Escalonamento de Contêineres com Método de Decisão Multicritério Acelerado por GPU

Leonardo R. Rodrigues[1], **Marcelo Pasin**[2], **Omir C. Alves Jr.**[1], **Maurício A. Pillon**[1], **Charles C. Miers**[1], **Guilherme P. Koslovski**[1]

[1]Programa de Pós-Graduação em Computação Aplicada (PPGCA)
Universidade do Estado de Santa Catarina (UDESC) - Joinville - SC - Brasil

[2]University of Neuchâtel (UniNE) - Institut d'informatique - Suíça

Resumo. *A utilização de contêineres passou a ser recentemente adotada como suporte para o provisionamento rápido de sistemas distribuídos. Microsserviços, processamento de fluxos de dados, computação nas bordas e outros sistemas complexos podem ser concretizados sob forma de contêineres. Entretanto, devido a heterogeneidade de configuração das requisições e a dimensionalidade dos data centers (DCs) hospedeiros, o escalonamento de contêineres é um problema NP-Difícil. Um caminho eficiente para amenizar a complexidade do escalonamento é a utilização do processamento paralelo de alto desempenho. Neste contexto, o presente trabalho apresenta o EMULAG: um escalonador multicritério acelerado por GPU. A função objetivo do escalonador representa a perspectiva do provedor, buscando a consolidação do DC. Uma análise experimental revelou que a solução é escalável, apresentando resultados superiores aos encontrados na literatura, mas com baixo tempo de processamento.*

Abstract. *Containers have been recently adopted as support for fast provisioning of distributed systems. They can be used to implement microservices, data flow processing, edge computing, and other complex systems. However, due to the settings heterogeneity of requests and dimensionality of the hosting DCs, container scheduling is an NP-Hard problem. An efficient path to ease the scheduler complexity is to use the of high-performance parallel processing. In this context, we present present EMULAG: a GPU-accelerated multi-criteria scheduler. The scheduler's objective function represents the providers perspective, aiming at data center (DC) consolidation. We present an experimental analysis revealing our solution is scalable and presents higher results than those found in the literature, but with lower processing time.*

1. Introdução

A complexidade no desenvolvimento e manutenção de sistemas distribuídos é um desafio para desenvolvedores e pesquisadores. Os contêineres foram criados como forma de simplificar o processo de instalação e de lançamento de serviços, oferecendo um provisionamento escalável. Desenvolvedores de microsserviços, infraestruturas para processamento de fluxos de dados, processamento na borda da Internet, entre outros sistemas complexos, adotam estrutura baseada em contêineres para realizar o seu provisionamento [Assuncao et al. 2018, Trihinas et al. 2018].

Dentre as tarefas administrativas de gerenciamento de contêineres, o escalonamento é uma parte crítica devido ao seu impacto direto no tempo de resposta e *Quality of Service* (QoS). No contexto deste artigo, o termo escalonador é utilizado para definir o conjunto de programas que realiza a associação de uma requisição de contêiner a um determinado servidor físico. Algoritmos de escalonamento devem considerar múltiplos atributos, configurações e critérios para a tomada de decisão e concepção de uma resposta. Respostas devem ser processadas em tempo computacional aceitável, caracterizando-o como um problema *on-line* [Havet et al. 2017, Vaucher et al. 2018]. A diversidade de configurações das requisições em termos de RAM e de CPU, o elevado número de recursos no *data center* (DC) e o considerável número de contêineres são fatores cruciais para o processamento desse problema *NP-Difícil*. 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[1]. Embora promissor, o desenvolvimento de escalonadores usando GPUs é complexo e exige uma releitura de algoritmos previamente desenvolvidos com arquiteturas tradicionais.

Expostos os fatos, o presente trabalho apresenta um escalonador para contêineres desenvolvido com foco no processamento em GPU. O escalonador é baseado no *Analytic Hierarchy Process* (AHP) [Saaty 2005] como método para processamento de requisições considerando múltiplos critérios. Como função objetivo, o escalonador atende requisições de forma a *consolidar* contêineres sobre um DC, ou seja, aglomerar tanto quanto possível os contêineres requisitados em um número mínimo de servidores. Além da paralelização em GPU, o escalonador aplica algoritmos para agrupamento de dados, diminuindo o número de comparações necessárias para tomada de decisão. A análise experimental é realizada considerando um DC com topologia *fat-tree* [Al-Fares et al. 2008, Singh et al. 2015], composto por mais de 20000 servidores, que hospedam requisições de contêineres oriundas dos rastros de execução do Google Borg [Reiss et al. 2011].

Em resumo, o trabalho apresenta três contribuições principais: (i) uma proposta de paralelização de algoritmos para embasar o escalonamento com o AHP; (ii) uma análise da relação entre o tamanho da requisição de contêineres e o número de servidores do DC, identificando qual a configuração combinada mínima que justifique o processamento em GPU; e (iii) uma 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.

O texto está organizado como segue. A Seção 2 discute a motivação, a definição do problema e os trabalhos relacionados. A Seção 3 apresenta o escalonador proposto (EMULAG), as estratégias de alocação utilizadas e a função objetivo a ser otimizada. As análises da escalabilidade e das soluções do escalonador são abordadas na Seção 4.

[1] 10^{12} operações de ponto flutuante por segundo.

2. Motivação e Definição do Problema

O escalonamento das requisições está diretamente atrelado a função objetivo adotada e ao cenário aplicado. Assim, a Seção 2.1 apresenta a definição do escalonamento de contêineres, enquanto a Seção 2.2 discute os trabalhos relacionados.

2.1. Escalonamento de Contêineres

No provisionamento de contêineres em DCs, a tarefa do escalonador é selecionar hospedeiros apropriados para atender as requisições, sendo organizados em três tipos de arquiteturas [Rodriguez and Buyya 2018]: centralizada, descentralizada e em dois níveis. A arquitetura centralizada, utilizada nos escalonadores do Docker Swarm[2] e Kubernetes[3], possui um único escalonador responsável pela tomada de decisão, realizada com uma visão global do ambiente em que estão inseridos. A arquitetura descentralizada, utilizada pelo Google Omega [Schwarzkopf et al. 2013], é utilizada para aumentar a escalabilidade, criando múltiplas réplicas do escalonador. Por fim, na arquitetura em dois níveis, utilizada pelo Apache Mesos [Hindman et al. 2011], os planos de gerenciamento de recursos e da aplicação são desacoplados. Um escalonador opera individualmente no gerenciamento do DC, enquanto outro realiza a concessão dos recursos para os contêineres.

Para exemplificar a atuação do escalonador de contêineres, a Figura 1 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.

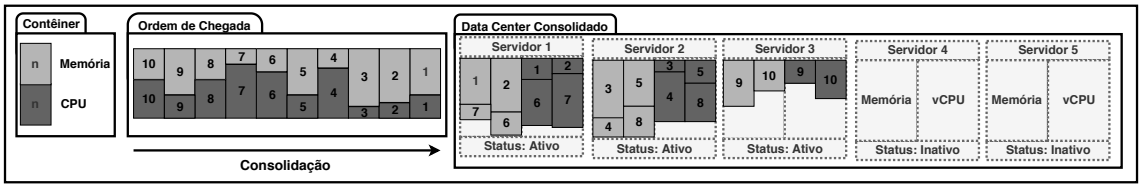


Figura 1. Escalonamento de contêineres em DC buscando a consolidação.

Como observado na Figura 1, 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. Dentre as alternativas para modelar e resolver o escalonamento multicritério (formulações lineares, heurísticas, entre outros), o método AHP é consolidado e utilizado em diversos cenários [Saaty 2005], sobretudo por permitir a definição dinâmica de pesos para análise dos critérios envolvidos. Em resumo,

[2]Sistema de orquestração de contêineres nativo do docker.

[3]Sistema de orquestração de contêineres desenvolvido pela Google, possui suporte para diversos tipos de contêineres.

o método AHP permite a classificação de diversos servidores candidatos, guiado por uma hierarquia organizacional de pesos. Cada atributo que define os recursos do DC, como capacidade de CPU e de RAM, espaço de armazenamento, vazão dos enlaces de comunicação, entre outros, são analisados considerando os pesos informados durante a configuração do AHP. Um provedor pode configurar um conjunto hierárquico de pesos buscando diferentes objetivos, como a distribuição dos recursos sobre o DC, diminuindo o impacto de falhas, ou a consolidação dos recursos, reduzindo o consumo energético [Cavalcanti et al. 2014, Vaucher et al. 2018, Havet et al. 2017].

2.2. 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, apresentamos a seguir trabalhos que buscam o escalonamento de contêineres e propostas relacionadas com alocação de máquinas virtuais com formulação *on-line*. Embora os problemas compartilhem fundamentos de formulação, o cenário baseado em contêineres é mais sensível ao tempo de resposta do escalonador [Guerrero et al. 2018].

Um escalonador de microsserviços hospedados em contêineres foi proposto buscando o balanceamento de carga de um DC [Guerrero et al. 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.

Em [Guo and Yao 2018] propuseram um escalonador que utiliza o algoritmo de divisão da vizinhança de contêineres, desempenhando função de equilibrar o balanceamento de carga e reduzir as distâncias entre os contêineres no DC. A análise experimental foi realizada em um DC homogêneo com uma topologia *fat-tree* configurada com $k = 18$, (*i.e.,* 1458 servidores), comparando o escalonador proposto, com dois outros, onde o primeiro algoritmo utiliza a meta-heurística *Particle Swarm Optimization* (PSO) e o segundo a técnica de espalhamento. A Seção 4 demonstra que a presente proposta permite a análise com dimensões superiores de DC.

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 and Chuenmuneewong 2017] em um DC com 13 servidores, utilizando uma amostra de 1% do arquivo de traços 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. Por sua vez, os autores [Vaucher et al. 2018] abordaram o escalonamento considerando requisitos de segurança para execução de contêineres, especificamente com suporte ao *Software Guard Extension* (SGX). O escalonador desenvolvido utiliza o algoritmo *bin packing* para selecionar os servidores hospedeiros. A análise experimental apresentada na Seção 4 se

inspira, quanto a origem das requisições de contêineres, em dois cenários previamente descritos [Havet et al. 2017] e [Vaucher et al. 2018].

Por fim, dentre os trabalhos revisados, não foram encontrados trabalhos que utilizam escalonadores de contêineres em conjunto com GPUs, visando melhorar o tempo de resposta, bem como a aplicação do escalonador em cenários reais. Por isso, tal fato revela-se um aspecto original no trabalho aqui apresentado. A Seção 3 apresenta em maiores detalhes a especificação do *Escalonador Multicritério de Contêineres Acelerado por GPU* (EMULAG).

3. EMULAG

O EMULAG é um escalonador centralizado (Seção 2.1), que realiza o processamento guiado pela utilização do método AHP, processando as requisições em GPU, e tendo como função objetivo a consolidação dos contêineres no DC. EMULAG apoia-se em um *framework* recentemente desenvolvido para realizar a alocação de recursos virtuais [Nesi et al. 2018a, Nesi et al. 2018b]. O escalonador é composto por três módulos principais: escalonamento, agrupamento e seleção multicritério. O módulo de escalonamento é responsável por orquestrar o fluxo de contêineres, *i.e.,* alocação e remoção dos contêineres. Este módulo possui duas políticas de escalonamento, a primeira garante que o somatório dos recursos alocados não é maior do que a capacidade do servidor físico, enquanto a segunda, garante que os contêineres devem ser alocados pela ordem de chegada das submissões. Caso o DC não possua um servidor capaz de comportar a requisição, esta então é postergada. O módulo de agrupamento analisa o DC e decompõe os recursos em grupos, auxiliando o escalonador a reduzir o espaço de busca. Finalmente, o módulo de seleção multicritério implementa o método AHP tanto em CPU quanto em GPU, conforme descrito na Seção 3.1, sendo responsável por realizar a classificação dos servidores de acordo com critérios preestabelecidos, guiado pela função objetivo adotada.

3.1. Estruturas do AHP

O método AHP é baseado em 4 funções principais: concepção da hierarquia, aquisição de dados, síntese e análise de consistência. A concepção da hierarquia é responsável pela definição do problema em forma hierárquica, através do conjunto de critérios (*i.e.,* as variáveis que impactam diretamente na solução do problema) e o conjunto de alternativas. A aquisição de dados é responsável por captar todas as informações pertinentes ao problema e realizar a comparação paritária entre os elementos da hierarquia. A etapa de síntese realiza os cálculos algébricos para construção da classificação das alternativas. Por fim, a análise de consistência verifica se a modelagem do problema é consistente e aplicável [Saaty 2005].

O modelo hierárquico para escolher qual é o servidor mais adequado a comportar um contêiner, a fim de otimizar a consolidação do DC (função objetivo), é apresentado na Figura 2. A hierarquia conta com três níveis, conectadas por arestas valoradas, o primeiro nível representa o foco da classificação dos servidores, buscando encontrar o servidor mais adequado para comportar o contêiner. O segundo nível contém os critérios analisados para compôr o cálculo da classificação dos servidores, sendo a configuração dos processadores, a memória RAM, e a fragmentação do DC. A fragmentação indica o número de servidores ativos hospedando ao menos um contêiner. O terceiro nível é composto por todos os servidores do DC.

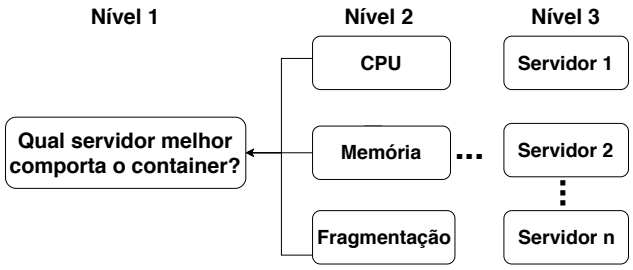


Figura 2. Hierarquia do AHP para análise multicritério do escalonamento.

O EMULAG realiza a distribuição de pesos (*i.e.,* valoração das arestas) na hierarquia de duas formas distintas. Inicialmente, os pesos fornecidos entre o primeiro e segundo nível da hierarquia são fixos, sendo distribuídos de forma igualitária para os critérios de CPU e RAM, enquanto que para o critério de fragmentação um peso maior é atribuído. A especificação dos valores dos pesos é apresentada na Subseção 4.2.2. Quanto maior o peso aplicado no critério de fragmentação, maior a tendência do escalonador aumentar a consolidação do DC, visando a função objetivo escolhida. Ainda, os pesos para o segundo e o terceiro níveis da hierarquia são adquiridos a cada execução do escalonador, ou seja, os valores são atualizados de acordo com os recursos disponíveis nos servidores candidatos. Fica evidente que a política de definição de pesos tem impacto na escolha feita pelo escalonador. Arbitrariamente, o EMULAG foi configurado buscando a consolidação do DC, mas outras distribuições de pesos podem ser futuramente investigadas.

3.2. Função Objetivo

O EMULAG tem como objetivo escalonar contêineres para consolidar o DC. Ele utiliza tanto quanto possível a capacidade disponível dos servidores ativos, de forma a deixar o máximo de servidores inativos. Sendo C o conjunto de contêineres solicitados e S o conjunto de servidores no DC, a função $\mathcal{M} : C \mapsto S$ indica o escalonamento de um contêiner i sobre um servidor s, com $\mathcal{M}(i) \in S; \forall i \in C$. Assim, a função objetivo deve buscar minimizar a razão $\frac{|S_a|}{|S|}$, considerando que S_a representa o conjunto de servidores ativos, ou seja, servidores que hospedam pelo menos um contêiner ($\mathcal{M}(i) \in S_a; \forall i \in C$) [Cavalcanti et al. 2014].

3.3. Aceleração dos Algoritmos em GPU

A estratégia adotada para escalonamento dos contêineres é dada pela combinação do AHP com o *Markov Cluster Algorithm* (MCL), ambos processados em GPU, conforme explicado no Algoritmo 1. MCL é um algoritmo de agrupamento de dados organizados em grafos, baseado em fluxos de dados e cadeia de Markov [Van Dongen 2001]. 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]. Inicialmente, EMULAG recebe o conjunto de servidores do DC (S) e o contêiner que deve ser escalonado ($i \in C$), retornado o mapeamento encontrado ($\mathcal{M}(i)$).

O Algoritmo 1 inicia realizando o agrupamento dos recursos do DC, aplicando o algoritmo MCL (linha 2). Com a obtenção dos grupos, um número reduzido de candidatos deve ser futuramente comparado pelo AHP. Ainda, como o EMULAG deve respeitar a função objetivo (nesse caso, consolidação), é necessário encontrar o grupo que melhor

Algoritmo 1: EMULAG: pseudocódigo do escalonamento de contêineres.

```
   Entrada: S, i ∈ C
   Saída: M(i)
1  início
2      grupos = MCL(S)
3      grupos_classificados = AHP(grupos)
4      para ∀g ∈ grupos_classificados faça
5          servidores = {∀h | h ⊂ g}
6          servidores_classificados = AHP(servidores)
7          para ∀h ∈ servidores_classificados faça
8              se QoS_respeitado(h, i) então
9                  M(i) = h
10                 retorna M(i)
11             fim
12         fim
13     fim
14     posterga_escalonamento(i)
15     retorna ∅
16 fim
```

comporta a requisição, desta forma, o algoritmo AHP é executado (linha 3). Posteriormente, os grupos são percorridos de acordo com a classificação multicritério efetuada pelo AHP. Para cada grupo, os servidores são novamente classificados para identificar a melhor opção de acordo com a função objetivo (linhas $4-6$). O servidor com maior valor de classificação é selecionado e a função $QoS\_respeitado(h, i)$ verifica se o servidor h realmente suporta a configuração do contêiner i (linhas $7-8$).

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 verificação de concordância de QoS inicia buscando o provisionamento do valor máximo solicitado para cada atributo (CPU e RAM). Se necessário, os valores são gradativamente reduzidos, respeitando o limite mínimo. Caso o servidor comporte a requisição, o escalonamento é realizado (linha 9) e o mapeamento é informado encerrando a execução (linha 10). Caso contrário, o próximo servidor é selecionado (linhas $7-12$), e se necessário, outros grupos são analisados (linhas $4-13$). Se todos os servidores e grupos forem analisados e nenhuma solução for encontrada, a requisição de contêiner é postergada para a próxima execução do escalonador, ou seja, a requisição retorna para fila e aguardará o encerramento de um provisionamento para escalonar (linha 14).

A implementação do AHP dispõe de um conjunto de 4 *kernels* [4] executados em pipeline.O primeiro *kernel* realiza a aquisição de dados, preparando a comparação paritária entre os elementos da hierarquia entre cada *thread*.

O primeiro *kernel* realiza a comparação paritária de todos os hosts do DC, cada *thread* realiza apenas uma comparação paritária. Cada *thread* possui um índice único para evitar conflitos de escrita na matriz resultante. O segundo *kernel* realiza a soma de cada coluna da matriz calculada na etapa anterior. Esta etapa foi desenvolvida utilizando métodos de *parallel reduction with avoidance bank conflicts* com memória compartilhada.

[4] Nomenclatura utilizada para definir funções executadas em GPU.

Por sua vez, a operação de síntese de dados é realizado pelo terceiro *kernel*. Esta etapa consiste na divisão de cada elemento da matriz por cada elemento do vetor, calculados pelo primeiro e segundo *kernels* respectivamente. Finalmente, o quarto *kernel* utiliza o *parallel reduction* para realizar a soma da matriz calculada no terceiro *kernel*. Após a sincronização das *threads* da GPU, uma multiplicação utilizando *parallel reduction* é realizada para gerar o vetor que contém os valores ranqueados dos hosts.

4. Análise Experimental

O escalonador EMULAG foi desenvolvido na linguagem de programação C++, e compilado com *g++ -O3 -fstack-protector-strong*. As diretivas de compilação utilizadas visam otimizar o escalonador (*-O3*), enquanto mantêm a segurança dos dados internos da memória (*-fstack-protector-strong*). A análise experimental é resultado da aplicação de traços reais do Google Borg [Reiss et al. 2011] ao escalonador EMULAG. O ambiente utilizado para realizar as análises consiste em uma GPU NVIDIA Titan XP (12GB) hospedada por uma máquina com um processador Intel i7 $-$ 5930K com 16GB de RAM DDR4 (2400MHz) executando GNU/Linux Archlinux 4.15.4 com CUDA 10.0.130 e GCC 8.2.1.

4.1. Métricas para Análise

Para analisar o desempenho do escalonador, quatro métricas foram selecionadas. O *tempo de execução* do escalonador é essencial para indicar o tempo de resposta do processo completo, contabilizando a representação dos dados e a execução dos algoritmos em CPU e GPU. Por sua vez, as métricas de *fragmentação* e *footprint* representam a utilização do DC. A primeira contabiliza o número de servidores ativos enquanto a segunda indica a carga efetivamente utilizada no servidor para hospedar os contêineres. Por fim, o *atraso* na execução dos contêineres quantifica a perspectiva dos usuários, indicando qual o impacto da decisão do AHP no tempo total de execução dos contêineres.

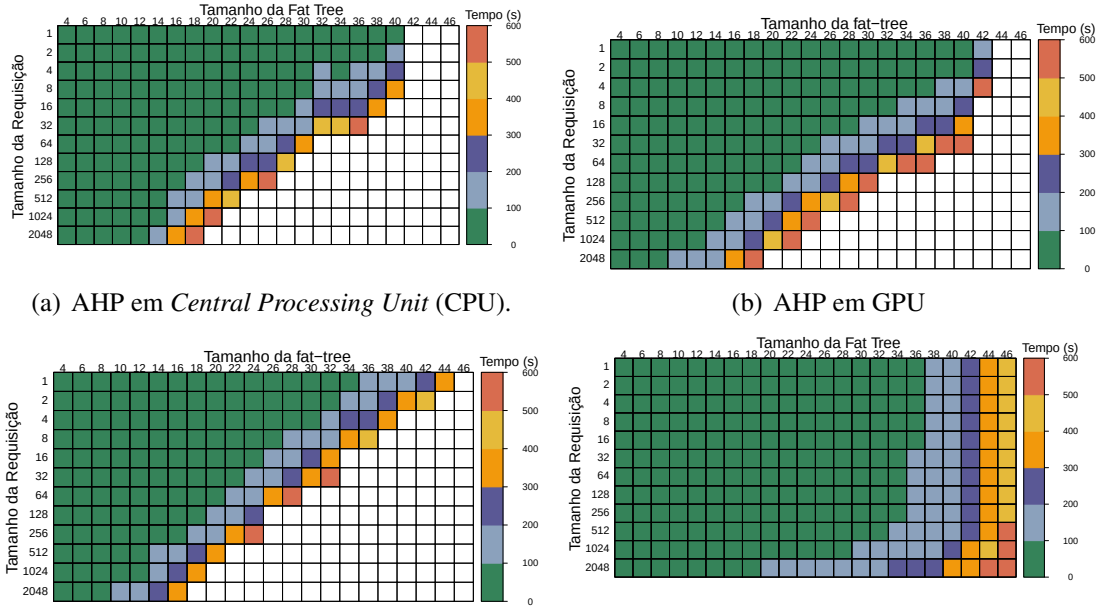
4.2. Cenários Experimentais e Discussão dos Resultados

Os experimentos desenvolvidos visam avaliar o algoritmo EMULAG em diferentes perspectivas. Para tanto, foram elaborados dois cenários: a análise de escalabilidade e aceleração e a análise da qualidade do escalonamento. Cada cenário será descrito a seguir.

4.2.1. Análise da Escalabilidade e Aceleração

Para quantificar a escalabilidade do EMULAG, duas dimensões de parâmetros foram variadas. A primeira dimensão é o tamanho da topologia do DC, seguindo o padrão *fat-tree*, sendo avaliada para valores entre $k = 4$ e $k = 46$ (*i.e.,* de 16 a 24334 servidores). A segunda dimensão representa o número de recursos requisitados, variando entre 1 até 2048 contêineres. Os limites superiores de contêineres e configuração de *fat-tree* foram definidos de acordo com a limitação de memória da GPU utilizada (neste caso, 12GB). 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 indicará a fronteira aceitável do tempo de processamento para escalonar as requisições em diferentes configurações de topologia. Para comparação, o Algoritmo 1 foi individualmente implementado em CPU e GPU.

As Figuras 3(a), 3(b), 3(d) apresentam o tempo necessário para escalonar utilizando EMULAG. Na Figura 3(c) é apresentado o tempo necessário para o escalonador realizar somente o agrupamento do DC (conforme discutido na Seção 3.3), não considerando os tempos de classificação e alocação dos contêineres.



(a) AHP em *Central Processing Unit* (CPU).

(b) AHP em GPU

(c) MCL em GPU.

(d) AHP e MCL em GPU

Figura 3. Tempo de processamento para escalonamento de contêineres em diferentes configurações da topologia *fat-tree*.

Os resultados apresentados na Figura 3 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 600 segundos, conforme indicado pela escala de cores. Células na cor branca indicam que o processamento não foi concluído no tempo máximo configurado. O eixo X representa o tamanho da topologia *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). O Algoritmo MCL em CPU não foi utilizado para realizar os testes, devido ao tempo computacional, para realizar os agrupamentos, ser superior as restrições de tempo existentes no problema abordado [Nesi et al. 2018a].

Inicialmente, de acordo com a Figura 3(a), a utilização do método AHP em CPU para escalonar um DC é viável para topologias pequenas, conforme atualmente citado pela literatura especializada (Seção 2.2). Ainda, a escalabilidade máxima da abordagem em CPU é uma *fat-tree* $k = 40$, tratando até requisições compostas por 8 contêineres. Enquanto isto, o AHP em GPU (Figura 3(b)) consegue escalar o problema, aumentando o tamanho do DC e a configuração das requisições. Porém, caso a topologia do DC seja inferior a $k = 14$, o algoritmo AHP em CPU apresenta um desempenho melhor se comparado ao AHP em GPU. Este fenômeno ocorre devido ao tempo necessário para

trafegar os dados entre a RAM e a GPU, para posteriormente realizar as chamadas de *kernel* da GPU.

Através da Figura 3(c) é possível observar que realizar o agrupamento do DC é uma tarefa escalável, agrupando inclusive um DC com topologia $k = 46$. 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. Em contraponto, a Figura 3(d) apresenta a abordagem híbrida (EMULAG completo), utilizando o MCL para realizar o agrupamento do DC uma única vez durante a execução do escalonador, e a cada nova requisição processando somente o método AHP em GPU. Deste modo, a abordagem torna-se escalável, sendo possível tratar um DC com topologia $k = 46$ com até 2048 contêineres. Por fim, os resultados revelam que o EMULAG supera as dimensões previamente consideradas para escalonamento de contêineres, permitindo o processamento de DCs e requisições em larga escala. Ainda, embora não discutido no presente trabalho, múltiplas GPUs podem ser combinadas para realizar o escalonamento.

4.2.2. Análise da Qualidade do Escalonamento

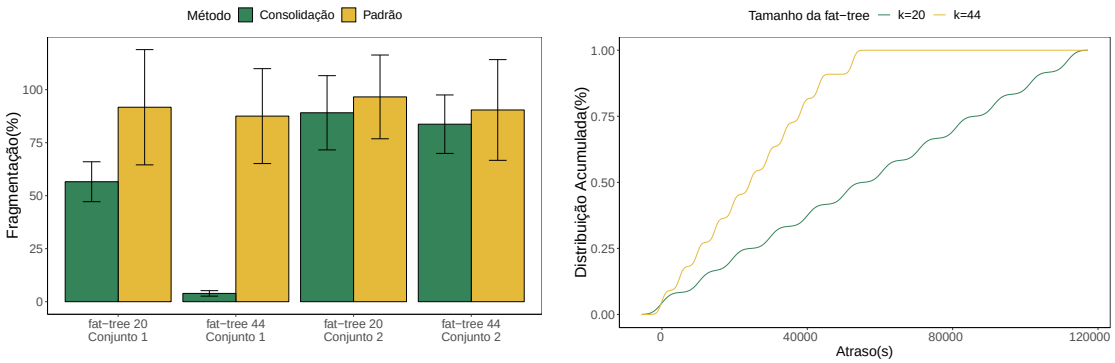
Após quantificar a escalabilidade de EMULAG, a segunda etapa da análise investiga a consolidação do DC seguindo as métricas da Seção 4.1. De acordo com a literatura especializada [Havet et al. 2017, Vaucher et al. 2018], o conjunto de requisições de contêineres é oriundo do rastro de execução do Google Borg [Reiss et al. 2011]. Especificamente, as requisições correspondem a 1 hora do rastro, selecionado entre 6480s e 10080s do arquivo original, totalizando 1313596 requisições. Cada requisição de contêiner é representada por uma quádrupla, $R(i) = (CPU_{max}, CPU_{min}, RAM_{max}, RAM_{min})$, representando a faixa de valores máxima e mínima de processadores e memória. Neste contexto, as requisições são mapeadas em um DC homogêneo, constituído por servidores compostos por 24 CPUs com 256GB de *Random Access Memory* (RAM), dispostas em uma topologia *fat-tree* [Al-Fares et al. 2008].

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 a capacidade máxima de um servidor (Conjunto 1) e considerando 10% da capacidade máxima do servidor (Conjunto 2). Quanto ao número de servidores, duas configurações de *fat-tree* foram analisadas, com $k = 20$ e $k = 44$. O limite superior ($k = 44$) é ligeiramente diferente do aplicado na Seção 4.2.1 devido ao aumento dos dados manipulados (deve respeitar o limite da GPU, 12GB). A análise foi realizada com duas distribuições de pesos no método AHP em GPU, a primeira representa a busca pela consolidação (Seção 3.2) ao indicar um maior peso para o critério de fragmentação possuindo peso $0,5$, enquanto os demais critérios possuem peso $0,25$, conforme apresentado na Seção 3.1 (Figura 2), 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 4(a). 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 método AHP 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 método AHP 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. O atraso máximo aplicado a uma requisição é o mesmo para os dois tamanhos da topologia analisada, 300 segundos. Entretanto, a Figura 4(b) 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 o 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.



(a) Análise da fragmentação do DC resultante das configurações do AHP.

(b) CDF do atraso no atendimento das requisições para o Conjunto 2.

Figura 4. *Footprint* e atrasos no escalonamento de contêineres.

Complementando a análise, os valores de *footprint* da memória e de CPU dos servidores são sintetizados na Tabela 1. Através da análise do *footprint* é possível observar que as requisições submetidas ao EMULAG ocupam uma proporção maior de CPU frente aos recursos de memória. A alteração da distribuição de pesos no AHP 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 havendo 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 se considerar a função objetivo de consolidação, é constatado que a aplicação do método AHP com maiores pesos 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	*fat-tree*	Configuração AHP	Footprint CPU	*Footprint* 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 1. Comparação do *Footprint* de RAM e CPU.

4.2.3. Discussão dos Resultados

Através da análise apresentada na Subseção 4.2.1 foi possível constatar o desempenho computacional do EMULAG, apresentando a escalabilidade dos métodos de agrupamento e classificação. Deste modo, é possível concluir que o método utilizado depende diretamente do tamanho do DC e do conjunto de requisições. A partir dos resultados obtidos, foi possível realizar as análises apresentadas na Subseção 4.2.2, constatando que o desempenho do escalonador é afetado não somente pelo tamanho do conjunto de requisições, mas também pelo comportamento em que estas são submetidas. Além disso, através da comparação entre a distribuição de pesos no método AHP, padrão e busca pela consolidação, foi indicado a simplicidade oferecida pelo método AHP para representação de funções objetivo, permitindo, futuramente, a configuração com outros cenários.

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 algumas requisições sejam postergadas. Este fato é apresentado através da Tabela 2, a qual apresenta a comparação do tempo médio necessário para escalonar os dois conjuntos de requisições definidos na Subseção 4.2.2, 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	*fat-tree*	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 2. Tempo de execução do EMULAG nos cenários analisados.

5. Consideracões & Trabalhos Futuros

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. Neste contexto, este artigo abordou aspectos relevantes através da discussão da utilização da abordagem multicritério acelerada por GPU para realizar o escalonamento de contêineres. Apesar da complexidade computacional do problema estudado, o EMULAG 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 20000 servidores. As análises experimentais demonstraram a sensibilidade que o método AHP possui frente ao problema analisado, realizando alocações de requisições através da ponderação de diversos critérios, conseguindo otimizar a função objetivo elencada. Em suma, as principais contribuições do trabalho foram: *(i)* desenvolver um escalonador de contêineres que considera de forma automática múltiplos critérios, sendo agnóstico a quantidade de variáveis consideradas, apresentando uma abordagem mista de algoritmos de agrupamento e métodos AHP acelerados em GPU; *(ii)* a análise da relação entre o tamanho da requisição de contêineres e o número de servidores do DC, identificando qual a configuração combinada mínima que justifique o processamento em GPU; *(iii)* 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.

Por fim, recomenda-se como trabalhos futuros a implementação de outros métodos de análise multicritério (*e.g.,* PROMETHEE, ELECTRE), para realizar comparações entre estes métodos, e, assim, identificar o comportamento do escalonamento através das métricas elencadas na Seção 4.1. Sugere-se, também, a utilização de múltiplas GPUs para realizar a alocação, permitindo que o escalonador execute diversas instâncias do módulo de classificação em paralelo, possibilitando reduzir ainda mais o tempo de computação.

Agradecimentos Os autores agradecem o apoio do Laboratório de Processamento Paralelo e Distribuído (LabP2D), FAPESC, UDESC e NVIDIA.

Referências

Al-Fares, M., Loukissas, A., and Vahdat, A. (2008). A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4):63–74.

Assuncao, M. D. d., da Silva Veith, A., and Buyya, R. (2018). Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. *Journal of Network and Computer Applications*, 103:1–17.

Cavalcanti, G. A. d. S., Obelheiro, R. R., and Koslovski, G. (2014). Optimal resource allocation for survivable virtual infrastructures. In *2014 10th Int. Conf. on the Design of Reliable Communication Networks (DRCN)*, pages 1–8.

Guerrero, C., Lera, I., and Juiz, C. (2018). Genetic algorithm for multi-objective optimization of container allocation in cloud architecture. *Journal of Grid Computing*.

Guo, Y. and Yao, W. (2018). A container scheduling strategy based on neighborhood division in micro service. In *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symp.*, pages 1–6. IEEE.

Havet, A., Schiavoni, V., Felber, P., Colmant, M., Rouvoy, R., and Fetzer, C. (2017). GENPACK: A generational scheduler for cloud data centers. In *2017 IEEE Int. Conf. on Cloud Engineering (IC2E)*, pages 95–104.

Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R. H., Shenker, S., and Stoica, I. (2011). Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22.

Kaewkasi, C. and Chuenmuneewong, K. (2017). Improvement of container scheduling for docker using ant colony optimization. In *Knowledge and Smart Technology (KST), 2017 9th Int. Conf. on*, pages 254–259. IEEE.

Nesi, L. L., Pillon, M. A., de Assunção, M. D., and Koslovski, G. P. (2018a). GPU-accelerated algorithms for allocating virtual infrastructure in cloud data centers. In *18th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing (CCGrid 2018)*.

Nesi, L. L., Pillon, M. A., de Assunção, M. D., Miers, C. C., and Koslovski, G. P. (2018b). Tackling virtual infrastructure allocation in cloud data centers: a gpu-accelerated framework. In *14th Int. Conf. on Network and Service Management (CNSM 2018)*.

Reiss, C., Wilkes, J., and Hellerstein, J. L. (2011). Google cluster-usage traces: format + schema. Technical report, Google Inc., Mountain View, CA, USA. Revised 2012.03.20. Posted at `http://code.google.com/p/googleclusterdata/wiki/TraceVersion2`.

Rodriguez, M. A. and Buyya, R. (2018). Container-based cluster orchestration systems: A taxonomy and future directions. *Software: Practice and Experience*.

Saaty, T. L. (2005). Making and validating complex decisions with the AHP/ANP. *Journal of Systems Science and Systems Engineering*, 14(1):1–36.

Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., and Wilkes, J. (2013). Omega: flexible, scalable schedulers for large compute clusters. In *Proc. of the 8th ACM European Conf. on Computer Systems*, pages 351–364. ACM.

Singh, A., Ong, J., Agarwal, A., Anderson, G., Armistead, A., Bannon, R., Boving, S., Desai, G., Felderman, B., Germano, P., et al. (2015). Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 183–197. ACM.

Trihinas, D., Tryfonos, A., Dikaiakos, M. D., and Pallis, G. (2018). Devops as a service: Pushing the boundaries of microservice adoption. *IEEE Internet Computing*, 22(3).

Van Dongen, S. M. (2001). *Graph clustering by flow simulation*. PhD thesis, University of Utrecht, Utrecht, Holanda.

Vaucher, S., Pires, R., Felber, P., Pasin, M., Schiavoni, V., and Fetzer, C. (2018). SGX-aware container orchestration for heterogeneous clusters. In *2018 IEEE 38th Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 730–741.