Escalonamento de Workflows em Múltiplos Provedores de Nuvem usando Programação Linear

Thiago A. L. Genez[1], **Luiz F. Bittencourt**[1] **& Edmundo R. M. Madeira**[1]

[1]Instituto de Computação
Universidade Estadual de Campinas (UNICAMP)
Av. Albert Einstein, 1252 – CEP 13.083-852 – Campinas – SP – Brasil
{thiagogenenz | bit | edmundo }@ic.unicamp.br

Abstract

Complex applications in e-science and e-business are usually modeled as workflows which execution in clouds can minimize the upfront investment in computational resources. Therefore, the development of algorithms to schedule workflows in clouds is of major interest. In this paper we propose and evaluate the use of different levels of timeline discretization utilizing linear programming to solve the problem of scheduling workflows with deadline constraints in multiple cloud providers. Simulations show that increasing the granularity of time discretization decreases the scheduler execution time, making it possible to achieve solutions faster when scheduling workflows that present larger execution times.

Keywords: Cloud computing, workflow scheduling, integer linear programming, time discretization.

Resumo

Aplicações complexas de e-Ciência e e-Business são geralmente modeladas como workflows cujas execuções em nuvens podem minimizar investimentos em infraestrutura computacional. Então, o desenvolvimento de algoritmos para escalonamento de workflows em nuvens é de grande interesse. Neste trabalho propomos e avaliamos o uso de diferentes níveis de discretização da linha do tempo por meio da programação linear para resolver o problema de escalonamento de workflows com deadline em múltiplos provedores de nuvem. Experimentos mostram que o aumento da granularidade da discretização do tempo diminui o tempo de execução do escalonador, possibilitando alcançar soluções mais rápidas no escalonamento de workflows que apresentam tempos de execução maiores.

Palavras-chave: Computação em nuvem, escalonamento de *workflow*, programação linear inteira, discretização do tempo.

1. INTRODUÇÃO

Computação em nuvem emergiu como um novo paradigma, onde hardware e software são entregues como serviços de utilidade geral e disponibilizados para os usuários através da Internet. Graças à camada de virtualização, construída sobre os recursos físicos, é possível otimizar o uso da infraestrutura física e, então, oferecer (virtualmente) diferentes tipos de hardware e software aos usuários da nuvem. Além disso, a computação em nuvem possibilita que os recursos virtualizados (serviços) sejam alugados e liberados de acordo com a necessidade dos usuários e tarifados por meio do modelo "pago-pelo-uso" (do inglês, *pay-as-you-go*) [19].

Várias aplicações complexas de e-Ciência e *e-Business* podem ser modeladas como *workflow*, compondo um conjunto de serviços a serem processados em uma ordem bem-definida [10]. Além disso, cada serviço pode depender de dados computados por outros serviços que o antecedem. Dessa forma, *workflows* são geralmente representados por um grafo acíclico direcionado – *Directed Acyclic Graph* (DAG) – para representar os serviços e suas dependências. Escalonamento de *workflow* é um problema NP-Completo e, como consequência, aprimoramentos em algoritmos e heurísticas são fundamentais para melhorar o desempenho de sua execução em nuvens.

Desenvolvemos um Programa Linear Inteiro (PLI) para escalonar *workflows* em nuvens [7], cuja execução é realizada por um provedor de software como serviço –

Software-as-a-Service (SaaS). Nesse trabalho, o cliente envia seu *workflow* para ser executado pelo provedor de SaaS, juntamente com um tempo de resposta (*deadline*) a ser obedecido. Entretanto, dependendo do tamanho do DAG e do *deadline*, a linha temporal discreta do PLI torna-se grande e, como conseqüência, o tempo de execução do escalonador aumenta consideravelmente. Dessa forma, esse escalonador fica limitado somente a DAGs pequenos (≈ 25 nós), pois não é possível encontrar soluções viáveis em tempo hábil para DAGs maiores, ou seja, com grande números de nós e dependências.

A contribuição deste trabalho está na proposta e avaliação do uso de diferentes níveis de granularização da linha do tempo no programa linear inteiro, que resolve o problema de escalonamento de *workflow* na computação em nuvem. Portanto, o principal objetivo deste artigo é mostrar que o aumento da granularidade da discretização do tempo permite ao escalonador encontrar mais rapidamente soluções viáveis para DAGs com grande número de nós e dependências; e, ao mesmo tempo, minimizar os custos monetários com a terceirização de recursos computacionais. Por uma questão de clareza, utilizamos apenas SaaS para referenciar ao provedor de nuvem, porém este trabalho também aplica-se aos provedores de plataforma como serviço – *Plataform-as-a-Service* (PaaS).

O restante deste artigo está organizado da seguinte maneira. Conceitos básicos e o problema do escalonamento são descritos na Seção 2, enquanto trabalhos relacionados são expostos na Seção 3. Uma visão geral do algoritmo usado no escalonador é apresentada na Seção 4. A Seção 5 descreve a discretização da linha do tempo proposta a esse escalonador, enquanto resultados experimentais são analisados na Seção 6. A Seção 7 finaliza este artigo com as conclusões e trabalhos futuros.

2. CONCEITOS BÁSICOS

Nesta seção introduzimos brevemente os conceitos básicos sobre a representação de *workflows*, o problema de escalonemnto e o cenário do escalonamento de *workflows* em nuvens deste trabalho.

2.1. REPRESENTAÇÃO DE *Workflow*

Um conjunto de serviços pode ser classificado, de modo geral, em duas categorias: serviços independentes ou serviços dependentes. A primeira categoria representa os serviços que podem ser executados em qualquer ordem, pois não realizam comunicação (troca de dados) entre si. A segunda, por outro lado, representa os serviços que possuem uma ordem para serem executados, pois há um fluxo de dados (comunicação entre serviços) bem-definido que deve ser respeitado. Neste artigo, vamos enfatizar o uso do conjunto de serviços dependentes, também chamado de *workflow*, na computação em nuvem.

Um *workflow* de serviços é geralmente representado por um grafo acíclico direcionado (DAG) $\mathcal{G} = \{\mathcal{U}, \mathcal{E}\}$, onde cada nó $u_i \in \mathcal{U}$ representa um serviço a ser executado e cada aresta (ou arco) $e_{i,j} \in \mathcal{E}$ representa uma dependência de dados entre o serviço i e j; ou seja, o peso associado a $e_{i,j}$ representa os dados produzidos por u_i e consumidos por u_j. Isto significa que u_j pode iniciar a sua execução somente após u_i finalizar e enviar todos os dados necessários a u_j. É importante frisar que, por ser um DAG, não há nenhum caminho direcionado que inicia e termina em u, para todo nó $u \in \mathcal{U}$; ou seja, estamos apenas interessados em escalonar *workflows* que não contenham ciclos. Portanto, um *workflow* de serviço compõe um conjunto de serviços a serem processados em uma ordem bem-definida e sem ciclos. Por uma questão de clareza, as palavras *workflow* e *DAG* serão utilizadas como sinônimos no restante deste artigo.

O DAG da Figura 1 representa um aplicativo de processamento do filtro de mediana de imagens [2]. Nessa figura, os rótulos dos nós e das arestas representam, respectivamente, custos de computação (número de instruções, por exemplo) e comunicação (bytes a transmitir, por exemplo). Esse DAG é composto por 14 nós, onde o nó 1 representa a operação de divisão (*fork*) da imagem a ser processada e é o primeiro nó a ser executado. Os nós 2-13 representam a função de processamento do filtro de mediana e podem iniciar suas execuções somente após receberem os dados do nó 1. Terminadas as execuções e o envio dos dados dos nós 2-13, o nó 14 pode começar sua execução, que representa a operação de união (*join*) das imagens inicialmente particionadas. É importante frisar que este trabalho não considera *DAGs condicionais*, onde as dependências de dados podem ser tratadas como uma composição de *Es* e *OUs* lógicos. Consideramos apenas o operador lógico *E*. Por exemplo, a tarefa 14 só pode iniciar sua execução após o término e o recebimento dos dados de *todos* os serviços 2-13.

Podemos assumir, sem perda de generalidade, que todo DAG tem apenas um nó de entrada (primeiro nó) e um nó de saída (último nó). Também assumimos que cada nó $n_i \in \mathcal{U}$ é indivisível e é executado em um único núcleo de processamento virtualizado. Todos os *workflows* considerados neste trabalho são estáticos, isto é, a topologia do DAG não é alterada durante o escalonamento.

2.2. O PROBLEMA DE ESCALONAMENTO

O escalonador é o elemento responsável em determinar qual parte do *workflow* será executada em qual recurso computacional, distribuindo os serviços para execução em paralelo. Em outras palavras, o problema de escalonamento de *workflow* consiste em, dado um conjunto de serviços dependentes e suas dependências, escolher em qual recurso cada serviço será executado. Ordem de pre-

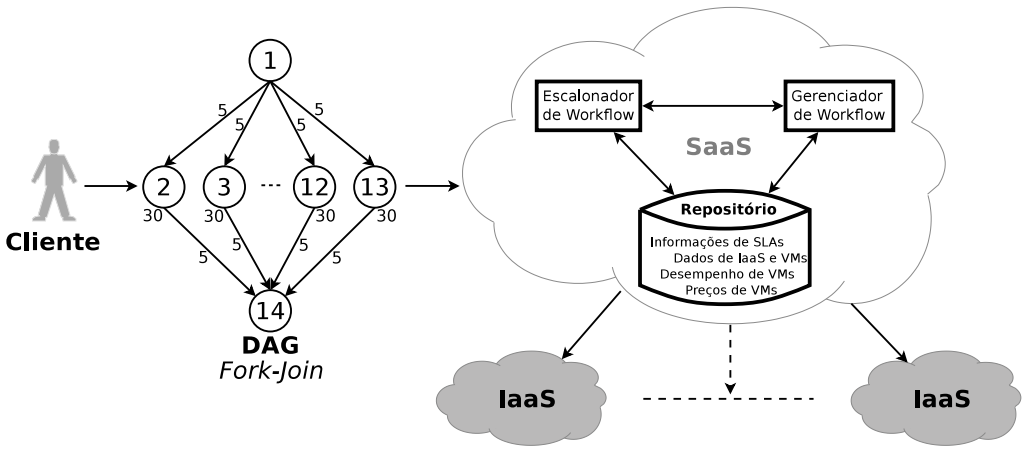


Figura 1. Submissão do DAG *fork-join* com 14 nós a ser executado pelo SaaS

cedência dos serviços, custos de execução (computação) dos serviços, custos de comunicação entre serviços, capacidades de processamento dos recursos e capacidades de transmissão de dados dos enlaces de rede, que interligam esses recursos, são informações importantes que o escalonador deve levar em consideração antes e/ou durante o escalonamento; o qual é realizado de acordo com uma *função objetivo* a ser otimizada.

Exemplos de funções objetivo encontradas na literatura são: minimizar o tempo de execução (*makespan*[1]) do *workflow* [1,14], minimizar o tempo de execução do escalonamento [3,12] ou minimizar custos monetários da execução do *workflow* [2]. Em sua forma geral, o problema de escalonamento de *workflow* é NP-Completo e o problema de otimização associado é NP-Difícil. Em outras palavras, não conhecemos nenhum algoritmo que gere soluções determinísticas ótimas em tempo polinomial. Portanto, algoritmos ótimos para realizar escalonamento de instâncias grandes levam muito tempo para serem executados, tornando, então, as heurísticas e outras abordagens sub-ótimas uma boa opção para realizar o escalonamento.

2.3. Cenário

Para providenciar o escalonamento de *workflows* em nuvens, utilizamos o cenário ilustrado na Figura 2. As principais entidades deste cenário são: clientes, provedor de SaaS e provedores de *Infrastructure-as-a-Service* (IaaS). Cada cliente é uma entidade responsável por submeter um ou mais *workflows* para serem executados pelo provedor de SaaS. Este, por sua vez, utiliza, se existirem, recursos próprios para executar esses *workflows*, de tal modo que deve obedecer os requisitos de *Quality of Service* (QoS) de cada *workflow* submetido. Entretanto, em picos de demanda, a infraestrutura própria do provedor de SaaS pode ser insuficiente para atender todos os clientes ao mesmo tempo. Assim, para garantir os requisitos de QoS das solicitações já aceitas e, ao mesmo tempo, não recusar nenhum pedido de futuros clientes, o provedor de SaaS precisa aumentar seu poder computacional de uma maneira rápida, simples e, principalmente, barata. É neste momento que entra em cena o provedor de IaaS, entidade responsável por fornecer *elasticidade* computacional. Portanto, neste cenário, consideramos que a infraestrutura computacional do provedor de SaaS é composta principalmente por recursos virtualizados alugados de vários provedores de IaaS (nuvem pública), mas também pode conter alguns recursos próprios (nuvem privada).

O provedor de SaaS pode alugar instâncias de máquinas virtuais do provedor de IaaS por meio de dois tipos de acordo de nível de serviço – *Service Level Agreements* (SLA): sob-demanda ou reservado. No SLA sob-demanda, o provedor de SaaS alugará máquinas virtuais sem compromissos de longo prazo e irá pagar somente pelas unidades de tempo (normalmente por hora) utilizadas. Em contrapartida, no SLA reservado, o provedor de SaaS irá alugar os recursos virtualizados com compromissos de longo prazo (1 a 3 anos, por exemplo) e pagará uma taxa antecipada para reservar cada máquina virtual. Entretanto, o provedor de SaaS receberá um desconto significativo sobre a tarifa das unidades de tempo utilizadas de cada instância reservada. O conceito de SLA adotado neste trabalho será semelhante ao conceito usado na *Amazon Elastic Compute Cloud* (*Amazon EC2*)[2].

Para garantir os requisitos de QoS aos clientes, o provedor de SaaS precisa ter certas garantias do funcionamento das máquinas virtuais alugadas. Dessa forma, esse cenário utiliza os conceitos de SLA através de dois níveis. Como pode ser visto na Figura 2, o primeiro nível contém todos os SLAs estabelecidos entre o provedor de SaaS e

[1]Tempo entre o início da execução do primeiro nó do DAG e o término da execução do último nó do DAG.

[2]`http://aws.amazon.com/ec2/`

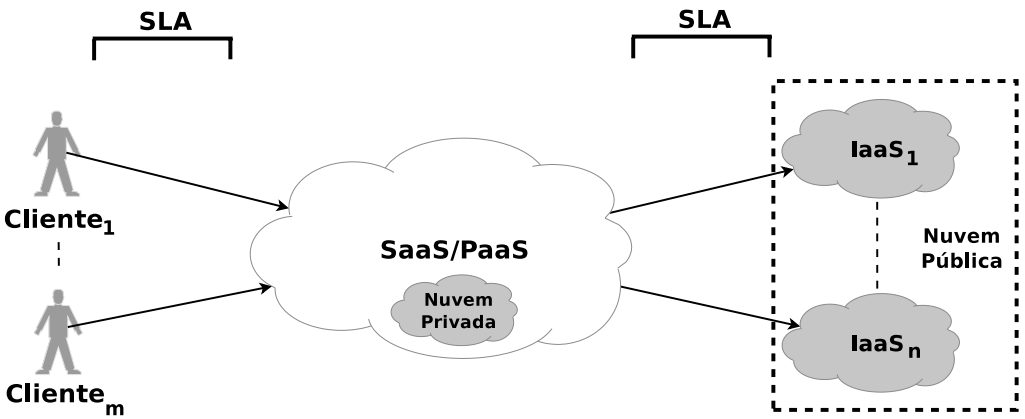


Figura 2. Cenário do escalonamento de *workflows* em nuvens

cada um dos seus clientes, enquanto o segundo nível contém todos os SLAs acordados entre o provedor de SaaS e cada provedor de IaaS. No primeiro nível, o cliente pode estipular, por exemplo, um tempo de resposta máximo (*deadline*) para a execução de seus *workflows*; enquanto no segundo nível, o provedor de SaaS pode alugar máquinas virtuais do provedor de IaaS através dos planos de *reserva* ou *sob demanda*. Portanto, a fim de maximizar o seu lucro, o provedor de SaaS deve minimizar não apenas gastos com nuvens públicas, mas também evitar quebras de SLAs do primeiro nível.

Consideramos esse cenário como um cenário realístico no ambiente da nuvem, pois o provedor de serviços (SaaS ou PaaS) recorre às máquinas virtuais a fim de trazer *elasticidade* ao seu poder computacional. Isto é, esses provedores de serviços não precisam, em picos de demanda, recusar pedidos de clientes quando a infraestrutura privada estiver ocupada ou for insuficiente.

3. Trabalhos Relacionados

Alguns trabalhos propõem soluções para a execução de serviços independentes em nuvem, mas apenas poucos trabalhos consideram a relação custo-benefício do uso de recursos alugados de várias nuvens públicas, a fim de disponibilizar serviços de execução de *workflows* focados em parâmetros de QoS definidos nos SLAs.

Wu *et al.* descrevem em [16] um algoritmo de alocação de recursos para os provedores de SaaS com o propósito de minimizar o custo da infraestrutura alugada e da violação de SLAs. Os autores apresentam políticas de custo-benefício para mapeamento e escalonamento estático de serviços independentes, com a intenção de maximizar o lucro do provedor de SaaS através do uso de vários provedores de IaaS. No entanto, os serviços independentes considerados pelos autores são aplicações corporativas e não científicas. Bossche *et al.* apresentam em [13] uma programação linear inteira binária para escalonar tarefas independentes em nuvens híbridas, obedecendo restrições de *deadline*. São apresentadas possíveis formas de avaliar o custo monetário computacional, com intuito de maximizar o uso do *datacenter* privado e de minimizar o custo monetário da execução de tarefas em nuvens públicas, enquanto satisfaz as restrições de QoS das aplicações. Embora esses trabalhos apresentem avanços no campo de escalonamento de tarefas nas nuvens, nenhum deles considera execução de *workflows*.

Yao *et al.* propõem em [17] um algoritmo de escalonamento de *workflow* em grades computacionais que minimiza o custo da execução do *workflow*, enquanto satisfaz o *deadline* estipulado pelo usuário. O algoritmo é formulado através de um PLI que realiza o escalonamento em duas etapas: (i) decompõe a restrição do tempo de execução do *workflow* (*deadline global*) em janelas de tempo para todas as tarefas (isto é, determina um *deadline local* para cada tarefa); e (ii) seleciona o melhor serviço que satisfaça a restrição da janela de tempo local. Embora esse escalonador minimize o custo monetário da execução do *workflow* respeitando um *deadline*, os autores consideram apenas *workflows sequenciais* para tentar diminuir o tempo de execução do escalonamento. Aliás, afirmam que todo *workflow* pode ser reduzido a um *workflow* sequencial equivalente, por meio do agrupamento dos nós paralelos em um único nó. O uso do *wokflow* sequencial pode facilitar e reduzir o tempo de execução do escalonamento; no entanto, pode diminuir a qualidade da solução.

Stefansson *et al.* descrevem em [12] um estudo de caso sobre o problema de escalonamento. Os autores apresentam uma comparação entre dois modelos matemáticos para esse problema, sendo um formulado com o uso do tempo discreto e outro com tempo contínuo. Em [6], Stefansson *et al.* também comparam o uso do tempo discreto e contínuo na formulação do problema de escalonamento como uma programação linear inteira. Embora o

tempo contínuo solucione casos de testes maiores e apresente uma melhor precisão nos escalonamentos, quando comparado ao uso do tempo discreto, nenhum desses trabalhos considera ambientes de computação em nuvem.

4. VISÃO GERAL DO ALGORITMO DE ESCALONAMENTO

Nesta seção vamos descrever a formulação do escalonador baseada em [7] cuja função objetivo é minimizar os custos monetários da execução do DAG, enquanto satisfaz o *deadline* estipulado no SLA do usuário. Esse escalonador leva em consideração o custo de computação de cada nó do DAG, o custo de comunicação das arestas do DAG, o poder de processamento dos recursos, os preços das máquinas virtuais e a largura de banda dos enlaces que conectam esses recursos. Ao considerar essas variáveis, o escalonador é capaz de decidir qual é o melhor recurso para executar cada nó do DAG. Portanto, o PLI decide entre (i) alugar máquinas virtuais caras e poderosas para acelerar a execução; ou (ii) alugar máquinas virtuais baratas para não desperdiçar recursos (ou dinheiro) e, ao mesmo tempo, cumprir os *deadlines* estipulados.

4.1. NOTAÇÕES E MODELAGEM DO PROBLEMA DE ESCALONAMENTO

O algoritmo de escalonamento é formulado através de PLI e utiliza as seguintes notações:

- $n = |\mathcal{U}|$: número de nós do DAG $\mathcal{G}$ ($n \in \mathbb{N}$);
- $\mathcal{W} = \{w_1, \ldots, w_n\}$: conjunto de demandas de processamento de cada nó $u_i \in \mathcal{U}$, expressa como o número de instruções a serem processadas ($w_i \in \mathbb{R}^+$);
- $f_{i,j}$: quantidade de unidades de dado a serem transmitidas entre os nós $u_i \in \mathcal{U}$ e $u_j \in \mathcal{U}$ ($f_{i,j} \in \mathbb{R}^+$);
- $\mathcal{H}(j) = \Big\{ij : i < j$, existe um arco do vértice i ao vértice j no DAG $\mathcal{G}\Big\}$: conjunto de predecessores imediatos de $u_j \in \mathcal{U}$;
- $\mathcal{D}_\mathcal{G}$: tempo de término (*deadline*) da execução do DAG $\mathcal{G}$ estipulado pelo cliente do provedor de SaaS;
- $\mathcal{I} = \{i_1, \ldots, i_m\}$: conjunto de provedores de IaaS que compõem a infraestrutura computacional do problema;
- δ_i: número máximo de máquinas virtuais que um cliente pode alugar do provedor de IaaS $i \in \mathcal{I}$ em qualquer instante de tempo t ($\delta_i \in \mathbb{N}$).

Seja $\mathcal{S}_i = \sigma_i^r \cup \sigma_i^o$ o conjunto composto por todos os SLAs que foram assinados entre o provedor de SaaS e o provedor de IaaS $i \in \mathcal{I}$. Esse conjunto é formado por dois subconjuntos disjuntos: (i) σ_i^r que contem os SLAs destinados às máquinas virtuais reservadas e que estão prontamente disponíveis e (ii) σ_i^o que inclui apenas os SLAs para máquinas virtuais alugadas sob demanda (*on-the-fly*). Sejam $\mathcal{V}_i^r$ e $\mathcal{V}_i^o$ os conjuntos disjuntos que contêm, respectivamente, máquinas virtuais associadas com os preços para os recursos reservados e sob-demanda do provedor de IaaS $i \in \mathcal{I}$. Assim, $\mathcal{V}_i = \mathcal{V}_i^r \cup \mathcal{V}_i^o$ representa o conjunto de máquinas virtuais disponíveis para serem alugadas do IaaS i. Neste trabalho, vamos assumir que cada máquina virtual $v \in \mathcal{V}_i$ está associada com um elemento v_o de $\mathcal{V}_i^o$ e com um elemento v_r de $\in \mathcal{V}_i^r$. Dessa forma, os respectivos v_r e v_o de v possuem a mesma configuração de hardware, mas são tarifados com preços diferentes. De modo geral, o preço por unidade de tempo de v_o é maior ou igual ao de v_r, sendo v_r e v_o associados a v.

Cada SLA $s_r \in \sigma_i^r$ está relacionado apenas com uma máquina virtual $v \in \mathcal{V}_i^r$, e cada SLA $s_o \in \sigma_i^o$ está associado somente com uma máquina virtual $v \in \mathcal{V}_i^o$. Além disso, consideramos outros dois parâmetros definidos nos contratos $s \in \mathcal{S}_i$: (i) o número $\alpha_s \in \mathbb{N}^+$ de máquinas virtuais solicitadas pelo provedor de SaaS e (ii) o tempo de duração $t_s \in \mathbb{N}^+$ desse acordo. Lembrando que o parâmetro t é normalmente estabelecido a longo prazo (1 a 3 anos, por exemplo) para os contratos $s_r \in \sigma_i^r$. Aliás, como o fornecimento de máquinas virtuais é limitado para cada cliente do provedor de IaaS [15], a restrição na equação (1) deve ser rigorosamente obedecida para qualquer instante de tempo t.

$$\sum_{s \in \mathcal{S}_i} \alpha_s \leq \delta_i, \quad \forall\, i \in \mathcal{I} \tag{1}$$

Portanto, os recursos alugados pelo provedor de SaaS estão presentes no conjunto $\mathcal{V}$, o qual é definido como:

$$\mathcal{V} = \left\{ \bigcup_{i=1}^{m} \mathcal{V}_i \quad \forall\, i \in \mathcal{I} : \mathcal{S}_i \neq \varnothing \right\} \tag{2}$$

onde $m = |\mathcal{I}|$ e $\mathcal{V}_i = \mathcal{V}_i^r \cup \mathcal{V}_i^o$. Em outras palavras, de acordo com os SLAs assinados com cada provedor de IaaS $i \in \mathcal{I}$, ou seja, os SLAs presentes em $\mathcal{S}_i$, o conjunto $\mathcal{V}$ é construído. A seguir, definimos algumas características importantes dos provedores de IaaS que também são intrínsecas à formulação do PLI:

- $m = |\mathcal{I}|$: número de provedores de IaaS, onde $m \in \mathbb{N}$;
- $\mathcal{P}_v$: número de núcleos de processamento da máquina virtual $v \in \mathcal{V}$, onde $\mathcal{P}_v \in \mathbb{N}^+$;
- $\mathcal{J}_v$: unidades de tempo que a máquina virtual $v \in \mathcal{V}$ leva para executar uma instrução, com $\mathcal{J}_v \in \mathbb{R}^+$;

- $\mathcal{L}_{v_g,v_h}$: unidades de tempo necessárias para transmitir uma unidade de dados sobre o enlace que conecta a máquina virtual $v_g \in \mathcal{V}$ e a máquina virtual $v_h \in \mathcal{V}$, com $\mathcal{L}_{v_g,v_h} \in \mathbb{R}^+$. Se $v_g = v_h$, então $\mathcal{L}_{v_g,v_g} = 0$;
- $\mathcal{B}_{i,v}$: variável binária que assume valor 1 se a máquina virtual $v \in \mathcal{V}$ pertence ao provedor de IaaS $i \in \mathcal{I}$; caso contrário, assume o valor 0;
- $\mathcal{C}_v$: preço para alugar a máquina virtual $v \in \mathcal{V}$ durante uma unidade de tempo, onde $\mathcal{C}_v \in \mathbb{R}^+$.

Uma configuração comum em ambientes reais é a diferença de preços entre a unidade de tempo cobrada das instâncias reservadas e sob demanda. Assim, conforme dito anteriormente, $\forall\, v_o \in \mathcal{V}_i^o$ e sua equivalente $v_r \in \mathcal{V}_i^r$, então $\mathcal{C}_{v_o} \geqslant \mathcal{C}_{v_r}$. Para finalizar, seja $\zeta = \{\sigma_1^r, \ldots, \sigma_m^r\}$ o conjunto composto por todos os SLAs de reservas de máquinas virtuais. Isto é, ζ contém informações sobre todas as máquinas virtuais reservadas pelo provedor de SaaS. Então, definimos a variável binária $\mathcal{K}_{s,v}$ que assume o valor 1 se o SLA $s \in \zeta$ está relacionado com a máquina virtual reservada $v \in \mathcal{V}$; caso contrário, assume o valor 0. Portanto, podemos deduzir que se $\mathcal{K}_{s,v} = 1$ e $\mathcal{B}_{i,v} = 1$, então $s \in \sigma_i^r$ e $v \in \mathcal{V}_i^r$.

O problema de escalonamento de *workflows* tratado neste trabalho pode ser enunciado como: *Encontre um mapeamento viável M entre os nós do DAG $\mathcal{G}$ e as máquinas virtuais de vários provedores de IaaS, de tal modo que (i) a soma do custo computacional monetário para todos os nós $u \in \mathcal{U}$ nas máquinas virtuais em $\mathcal{V}$ seja mínimo, (ii) as dependêcias entre os nós do DAG $\mathcal{G}$ não sejam violadas e (iii) o tempo total de execução do mapeamento M (makespan) $\mathcal{M}_\mathcal{G}$ seja no máximo igual ao prazo (deadline) exigido pelo usuário, ou seja, $\mathcal{M}_\mathcal{G} \leq \mathcal{D}_\mathcal{G}$.*

4.2. Formulação do Programa Linear Inteiro

A formulação do PLI leva em consideração a quantidade de núcleos de processamento da máquina virtual para tomar decisões no escalonamento, e também o número limitante (δ_i) de máquinas virtuais que podem ser instanciadas para cada usuário em um determinado tempo t. O programa linear inteiro soluciona o problema de escalonamento através das variáveis binárias x e y e da constante C_v, conforme descrito a seguir:

- $x_{u,t,v}$: variável binária que assume o valor 1 se o nó u termina sua execução no instante de tempo t na máquina virtual v, caso contrário, esta variável assume o valor 0;
- $y_{t,v}$: variável binária que assume o valor 1 se a máquina virtual v está sendo utilizada no instante de tempo t, caso contrário, esta variável assume o valor 0;
- $\mathcal{C}_v$: constante que assume o custo por unidade tempo da máquina virtual v.

Antes de formular um modelo matemático para qualquer problema de escalonamento, é estritamente necessário levar em consideração a representação do parâmetro *tempo*. O tempo pode ser representado por valores discretos ou contínuos [6]. A essência do problema de escalonamento de *workflows* requer, de modo geral, o uso de algumas variáveis discretas para representar decisões discretas. Por exemplo, atribuições de recursos e alocações de tarefas atômicas ao longo do tempo são algumas atividades que representam decisões discretas envolvidas no escalonamento de *workflows* [12]. Portanto, a formulação deste PLI considera apenas intervalos discretos de tempo. A Seção 5 apresenta maiores detalhes sobre a representação da linha do tempo desta formulação.

Seja o conjunto $\mathcal{T} = \{1, \ldots, \mathcal{D}_\mathcal{G}\}$ a linha do tempo da execução do *workflow*, que varia de 1 ao *deadline* $\mathcal{D}_\mathcal{G}$ estipulado pelo cliente do provedor de SaaS. Portanto, o PLI é formulado da seguinte maneira [7]:

$$\text{Minimize} \quad \sum_{t\in\mathcal{T}} \sum_{v\in\mathcal{V}} y_{t,v} \times \mathcal{C}_v, \quad \text{sujeito a :}$$

$$\text{(C1)} \quad \sum_{t\in\mathcal{T}} \sum_{v\in\mathcal{V}} x_{u,t,v} = 1; \quad \forall\, u \in \mathcal{U};$$

$$\text{(C2)} \quad \sum_{u\in\mathcal{U}} \sum_{v\in\mathcal{V}} \sum_{t=1}^{\lceil w_u \times \mathcal{J}_v \rceil} x_{u,t,v} = 0;$$

$$\text{(C3)} \quad \sum_{s=1}^{t-\lceil w_z \times \mathcal{J}_r + f_{u,z} \times \mathcal{L}_{i,j} \rceil} x_{u,s,v} \geq \sum_{s=1}^{t} x_{z,s,r} \quad \forall\, z \in \mathcal{U}, \quad \forall\, u \in \mathcal{H}(z), \quad \forall\, r, v \in \mathcal{V}, \quad \forall\, t \in \mathcal{T}, \quad \forall\, i, j \in \mathcal{I} \mid \mathcal{B}_{i,v} = 1, \quad \mathcal{B}_{j,r} = 1;$$

$$\text{(C4)} \quad \sum_{u\in\mathcal{U}} \sum_{s=t:t\leq \mathcal{D}_\mathcal{G} - \lceil w_u \times \mathcal{J}_v \rceil}^{t+\lceil w_u \times \mathcal{J}_v \rceil - 1} x_{u,s,v} \leq \mathcal{P}_v \quad \forall\, v \in \mathcal{V}, \quad \forall\, t \in \mathcal{T};$$

$$\text{(C5)} \quad \sum_{s=t-\lceil w_u \times \mathcal{J}_v \rceil + 1}^{t} y_{s,v} \geq x_{u,t,v} \times \Big(\lceil w_u \times \mathcal{J}_v \rceil \Big) \quad \forall\, u \in \mathcal{U}, \quad \forall\, v \in \mathcal{V}, \quad \forall\, t \in \{\lceil w_u \times \mathcal{J}_v \rceil, \ldots, \mathcal{D}_\mathcal{G}\};$$

$$\text{(C6)} \quad \sum_{v \in \mathcal{V}} y_{t,v} \leq \delta_i \quad \forall\, i \in \mathcal{I}, \quad \forall\, t \in \mathcal{T} \mid \mathcal{B}_{i,v} = 1;$$

$$\text{(C7)} \quad \sum_{v \in \mathcal{V}} y_{t,v} \leq \alpha_s \quad \forall\, s \in \zeta, \quad \forall\, t \in \mathcal{T} \mid \mathcal{K}_{s,v} = 1;$$

$$\text{(C8)} \quad x_{u,t,v} \in \{0,1\} \quad \forall\, u \in \mathcal{U}, \quad \forall\, t \in \mathcal{T}, \quad \forall\, v \in \mathcal{V};$$

$$\text{(C9)} \quad y_{t,v} \in \{0,1\} \quad \forall\, t \in \mathcal{T}, \quad \forall\, v \in \mathcal{V};$$

As restrições em (C1) especificam que todo serviço deve ser executado em algum momento e em uma única máquina virtual. A restrição em (C2) determina que um nó u do DAG não pode terminar sua execução até que todas suas instruções tenham sido executadas na máquina virtual v. As restrições em (C3) estabelecem que um nó z do DAG não pode iniciar a sua execução até que todos os nós que o precedem tenham terminado seus processamentos e os dados resultantes tenham chegado à máquina virtual que executará z. As restrições em (C4) estipulam que o número de nós do DAG em execução em uma máquina virtual v, em um determinado tempo t, não pode exceder o número de núcleos de processamento de v.

As restrições em (C5) determinam que uma máquina virtual deve permanecer ativa (ou seja, com status *sendo usado* habilitado na variável y), enquanto ela estiver executando os nós que a exigem. As restrições em (C6) especificam que o número de máquinas virtuais reservadas somado com o número de máquinas virtuais alugadas sob demanda não pode exceder o número máximo permitido para cada provedor de IaaS. As restrições em (C7) estabelecem que a quantidade de máquinas virtuais sendo utilizadas não pode exceder o limite estipulado no SLA. As duas últimas restrições, (C8) e (C9), especificam que as variáveis deste programa linear inteiro (x e y) só irão assumir os valores binários.

5. LINHA DO TEMPO

O problema de escalonamento é conhecido por ser um problema difícil de ser modelado e resolvido de modo eficiente [12]. A granularidade da linha do tempo é a questão-chave desse problema, quando utilizamos programação linear inteira. O tempo pode ser classificado em duas principais categorias [6]: discreto e contínuo. No primeiro modo, os escalonadores são formulados por meio da abordagem da discretização do tempo; isto é, a linha do tempo é divida em intervalos de tempo iguais. Assim, as decisões do escalonador, tal como início (ou término) de um evento, devem ser feitas somente no começo (ou no final) desses intervalos. No modo contínuo, por outro lado, as decisões do escalonador são realizadas em qualquer instante de tempo pertencente ao intervalo real $[1, deadline]$. Embora a linha temporal contínua aumente a precisão do escalonador, ou seja, apresenta uma granularidade mais fina, o modelo matemático torna-se mais complexo [18], assim como torna-se mais complexa a obtenção de soluções ótimas inteiras para o PLI.

Como dito anteriormente, a natureza do problema de escalonamento requer, de modo geral, o uso de algumas variáveis discretas para representar decisões discretas, como, por exemplo, atribuições de recursos e alocações de tarefas (ou serviços) ao longo do tempo [12]. Além disso, os provedores de IaaS normalmente contabilizam o uso das máquinas virtuais de acordo com as unidades de tempo inteiras utilizadas através do modelo *pay-as-you-go*. Aliás, as unidades de tempo parcialmente consumidas são geralmente cobradas como se fossem unidades de tempo completas[3]. Portanto, devido à cobrança das máquinas virtuais serem feitas por unidades de tempo inteiras, o escalonador apresentado neste trabalho utiliza apenas tempo discreto para representar a execução de DAGs. No entanto, dependendo do nível de granularidade da linha do tempo e do tamanho do DAG, o tempo de execução do escalonador pode ser alto para o contexto da computação em nuvem; resultando, então, em nenhuma solução viável em tempo hábil. Neste trabalho avaliamos o uso de diferentes níveis de fragmentação do tempo no escalonamento de diferentes tipos de DAGs.

O parâmetro (ou dimensão) *tempo* é o principal gargalo que impede a escalabilidade do escalonador apresentado em [7]. Devido ao elo entre a precedência dos nós do DAG e a NP-Completude do problema de escalonamento, tanto o tempo de execução do *solver* quanto o conjunto $\mathcal{T}$ crescem rapidamente quando tentamos escalonar DAGs com grande número de nós e dependências. Em outras palavras, a principal variável do programa linear inteiro é uma matriz binária de três dimensões ($x_{u,t,v}$: nó u, tempo t e máquina virtual v) e, dessa forma, quanto maior o tamanho da matriz binária, maior o tempo de execução do *solver*. Assim, uma alternativa utilizada para tentar reduzir o tamanho dessa matriz binária foi por meio da técnica do *aumento da granularidade da discretização do tempo*. Isto é, com o tempo discretizado em unidades maiores, podemos reduzir o número de elementos do conjunto $\mathcal{T}$ e, consequentemente, diminuir a dimensão do tempo na matriz binária. Portanto, com um conjunto $\mathcal{T}$ de tamanho reduzido, o tempo de execução do *solver* tende a diminuir.

O uso de intervalos de tempo curtos (granularidade fina) e/ou linha temporal longa (*deadline* alto) pode aumentar consideravelmente a quantidade de intervalos de tempo discretizado do conjunto $\mathcal{T}$ usado no PLI. Aliás, o aumento do *deadline* e/ou do grafo são exemplos que implicam diretamente no aumento do tamanho do conjunto

[3] A *Amazon EC2* realiza esta prática em `http://aws.amazon.com/ec2/pricing/`

Tabela 1. Provedor de IaaS A

Tipo	Núcleo	Instrução Por Núcleo	Preço Demanda	Preço Reserva
P	1	1.5	\$0.13	\$0.045
M	2	1.5	\$0.20	\$0.070

Tabela 2. Provedor de IaaS B

Tipo	Núcleo	Instrução Por Núcleo	Preço Demanda	Preço Reserva
P	1	2	\$0.17	\$0.045
M	2	2	\$0.30	\$0.059
G	3	2	\$0.40	\$0.140
EG	4	2	\$0.52	\$0.183
EG2	8	2	\$0.90	\$0.316

Tabela 3. Provedor de IaaS C

Tipo	Núcleo	Instrução Por Núcleo	Preço Demanda	Preço Reserva
P	1	2	\$0.15	\$0.052
M	2	2	\$0.25	\$0.088
G	4	2.5	\$0.50	\$0.176
EG	8	2.5	\$0.80	\$0.281

Tabela 4. Máquinas virtuais reservadas para provedor de SaaS

Tipo	IaaS	VM	Quantidade
Reservada	A	P	1
Reservada	A	M	1
Reservada	B	P	1
Reservada	B	M	1

$\mathcal{T}$. Em suma, quanto maior o tamanho de $\mathcal{T}$, maior o espaço de busca de soluções e, consequentemente, maior o tempo gasto pelo escalonador para encontrar alguma solução viável. Isto é, o escalonamento pode se tornar em um problema computacionalmente caro de ser solucionado de forma rápida, tornando-se, portanto, em um problema sem solução em tempo hábil [6]. Assim, definimos λ como um *fator multiplicativo* que determina a granularidade do tempo discreto de $\mathcal{T}$, o qual é reescrito na equação (3).

$$\mathcal{T} = \left\{ \lambda, 2\lambda, 3\lambda, 4\lambda, \ldots, \Lambda \right\} \quad \middle| \quad \Lambda \leq \mathcal{D}_{\mathcal{G}} \quad \text{e} \quad \lambda \in \mathbb{N}^{+} \qquad (3)$$

No entanto, o uso de intervalos de tempo longos (granularidade grossa) pode diminuir consideravelmente a quantidade de intervalos de tempo de $\mathcal{T}$ e, com isso, o escalonamento pode se tornar inviável, pois o escalonador não encontrará unidades de tempo suficientes para representar todas as dependências dos nós do DAG. Além disso, a granularidade grossa pode aumentar o custo monetário das soluções viáveis, pois com um conjunto $\mathcal{T}$ de tamanho reduzido, o escalonador é obrigado escolher máquinas virtuais mais rápidas e consequentemente mais caras. Além disso, alguns núcleos de processamento dessas máquinas podem não ser utilizados e, portanto, pode haver desperdício de recursos (ou dinheiro). Portanto, há um claro *trade-off* entre utilizar intervalos de tempo curtos para obter um escalonamento mais preciso ou usar intervalos de tempo longos para diminuir o tempo de execução do escalonamento. Dessa forma, em nossas simulações, variamos o fator multiplicativo λ para tentar diminuir o tempo de execução do escalonamento de DAGs com uma grande quantidade de nós e dependências. É importante ressaltar que utilizamos essa abordagem sem aplicar o relaxamento linear nas variáveis do PLI, ou seja, as variáveis x e y permaneceram binárias.

6. AVALIAÇÃO

O programa linear inteiro foi implementado em *Java*[4] e as simulações foram realizadas por meio do *solver*[5] *IBM ILOG CPLEX Optimizer*[6] com as configurações internas padrão. Nas simulações foram utilizados apenas grafos que representam aplicações do mundo real, tais como *fork-join* (Figura 1), *Montage* [5] e *LIGO* [11] (Figura 3); além disso, utilizamos três provedores de IaaS distintos. As métricas utilizadas neste trabalho para avaliar os resultados foram: o *custo monetário do escalonamento* (em unidades monetárias), o *makespan do workflow* (em unidades de tempo), o *tempo de execução do escalonador* (em segundos) e a *porcentagem de soluções inviáveis* (porcentagem em que nenhuma solução foi encontrada). Nesta seção vamos apresentar o cenário das simulações, o ambiente de simulação e os resultados obtidos.

6.1. CONFIGURAÇÃO DAS SIMULAÇÕES

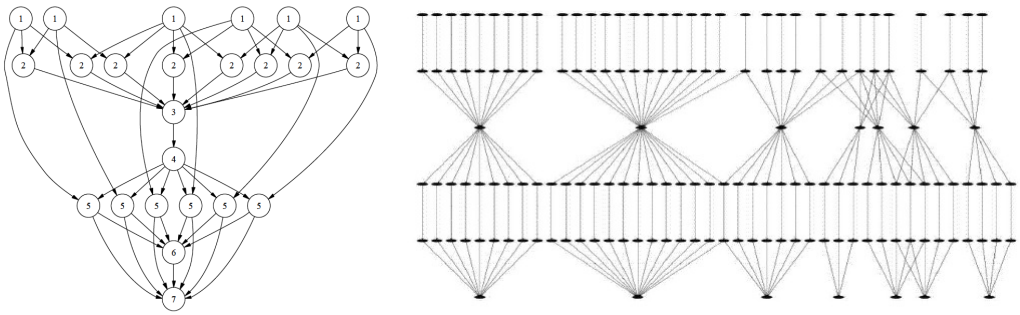
Utilizamos três provedores de IaaS em nossas simulações, sendo que cada provedor define seus próprios preços das máquinas virtuais para os planos de reservas e sob-demanda. As Tabelas 1, 2 e 3 mostram, respectivamente, as opções de máquinas virtuais dos provedores de IaaS A, B e C. Além disso, a Tabela 4 mostra todos os SLAs de máquinas virtuais reservadas estabelecidos pelo provedor de SaaS, isto é, descreve os elementos do conjunto ζ. O número máximo de máquinas virtuais que podem ser alugadas a partir de cada provedor de IaaS foi estipulado da seguinte maneira: $\delta_A = 4$, $\delta_B = 7$, $\delta_C = 2$. O conjunto ζ, os valores de δ e demais detalhes da modelagem matemática do PLI estão descritos na Seção 4.

As nuvens públicas geralmente não fornecem informações sobre a qualidade de serviço dos enlaces internos

[4] `http://www.java.com/`

[5] O termo genérico *solver* é usado para indicar um software ou parte de um software que resolve um problema matemático.

[6] `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`



(a) Montage [5] (b) LIGO [11]

Figura 3. Dois *Worfklows* científicos utilizados nos experimentos.

(entre as máquinas virtuais) e enlaces externos (entre os provedores de IaaS). Portanto, assumimos que a largura de banda dos enlaces internos (dentro do mesmo provedor de IaaS) é maior do que a largura de banda dos enlaces externos, pois é uma suposição razoável em ambientes reais. Isso se reflete em nossa simulação gerando aleatoriamente um valor para $\mathcal{L}$ no intervalo $[2, 3]$ para os enlaces entre nuvens (provedores) diferentes, enquanto que para os enlaces entre máquinas virtuais dentro da mesma nuvem, o valor de $\mathcal{L}$ pertence ao intervalo $[0.1, 0.2]$. Portanto, através das máquinas virtuais já reservadas e das que ainda poderão ser alugadas sob-demanda, nosso objetivo é que o custo monetário do escalonamento seja mínimo. A propósito, as palavras *solver* e *escalonador* vão ser utilizadas de formas intercambiáveis nesta seção.

6.2. *IBM CPLEX Solver*

Para solucionar um PLI, o *CPLEX* utiliza, por padrão, um algoritmo proprietário denominado *dynamic search*. Embora seja um algoritmo de código fechado, o *dynamic search* é baseado no método de enumereção *Branch & Cut* [8], um dos tradicionais algoritmos para solucionar problemas de PLI. Aliás, esse método é uma combinação de outros dois, são eles: *Branch & Bound* [9] e *Planos de Corte* [4]. Basicamente, o algoritmo de *Branch & Bound* consiste em enumerar, sistematicamente, o espaço de soluções em uma árvore de enumeração, descartando os ramos (não explorados) que levam em soluções inviáveis ou soluções piores que as já encontradas. O termo *Branch* (em português, ramificar) é o processo de dividir um problema em dois ou mais subproblemas menores e mutuamente exclusivos; enquanto o termo *Bound* (em português, poda) é o procedimento que calcula limitantes (superior e inferior) para o valor da solução ótima obtida em cada subproblema produzido no processo de remificação. Assim, baseado no limitantes calculados, o ramo é *podado* da árvore de enumeração. Enfim, o *Branch & Bound* processa quebrando o espaço de soluções viáveis em subproblemas cada vez menores até que uma solução ótima seja alcançada.

O algoritmo de *Planos de Corte*, por sua vez, consiste em adicionar novas restrições na formulação do problema, a fim de reduzir o espaço de busca de soluções[7,8]. Em outras palavras, a ideia é utilizar hiperplano (chamados de planos de cortes) que "cortem" soluções não desejáveis do poliedro problema inicial. É importante frisar que, decidir se uma reta representa um plano de corte é um problema NP-difícil e, na prática, utiliza-se heurísticas para tomar essa decisão. Portanto, ao adicionar o método *Planos de Corte* no algoritmo de *Branch & Bound* temos o *Branch & Cut*.

6.3. RESULTADOS

As simulações foram executadas em um processador Intel® Xeon $X5650$ CPU $2.67GHz$ com $16GB$ de memória RAM. Realizamos 30 simulações para cada DAG com as configurações dos provedores de IaaS e dos SLAs presentes nas Tabelas 1 a 4. Em cada simulação, o custo de computação de cada nó do DAG e os custos de comunicação de cada dependência foram obtidos aleatoriamente do intervalo $[1, 3]$. Variamos o fator multiplicativo λ para avaliar a relação entre as métricas *custo monetário* e *tempo de execução do solver*. Entretanto, também vamos disponibilizar os gráficos das métricas *makespan* e *números de soluções inviáveis* para mostrar o efeito da variação de λ nessas métricas. O PLI foi executado com tempo limite de 10 minutos para cada simulação.

Para representar os possíveis valores dos *deadlines* solicitados pelos clientes do provedor de SaaS, executamos as simulações com $\mathcal{D}_G$ variando de $\mathcal{T}_{max} \times 2/7$ à $\mathcal{T}_{max} \times 6/7$ em etapas de $1/7$, onde $\mathcal{T}_{max}$ é o *makespan* da execução sequencial de todos os nós do DAG em um único recurso cuja tarifa é a mais barata e, provavel-

[7] O espaço de soluções é representado por um polígono convexo, ou para dimensões maiores, poliedros convexos.

[8] Quando o espaço de busca de solução é reduzido, o poliedro é geralmente denominado *poliedro apertado*, uma vez que são eliminadas folgas de soluções fracionárias.

mente, a máquina virtual mais lenta. *Deadlines* iguais à $\mathcal{T}_{max} \times 1/7$ apresentaram somente soluções inviáveis, enquanto *deadlines* iguais à $\mathcal{T}_{max} \times 7/7$ podem ser trivialmente alcançados colocando todas as tarefas no recurso mais barato. É importante frisar que o divisor 7 foi escolhido apenas para avaliar a evolução da discretização com o aumento do *deadline* e, portanto, outros divisores poderiam ser utilizados. As médias apresentadas nos gráficos são sobre 30 simulações de cada DAG para cada $\mathcal{D}_\mathcal{G}$, com um intervalo de confiança de 95%. Em alguns pontos, o intervalo de confiança é pequeno e, portanto, a sua visualização torna-se imperceptível.

Como dito anteriormente, o aumento do valor de λ implica na redução do número de elementos do conjunto $\mathcal{T}$. Dessa forma, a fim de cumprir o *deadline*, o *solver* tende a alugar máquinas virtuais caras e com alta capacidade de processamento (geralmente com vários núcleos de processamento virtuais) para escalonar todos os nós do DAG. Em outras palavras, a redução do tamanho do conjunto $\mathcal{T}$ pode gerar uma "penalidade" no escalonamento, pois quando aumentamos a discretização da linha do tempo, podemos desperdiçar núcleos virtualizados e, com isso, o escalonamento tende a ficar mais caro. Por exemplo, uma unidade de tempo em $\lambda = 2$ equivale a duas unidades de tempo em $\lambda = 1$; ou seja, se o *solver* escalona um nó do DAG para ser executado em uma unidade de tempo em $\lambda = 1$, então esse nó também será executado em uma unidade de tempo em $\lambda = 2$. Em suma, haverá um custo monetário adicional de uma unidade de tempo (não utilizada) na máquina virtual em que esse nó for escalonado quando $\lambda = 2$; aumentando, então, o custo monetário total do escalonamento. Enfim, para cada valor de λ, o *solver* dará uma estimativa[9] do valor monetário do escalonamento.

A Figura 4 mostra a evolução do tempo de execução do *solver* em diversos cenários. Quando aumentamos o tamanho do DAG (números nós e dependência entre nós), também aumentamos, indiretamente, (i) o número de elementos do conjunto $\mathcal{T}$; (ii) o número de possibilidades de soluções para o escalonamento; e (iii) o tempo de execução do *solver*. Por exemplo, para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$, o *solver* chegou a demorar, em média, 1.15, 141.43, 1652.92 e 3456.34 segundos para encontrar alguma solução viável para o DAG *fork-join* com 10, 20, 30 nós e o DAG *Montage*, respectivamente. Portanto, devido à NP-Completude do problema de escalonamento, encontrar uma solução ótima para o DAG *fork-join* de 30 nós

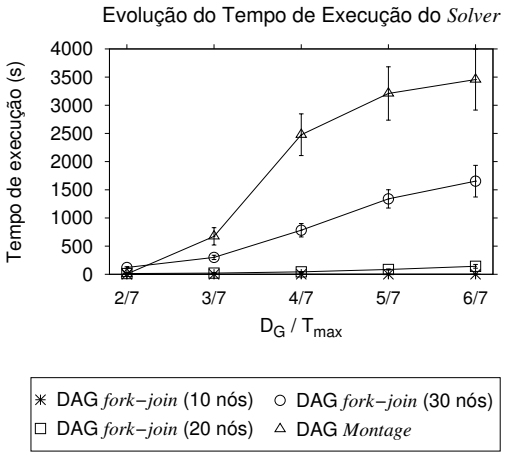


Figura 4. Comparação do tempo de execução do *solver* com aumento do tamanho do DAG (mais detalhes em [7]).

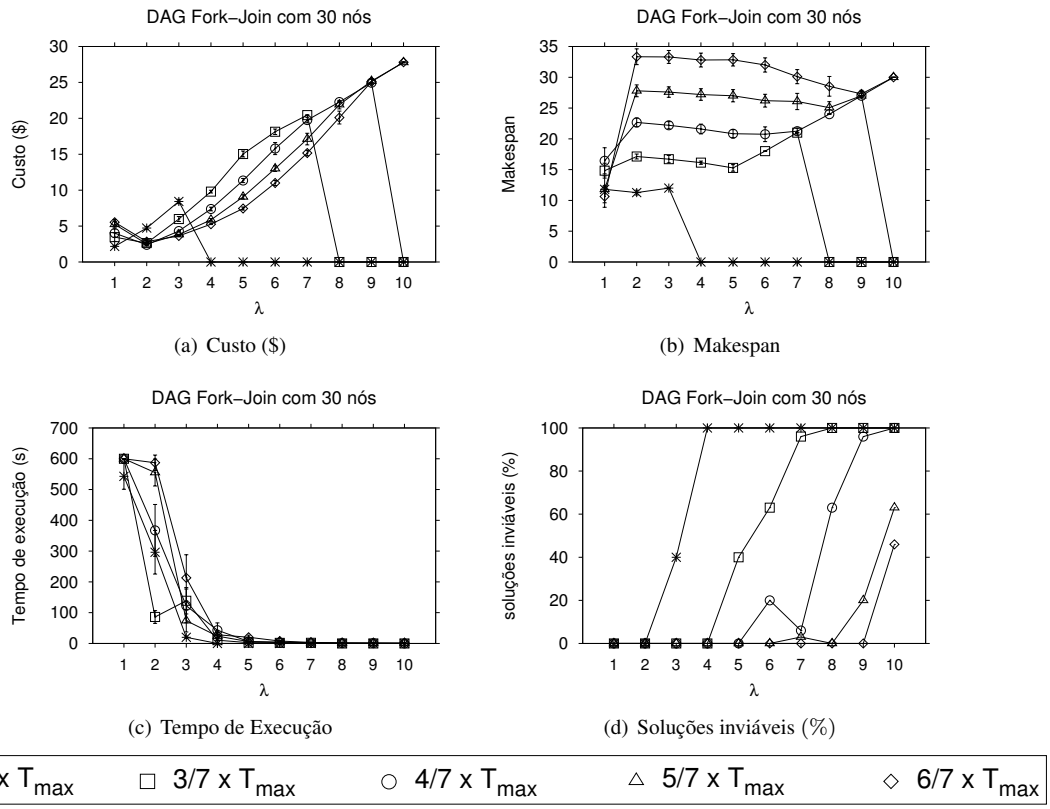
e o DAG *Montage* pode levar horas ou até mesmo dias. Dessa forma, o aumento do nível da granularidade da discretização da linha do tempo visa reduzir $\mathcal{T}$ e, consequentemente, o tempo de execução do *solver*. A seguir, analisamos resultados onde aplicamos diferentes fatores de discretização da linha do tempo.

6.3.1. DAG *fork-join* com 30 nós: A Figura 5 mostra resultados das simulações para o DAG *fork-join* com 30 nós. Em $\lambda = 1$ e $\lambda = 2$, o *solver* encontrou soluções para todos os $\mathcal{D}_\mathcal{G}$s, ou seja, 0% de soluções foram inviáveis. Já para $\lambda = 2$ e $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 2/7$, o escalonamento foi 1.83 vezes mais rápido, mas com um custo monetário médio duas vezes maior. Mantendo $\lambda = 2$ e aumentando o *deadline*, o escalonamento tornou-se mais rápido e com custos mais baixos. Por exemplo, para $\lambda = 2$ e $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 3/7$, o *solver* conseguiu encontrar, em média, soluções 24% mais baratas e 7 vezes mais rápidas em relação as soluções para o mesmo $\mathcal{D}_\mathcal{G}$ em $\lambda = 1$.

Em $\lambda = 1$, a solução com menor *deadline* ($\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 2/7$) teve o menor custo monetário médio, devido ao escalonamento terminar antes do limite de tempo de 10 minutos. Note que o custo monetário médio foi menor nesse caso porque o *solver* conseguiu alcançar a solução ótima para algumas entradas. Além disso, para as demais entradas, o *solver* conseguiu alcançar níveis profundos da árvore de enumeração e, com isso, encontrou soluções viáveis (não-ótimas) com custos monetários reduzidos. No entanto, para os demais *deadlines* em $\lambda = 1$, a maioria das soluções encontradas pelo *solver* foram obtidas na raiz da árvore de enumeração. Aliás, para $\mathcal{D}_\mathcal{G} \geq \mathcal{T}_{max} \times 5/7$, algumas das soluções foram encontradas somente pelas heurísticas internas[10] do *CPLEX*, antes

[9]Esta estimativa contradiz o modelo de tarifação *pay-as-you-go* da computação em nuvem, pois contabiliza o *não* uso da máquina virtual. Esse problema ocorre porque o *solver* "pensa" que uma unidade de tempo discretizada, por exemplo em $\lambda = 3$, será totalmente utilizada pelo nó que lhe for atribuído. Assim, dada uma solução viável em $\lambda = 3$, há uma necessidade de transformar essa solução em $\lambda = 1$ e verificar quais unidades de tempo que não foram utilizadas e que estão sendo contabilizadas no custo monetário do escalonamento, para que possamos descontar esse custo adicional. No entanto, essa tarefa está como trabalhos futuros.

[10]Heurísticas de código fechado.



(a) Custo ($)

(b) Makespan

(c) Tempo de Execução

(d) Soluções inviáveis (%)

Figura 5. Resultados do DAG *fork-join* com 30 nós

mesmo de iniciar o método de enumeração *Branch & Cut*. Isso explica o porquê das soluções dos demais *deadlines* serem mais caras que as soluções de $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 2/7$. Resumindo, o tamanho do conjunto de tempo $\mathcal{T}$ (e o espaço de soluções) de $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 2/7$ era menor que dos demais *deadlines*, e por isso o *solver* conseguiu avançar níveis profundos na árvore de enumeração.

Por outro lado, em $\lambda = 2$, o *solver* conseguiu achar soluções antes do limite de tempo para a maioria dos *deadlines* e, conforme esperado, o custo monetário médio das soluções de $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 2/7$ aumentou (devido aos desperdícios de recursos não utilizados). Entretanto, para os demais *deadlines*, o custo monetário médio reduziu devido ao aumento da discretização da linha do tempo. Isso ocorreu pelo seguinte motivo: em $\lambda = 1$, ora a solução era encontrada apenas na raiz da árvore de enumeração, ora a solução era dada pelas heurísticas internas do *CPLEX* antes de iniciar *Branch & Cut*. Já em $\lambda = 2$, o *solver* conseguiu alcançar níveis mais profundos da árvore de enumeração e, dessa maneira, encontrou soluções melhores que em $\lambda = 1$. Em outras palavras, quando aumentamos a granularidade da linha do tempo para $\lambda = 2$, podemos "dar continuidade" na busca da árvore de enumeração de $\lambda = 1$ e, com isso, alcançar soluções melhores. Note que colocamos a expressão *dar continuidade* entre aspas, pois a árvore de $\lambda = 1$ pode não ser a mesma de $\lambda = 2$.

Uma conjectura tirada dos resultados das simulações onde variamos o fator λ é a seguinte: seja um DAG $\mathcal{G}$ e seu respectivo $\mathcal{D}_\mathcal{G}$, se a solução ótima de $\mathcal{G}$ com $\mathcal{D}_\mathcal{G}$ é possível de ser encontrada em $\lambda_\mathcal{A}$ e em $\lambda_\mathcal{B}$, tal que $\lambda_\mathcal{A} < \lambda_\mathcal{B}$, $\lambda_\mathcal{A}, \lambda_\mathcal{B} \in \mathbb{N}^+$, então o custo monetário da solução ótima em $\lambda_\mathcal{A}$ tende a ser menor ou igual ao custo monetário da solução encontrada em $\lambda_\mathcal{B}$. Além disso, o tempo de execução do *solver* em $\lambda_\mathcal{A}$ tende a ser maior ou igual ao tempo de execução do *solver* em $\lambda_\mathcal{B}$.

Quando aumentamos a discretização do tempo para $\lambda = 3$, o *solver* encontrou soluções com custos monetários maiores para todos os *deadlines*, como mostra a Figura 5(a). Lembrando que para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 2/7$ esse aumento começou a ocorrer em $\lambda = 2$. Embora haja um aumento significativo nos custos monetários das soluções encontradas para $\lambda \geq 3$, o tempo de execução do escalonamento diminuiu consideravelmente. Por exemplo, na comparação entre as simulações com $\lambda = 1$ e $\lambda = 4$, para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 4/7$, temos uma redução de aproximadamente 93% no tempo de execução do *solver* e, em contrapartida, um aumento de aproximadamente 83% no custo monetário.

O aumento de λ, acompanhado da redução de $\mathcal{D}_\mathcal{G}$, diminui a linha temporal do escalonamento e, portanto, au-

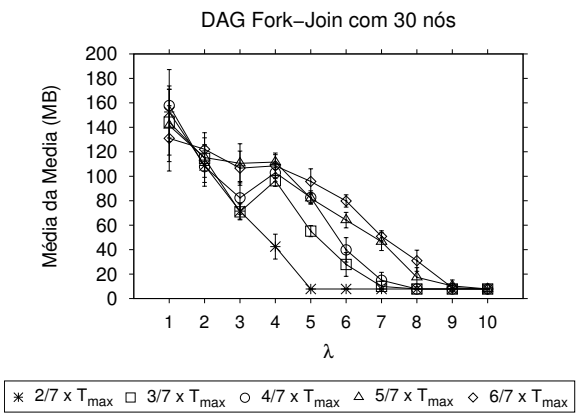


Figura 6. Consumo de memória RAM das simulações do DAG *fork-join* com 30 nós

menta o número de soluções inviáveis. Isso ocorre pelo motivo de não existirem intervalos de tempo suficientes para todos os nós do DAG. Por exemplo, o aumento de soluções inviáveis para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 2/7$ ocorre apenas quando $\lambda = 3$, porém para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$ esse aumento acontece somente quando $\lambda = 9$. Note que, quando a porcentagem de soluções inviáveis é 100%, os valores do custo monetário e do *makespan* são zero, pois não houve uso de máquinas virtuais; essa observação também é válida para os resultados dos demais DAGs.

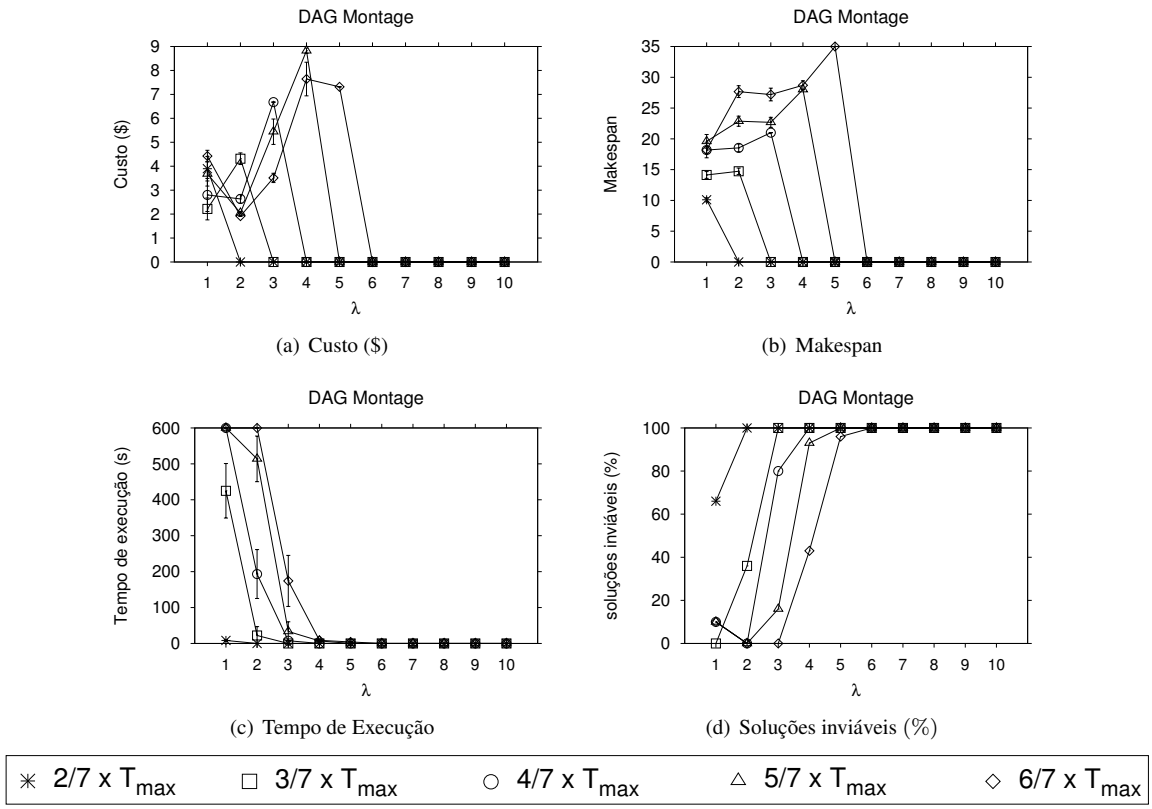
A Figura 6 mostra a média da média do consumo de memória RAM das simulações do DAG *fork-join* com 30 nós. Como podemos analisar, à medida que aumentamos a discretização da linha do tempo, o *solver* tende a usar menos memória RAM, pois o escalonamento tende a ficar menos complexo com a redução do conjunto $\mathcal{T}$. Por exemplo, para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 5/7$ e $\lambda = 5$, o consumo de memória foi, em média, 42% menor que as simulações para $\lambda = 1$; lembrando que a porcentagem de soluções inviáveis para esse *deadline* foi 0% em $\lambda = 1$ e $\lambda = 5$. Quando o *solver* descobre rapidamente que o escalonamento é inviável, o consumo de memória é baixo como em $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 2/7$ em $\lambda = 5$.

6.3.2. DAG *Montage*: Os resultados das simulações para o DAG *Montage* estão ilustrados na Figura 7. Em $\lambda = 1$, o *solver* conseguiu encontrar soluções viáveis para todas as 30 simulações somente para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 3/7$; ou seja, esse *deadline* teve 0% de soluções inviáveis em $\lambda = 1$, como mostra a Figura 7(d). No entanto, entre essas soluções viáveis, 80% foram ótimas (com custos mínimos), pois foram encontradas antes do limite de tempo de 10 minutos; e os 20% restantes foram encontrados em níveis menos profundos da árvore de enumeração, pois a execução do *solver* foi abortada pelo limite de tempo e, com isso, a melhor solução teve que ser escolhida até esse momento. Portanto, por ter uma grande quantidade de soluções ótimas, o custo monetário médio das simulações para esse *deadline* foi o menor de todos, como ilustra a Figura 7(a) em $\lambda = 1$.

A porcentagem de soluções inviáveis para as simulações com $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 2/7$ foi de 66% em $\lambda = 1$. Esse valor foi alto porque o tamanho do conjunto $\mathcal{T}$ era pequeno o suficiente para que todos os nós do *Montage* fossem escalonados com sucesso. Como conseqüência do tamanho reduzido de $\mathcal{T}$, o tamanho do conjunto de solução também se torna menor e, portanto, o *solver* consegue descobrir de forma rápida se há ou não alguma solução viável para o escalonamento. Isso explica o motivo dos valores baixos do tempo de execução do *solver* para esse *deadline*, como mostra a Figura 7(c). Aliás, para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 2/7$ em $\lambda = 2$, o tamanho do conjunto $\mathcal{T}$ tornou-se ainda menor e, portanto, o *solver* não teve intervalos de tempo suficientes para escalonar todos os nós do DAG *Montage*. Dessa forma, explicamos o porquê do *solver* não ter encontrado soluções viáveis para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 2/7$ quando $\lambda \geq 2$.

Por outro lado, ainda em $\lambda = 2$, foi possível encontrar rapidamente soluções viáveis com custos menores para alguns *deadlines*. Por exemplo, para as simulações com $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 4/7$, enquanto o custo médio do escalonamento foi reduzido em 6%, o tempo médio de execução do *solver* foi reduzido em 68%, quando comparado às soluções para o mesmo $\mathcal{D}_\mathcal{G}$ em $\lambda = 1$. Mantendo λ ainda igual a 2 e aumentando o *deadline*, a redução do custo monetário médio ficou ainda maior; 44% para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 5/7$ e 56.5% para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$. Por outro lado, o tempo de execução reduziu somente para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 5/7$ e no valor de 14.5%, enquanto para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$, essa redução ocorreu apenas em $\lambda = 3$ que foi de 71.2%. Embora o custo monetário médio de $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$ tenha aumentado 83% de $\lambda = 2$ para $\lambda = 3$, ele foi 20.5% menor quando comparado ao custo médio em $\lambda = 1$.

Devido aos desperdícios de recursos impostos pela técnica da discretização do tempo, já era esperado que o aumento da granularidade da linha temporal nas simulações, onde o *solver* encontrou somente soluções ótimas, resultasse em soluções mais caras. Por exemplo, para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 3/7$ de $\lambda = 1$ para $\lambda = 2$, houve um aumento de 194% no custo monetário; em contrapartida, houve uma redução de 95% no tempo de execução do *solver*. Por outro lado, para os demais *deadlines* ($\mathcal{D}_\mathcal{G} \geq \mathcal{T}_{max} \times 4/7$) o aumento da discretização do tempo ajudou o *solver* a encontrar soluções viáveis que reduziram o custo monetário médio. Por exemplo, para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 4/7$, o *solver* foi capaz de encontrar soluções viáveis com um custo monetário médio 6% menor pelo seguinte motivo: entre 90% das soluções viáveis em $\lambda = 1$, 37% foi obtida na raiz da árvore de enumera-



(a) Custo ($)

(b) Makespan

(c) Tempo de Execução

(d) Soluções inviáveis (%)

Figura 7. Resultados do DAG *Montage*

ção, enquanto 62% das soluções viáveis restantes foram encontradas em níveis mais profundos na árvore, porém nenhuma ótima. Por outro lado, em $\lambda = 2$, 100% das soluções viáveis foram ótimas, reduzindo, portanto, o custo monetário médio.

No entanto, como comentado anteriormente, a redução dos custos monetários médios de $\lambda = 1$ para $\lambda = 2$ foram maiores para as simulações com $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 5/7$ e $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$, cujos valores foram 44% e 56.5%, respectivamente. A redução foi maior nesses casos porque a maioria das soluções viáveis encontradas em $\lambda = 1$ foram obtidas através das heurísticas internas do *CPLEX*, cujos resultados carecem de garantias teóricas. Em resumo, ao solucionar um programa linear inteiro, as heurísticas do *CPLEX* podem encontrar uma ou mais soluções viáveis, as quais satisfazem todas as restrições e todas as condições de integralidade; porém, sem garantia de otimalidade das soluções. Enfim, o aumento da discretização do tempo para $\lambda = 2$ reduziu o tamanho do conjunto $\mathcal{T}$ e, consequentemente, o tamanho do conjunto de soluções. Dessa forma, as heurísticas gastaram pouco tempo para encontrar alguma solução viável e, com isso, o método *Branch & Cut* foi iniciado rapidamente pelo *solver*, o qual conseguiu achar soluções melhores em $\lambda = 2$.

O *makespan* teve um comportamento semelhante aos DAGs anteriores, tendo um aumento acentuado apenas para *deadlines* longos. Por exemplo, de $\lambda = 1$ para $\lambda = 2$, o *makespan* aumentou 3.5% para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 3/7$ e 57% para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$. Como o DAG *Montage* possui vários nós e dependências, não foi possível encontrar soluções viáveis para $\lambda \geq 5$, pois o conjunto $\mathcal{T}$ tornava-se insuficiente para todos os nós do DAG.

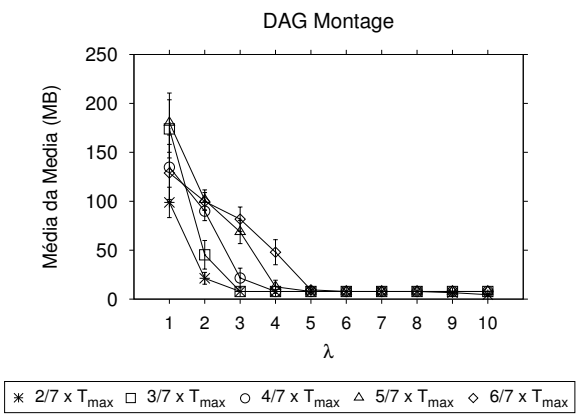


Figura 8. Consumo de memória RAM das simulações do DAG *Montage*

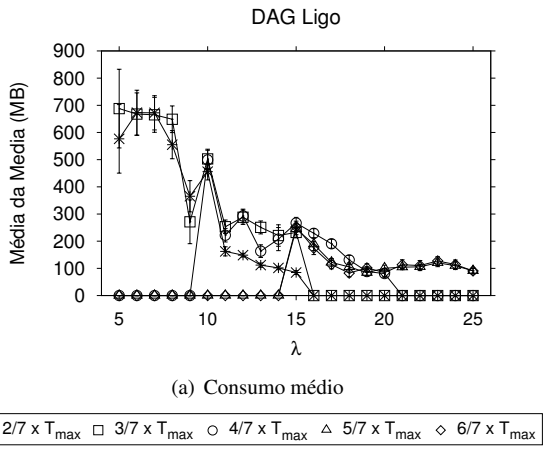
O consumo médio de memória RAM usado pelo *solver* para simular o DAG *Montage* está ilustrado na Figura 8. Esse consumo de memória também teve um comportamento semelhante às simulações anteriores, ou seja, houve uma redução do uso de memória RAM com aumento de λ. Por exemplo, para as simulações com $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$ de $\lambda = 1$ para $\lambda = 2$, o uso de memória do *solver* foi reduzido, em média, 22.5%, enquanto de $\lambda = 2$ para $\lambda = 3$ essa redução foi de 18% em média.

6.3.3. DAG *LIGO*: Quando usamos DAGs com maior número de nós, o escalonamento começou a encontrar soluções viáveis em tempo hábil com o aumento da discretização do tempo, como mostram os resultados das simulações do DAG *LIGO* na Figura 9, cuja topologia está ilustrada na Figura 3(b). Entretanto, devido à grande quantidade de nós, tivemos que variar o valor de λ de acordo com o valor de $\mathcal{D}_\mathcal{G}$, pois com o conjunto $\mathcal{T} = \{1, \ldots, 10\}$, o *solver* teve dificuldades de encontrar soluções viáveis em 10 minutos de simulação para todos os *deadlines*. Assim, simulamos o DAG *LIGO* da seguinte maneira:

- Para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 2/7$ e $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 3/7$, usamos o conjunto $\mathcal{T} = \{5, \ldots, 15\}$;
- Para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 4/7$, usamos o conjunto $\mathcal{T} = \{10, \ldots, 20\}$;
- Para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 5/7$ e $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$, usamos o conjunto $\mathcal{T} = \{15, \ldots, 25\}$

É importante ressaltar que, para cada *deadline*, o menor e o maior elemento do conjunto $\mathcal{T}$ foram escolhidos de acordo com as primeiras simulações do *LIGO* e, como podemos observar na Figura 9(d), nenhum *deadline* teve soluções viáveis para todos os valores em $\mathcal{T}$. Além disso, ao utilizar uma granularidade fina ($\lambda \leq 5$), o *solver* não conseguiu achar soluções viáveis em tempo hábil para nenhum *deadline*. Em suma, as soluções viáveis para esse tipo de DAG foram possíveis apenas com uma redução significativa do conjunto do tempo $\mathcal{T}$ (ou um aumento significativo de λ). Por exemplo, para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 2/7$, o escalonamento foi possível apenas para $\lambda \in \{6, 9\}$, sendo que o *solver* ainda foi abortado pelo limite de tempo de 10 minutos para todos esses valores de λ. Isto é, mesmo aumentando a granularidade da linha do tempo (ou reduzindo o conjunto $\mathcal{T}$), o *solver* utilizou os 10 minutos e retornou a melhor solução viável encontrada. A propósito, para esse *deadline*, o menor custo monetário médio ocorreu em $\lambda = 6$, o qual foi 12%, 8.5% e 17% menor que os valores em λ iguais a 7, 8 e 9, respectivamente.

Contendo 168 nós, o *LIGO* é o maior DAG simulado neste tabalho; contra os 30 nós do *fork-join* e os 24 do

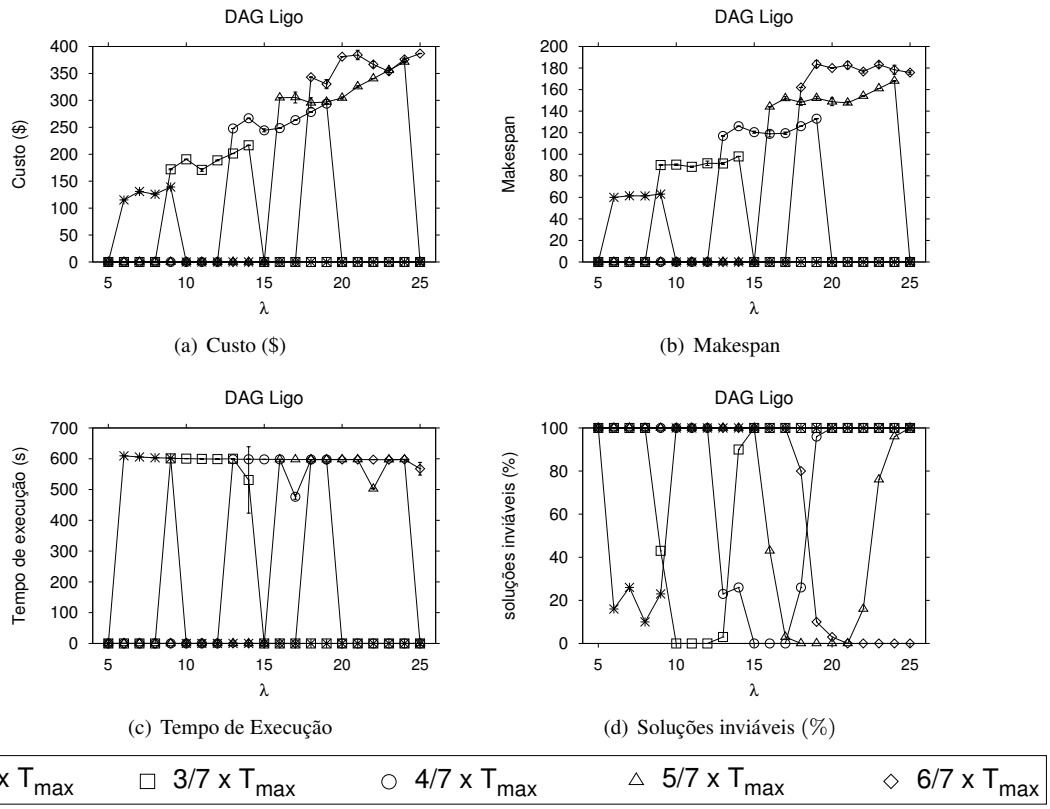


(a) Consumo médio

Figura 10. Consumo de memória RAM das simulações do DAG *LIGO*

Montage. Dessa forma, em comparação aos DAGs menores, o aumento do multiplicador de $\mathcal{T}_{max}$ no *LIGO*, ou seja, o aumento do *deadline*, implica em um aumento maior no conjunto $\mathcal{T}$. Logo, o escalonamento torna-se computacionalmente caro de ser solucionado mais rapidamente em DAGs maiores. Portanto, para diminuir o tamanho do conjunto $\mathcal{T}$ no escalonamento do DAG *LIGO*, temos que aumentar consideravelmente o valor de λ. Por exemplo, para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 3/7$, foi possível encontrar soluções viáveis somente quando $\lambda \in \{9, 14\}$, e para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$, apenas quando $\lambda \in \{18, 25\}$. Além disso, os diferentes níveis de discretização da linha do tempo podem nos ajudar a escolher um valor para λ que gere um custo mínimo para um determinado *deadline* do escalonamento, como, por exemplo, $\lambda = 15$ para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 4/7$ e $\lambda = 19$ para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$. Portanto, dependendo do tamanho do DAG e do *deadline*, temos que aumentar ou diminuir a discretização do tempo para que o *solver* possa encontrar alguma solução viável em tempo hábil ao escalonamento.

A Figura 10 apresenta o consumo médio de memória RAM das simulações do DAG *LIGO*. Como podemos analisar, quanto maior o valor de λ, o *solver* tende a consumir menos memória RAM devido à diminuição da complexidade computacional do problema. Isto é, conseguimos diminuir consideravelmente o tamanho da matriz binária de três dimensões, comentado no início desta seção, o qual está estritamente relacionado com o tamanho do conjunto de soluções do escalonamento. No entanto, o aumento de λ não diminuiu o tempo de execução do *solver*, mas o ajudou a avançar na árvore de enumeração mais rápido. Portanto, se não aplicarmos vários níveis de granularidade da linha do tempo, ou seja se utilizarmos apenas $\lambda = 1$, provavelmente não teríamos nenhuma solução em tempo hábil e o *solver* poderia requisitar uma grande quantidade de memória RAM.



(a) Custo ($)

(b) Makespan

(c) Tempo de Execução

(d) Soluções inviáveis (%)

Figura 9. Resultados do DAG *LIGO*

7. CONCLUSÃO

Neste trabalho apresentamos uma abordagem para reduzir o tempo de execução de DAGs (com vários nós e várias dependências) através do uso de diferentes níveis de discretização da linha do tempo. Realizamos simulações com vários DAGs que representam aplicações do mundo real, tais como: *Montage*, *Ligo* e DAG *fork-join*. Quando simulamos os DAGs com $\lambda = 1$ e *deadlines* altos, o tamanho do conjunto de intervalos discretos $\mathcal{T} = \{1, \ldots, \mathcal{D}_\mathcal{G}\}$ pode ser suficientemente grande de tal modo que impede que se encontre solução viável em tempo hábil. Portanto, os resultados apresentados neste trabalho sugerem que o aumento da granularidade da discretização do tempo pode ser eficaz na redução do tempo de execução de DAGs com vários nós e várias dependências. Além disso, o aumento do valor de λ também trouxe redução de custos monetários em alguns casos, permitindo o *solver* de "avançar" na árvore de enumeração de λ menores. Por outro lado, o aumento de λ pode resultar em desperdícios de recursos no escalonamento e, consequentemente, no aumento do custo monetário. Portanto, a abordagem do aumento da granularidade da linha temporal pode auxiliar o provedor de SaaS a negociar os SLAs com seus clientes.

Como dito anteriormente, há uma necessidade em criar uma heurística para corrigir o erro (desperdício de recursos) inserido pelo aumento da discretização do tempo, como, por exemplo, obter um escalonamento realizado com $\lambda = 5$ e calcular os novos custos monetários e *makespans* em $\lambda = 1$, porém sem reescalonar o DAG. Dessa forma, conseguimos corrigir esse erro inserido e visualizar os instantes de tempo não utilizados em $\lambda = 1$, mas usados $\lambda = 5$. Além disso, podemos utilizar esses instantes de tempo não utilizados para escalonar nós de outros DAGs, tornando o escalonador em um de múltiplos DAGs. A criação desse algoritmo é considerado como trabalho futuro. Também podemos apontar como trabalhos futuros o relaxamento do programa linear inteiro, com o desenvolvimento de heurísticas não iterativas para encontrar soluções viáveis inteiras sobre as execuções relaxadas, e o escalonamento de múltiplos DAGs e DAGs dinâmicos em nosso cenário.

Referências

[1] L. F. Bittencourt and E. R. M. Madeira. Fulfilling task dependence gaps for workflow scheduling on grids. In *Proceedings of the 2007 3rd International IEEE Conference on Signal-Image Technologies and Internet-Based System*, SITIS '07, pages 468–

475, Washington, DC, USA, 2007. IEEE Computer Society.

[2] L. F. Bittencourt and E. R. M. Madeira. HCOC: a cost optimization algorithm for workflow scheduling in hybrid clouds. *Journal of Internet Services and Applications*, 2:207–227, 2011.

[3] C.G. Chaves, D.M. Batista, and N.L.S. da Fonseca. Scheduling grid applications on clouds. In *Proceedings of the 2010 IEEE Global Telecommunications Conference*, GLOBECOM'10, pages 1 –5, dec. 2010.

[4] G. Cornuéjols. Revival of the gomory cuts in the 1990's. *Annals of Operations Research*, 149:63–66, 2007.

[5] E. Deelman, G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Journal of Scientific Programming*, 13:219–237, July 2005.

[6] C. Floudas and X. Lin. Mixed integer linear programming in process scheduling: Modeling, algorithms, and applications. *Annals of Operations Research*, 139:131–162, 2005.

[7] T. A. L. Genez, L. F. Bittencourt, and E. R. M. Madeira. Workflow scheduling for SaaS / PaaS cloud providers considering two SLA levels. In *Proceedings of the 2012 IEEE/IFIP Network Operations and Management Symposyum*, NOMS'12, pages 906 –912, april 2012.

[8] Ralph E. Gomory. Outline of an algorithm for integer solutions to linear programs and an algorithm for the mixed integer problem. In *Proceedings of the 50 Years of Integer Programming 1958-2008*, pages 77–103. Springer Berlin Heidelberg, 2010.

[9] A. H. Land and A. G. Doig. An automatic method for solving discrete programming problems. In *Proceedings of the 50 Years of Integer Programming 1958-2008*, pages 105–132. Springer Berlin Heidelberg, 2010.

[10] G. Papuzzo and G. Spezzano. Autonomic management of workflows on hybrid grid-cloud infrastructure. In *Proceedings of the 7th International Conference on Network and Services Management*, CNSM '11, pages 230–233, Laxenburg, Austria, Austria, 2011. International Federation for Information Processing.

[11] A. Ramakrishnan, G. Singh, H. Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Meyers, and M. Samidi. Scheduling data-intensiveworkflows onto storage-constrained distributed resources. In *Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid*, CCGRID'07, pages 401–409, Washington, DC, USA, 2007. IEEE Computer Society.

[12] H. Stefansson, S. Sigmarsdottir, P. Jensson, and N. Shah. Discrete and continuous time representations and mathematical models for large production scheduling problems: A case study from the pharmaceutical industry. *European Journal of Operational Research*, 215(2):383 – 392, 2011.

[13] R. Van den Bossche, K. Vanmechelen, and J. Broeckhove. Cost-optimal scheduling in hybrid IaaS clouds for deadline constrained workloads. In *Proceedings of the 2010 3rd IEEE International Conference on Cloud Computing*, CLOUD'10, pages 228 –235, july 2010.

[14] X. Wang, C. S. Yeo, R. Buyya, and J. Su. Optimizing the makespan and reliability for workflow applications with reputation and a look-ahead genetic algorithm. *Journal of Future Generation Computer Systems*, 27:1124–1134, October 2011.

[15] L. Wu, S. Garg Kumar, and R. Buyya. SLA-based admission control for a software-as-a-service provideri cloud computing environments. Technical Report CLOUDS-TR-2010-7, University of Melbourne, Australia, sep. 2010.

[16] Linlin Wu, S.K. Garg, and R. Buyya. SLA-based resource allocation for software as a service provider (SaaS) in cloud computing environments. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2011)*, pages 195 –204, may 2011.

[17] Y. Yao, J. Liu, and L. Ma. Efficient cost optimization for workflow scheduling on grids. In *Proceedings of the 2010 International Conference on Management and Service Science*, MASS'10, pages 1 –4, aug. 2010.

[18] K. L. Yee and N. Shah. Improving the efficiency of discrete time scheduling formulation. *Journal of Computers Chemical Engineering*, 22, Supplment 1(0):S403 – S410, 1998.

[19] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, pages 7–18, 2010.