial Engineering and Operations Management**. Bali, Indonesia: [s.n.], 2014. p. 7–9.

KOZHIRBAYEV, Z.; SINNOTT, R. O. A performance comparison of container-based technologies for the cloud. **Future Generation Computer Systems**, Elsevier, v. 68, p. 175–182, 2017.

LXC. **Linux Containers**. 2019. Disponível em: <<https://www.linuxcontainers.org/lxc/>>.

MARCONDES, A. H. et al. Executing distributed applications on sdn-based data center: A study with nas parallel benchmark. In: **IEEE. 2016 7th International Conference on the Network of the Future (NOF)**. [S.l.], 2016. p. 1–3.

MESOS. **Program against your datacenter like it's a single pool of resources**. 2019. Acessado em: 26/05/2019. Disponível em: <<http://mesos.apache.org/>>.

MOREIRA, R. A. Análise multicritério dos projetos do sebrae/rj através do ELECTRE IV. **Faculdade de Economia e Finanças IBMEC**, 2007.

MOTA, C. M. d. M.; ALMEIDA, A. T. d. Método multicritério ELECTRE IV-H para priorização de atividades em projetos. **Pesquisa Operacional**, SciELO Brasil, v. 27, n. 2, p. 247–269, 2007.

NESI, L. L. et al. GPU-accelerated algorithms for allocating virtual infrastructure in cloud data centers. In: . Washington, USA: IEEE, 2018.

NESI, L. L. et al. Tackling virtual infrastructure allocation in cloud data centers: a gpu-accelerated framework. In: **14th Int. Conf. on Network and Service Management (CNSM 2018)**. Rome, Italy: IEEE, 2018.

PANWAR, N. et al. Topsis–pso inspired non-preemptive tasks scheduling algorithm in cloud environment. **Cluster Computing**, Springer, p. 1–18, 2018.

PIRAGHAJ, S. F. et al. Containercloudsim: An environment for modeling and simulation of containers in cloud data centers. **Software: Practice and Experience**, Wiley Online Library, v. 47, n. 4, p. 505–521, 2017.

RAUEN, F. J. Pesquisa científica: discutindo a questão das variáveis. **SIMFOP: Santa Catarina**, 2012.

REISS, C.; WILKES, J.; HELLERSTEIN, J. L. **Google cluster-usage traces: format + schema**. Mountain View, CA, USA, 2011. Revised 2012.03.20. Posted at <<http://code.google.com/p/googleclusterdata/wiki/TraceVersion2>>.

RODRIGUES, L. R. et al. Cloud broker proposal based on multicriteria decision-making and virtual infrastructure migration. **Software: Practice and Experience**, Wiley Online Library, v. 49, n. 9, p. 1331–1351, 2019.

RODRIGUES, L. R. et al. Proposta de um escalonador multicritério acelerado por gpu para nuvens openstack. In: **Anais da XIX Escola Regional de Alto Desempenho da Região Sul**. Porto Alegre, RS, Brasil: SBC, 2019. ISSN 2595-4164. Disponível em: <<https://ojs.sbc.org.br/index.php/erads/article/view/7088>>.

RODRIGUES, L. R.; KOSLOVSKI, G. P.; ALVES JUNIOR, O. C. Multicriteria analysis for iaas cloud providers selection. In: **Proceedings of the XIV Brazilian Symposium on Information Systems**. New York, NY, USA: ACM, 2018. (SBSI'18), p. 15:1–15:8. ISBN 978-1-4503-6559-8. Disponível em: <<http://doi.acm.org/10.1145/3229345.3229361>>.

RODRIGUES, L. R. et al. Escalonamento de contêineres com método de decisão multicritério acelerado por gpu. In: **Anais do XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos**. Porto Alegre, RS, Brasil: SBC, 2019. p. 515–528. ISSN 2177-9384. Disponível em: <<https://ojs.sbc.org.br/index.php/sbrs/article/view/7383>>.

RODRIGUES, L. R. et al. Network-aware container scheduling in multi-tenant data center. **IEEE Global Communications Conference**, 2019.

RODRIGUEZ, D. S. S.; COSTA, H. G.; CARMO, L. D. Métodos de auxílio multicritério à decisão aplicados a problemas de pcp: Mapeamento da produção em periódicos publicados no brasil. **Gestão & Produção**, v. 20, n. 1, p. 134–146, 2013.

RODRIGUEZ, M. A.; BUYYA, R. Container-based cluster orchestration systems: A taxonomy and future directions. **Software: Practice and Experience**, Wiley Online Library, 2018.

- ROST, M.; DÖHNE, E.; SCHMID, S. Parametrized complexity of virtual network embeddings: Dynamic & linear programming approximations. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 49, n. 1, p. 3–10, fev. 2019. ISSN 0146-4833.
- SAATY, T. L. How to make a decision: the analytic hierarchy process. **European journal of operational research**, Elsevier, v. 48, n. 1, p. 9–26, 1990.
- SAATY, T. L. **Fundamentals of decision making and priority theory with the analytic hierarchy process**. [S.l.]: Rws Publications, 2000. v. 6.
- SHARMA, P. et al. Containers and virtual machines at scale: A comparative study. In: **Proceedings of the 17th International Middleware Conference**. Trento, Italy: ACM, 2016. p. 1.
- SHEIKHAN, M.; MOHAMMADI, N. Time series prediction using pso-optimized neural network and hybrid feature selection algorithm for iee load data. **Neural computing and applications**, Springer, v. 23, n. 3-4, p. 1185–1194, 2013.
- SILVA, V. B. de S.; SCHRAMB, F.; CARVALHO, H. R. C. de. O uso do método PROMETHEE para seleção de candidatos à bolsa-formação do pronatec. **Production**, Scielo Brasil, v. 24, n. 3, p. 548–558, 2014.
- SOUZA, F. R. de et al. Qos-aware virtual infrastructures allocation on sdn-based clouds. In: **2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)**. Madrid, Spain: IEEE, 2017. p. 120–129.
- SOUZA, F. R. de et al. Qvia-sdn: Towards qos-aware virtual infrastructure allocation on sdn-based clouds. **Journal of Grid Computing**, Mar 2019. ISSN 1572-9184. Disponível em: <<https://doi.org/10.1007/s10723-019-09479-x>>.
- SUO, K. et al. An analysis and empirical study of container networks. In: **INFOCOM 2018-IEEE Conference on Computer Communications**. Honolulu, HI, USA: IEEE, 2018. p. 189–197.
- THÖNES, J. Microservices. **IEEE software**, IEEE, v. 32, n. 1, p. 116–116, 2015.
- TRIHINAS, D. et al. Devops as a service: Pushing the boundaries of microservice adoption. **IEEE Internet Computing**, v. 22, n. 3, May 2018. ISSN 1089-7801.
- VERMA, A. et al. Large-scale cluster management at google with borg. In: **Proceedings of the Tenth European Conference on Computer Systems**. Bordeaux, France: ACM, 2015. p. 18.
- WAZLAWICK, R. **Metodologia de pesquisa para ciência da computação**. Brasil: Elsevier Brasil, 2017. v. 2.
- WHAIDUZZAMAN, M. et al. Cloud service selection using multicriteria decision analysis. **The Scientific World Journal**, Hindawi Publishing Corporation, v. 2014, 2014.
- XAVIER, M. G. et al. Performance evaluation of container-based virtualization for high performance computing environments. In: **2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing**. Belfast, UK: IEEE, 2013. p. 233–240.

YU, M. et al. Rethinking virtual network embedding: substrate support for path splitting and migration. **ACM SIGCOMM Computer Communication Review**, ACM, v. 38, n. 2, p. 17–29, 2008.

ZHU, Y.; WANG, Y.; WANG, F. 10 observations on google cluster trace+ 2 measures for cluster utilization enhancement. **arXiv preprint arXiv:1508.02111**, 2015.

ZHU, Z.; TANG, X. Deadline-constrained workflow scheduling in iaas clouds with multi-resource packing. **Future Generation Computer Systems**, Elsevier, v. 101, p. 880–893, 2019.