São José dos Campos - SP
LV SBPO
LV Simpósio Brasileiro de Pesquisa Operacional
São José dos Campos, SP - 6 a 9 de novembro de 2023

Escalonamento de *Containers* em *Clusters* Kubernetes Usando uma Abordagem de Programação Linear Inteira Mista

Camilla Martins, Pedro Nuno Moura, Sidney C. de Lucena, Carlos Alberto V. Campos
Programa de Pós–Graduação em Informática
Universidade Federal do Estado do Rio de Janeiro (UNIRIO)
Av. Pasteur 458, Urca – Rio de Janeiro, RJ -– Brasil
camilla.martins@unirio.edu.br,{pedro.moura,sidney,beto}@uniriotec.br

RESUMO

Grande parte dos serviços providos na Internet residem em ambientes virtualizados de provedores de "nuvem", cujas infraestruturas de TI adotam, em sua grande maioria, a técnica de virtualização de *containers* para abrigar as diversas aplicações que devem rodar de forma independente e isolada. O gerenciamento desses *containers* é predominantemente realizado pela solução Kubernetes, contudo, o algoritmo padrão de alocação de *containers* em máquinas físicas (nós), usado pelo Kubernetes, não é capaz de otimizar a relação custo/desempenho do ambiente em nuvem. Para solucionar isso, este trabalho propõe uma abordagem de programação linear inteira mista baseada no *Capacitated Facility Location Problem* para otimizar a distribuição de *containers* em nós de um *cluster* Kubernetes, levando em consideração o custo de instalação de um novo nó, o histórico de falhas dos nós em funcionamento e o consumo de recursos por cada aplicação rodando em um *container*. Os resultados experimentais mostram que a abordagem proposta é capaz de melhorar significativamente o desempenho do *cluster* Kubernetes em relação à estratégia convencional de escalonamento, ao mesmo tempo em que garante que as demandas dos aplicativos sejam atendidas de forma eficiente.

PALAVRAS CHAVE. Problema de Localização de Instalações, Programação Linear Inteira Mista, Kubernetes, Escalonamento de *Containers*.

ABSTRACT

A large part of the services provided on the Internet resides in virtualized environments of "cloud" providers, whose IT infrastructures adopt, for the most part, the *containers* virtualization technique to house the various applications that must run independently and isolated. The management of such *containers* predominantly adopts the Kubernetes solution, whose standard algorithm for allocating *containers* on physical machines (nodes) is not capable of optimizing the cost/performance ratio of the cloud environment. As a solution to this problem, this work proposes a mixed integer linear programming approach based on the *Capacitated Facility Location Problem* to optimize the distribution of *containers* in nodes of a Kubernetes *cluster*, taking into account the cost of installing a new node, the history of failures of the nodes in operation and the consumption of resources by each application running in a *container*. The experimental results show that the proposed approach significantly improves the performance of Kubernetes *cluster* concerning the conventional scheduling strategy while ensuring the efficient execution of applications demands.

KEYWORDS. Facility Location Problem, Mixed Integer Linear Programming, Kubernetes, Container Scheduling.

DOI: 10.59254/sbpo-2023-175211
http://live-proceedings.pantheonsite.io/p/175211

1. Introdução

Na área de Tecnologia da Informação e Comunicações (TIC), uma das técnicas mais empregadas atualmente por provedores de serviço em nuvem é a virtualização baseada em *containers*. Segundo a empresa Docker (Docker [2023]), um *container* pode ser definido como uma unidade de *software* capaz de agrupar o código de uma aplicação e todas as suas dependências, possibilitando que o respectivo aplicativo seja executado de forma rápida e confiável, ainda que se mude o ambiente computacional. Trata-se, portanto, de um recurso computacional virtualizado que se comporta como uma máquina (física) independente, dimensionada para as necessidades computacionais da aplicação que é executada.

No escopo das tecnologias e soluções de mercado voltadas para a "orquestração" de *containers*, o Kubernetes é, segundo a própria documentação oficial (CNCF [2023a]), um sistema de código aberto que automatiza, em larga escala, a implantação, o dimensionamento e o gerenciamento de aplicativos em *containers*. O Kubernetes trabalha com o conceito de *cluster*, que é a entidade que agrupa todas as máquinas físicas gerenciadas por uma mesma instância do Kubernetes. Nessas máquinas físicas, chamadas de nós, rodam os *pods*, onde são executadas as aplicações, sendo que um *pod* pode conter um ou mais *containers*. O plano de controle do Kubernetes fica, portanto, encarregado de distribuir os *pods* pelos nós disponíveis, conforme as necessidades de escalabilidade e demanda de uso (*workloads*) das aplicações.

O escalonador padrão do Kubernetes, chamado de Kube-scheduler, ao fazer a distribuição dos *pods* pelos nós, não considera certos parâmetros que podem afetar o custo ou o desempenho geral da infraestrutura de nuvem. Dentre eles, podemos citar o custo individual de cada nó que precisa ser colocado em funcionamento, a quantidade de recursos consumida por cada *pod* associado a um serviço sendo demandado, ou a quantidade média de erros (falhas) registradas para um dado nó em funcionamento.

Este artigo propõe modelar o problema da alocação de *pods* em nós, num ambiente de *cluster* Kubernetes, como um problema de otimização combinatória baseado no *Capacitated Facility Location Problem*, que, por sua vez, é resolvido através da construção de uma formulação por Programação Linear Inteira Mista. A validação da proposta se deu em ambiente simulado, que buscou reproduzir um cenário realista de uso do Kubernetes, incluindo seu *workload*. Os resultados experimentais confirmaram a eficácia da formulação e sua vantagem em relação ao escalonador padrão do Kubernetes.

As contribuições do trabalho residem na construção de uma modelagem que permita identificar o que seria um nível ótimo de alocação de recursos em um *cluster* Kubernetes pelo menor preço de alocação e de custo de nó, utilizando a maior referência de pods que conseguirmos.

O restante deste artigo está estruturado da seguinte forma: a Seção 2 aborda os trabalhos relacionados; a Seção 3 descreve o funcionamento do Kubernetes e do seu escalonador padrão; a Seção 4 define o problema a ser tratado; a Seção 5 propõe a modelagem do problema de escalonamento e detalha sua formulação; a Seção 6 traz a validação experimental da proposta; por fim, a Seção 7 apresenta a conclusão e os trabalhos futuros.

2. Trabalhos Relacionados

Dentre os trabalhos na literatura pertinente e que possuem relação com o tema aqui abordado, Zhang et al. [2021] aborda um problema $\mathcal{NP}$-Difícil na alocação de recursos em um cenário de *Multi-Access Edge Computing* (MEC) com mobilidade. No cenário abordado, veículos usuários de um aplicativo de transporte em carros de passeio precisam estabelecer conexões sem fio, enquanto se movimentam, a alguma base de rádio dentre as que abrigam os servidores MEC, conforme o raio de alcance dessas bases. Essas bases possuem capacidade limitada e precisam ser devidamente dimensionadas para evitar sobrecarregadas. O trabalho propõe um modelo de otimização

DOI: 10.59254/sbpo-2023-175211
http://live-proceedings.pantheonsite.io/p/175211

para dimensionar a alocação de recursos nos servidores MEC instalados nas bases, buscando maximizar a disponibilidade de conexões para os veículos e minimizar latências. Há similaridades com o problema abordado neste artigo, porém a dinâmica na alocação de recursos tem natureza diferente da que é tratada neste cenário.

Botez et al. [2021] aborda o problema de escalabilidade de funções virtualizadas de rede, ou *Network Functions Virtualization* (NFV), em ambientes de computação em nuvem voltados para aplicações no estado da arte de redes 5G. Nesse trabalho, o Kubernetes é usado como orquestrador para escalar as funções de rede (*User Plane Functions*, ou UPS), porém a solução não contempla um modelo em larga escala.

Wojciechowski et al. [2021] apresenta uma forma de otimizar o escalonamento de *workloads* sensíveis à latência no Kubernetes, visando diminuir o tempo de atendimento às requisições. O cenário de aplicação é tipicamente o de redes 5G fazendo uso de MEC. O modelo apresentado busca verificar se um nó tem ou não capacidade para suportar a alocação de um novo *pod*, segundo algumas métricas de desempenho e seus níveis de prioridade.

O trabalho de Zhong e Buyya [2020] propõe uma estratégia de alocação de tarefas em *containers* de um *cluster* Kubernetes, objetivando uma redução de custo. Isso se dá através da redução do tamanho do *cluster* conforme o encerramento dessas tarefas, o que é possível a partir de ajustes nas configurações dos *containers* e do *cluster* em si. Na nossa abordagem, não há modificações do *cluster* ou dos *containers* em execução, com o algoritmo de escalonamento atuando apenas sobre novas demandas de alocação de *pods*. Além disso, o modelo que propomos admite penalizações aos nós do *cluster* conforme a incidência de erros na execução das tarefas respectivamente alocadas.

Por seu turno, em relação ao contexto histórico do domínio de Otimização Combinatória, o trabalho de Cornuéjols et al. [1983], lançou as bases para abordagens eficazes na alocação de recursos em redes e definiu o *Facility Location Problem*. Este aborda o desafio de selecionar locais para instalações de galpões de estoque, considerando a minimização dos custos de operação e transporte, que corresponde a uma questão central em muitos problemas práticos de escalonamento e alocação. Analogamente, o trabalho de Walker e Chaudhuri [1995] traz uma introdução ao problema de escalonamento, definindo-o formalmente, apresentado diferentes variações de tal problema e apresenta quatro algoritmos comumente empregados na abordagem a problemas de escalonamento, em que três deles consistem em algoritmos construtivos que selecionam iterativamente a operação a ser escalonada, agindo de maneira gulosa e produzindo soluções razoáveis rapidamente, à medida que o último corresponde a uma formulação de Programação Inteira.

Além desses, outro trabalho relevante é o de Fischetti et al. [2016], no qual é considerado o *Capacitated Facility Location Problem* em sua versão clássica e uma variante, de modo que foram feitos experimentos considerado o uso da estratégia de decomposição na formulação de Programação Linear Inteira Mista. O trabalho de Kuehn e Hamburger [1963] também trouxe impressões valiosas sobre o transporte de recursos em redes, embora seu foco tenha sido em problemas de transporte fixo, as técnicas de otimização discutidas são pertinentes para discutir processos de escalonamento.

Finalmente, em Fischetti et al. [2017], é oferecida uma perspectiva atualizada sobre o *Uncapacitated Facility Location Problem*, considerando o reprojeto necessário nas estratégias de decomposição clássicas para que se tire proveito das novas tecnologias de hardware e software. A estratégia proposta permitiu uma aceleração na decomposição do PLIM e foi possível resolver à otimalidade instâncias da literatura que ainda estavam em aberto, isto é, para as quais não se conhecia ainda a solução ótima.

DOI: 10.59254/sbpo-2023-175211
http://live-proceedings.pantheonsite.io/p/175211

3. Escalonamento de *Pods* no Kubernetes

Esta seção descreve o funcionamento do escalonador padrão de um *cluster* Kubernetes, o **Kube-scheduler**, responsável pela alocação de *pods* aos nós ativos do *cluster*. A alocação é realizada com base num critério de ranqueamento para os nós válidos e passíveis de receberem *pods* (CNCF [2023b]).

A criação de um *pod* costuma ser demandada por alguma aplicação em execução ou pela equipe de desenvolvedores e/ou operadores dos serviços em nuvem. Por intermédio de alguns parâmetros de configuração sempre que um novo *pod* é criado, é possível associar prioridades a determinados nós, a critério dos administradores, que são usadas para determinar a alocação deste novo *pod*. Dentre esses parâmetros, destacam-se o **NodeAffinity**, o **TaintToleration**, o **PodTopologySpread** e o **InterPodAffinity**.

O **NodeAffinity** é utilizado para especificar as preferências e restrições relacionadas aos nós do *cluster*. É usado para definir onde um *pod* será executado, como em nós que tenham um determinado rótulo, ou para evitar que um *pod* seja executado em nós com um rótulo específico. O **InterPodAffinity** permite especificar as preferências de execução de alguns *pods* em relação a outros *pods*. Pode ser usado para que grupos de *pods* sejam executados num determinado nó ou em nós separados. O **PodTopologySpread** controla a distribuição dos *pods* em relação à topologia do *cluster*, ajudando a garantir alta disponibilidade e a tolerar falhas através da distribuição de *pods* de um mesmo grupo em diferentes nós do *cluster*. E por último, o **TaintToleration** permite que sejam especificados *taints* (marcas) nos nós do *cluster* para, em seguida, definir quais *pods* os "toleram". Isso é útil para restringir quais *pods* podem ser executados em nós específicos.

Existem outros tipos de rótulos ou parametrizações que, de acordo com os pesos definidos pelo escalonador, irão determinar onde um *pod* será executado. Entretanto, não há qualquer parâmetro disponibilizado por padrão pelo orquestrador que faça uma ponderação em relação ao custo individual de um nó em uma infraestrutura física local ou em nuvem. Sendo assim, este artigo tem por objetivo a proposição de um algoritmo que faça uma alocação de *pods* a nós que não apenas replique o método padrão disponibilizado pelo Kube-scheduler, com suas possíveis variações, mas que seja capaz de alocar *pods* visando uma redução de custos e um melhor aproveitamento da infraestrutura existente.

O Algoritmo 1 descreve a lógica de alocação de *pods* a nós realizada pelo *Kube-scheduler*. Assim, a entrada do algoritmo corresponde a uma lista de *pods* P a serem alocados e uma lista de nós disponíveis N. Inicialmente, na linha 3, o algoritmo inicializa um $mapeamento$ vazio de *pods* a nós, que guardará a alocação a ser feita. Após isso, para cada pod $p \in P$ e cada nó $n \in N$, é verificado se o nó n possui capacidade disponível suficiente para atender ao pod p, isto é, se, mesmo considerando os pods já alocados a n nesse ponto, tal nó ainda tem capacidade disponível que atenda ao requerimento de uso de recursos do pod p (linhas 4 - 6). Se essa condição é satisfeita, o pod p é alocado ao nó n e essa alocação é adicionada na variável $mapeamento$, passando-se a considerar então o próximo pod da lista (linhas 7 - 9). Por sua vez, nas linhas 12 e 13, caso não haja nenhum nó que tenha capacidade disponível suficiente para atender ao uso de recursos do pod p, então p não é alocado a nenhum nó. O algoritmo executa esses passos até que todos os pods em P tenham sido considerados e, ao fim, na linha 16, retorna a alocação obtida na variável $mapeamento$.

4. Problema

Um novo *pod* deve ser sempre alocado para o nó mais barato dentre aqueles disponíveis e que possuam recursos suficientes para acomodá-lo. Supondo um dado *slot* de tempo t, haverá sempre uma determinada quantidade de *pods* em execução para cada um dos nós ativos. A medida que os *slots* de tempo vão passando – por exemplo, $t + 1$, $t + 2$ e assim por diante –, a quantidade

DOI: 10.59254/sbpo-2023-175211
http://live-proceedings.pantheonsite.io/p/175211

Algorithm 1 Algoritmo do Kube-scheduler

Entrada: Lista de pods pendentes P, lista de nós disponíveis N
Saída: Mapeamento de pods em P a nós em N
Inicialize um mapeamento vazio de pods a nós: $mapeamento \leftarrow \{\}$
for cada pod $p \in P$ **do**
  for cada nó $n \in N$ **do**
    if n tem capacidade disponível suficiente para servir p **then**
      $mapping \leftarrow (p, n)$
      Atualize a capacidade do nó n baseado no uso de recursos do pod p
      continue {Avance para o próximo pod}
    end if
  end for
  if não há nó $n \in N$ que tenha capacidade disponível para atender p **then**
    continue {Esse pod não será alocado a nenhum nó}
  end if
end for
return $mapeamento$

de *pods* em cada nó naturalmente se altera, em função do término de algumas execuções e das novas alocações que porventura tenham surgido. A cada novo *slot* de tempo, avalia-se a quantidade necessária de nós para atender tais demandas conforme a capacidade individual dos nós existentes, a quantidade de *pods* existentes no *cluster*, os recursos em uso por cada *pod* e os recursos demandados pelo novos *pods* que foram gerados.

Sendo assim, postulamos aqui que a solução ótima para este problema envolve a alocação de *pods* a nós buscando minimizar o custo associado ao resultado final desta alocação, de acordo com a disponibilidade de recursos de cada nó. Além disso, percebemos que o problema aqui tratado guarda similaridade com o *Capacitated Facility Location Problem* (CFLP), que é um problema sabidamente $\mathcal{NP}$-Difícil existente na literatura [Liu, 2009]. Enquanto o CFLP busca as melhores localizações para novas instalações, nós buscamos o melhor nó para a alocação de um determinado *pod*. O objetivo de ambos os problemas é o mesmo: minimizar os custos e maximizar a eficiência do sistema, buscando a melhor combinação que leve a isso.

O problema abordado envolve um conjunto de $N = \{n_1, n_2, \ldots, n_p\}$ nós disponíveis no *cluster* Kubernetes para o orquestrador alocar cada um dos $P = \{p_1, p_2, \ldots, p_k\}$ *pods* a esses nós a cada *slot* de tempo t. Esta alocação envolve também decidir quais nós serão ou não abertos, a cada t, para atender tal demanda. Mais especificamente, deseja-se obter uma alocação que minimize o custo a ela associado, onde o custo de uma alocação é definido pelos seguintes critérios: (i) custo de abertura dos nós utilizados; (ii) custo de alocação de cada *pod* aos nós; e (iii) custo relacionado à quantidade de erros registrados num dado nó em decorrência dos *pods* que rodaram ou ainda rodam neste nó.

O cenário que exemplifica o problema que aqui lidamos é o seguinte: num dado *slot* de tempo t, para cada nó i do *cluster* uma determinada quantidade de *pods* P_i é alocada. Nos *slots* de tempo seguintes, mais *pods* podem ser inseridos num dado nó i, chegando a um momento em que sua capacidade chega ao limite e o orquestrador entende que é necessária a abertura de outras máquinas (nós), a depender de como está a configuração dos demais nós, já abertos, para lidar com este tipo de situação. Quanto maior o número total de nós abertos, maior a quantidade total de

DOI: 10.59254/sbpo-2023-175211
http://live-proceedings.pantheonsite.io/p/175211

recursos em uso. Quanto maior a quantidade de recursos em uso, o que também inclui o número de nós abertos, maior é o custo operacional da alocação.

A Figura 1 ilustra um exemplo de dimensionamento do número de nós necessários para atender um dado conjunto de *pods* em execução. Além do que já foi descrito em termos de alocação de *pods* a nós i, podemos verificar que no slot t, temos um cluster *Kubernetes* de três nós com 5 *pods* em cada nó. No slot $t+1$, dois *pods* foram removidos do nó 3, deixando-o com apenas 3 *pods*. No slot $t+2$, mais 2 *pods* foram removidos, sobrando apenas um *pod*. Por fim, no *slot* $t+3$ o nó 3 não é mais necessário, visto que os *pods* em atividade foram removidos. Portanto, a manutenção de um terceiro nó aberto tornaria a solução mais cara e o mesmo seria inutilizado. O problema consiste, portanto, em um *slot* de tempo t, estabelecer um arranjo de alocação de *pods* a nós do *cluster* que resulte no menor custo operacional possível, conforme os critérios adotados para tal. De maneira sintética, este problema pode corresponder à seguinte pergunta de pesquisa: qual é a alocação que permite ao *cluster* Kubernetes atender a demanda existente de *pods* utilizando o menor número de nós e incorrendo no menor custo possível?

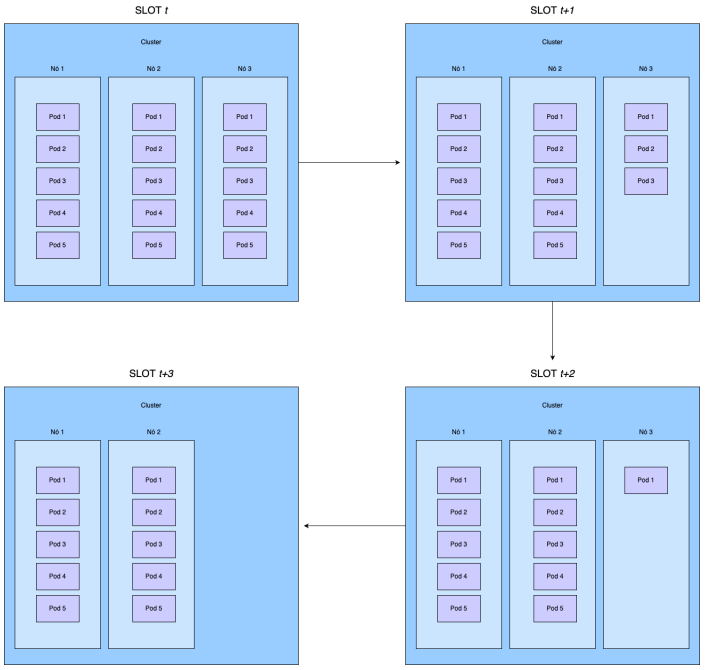


Figura 1: Representação de cluster Kubernetes recebendo alocações em diferentes slots de tempo

5. Modelagem

Conforme definido anteriormente, o Kubernetes é um orquestrador de *containers* e estes *containers* ficam em uma mínima unidade do *cluster*, chamada de *pod*. Estes *pods* são responsáveis por escalar os serviços (aplicações) e atender às demandas direcionadas a esses serviços, dentro da capacidade de uso do *cluster*. A capacidade do *cluster* está diretamente relacionada aos "tamanhos" (capacidades) das máquinas computacionais alocadas para cada nó funcionar, pois os nós são nada mais do que máquinas virtuais ou físicas acopladas a este orquestrador.

Com base nas definições apresentadas na Seção 4, foi construída uma formulação de Programação Linear Inteira Mista (PLIM) baseada no clássico problema de *Capacitated Facility Location Problem* (CFLP) [Liu, 2009], que objetiva apontar a localização ideal para novas fábricas

DOI: 10.59254/sbpo-2023-175211
http://live-proceedings.pantheonsite.io/p/175211

de forma que o custo de atendimento aos consumidores seja minimizado. Como dissemos, o CFLP guarda grande similaridade com o problema aqui abordado e, por isso, resolvemos utilizá-lo como base. A descrição das variáveis e dos coeficientes adotados, como conjuntos de *pods*, de nós, capacidade dos recursos, associação entre nó e *pod*, dentre outros, está apresentada na Tabela 1. É importante citar que, nesta modelagem, olha-se somente para um instante de tempo *t* fixado e, por isso, t não aparece como mais um índice das variáveis e coeficientes.

Esta formulação de minimização de custo obtida é sujeita a um total de seis conjuntos de restrições. A restrição (1) define que o número de nós sendo utilizados deve ser de pelo menos um. Por sua vez, a restrição (2) afirma que um *pod* j só pode estar alocado a um nó i que esteja aberto. Em (3), é dito que, para cada nó i, deve haver pelo menos um *pod* j alocado ao nó i. Em (4), a quantidade de recursos utilizados por todos os *pods* alocados a um nó i não pode ser superior à capacidade de recursos do nó i. Finalmente, as restrições (6) e (7) informam que x_i e $y_{i,j}$ são variáveis binárias.

n	Número total de nós pertencentes a um *cluster* C
m	Número total de *pods* associados a quaisquer nós do *cluster* C
x_i	Variável de decisão que designa se um nó i está aberto ou não
$y_{i,j}$	Variável de decisão que designa se um *pod* j é atendido por um nó i
u_j	Quantidade de recursos utilizados por um *pod* j
e_j	Quantidade de erros de um *pod* j
U_i	Capacidade de recursos de um nó i
α_i	Custo operacional associado à abertura (uso) do nó i
β_i	Custo operacional associado à alocação de um *pod* a um nó i
γ_i	Custo operacional associado ao acúmulo de registros de erro e_j relativos a cada *pod* j alocado ao nó i

Tabela 1: Descrição das variáveis e coeficientes usados na formulação do problema.

A função objetivo busca alocar *pods* ao nó i que possua um menor custo de alocação β_i, assim como evitar a penalização (α_i) pela abertura de novos nós. Além disso, busca-se evitar a alocação de *pods* a nós que acumulem um maior registro de erros e_j vinculados aos *pods* j associados a esses nós.

DOI: 10.59254/sbpo-2023-175211
http://live-proceedings.pantheonsite.io/p/175211

$$\min \quad \sum_{i=1}^{n} \left(\alpha_i x_i + \beta_i \sum_{j=1}^{m} y_{i,j} + \gamma_i \sum_{j=1}^{m} e_j y_{i,j} \right)$$

sujeito a

$$\sum_{i=1}^{n} x_i \geq 1 \tag{1}$$

$$y_{i,j} \leq x_i \qquad i = 1, \ldots, n \qquad j = 1, \ldots, m \tag{2}$$

$$\sum_{j=1}^{m} y_{i,j} \geq 1 \qquad i = 1, \ldots, n \tag{3}$$

$$\sum_{j=1}^{m} u_j y_{i,j} \leq U_i \qquad i = 1, \ldots, n \tag{4}$$

$$x_i \in \{0, 1\} \qquad i = 1, \ldots, n \tag{5}$$

$$y_{i,j} \in \{0, 1\} \qquad i = 1, \ldots, n \qquad j = 1, \ldots, m \tag{6}$$

6. Avaliação Experimental

Nesta seção, é descrita a avaliação experimental conduzida, a fim de se comparar uma implementação do escalonador padrão do Kubernetes com o algoritmo baseado na formulação proposta.

6.1. Metodologia

Com o propósito de validar e avaliar a eficácia da solução proposta, foi gerado um conjunto de instâncias artificiais considerando os seguintes pares de números de nós e de *pods*: $N = \{10, 20, 50, 100, 200\}$ e $P = \{50, 100, 200, 500, 1000, 5000, 10000\}$.

Esses pares foram usados como entrada para dois algoritmos de escalonamento codificados em Java: um deles simulava o Kube-scheduler padrão, onde os *pods* eram instanciados nos nós de forma aleatória, e o outro implementa, através do resolvedor Gurobi[1], a metodologia de escalonamento customizada para o Kube-scheduler, atendendo a modelagem proposta neste artigo.

Para gerar artificialmente as instâncias de entrada para ambos escalonadores, foram sorteados valores para os parâmetros U_i, u_j, α_i e β_i a cada combinação de número de *pods* e número de nós. O parâmetro γ_i foi considerado zero em todos os casos para que a variável e_j, relacionada ao erro registrado para cada *pod* j, não seja computada em nenhum dos escalonadores.

A faixa de valores usada para o sorteio do parâmetro U_i, associado a cada nó i da instância de entrada de um dado par de número de *pods* e número de nós, teve como limite inferior o número de *pods* m sobre o número de nós n mais um, ou seja, $m/(n+1)$. Já o limite superior foi igual ao número de *pods* multiplicado por dois, ou seja, $2n$. Com relação aos valores sorteados para o parâmetro u_j, associado a cada *pod* j, usou-se o intervalo entre 1 e 10.

Quanto ao custo de alocação α_i, associado à abertura de um nó i, foi sorteado um valor entre o limite mínimo de 1 e o limite máximo de $4n$, que é o número de nós multiplicado por quatro. O mesmo intervalo se aplica para o sorteio do valor β_i, relacionado à alocação de um *pod* qualquer a um nó i.

[1] https://www.gurobi.com/

DOI: 10.59254/sbpo-2023-175211
http://live-proceedings.pantheonsite.io/p/175211

Para fins de obtenção da estimativa de tempo de execução dos algoritmos, foram executadas 10 rodadas de cada instância de entrada, sem modificações entre elas, e computado o tempo médio de execução dessas 10 rodadas. Já com relação à decisão de parada para a implementação da solução proposta usando o resolvedor Gurobi, foi usado o parâmetro `MIPGap` configurado em 0,01 [Gurobi Optimization, 2023b].

6.2. Ambiente Computacional

O ambiente computacional adotado nos experimentos possui a seguinte configuração: 8 núcleos de CPU Apple M1, 16 GB de memória RAM e sistema operacional MacOS Monterey 12.3.1. A implementação de ambas as abordagens foi feita na linguagem de programação Java na versão 17.0.4.1 [Oracle, 2023]. Para executar a formulação de PLIM proposta, descrita na Seção 5, foi utilizada a versão 10.0.0 do resolvedor Gurobi a partir de sua API para Java [Gurobi Optimization, 2023a]. Os códigos desenvolvidos nesta pesquisa, assim como o resultado dos experimentos, estão disponíveis publicamente em um repositório criado no Github[2].

6.3. Resultados

Os resultados obtidos nos experimentos computacionais seguem expostos nas Tabelas 2 e 3. A Tabela 2 apresenta, para cada configuração de nós e *pods* testada, o valor da solução obtida e o tempo médio, em millisegundos, das 10 rodadas para a implementação do Kube-scheduler padrão.

Número de pods	Número de nós	Valor da solução	Tempo (ms)
10	10	399	< 1
10	20	446	< 1
10	50	1244	< 1
10	100	3854	< 1
10	200	7667	< 1
50	10	1495	1
50	20	2016	< 1
50	50	4228	< 1
50	100	13578	< 1
50	200	27938	< 1
100	10	2407	< 1
100	20	3954	< 1
100	50	15050	< 1
100	100	19329	< 1
100	200	56409	< 1
200	10	5342	< 1
200	20	7230	< 1
200	50	8709	< 1
200	100	40981	< 1
200	200	88353	< 1
500	10	9776	< 1
500	20	20560	< 1
500	50	58032	< 1
500	100	96226	< 1
500	200	121867	< 1
1000	10	28037	< 1
1000	20	48455	< 1
1000	50	125517	< 1
1000	100	196462	< 1
1000	200	498364	< 1
5000	10	131090	14
5000	20	188764	11
5000	50	511526	11

[2]https://github.com/camilla-m/facility-problem

DOI: 10.59254/sbpo-2023-175211
http://live-proceedings.pantheonsite.io/p/175211

5000	100	731992	11
5000	200	2113569	12
10000	10	175960	48
10000	20	480883	49
10000	50	895451	45
10000	100	1728710	52
10000	200	2356808	50

Tabela 2: Resultados obtidos pela implementação do Kube-scheduler padrão para as instâncias artificiais geradas.

Por seu turno, a Tabela 3 expõe, para as mesmas configurações de nós e *pods*, o valor da solução obtida e o tempo médio, em millisegundos, das 10 rodadas para a formulação de PLIM proposta. Adicionalmente, é exibida na última coluna a taxa entre o valor das soluções obtidas pela implementação do *Kube-scheduler* padrão e pela formulação de PLIM proposta.

Número de pods	**Número de nós**	**Valor da solução**	**Tempo (ms)**	**Taxa**
10	10	250	< 1	1.60
10	20	158	1	2.82
10	50	356	10	3.49
10	100	586	5	6.57
10	200	678	9	11.30
50	10	573	12	2.61
50	20	1195	7	1.69
50	50	873	8	4.84
50	100	1640	45	8.28
50	200	1964	31	14.22
100	10	1155	1	2.08
100	20	1270	5	3.11
100	50	1530	18	9.83
100	100	1718	52	11.25
100	200	1539	94	36.65
200	10	2653	3	2.01
200	20	2348	5	3.07
200	50	1039	43	8.38
200	100	3641	123	11.25
200	200	1912	234	46.20
500	10	3383	8	2.88
500	20	5428	15	3.78
500	50	6361	112	9.12
500	100	11634	54	8.27
500	200	5568	815	21.88
1000	10	4354	66	6.43
1000	20	3319	38	14.59
1000	50	17289	56	7.26
1000	100	21298	1160	9.22
1000	200	20331	3220	24.51
5000	10	60671	102	2.16
5000	20	60130	843	3.14
5000	50	53105	310	9.63
5000	100	28751	609	25.45
5000	200	70832	1367	29.83
10000	10	94909	166	1.85
10000	20	137247	395	3.50
10000	50	90058	8413	9.94
10000	100	77527	11418	22.29

DOI: 10.59254/sbpo-2023-175211
http://live-proceedings.pantheonsite.io/p/175211

10000	200	63702	3038	36.99

Tabela 3: Resultados obtidos pela implementação da formulação de PLIM para as instâncias artificiais geradas.

6.4. Discussão dos resultados

Conforme dito, o objetivo dos experimentos foi o de validar a solução proposta tendo como base de comparação o escalonador padrão do Kubernetes. Para fins desta comparação, usamos como métrica o ganho, em termos de custo computacional, refletido no valor da solução, da nossa proposta contra a do escalonador padrão e também o tempo gasto em ambas as propostas.

Pode-se perceber que, na Tabela 3, para algumas instâncias com o mesmo número de *pods*, o custo da solução obtida pode aumentar quando se aumenta o número de nós, diferentemente do que seria esperado. Isso acontece devido ao sorteio aleatório que é feito para a geração das instâncias artificiais, que pode chegar a configurações, por exemplo, em que os nós possuem capacidades baixas com custos de abertura e de alocação altos.

Em relação ao custo computacional, expresso no valor das soluções obtidas pelos dois algoritmos para as instâncias artificiais geradas, tem-se uma diferença média de aproximadamente uma ordem de grandeza obtida pela solução proposta, conforme exposto na Tabela 3. Tal fato implica, sob um ponto de vista de ordem prática, num ganho em termos de utilização da infraestrutura física, ganho esse trazido pela nossa proposta.

Por outro lado, deve-se destacar o fato de que a alocação de *pods* a nós é uma atividade considerada crítica, e que, portanto, deve ser realizada dentro de um intervalo de tempo limite imposto pelo cenário de uso. Nesse sentido, fica evidente que a solução proposta possui um tempo de execução muito maior que a do escalonador padrão, em função do método usado para se buscar a solução ótima do problema.

7. Conclusão

Este trabalho propôs um método baseado em uma formulação de Programação Linear Inteira Mista cujo objetivo é otimizar o custo operacional de alocação de *pods* a nós em um *cluster* Kubernetes. A solução baseia-se na quantidade disponível de espaço nos nós do *cluster* e no custo de alocação de um *pod* a um nó, além do próprio custo de abertura de um nó no *cluster*. A formulação proposta foi comparada com uma implementação do escalonador padrão do Kubernetes em relação ao custo das soluções obtidas e ao tempo de processamento. Os resultados obtidos validaram a aplicação da proposta nos cenários de uso estudados.

Para trabalhos futuros, deseja-se realizar experimentos em nuvem com um cenário real de uso, a partir da modificação do algoritmo do escalonador padrão do Kubernetes, utilizando-se a formulação proposta. Busca-se ainda trabalhar em uma proposta que leve em consideração um horizonte composto de instantes de tempo, de modo que se resolva um problema de alocação, por meio da formulação, para cada instante. Isso naturalmente introduzirá uma complexidade maior ao problema, sobretudo porque será necessário adotar uma fila a fim de contemplar os *pods* que porventura não conseguirem sua alocação em um dado instante de tempo.

Referências

Botez, R., Costa-Requena, J., Ivanciu, I.-A., Strautiu, V., e Dobrota, V. (2021). Sdn-based network slicing mechanism for a scalable 4g/5g core network: A kubernetes approach. *Sensors*, 21(11). ISSN 1424-8220. URL https://www.mdpi.com/1424-8220/21/11/3773.

DOI: 10.59254/sbpo-2023-175211
http://live-proceedings.pantheonsite.io/p/175211

CNCF (2023a). Kubernetes. https://kubernetes.io/. Accessed: 2023-05-09.

CNCF (2023b). Kubernetes - kube-scheduler. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/. Accessed: 2023-04-22.

Cornuéjols, G., Nemhauser, G., e Wolsey, L. (1983). The uncapicitated facility location problem. Technical report, Cornell University Operations Research and Industrial Engineering.

Docker (2023). What's a container? - docker. https://www.docker.com/resources/what-container/. Accessed: 2023-05-09.

Fischetti, M., Ljubić, I., e Sinnl, M. (2017). Redesigning benders decomposition for large-scale facility location. *Management Science*, 63(7):2146–2162.

Fischetti, M., Ljubić, I., e Sinnl, M. (2016). Benders decomposition without separability: A computational study for capacitated facility location problems. *European Journal of Operational Research*, 253(3):557–569. ISSN 0377-2217. URL https://www.sciencedirect.com/science/article/pii/S0377221716301126.

Gurobi Optimization (2023a). Java API Overview. https://www.gurobi.com/documentation/current/refman/java_api_overview.html.

Gurobi Optimization (2023b). MIPGap. https://www.gurobi.com/documentation/9.5/refman/mipgap2.html.

Kuehn, A. A. e Hamburger, M. J. (1963). A heuristic program for locating warehouses. *Management science*, 9(4):643–666.

Liu, B. (2009). *Facility Location Problem*, p. 157–165. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-540-89484-1. URL https://doi.org/10.1007/978-3-540-89484-1_11.

Oracle (2023). Java SE 17 Archive Downloads. https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html.

Walker, R. e Chaudhuri, S. (1995). Introduction to the scheduling problem. *IEEE Design Test of Computers*, 12(2):60–69.

Wojciechowski, L., Opasiak, K., Latusek, J., Wereski, M., Morales, V., Kim, T., e Hong, M. (2021). Netmarks: Network metrics-aware kubernetes scheduler powered by service mesh. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, p. 1–9.

Zhang, J., Chen, S., Wang, X., e Zhu, Y. (2021). Deepreserve: Dynamic edge server reservation for connected vehicles with deep reinforcement learning. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, p. 1–10.

Zhong, Z. e Buyya, R. (2020). A cost-efficient container orchestration strategy in kubernetes-based cloud computing infrastructures with heterogeneous resources. *ACM Trans. Internet Technol.*, 20 (2). ISSN 1533-5399. URL https://doi.org/10.1145/3378447.

DOI: 10.59254/sbpo-2023-175211
http://live-proceedings.pantheonsite.io/p/175211