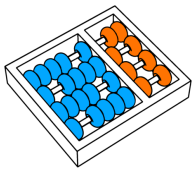
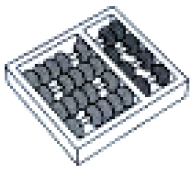


Thiago Augusto Lopes Genez

"Escalonamento de *Workflows* para Provedores de SaaS/PaaS Considerando Dois Níveis de SLA"

CAMPINAS
2012



Universidade Estadual de Campinas
Instituto de Computação

Thiago Augusto Lopes Genez

"Escalonamento de *Workflows* para Provedores de SaaS/PaaS Considerando Dois Níveis de SLA"

Orientador(a): **Prof. Dr. Edmundo Roberto Mauro Madeira**

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Estadual de Campinas para obtenção do título de Mestre em Ciência da Computação.

Este exemplar corresponde à versão final da Dissertação defendida por Thiago Augusto Lopes Genez, sob orientação de Prof. Dr. Edmundo Roberto Mauro Madeira.

Assinatura do Orientador(a)

CAMPINAS
2012

FICHA CATALOGRÁFICA ELABORADA POR
MARIA FABIANA BEZERRA MULLER - CRB8/6162
BIBLIOTECA DO INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E
COMPUTAÇÃO CIENTÍFICA - UNICAMP

Genez, Thiago Augusto Lopes, 1987-
G287e Escalonamento de workflows para provedores de SaaS/PaaS considerando dois níveis de SLA / Thiago Augusto Lopes Genez. – Campinas, SP : [s.n.], 2012.

Orientador: Edmundo Roberto Mauro Madeira.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

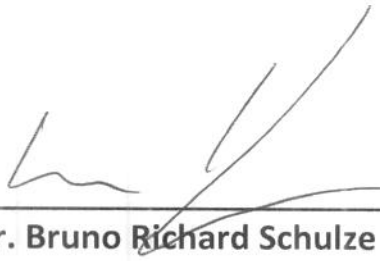
1. Computação em nuvem. 2. Programação linear. 3. Fluxo de trabalho. 4. Escalonamento da produção. I. Madeira, Edmundo Roberto Mauro,1958-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

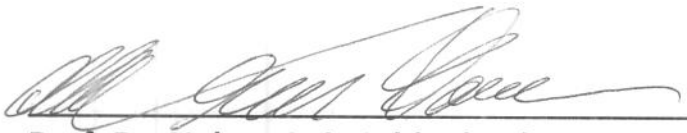
Título em inglês: Workflow scheduling for SaaS / PaaS cloud providers considering two SLA levels
Palavras-chave em inglês:
Cloud computing
Linear programming
Workflow
Scheduling production
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Edmundo Roberto Mauro Madeira [Orientador]
Bruno Richard Schulze
Nelson Luis Saldanha da Fonseca
Data de defesa: 27-09-2012
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 27 de Setembro de 2012, pela Banca examinadora composta pelos Professores Doutores:



Prof. Dr. Bruno Richard Schulze
LNCC



Prof. Dr. Nelson Luis Saldanha da Fonseca
IC / UNICAMP

Prof. Dr. Edmundo Roberto Mauro Madeira
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Escalonamento de *Workflows* para Provedores de SaaS/PaaS Considerando Dois Níveis de SLA

Thiago Augusto Lopes Genez[1]

27 de Setembro de 2012

Banca Examinadora:

- Prof. Dr. Edmundo Roberto Mauro Madeira (Orientador)
- Prof. Dr. Bruno Richard Schulze
  LNCC
- Prof. Dr. Nelson Luis Saldanha da Fonseca
  IC - UNICAMP
- Prof. Dr. Daniel Macêdo Batista (Suplente)
  IME - USP
- Prof. Dr. Luiz Eduardo Buzato (Suplente)
  IC - UNICAMP

[1]O aluno recebeu recursos financeiros da CAPES entre 03/2010 e 02/2009; e da FAPESP (processo 2010/14433-8) entre 03/2011 e 02/2012

Resumo

Computação em nuvem oferece utilidades computacionais de acordo com a necessidade do usuário através do modelo "pago-pelo-uso". Usuários podem fazer o uso da nuvem através dos provedores de software (*Software as a Service* – SaaS), de plataforma (*Plataform as a Service* – PaaS) ou de infraestrutura (*Infrastructure as a Service* – IaaS). Computação em nuvem está atualmente sendo muito utilizada por instituições para evitar custos de manutenção e investimentos iniciais em infraestrutura computacional. Por exemplo, as aplicações complexas de e-Ciência e *e-Business* requisitam, hoje em dia, um poder computacional cada vez maior, o qual é tradicionalmente superior ao montante que está disponível nas instalações de uma única instituição. Assim, as nuvens podem ser utilizadas para fornecer recursos de computação extras sempre que necessário, trazendo *elasticidade* ao poder computacional local (ou privado). Essas aplicações complexas podem ser modeladas como *workflows*, compondo um conjunto de serviços a serem processados em uma ordem bem-definida; ou seja, podem haver dependências de dados entre serviços que geram precedência de execução. *Workflows* são geralmente representados por grafos acíclicos direcionados (*Directed Acyclic Graphs* – DAGs) para representar os serviços e suas dependências. O escalonador é o componente responsável por decidir de que forma a distribuição desses serviços será realizada; porém, o problema de escalonamento de *workflows*, em sua forma geral, é NP-Completo. Devido à insuficiência de recursos próprios, os usuários podem recorrer aos serviços de execução de *workflows* disponibilizados pelos provedores de SaaS. Estes, por outro lado, podem ter apenas recursos privados para lidar com um pequeno número de clientes e, portanto, alugam recursos de provedores de infraestrutura para executar os *workflows* quando necessário. Nesta dissertação, consideramos um provedor de SaaS com dois níveis de acordo de nível de serviço (*Service Level Agreement* – SLAs), dos quais um representa os SLAs estabelecidos com seus clientes, enquanto o outro contém os SLAs acordados com cada provedor de IaaS. Portanto, através do algoritmo de escalonamento de *workflows* apresentado neste trabalho, pretendemos que o provedor de SaaS (i) obedeça os requisitos de qualidade de serviço (*Quality of Service* – QoS) de cada *workflow*; (ii) minimize quebras de SLAs de seus clientes; e (iii) minimize gastos monetários envolvidos com a terceirização de recursos computacionais.

Abstract

Cloud computing offers utility computing according to the user's needs in a "pay-per-use" basis. Customers can make use of the cloud via Software as a Service (SaaS), Plataform as a Service (PaaS), or Infrastructure as a Service (IaaS) providers. Cloud computing is now being widely used by institutions to avoid maintenance costs and upfront investment in computing infrastructure. For instance, e-Science and e-Business complex applications are nowadays requiring an increasing computational power, which is traditionally exceeding the amount that is available within the premises of a single institution. Therefore, clouds can be used to provide additional computing resources where necessary, bringing *elasticity* to the local (or private) computational power. These complex applications can be modeled as workflows, composing a set of services to be processed in a well-defined order, in other words, there may be data dependencies between services that yield in precedence constraints. Thus, workflows are usually represented by directed acyclic graphs (DAGs) to represent the services and their dependencies. The scheduler is the component responsible for deciding how the distribution of such services will occur, however, the workflow scheduling problem, in its general form, is NP-Complete. Due to lack of privately owned resources, customers can use workflow execution service that is provided by a SaaS provider. On the other hand, SaaS providers may only have private resources to cope with a small number of clients, therefore, leasing resources from infrastructure providers to execute their customer's workflows when needed. In this thesis, we consider a SaaS provider with two levels of service level agreement (SLA). One level includes all SLAs made with its customer, while another one contains all SLAs signed with each IaaS provider. Therefore, through the workflow scheduling algorithm presented in this work, we intend that the SaaS provider (i) meets each workflow's quality of service (QoS); (ii) minimizes breaks of its customer's SLA; and (iii) minimizes the monetary costs involved with outsourcing computational resources.

Agradecimentos

Primeiramente, agradeço à minha família, principalmente aos meus pais, Carlos Augusto e Jandira, e ao meu irmão, Lucas, pelo carinho e apoio em todas as minha decisões para que eu chegasse até esta etapa de minha vida. Agradeço em especial ao meu finado pai e a minha mãe por terem me ensinado a arte de pensar com rigor e disciplina, propiciando-me a fundamentação básica, sem a qual esta dissertação não teria sido escrita. Agradeço, de forma muito carinhosa, a atuação do meu irmão no período do desenvolvimento deste trabalho e também pela amizade e dedicação comigo, além de fazer companhia a minha mãe enquanto estive fora de Londrina nestes últimos anos. Ao meu primo, Giorge, e sua esposa Angela pelo apoio familiar aqui em Campinas.

À Amanda, minha namorada, pelo apoio, pelos momentos difíceis, pelos momentos agradáveis, pelo carinho, pelos filmes, pelas viagens, pelas risadas, pela compreensão durante minha ausência nestes últimos anos, pelo companheirismo, pela amizade, pelo incentivo, pela correção desta dissertação e por todas as outras coisas indescritíveis desfrutadas durante esse tempo juntos. À sua irmã Flávia, ao seu cunhado Rodrigo, à sua mãe Silvana e ao seu pai Luiz Antônio, pela amizade e pelos momentos agradáveis.

Ao meu orientador, Professor Edmundo Madeira, pela orientação sempre atenciosa, correta e cuidadosa.

Ao Professor Luiz Bittencourt, amigo e companheiro, pelo trabalho em conjunto e pelas dicas e sugestões em relação ao desenvolvimento desta dissertação, e também pelos papos nerds (e não nerds) de computeiros.

Ao Carlos Eduardo de Andrade, amigo e companheiro de pós-graduação, membro do Laboratório de Otimização Combinatória (LOCo), que contribuiu com dicas preciosas para o desenvolvimento do programa linear inteiro apresentado neste trabalho.

A todos amigos do Laboratório de Redes de Computadores (LRC) que contribuíram pelas dicas, pelas sugestões, pelos bate-papos na sala de café e nos corredores, pelas companhias no laboratório (durante a semana, nos finais de semanas, nos feriados e nas madrugadas) e pelos momentos de descontração. Alisson Pontes, Carlos Augusto Froldi, Carlos Senna, Carlos Trujillo, Cesar Chaves, Daniel Batista, Daniel Lago, Esteban Rodriguez, Flávio Kubota, Geraldo Silva, Guilherme Russi, Gustavo Alkmim, Jorge de Oliveira,

Juliana Borin, Juliana de Santi, Luciano Chaves, Mariana Dias, Milton Soares, Neumar Malheiros, Pedro Gomes, Rafael Lopes, Rafael Rosa, Rafael Scaraficci, Rodolfo Meneguette, Tiago Andrade e Walisson Pereira.

A todos os amigos e amigas de Campinas e de Londrina que, de uma forma ou de outra, fizeram parte da minha vida nesses últimos 2 anos e que contribuíram com sua amizade, tornando a minha vida agradabilíssima. Não irei citar nomes para não deixar ninguém de fora, porém todos estão implicitamente incluídos aqui e expresso a minha profunda gratidão por fazer parte da minha felicidade.

Aos funcionários e professores do Instituto de Computação (IC) que colaboraram com a minha formação e se esforçaram para que eu pudesse desenvolver meu trabalho com o melhor apoio possível.

À CAPES e FAPESP pela bolsa de mestrado.

Agradecer a todos que me ajudaram a desenvolver esta dissertação de mestrado não é uma tarefa fácil. O principal problema de um agradecimento seletivo não é decidir quem incluir, mas decidir quem não mencionar. Se nestes agradecimentos não há nenhuma referência direta à você que está lendo-os, e você tem motivos claros que contribuíram com este trabalho, então, primeiramente, peço-lhe desculpas e, em seguida, permito-lhe escrever de próprio punho seu nome nas linhas em branco seguintes.

Sumário

Lista de Abreviaturas

Amazon EC2	*Amazon Elastic Compute Cloud*, p. 3
Amazon S3	*Amazon Simple Storage Service*, p. 9
API	*Application Programming Interface*, p. 13
ARPANet	*Advanced Research Projects Agency Network*, p. 1
BRS	*Best Resource Selection*, p. 31
BMT	*Begin-Minimum Time*, p. 48
BMEMT	*Begin-Minimum End-Maximum Time*, p. 48
BPEL	*Business Process Execution Language*, p. 32
CPU	*Central Processing Unit*, p. 9
CRM	*Customer Relationship Management* , p. 13
DaaS	*Data as a Service*, p. 11
DAG	*Directed Acyclic Graph*, p. 3
DIaaS	*Data Integrity as a Service*, p. 11
Enomaly ECP	*Enomaly Elastic Computing Platform*, p. 12
GLPK	*GNU Linear Programming Kit*, p. 43
HGP	*Heteroscedastic Gaussian Processes*, p. 29
HuaaS	*Human as a Service*, p. 11
HCOC	*Hybrid Cloud Optimized Cost*, p. 30
IaaS	*Infrastructure as a Service*, p. 3
JNI	*Java Native Interface*, p. 55
JVM	*Java Virtual Machine*, p. 55
NaaS	*Network as a Service*, p. 11
NIST	*National Institute of Standards and Technology*, p. 8

OPF	*Open Grid Forum*, p. 24
PSO	*Particle Swarm Optimization*, p. 31
PaaS	*Plataform as a Service*, p. 3
PL	Programa Linear
PLI	Programa Linear Inteiro, p. 27
PQ	Programa Quadrático
PIM	Programa Inteiro Misto, p. 33
QoS	*Quality of Service*, p. 4
RAM	*Random Access Memory*, p. 10
RDPSO	*Revised Discrete Particle Swarm Optimization*, p. 31
SaaS	*Software as a Service*, p. 3
SCIP	*Solving Constraint Integer Programs*, p. 43
SOC	*Service Oriented Computing*, p. 1
SLA	*Service Level Agreements*, p. 2
TVM	*Task and Virtual Machine*, p. 28
TI	Tecnologia da Informação, p. 10
VM	*Virtual Machine*, p. 4
VMM	*Virtual Machine Monitor*, p. 11

Lista de Algoritmos

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

À medida que a tecnologia avança, temos computadores com maior poder de processamento, maior quantidade de memória e maior capacidade de armazenamento. Paralelamente a esse avanço tecnológico, as aplicações (software) estão cada vez mais exigindo uma demanda maior por computação e, dessa forma, os supercomputadores podem não ser suficientes para suprir todas as necessidades dessas aplicações. Com intuito de sanar esse problema, o campo da computação distribuída vêm substituindo os supercomputadores na execução de aplicações que solicitam alto poder de processamento e alta capacidade de armazenamento. A computação distribuída, também denominada sistema distribuído, é uma referência à computação paralela e descentralizada, cujo principal objetivo é executar uma tarefa em comum [13].

O desenvolvimento dos sistemas distribuídos ocorreu rapidamente a partir de tecnologias de computadores pessoais, primeiramente através de *clusters* de computadores homogêneos, em seguida pela computação em grade (com recursos heterogêneos) e agora com a *computação em nuvem* [57]. Em outras palavras, a computação em nuvem surgiu como um paradigma mais recente da computação distribuída e, atualmente, atrai um grande interesse de pesquisadores principalmente nas áreas da própria Computação Distribuída e da Computação Orientada a Serviço – *Service Oriented Computing* (SOC) – [79]. De forma geral, todas essas derivações da computação distribuída se concentram em disponibilizar um alto poder de computação para um grande número de usuários finais de forma confiável, eficiente, escalável e, além disso, de baixo custo monetário. Além disso, a tendência dessas tecnologias é fornecer computação como um serviço de utilidade, ou seja, estando disponível sempre que o usuário solicitar, muito parecido com os tradicionais serviços utilitários públicos, tais como, eletricidade, gás, água e telefonia [36].

A essência do conceito de *computação utilitária* foi descrita em 1969, quando *Leonard Kleinrock*, um dos principais cientistas no desenvolvimento da *Advanced Research Projects Agency Network* (ARPANet), a rede precursora da Internet, disse [45, 46]: *até*

agora, as redes de computadores estão ainda na sua infância, mas à medida que cresçam e se tornem mais sofisticadas, nós provavelmente veremos a disseminação de "utilitários computacionais" que, como os atuais serviços de eletricidade e telefone, atenderão casas individuais e escritórios em todo o país. Essa visão de computação utilitária tem transformado toda a indústria de computação e, além disso, vem se concretizando cada vez mais com a viabilização em larga escala da computação em nuvem, onde os recursos computacionais estão disponíveis como serviços através da Internet, fornecidos sob demanda e tarifados de acordo com o modelo "pago-pelo-uso" [18]. Portanto, os consumidores (usuários da computação em nuvem) não precisam investir fortemente ou se deparar com dificuldades na construção e manutenção de uma infraestrutura complexa de computação. Enfim, após a computação em *cluster* e a computação em grade, podemos dizer que a maturidade da computação utilitária está ocorrendo através da computação em nuvem.

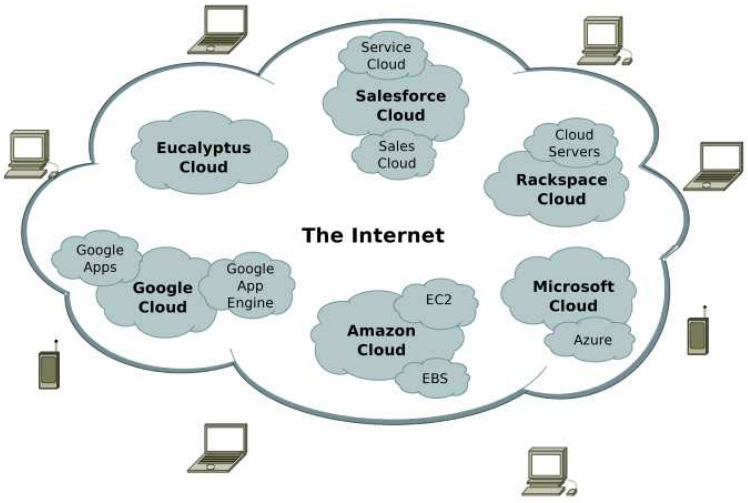


Figura 1.1: Serviços da computação em nuvem (figura retirada de [42])

O paradigma da computação em nuvem pode ser visto como uma infraestrutura escalável, pois suporta e interliga vários serviços de computação (ver Figura 1.1). Dessa forma, a nuvem permite o uso dos padrões da SOC, permitindo aos usuários estabelecer ligações entre serviços, organizando-os como *workflows* ao invés de construir apenas aplicações tradicionais que são executadas em computadores individuais [16]. Segundo Buyya *et al.* em [18, 19] o termo *nuvem* consiste em: *um conjunto de computadores interconectados e virtualizados que são dinamicamente fornecidos e apresentados como um ou mais recursos de computação unificados com base em acordos de nível de serviço – Service Level Agreementss* (SLAs) *– estabelecidos por meio da negociação entre o provedor de serviço e o usuário (consumidor).* Além disso, os detalhes de hardware são abstraídos dos usuários, os quais não precisam ter conhecimento ou experiência sobre a infraestrutura tecnológica

que a nuvem utiliza.

A computação em nuvem é dividida de acordo com os serviços fornecidos: *Software as a Service* (SaaS), *Plataform as a Service* (PaaS) e *Infrastructure as a Service* (IaaS). No SaaS, o usuário usa um aplicativo, mas não controla o ambiente de execução; *Google Apps*[1] e *Salesforce.com*[2] são exemplos desse modelo. No PaaS, os usuários utilizam um ambiente para hospedar suas aplicações; *Google App Engine*[3] e *Microsoft Azure*[4] são exemplos de PaaS. No IaaS, o usuário usa recursos de computação, tais como poder de processamento e armazenamento. Nesse modelo, o usuário tem mais controle no ambiente computacional, incluindo a execução de aplicativos; *Amazon Elastic Compute Cloud* (*Amazon EC2*)[5], *Rackspace Cloud*[6] e *Eucalyptus*[7] são bons exemplos de IaaS. Todos esses exemplos estão citados na Figura 1.1.

Portanto, o ambiente da computação em nuvem permite alugar recursos sob demanda para melhorar a capacidade computacional disponível em sistemas privados (computadores pessoais, *clusters* e grades), fornecendo recursos computacionais quando necessário. Em outras palavras, a nuvem consegue fornecer uma *elasticidade* ao ambiente computacional do usuário, aumentando ou diminuindo o poder computacional de acordo com a demanda de suas aplicações e, além disso, de forma rápida, fácil e de baixo custo monetário; concretizando a visão de computação utilitária de Leonard Kleinrock.

1.1 Motivação, Objetivos e Contribuições

Várias aplicações complexas de e-Ciência e *e-Business* podem ser modeladas como *workflows*, compondo um conjunto de serviços a serem processados em uma ordem bem-definida [16]. Isto é, cada serviço pode depender de dados computados por outros serviços que o antecedem. Assim, devido a esse acoplamento entre serviços, *workflows* são geralmente representados por grafos acíclicos direcionados – *Directed Acyclic Graph* (DAG) – para representar os serviços e suas dependências. Em outras palavras, DAGs são *workflows* em uma forma mais restrita, pois representam aplicações compostas de serviços que não possuem ciclos de dependência. Desse modo, quando tais aplicações são executadas em recursos alugados da nuvem, a maneira como esses serviços são distribuídos para a execução e a ordem dessas execuções são fatores decisivos, que estipulam tanto o desempenho quanto o custo monetário do escalonamento. Assim, o desenvolvimento de algoritmos

[1] `http://www.google.com/apps/`
[2] `http://www.salesforce.com/`
[3] `https://developers.google.com/appengine/`
[4] `http://www.windowsazure.com`
[5] `http://aws.amazon.com/ec2/`
[6] `http://www.rackspace.com/cloud/`
[7] `http://www.eucalyptus.com/`

para escalonamento de DAGs em nuvens, considerando suas dependências e as características dos provedores nuvens, são de grande interesse atualmente [12, 16, 31, 38, 58, 80]. Por exemplo, uma das requisições para a execução desses DAGs é que o tempo de resposta (*deadine*) seja curto e, para que isso se torne possível, um poder de processamento deve ser adquirido de forma rápida caso os recursos privados forem insuficientes; característica de *elasticidade* computacional oferecida pela computação em nuvem.

Geralmente os usuários não possuem recursos computacionais próprios que sejam suficientes para executar seus DAGs e, desse modo, podem recorrer aos serviços de execução de *workflows* disponibilizados na nuvem pelos provedores de SaaS e de PaaS[8]. Note que, a solicitação destes serviços pode ser sazonal e, portanto, o provedor de SaaS deve estar preparado não só para atender picos de demanda, mas também fornecer uma boa qualidade de serviço – *Quality of Service* (QoS) – definida no SLA de cada um dos seus clientes. Além disso, os recursos privados do provedor de SaaS devem ser suficientes para lidar com uma grande quantidade de clientes, o que implica em altos custos de manutenção e subutilização em épocas fora do pico de demanda. Para contornar esta situação, uma alternativa rápida e de baixo custo é a terceirização de recursos através de máquinas virtuais providenciadas pelos provedores de IaaS. Essa alternativa traz *elasticidade* ao poder computacional do provedor de SaaS. Enfim, com um recurso computacional *elástico*, o provedor de SaaS pode minimizar quebras de SLA acordados com cada cliente e maximizar seu lucro.

Entretanto, o problema de escalonamento de DAGs é, em geral, NP-Completo e, como consequência, aprimoramentos em algoritmos, técnicas e heurísticas são fundamentais para se obter um bom balanceamento entre o tempo de execução do escalonador, a complexidade e a qualidade do escalonamento [13]. A propósito, em um ambiente composto principalmente por recursos alugados da computação em nuvem, esse problema adquire novas variáveis e torna-se ainda mais complexo, como, por exemplo, determinar quais provedores de IaaS e quais tipos de máquinas virtuais – *Virtual Machine* (VM) – devem ser alugadas para que o escalonamento não tenha um custo monetário alto e, além disso, com um *makespan* (duração do escalonamento) também alto. Isto é, máquinas virtuais rápidas são geralmente mais caras na nuvem e, consequentemente, processarão o DAG mais rapidamente, diminuindo o *makespan*. Portanto, para aperfeiçoar a execução de serviços dependentes representados por DAGs em nuvens, devemos considerar algumas dificuldades, das quais se destacam: (i) a heterogeneidade de máquinas virtuais com um poder de processamento e uma tarifa associada; e (ii) o tempo de comunicação entre essas máquinas virtuais. A heterogeneidade implica em uma complexidade na seleção de

[8]Por uma questão de clareza, iremos utilizar no restante deste documento, apenas o termo SaaS para referenciar o provedor de nuvem que disponibiliza serviço de execução de *workflows*, porém este trabalho também se aplica aos provedores de PaaS.

recursos para execução dos serviços, enquanto o tempo de comunicação pode atrasar o escalonamento (*deadine*).

Nesta dissertação, abordamos o problema de escalonamento de serviços dependentes representados por grafos acíclicos direcionados em nuvens públicas, levando em consideração características inerentes a esse ambiente computacional. As contribuições apresentadas nesta dissertação são algoritmos para escalonamento de serviços dependentes representados por DAGs que incluem:

- Um programa linear inteiro que considera o escalonamento em dois níveis de SLA;
- Duas heurísticas para obter soluções inteiras da versão relaxada desse programa linear inteiro;
- Um método para aumentar a granularidade da discretização da linha do tempo com intuito de reduzir o tempo de execução do escalonador;
- A implementação do escalonador proposto no solucionador matemático *IBM ILOG CPLEX Optimizer*[9] na linguagem de programação *Java*[10];
- Uma avaliação de desempenho do escalonador de *workflows* em nuvens proposto, através de um processo comparativo entre os resultados obtidos nas simulações realizadas no *CPLEX*.

1.2 Organização da Dissertação

Os capítulos desta dissertação estão organizados como segue:

- O Capítulo 2 contém uma revisão sobre os conceitos necessários para a compreensão desta dissertação. São discutidos detalhes sobre a computação em nuvem, incluindo a descrição de *workflows* de serviços como um grafo acíclico direcionado (DAG) e o problema de escalonamento desses *workflows* de forma geral. Também são abordados conteúdos sobre acordo de nível de serviço (SLA), assim como o cenário de escalonamento de *workflows* em nuvens desenvolvido neste trabalho;
- O Capítulo 3 apresenta uma ampla revisão bibliográfica dos algoritmos de escalonamento de *workflows*, os quais foram categorizados de acordo com a infraestrutura computacional utilizada, as quais são: grades computacionais, nuvens híbridas, nuvens comunitárias e nuvens públicas. As principais vantagens e desvantagens de

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

cada um desses algoritmos também são discutidas, incluindo informações a respeito do escalonador proposto nesta dissertação;

- O Capítulo 4 detalha o escalonador proposto como um programa linear inteiro (PLI), incluindo as notações matemáticas, a modelagem do problema de escalonamento em nuvens, as variáveis binárias e as restrições lineares desse PLI. Também são apresentadas as abordagens para solucionar o programa linear inteiro, através de heurísticas, relaxação linear e aumento da discretização da linha do tempo;

- O Capítulo 5 mostra os resultados obtidos durante a avaliação de desempenho do escalonador proposto, bem como as configurações e os parâmetros utilizados durante o processo de simulação. Toda avaliação foi realizada através de simulações no *solver IBM ILOG CPLEX Optimizer*;

- O Capítulo 6 conclui esta dissertação, destacando as principais contribuições e potenciais extensões para o trabalho realizado.

Capítulo 2

Fundamentos Básicos

Este capítulo aborda alguns conceitos básicos sobre os assuntos envolvidos nesta dissertação. Primeiramente, na Seção 2.1, são apresentadas a definição e as principais características essenciais da computação em nuvem, e os principais serviços oferecidos pelos provedores de nuvem. Na sequência, a Seção 2.2 descreve o conceito de *workflow* de serviço, e como representá-lo através de um grafo acíclico direcionado (DAG). O problema de escalonamento de *workflows* é apresentado na Seção 2.3, enquanto a Seção 2.4 aborda o acordo de nível de serviço (SLA), incluindo o conceito e a aplicabilidade na computação em nuvem. Na Seção 2.5 é descrito o cenário de escalonamento de *workflows* em nuvens desenvolvido neste trabalho. Para finalizar este capítulo, apresentamos, na Seção 2.6, algumas considerações finais sobre os assuntos tratados neste capítulo.

2.1 Computação em Nuvem

Esta seção apresenta uma visão geral do paradigma da computação em nuvem, incluindo a definição e as características essencias na Seção 2.1.1. As tecnologias relacionadas com a computação em nuvem são descritas na Seção 2.1.2, enquanto a Seção 2.1.3 apresenta os tipos de serviços disponibilizados por esse paradigma. Por fim, a Seção 2.1.4 descreve os três tipos de nuvens existentes na computação em nuvem.

2.1.1 Definição e Características Essenciais

Com o rápido desenvolvimento de tecnologias de processamento e de armazenamento, os recursos de computação tornaram-se mais baratos, mais poderosos e mais onipresentes na vida moderna. Essa evolução tecnológica permitiu o desenvolvimento de um novo modelo de computação, denominado *computação em nuvem*. A computação em nuvem, em outras palavras, 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" (em inglês, as expressões *pay-as-you-go*, *charge-per-use* ou *pay-per-use*) [5, 18, 42, 85].

A computação em nuvem possui várias definições na literatura [5, 6, 18, 34, 42, 71, 74, 85–87]. Nesta dissertação, porém, vamos adotar a versão final publicada recentemente (setembro de 2011) pelo *National Institute of Standards and Technology* (NIST). Assim, a definição do termo *computação em nuvem* dada pelo NIST em [56] é a seguinte:

> *Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*

Além disso, o NIST também definiu 5 características essenciais para a nuvem, as quais estão listadas a seguir:

- **Autoatendimento sob demanda:** O usuário pode solicitar por conta própria os recursos de computação (software e hardware como serviço) automaticamente e conforme necessário, sem a necessidade de intervenção humana com cada provedor de serviço (IaaS, PaaS ou SaaS – descritos na Seção 2.1.3);
- **Amplo acesso aos serviços:** Os recursos estão disponíveis através da Internet e são acessados por qualquer dispositivo que conecta nessa rede;
- ***Resource pooling:***[1] Os recursos computacionais (físicos ou virtuais) do provedor de serviço são agrupados através do modelo *multi-tenant*[2], com intuito de servir

[1] *Resource pooling* é um termo usado para descrever a situação onde vários recursos se comportam como se fossem apenas um [76]. O objetivo dessa técnica é aumentar a confiabilidade, a flexibilidade e a eficiência dos recursos como um todo.

[2] "Múltiplos-inquilinos", do inglês *multi-tenant*, refere-se a um princípio da arquitetura de software, onde uma *única* instância do software é executada em um servidor para atender às requisições de múltiplos clientes (inquilinos). Aplicando essa ideia no hardware, o *multi-tenant* ocorre através da virtualização, onde cada inquilino recebe recursos (com uma pilha de tecnologia especificada antecipadamente) que são alocados dinamicamente a partir de um *pool* de recursos. Portanto, uma aplicação *multi-tenant* facilita (i) a escalabilidade, para que a infraestrutura suporte a adição de novos usuários; (ii) a segurança das informações, pois os recursos virtuais (software e hardware) são logicamente isolados; e (iii) a confiabilidade, para que as requisições dos usuário sejam atendidas em um tempo adequado. Além disso, por ter apenas uma única instância do software ou um *pool* de recursos, a manutenção torna-se mais simples [40, 88].

múltiplos usuários. Além disso, esses recursos são alocados, desalocados e realocados dinamicamente, conforme a demanda. Aliás, o usuário geralmente não conhece a localização geográfica exata dos recursos fornecidos, mas o provedor de nuvem pode especificar uma localização com um alto nível de abstração, como por exemplo, país, estado ou região;

- **Elasticidade rápida:** Os provedores de nuvem fornecem e liberam os recursos computacionais de acordo com a demanda do usuário. Em outras palavras, de uma maneira *elástica*, os recursos são rapidamente "aumentados" ou "diminuídos" sob demanda. Esta elasticidade permite que o usuário visualize a nuvem como um ambiente composto por *recursos computacionais infinitos*, permitindo, então, adquirir (alugar) recursos em qualquer quantidade e em qualquer momento. É importante frisar que a elasticidade é possível graças à camada de virtualização;

- **Serviço mensurado:** Cada recurso fornecido pelos provedores de serviços é controlado e tarifado através de medições realizadas em um nível de abstração apropriada para o tipo do serviço. Por exemplo, os recursos de processamento podem ser cobrados por hora[3] (CPU[4]$/hora$) e os serviços de armazenamento por mês[5] ($bytes/mês$). Desta maneira, na computação em nuvem, cada serviço é tarifado através do modelo "pago-pelo-uso".

Logo, as principais atratividades do uso da computação em nuvem hoje em dia são: (i) rápida flexibilidade de entrega/liberação dos serviços, graças à característica de *elasticidade*; (ii) baixo custo operacional, devido ao modelo de tarifação pagos-pelo-uso; (iii) redução de investimentos (iniciais) em infraestrutura computacional própria e (iv) transferência do capital gasto em recursos privados para os negócios operacionais. Dessa forma, através da computação em nuvem, os usuários precisam se concentrar apenas na criação de seus negócios, e não com as peculiaridades dos recursos computacionais, como por exemplo, instalações, configurações, atualizações e manutenções nos softwares e/ou hardwares.

2.1.2 Tecnologias Relacionadas

A computação em nuvem não é uma tecnologia completamente nova [71, 85], e sim um modelo de computação com novas perspectivas de negócios [18]. Em outras palavras, a essência da nuvem é composta por um conjunto de tecnologias já existentes (grades computacionais, utilidades computacionais e virtualização) com intuito de criar um novo

[3]Métrica adotada na *Amazon EC2* em `http://aws.amazon.com/ec2/`
[4]*Central Processing Unit* (CPU)
[5]Métrica adotada na *Amazon Simple Storage Service* (*Amazon S3*) em `http://aws.amazon.com/s3/`

modelo de negócio que atenda às exigências tecnológicas e demandas econômicas atuais da Tecnologia da Informação (TI). Uma breve comparação entre a computação em nuvem e essas tecnologias existentes é apresentada a seguir.

***Grades computacionais*:** Uma grade computacional é um sistema geograficamente distribuído, heterogêneo, multi-institucional, colaborativo e dinâmico, onde qualquer recurso computacional é um potencial colaborador. Com objetivo de resolver problemas (científicos) de grande complexidade e/ou grande volume de dados, as grades computacionais têm alto poder de processamento, pois dividem as execuções de tarefas entre os diversos recursos heterogêneos conectados por meio de uma rede local ou pela própria Internet [15]. Muitas das características encontradas em grades computacionais também são encontradas na computação em nuvem, pois ambos os modelos possuem objetivos comuns, tais como [71]: (i) redução dos custos computacionais, (ii) compartilhamento de recursos e (iii) escalabilidade alta. Esses objetivos são semelhantes, pois ambos os modelos utilizam, de modo geral, hardware operados por terceiros. Entretanto, também há diferenças entre esses dois paradigmas, tais como: (i) a maneira como o compartilhamento de recursos é realizado e (ii) a virtualização [9, 11, 71, 85]. Em grades computacionais, os recursos são geralmente compartilhados entre as organizações por meio do modelo "mais justo possível" [44], enquanto na computação em nuvem o usuário tem a sensação de que o recurso providenciado é dedicado somente a ele. Essa sensação ocorre por causa da virtualização que "isola" virtualmente os recursos alugados[6]. Embora seja pouco explorada em grades, a virtualização abrange geralmente a camada de software [9]; enquanto em nuvens, a virtualização é realizada geralmente em termos de recursos de hardware. Detalhes sobre a virtualização em nuvem são esclarecidos abaixo.

***Utilidade Computacional*:** Utilidade computacional refere-se a um modelo de negócios no qual os recursos de computação (CPU, espaço de armazenamento, aplicativos, etc.) são colocados à disposição do usuário de acordo com a sua necessidade (sob demanda) e cobrados pelo uso (*pay-as-you-go*). Dessa maneira, os prestadores de serviços podem melhorar a utilização dos recursos físicos e minimizar os custos operacionais [85, 86]. A ideia da utilidade computacional é semelhante aos tradicionais serviços utilitários públicos, tais como, eletricidade, gás, água e telefonia; isto é, uma ideia antiga.

***Virtualização*:** Um dos componentes principais da computação em nuvem é a virtualização. Essa tecnologia abstrai as características físicas do hardware e fornece recursos

[6] Os recursos alugados da nuvem são "opacos" aos usuários, pois os provedores de nuvem não fornecem detalhes do hardware virtual, como por exemplo, *clock* do processador, frequência do barramento da memória *Random Access Memory* (RAM), etc.

virtualizados para as aplicações de alto nível. Assim, através da virtualização, uma máquina física é, de modo geral, logicamente dividida em um monitor de máquina virtual – *Virtual Machine Monitor* (VMM) – e em várias máquinas virtuais – VMs [51]. É importante ressaltar que cada máquina virtual é independente das outras, podendo ter seu próprio sistema operacional, aplicativos e serviços. O VMM, também conhecido como *hypervisor*, é responsável pelo controle e virtualização dos recursos físicos compartilhados entre as máquinas virtuais, como por exemplo, processadores, memória RAM, disco rígido e dispositivos de entrada e saída. Outras funcionalidades do VMM incluem: (i) isolar o ambiente de cada VM, pois execuções de aplicativos (e serviços) de uma determinada VM não podem ser alteradas (e infuenciadas) por outras VMs; (ii) resolver o conflito entre VMs e (iii) escalonar qual VM vai executar (usar os recursos físicos) a cada instante de tempo [54]. Portanto, cada máquina virtual é basicamente composta por um sistema operacional próprio e por recursos computacionais virtuais, tais como, CPU virtual, memória RAM virtual e discos rígidos virtuais. Enfim, através da virtualização e dos conceitos de *multi-tenant*, a computação em nuvem consegue facilmente compartilhar recursos computacionais de *clusters* de servidores físicos em recursos virtuais, permitindo alocar, desalocar e realocar dinamicamente (e *elasticamente*) os recursos virtuais de acordo com a necessidade do usuário.

2.1.3 Provedores de Serviços Disponíveis em Nuvem

De acordo com o tipo de serviço fornecido, a computação em nuvem é dividida em três modelos de serviço (provedores de serviço): IaaS, PaaS e SaaS. Embora essa divisão seja oficialmente definida pelo NIST em [56], na literatura podemos encontrar outros modelos, como por exemplo: *Data as a Service* (DaaS) [3,72,92], *Network as a Service* (NaaS) [92], *Data Integrity as a Service* (DIaaS) [55] e *Human as a Service* (HuaaS) [49]. Entretanto, estes modelos são classificados de acordo com aqueles três estipulados pelo NIST [42].

A Figura 2.1 mostra a arquitetura da computação em nuvem, bem como a relação entre os três modelos de serviços e os recursos gerenciados em cada camada da arquitetura. A camada de hardware é responsável por gerenciar (e configurar) os recursos físicos da nuvem, incluindo servidores, roteadores e *switches*. Além disso, também é responsável por gerenciar e monitorar os sistemas de tráfego de rede, tolerância a falhas de hardware, energia e resfriamento. Isto é, a camada de hardware é geralmente implementada através de *datacenters*, onde servidores são organizados em *racks* e interligados através de *switches* e roteadores [85].

Imediatamente acima da camada de hardware está localizada a camada de infraes-

Figura 2.1: Arquitetura da computação em nuvem (adaptado de [85]).

trutura, onde os recursos físicos são virtualizados. *KVM*[7], *Xen*[8] e *VMware*[9] são exemplos de *hypervisors* utilizados para realizar essa virtualização, dos quais *Xen* é a base do *hypervisor* da *Amazon EC2* e do *Citrix Systems*[10] [20]. Para gerenciar as máquinas virtuais e os *hypervisors*, o provedor de IaaS utiliza uma ferramenta denominada "plataforma de gerenciamento de nuvem", como por exemplo, *Enomaly Elastic Computing Platform* (*Enomaly ECP*)[11], *Eucalyptus*[12], *OpenNebula*[13], *oVir*[14] e *Openstack*[15]; todas de código aberto (*open-source*) [20]. A camada de infraestrutura, também conhecida como camada de virtualização, e a de hardware pertencem, de modo geral, ao mesmo provedor de IaaS. Assim, o modelo de "infraestrutura como um serviço" (IaaS) refere-se ao fornecimento de utilidades computacionais sob demanda, geralmente em termos de máquinas virtuais. Através da infraestrutura virtual alugada, o usuário possui o total controle sobre esse ambiente, desde o sistema operacional até as execuções de suas aplicações; mas, não controla a infraestrutura física da nuvem em si. Por exemplo, não consegue gerenciar em qual hardware físico as suas máquinas virtuais serão executadas e nem impedir o compartilhamento do hardware físico com outros usuários [34,85,87]. Exemplos de provedores de

[7] http://www.linux-kvm.org/
[8] http://xen.org/
[9] http://www.vmware.com/
[10] http://www.citrix.com/
[11] http://www.enomaly.com/
[12] http://www.eucalyptus.com/
[13] http://opennebula.org/
[14] http://www.ovirt.org/
[15] http://openstack.org/

IaaS incluem *Amazon EC2*, *Citrix Systems*, *GoGrid*[16], *Rackspace Cloud*[17] e *Flexiscale*[18]. Também se encaixam na categoria de IaaS os serviços de armazenamento, cujos exemplos são: *Amazon S3*, *Amazon SimpleDB*[19] e *Dropbox*[20].

A camada de plataforma, situada acima da camada de infraestrutura, fornece um ambiente para desenvolver aplicações *web*, tais como: linguagens de programação, ferramentas de desenvolvimento e ambientes de execuções e testes. Isto é, o modelo de "plataforma com um serviço" (PaaS) provê serviços para que as aplicações *web* sejam totalmente implementadas, testadas e implantadas em nuvens pelos usuários programadores. Entretanto, o usuário programador não consegue gerenciar (ou customizar) a infraestrutura computacional subjacente (servidores, rede, armazenamento e sistemas operacionais, por exemplo) e, portanto, os recursos subjacentes são utilizados pelas aplicações de forma transparente através de interfaces de programação de aplicativos – *Application Programming Interfaces* (APIs). Assim, o usuário deve se concentrar apenas no desenvolvimento de suas aplicações e não nas peculiaridades de hardware. De fato, quando finalizadas, as aplicações *web* são geralmente disponibilizadas na camada acima (camada de aplicação) para serem utilizadas pelos usuários finais [34, 42, 85, 87]. Exemplos de provedores de PaaS incluem o *Google App Engine*[21], que oferece um ambiente de desenvolvimento de aplicações *Web* em *Java* ou em *Python*, e o *Windows Azure Platform*[22], onde aplicações podem ser desenvolvidas usando bibliotecas *.NET Framework* da *Microsoft*.

A camada de aplicação, situada no nível mais alto da hierarquia, oferece diversas aplicações *web* já criadas (serviços) para os usuários finais, por meio de dispositivos que possuem acesso à Internet. Assim, no modelo de SaaS, o usuário final não administra (e nem controla) a infraestrutura computacional subjacente e a plataforma da aplicação *web*, apenas algumas configurações específicas (e limitadas) são liberadas ao usuário. Por exemplo, se a aplicação *web* permitir, a interface gráfica pode ser customizada de acordo com o gosto do usuário. Uma das principais vantagens do SaaS é que novas funcionalidades podem ser adicionadas ao software de modo transparente, ou seja, o serviço pode ser atualizado sem que os usuários percebam essas modificações [34, 85, 87]. Exemplo de SaaS para uso corporativo é o serviço providenciados pelo *Salesforce.com*[23], que oferece ferramentas para análise de negócios e gestão de relacionamento com clientes – *Customer Relationship Management* (CRM). Outros exemplos para uso pessoal são os aplicati-

[16] `http://www.gogrid.com/`
[17] `http://www.rackspace.com/cloud/`
[18] `http://www.flexiscale.com/`
[19] `http://aws.amazon.com/simpledb/`
[20] `http://www.dropbox.com/`
[21] `http://code.google.com/appengine/`
[22] `http://www.windowsazure.com/en-us/`
[23] `http://www.salesforce.com/`

vos oferecidos pelo *Google Apps*[24], dos quais incluem o *Google Mail (Gmail)*[25], o *Google Calendar*[26] e o *Google Docs*[27].

2.1.4 Tipos de Nuvens

Adotar a computação em nuvem nas empresas ainda é uma alternativa duvidosa, pois há muitas questões que devem ser levadas em consideração, como, por exemplo, deslocar a execução dos aplicativos corporativos (e, consequentemente, os dados empresariais) para um ambiente público e compartilhado. Embora uma máquina virtual possa ter um ambiente de computação isolado, ela geralmente compartilha o hardware com outras máquinas virtuais. Aliás, a partir do momento em que os dados são colocados em nuvens públicas, é difícil descobrir, com precisão, a localização geográfica de onde eles vão estar. Além disso, uma empresa pode, por exemplo, estar interessada em minimizar custos monetários, enquanto outras em alta confiabilidade e segurança de suas informações. Portanto, de acordo com a disponibilidade dos serviços, a nuvem é classificada em quatro tipos diferentes [56, 71, 85]: pública, comunitária, privada e híbrida; as quais são descritas a seguir.

***Nuvem Pública*:** São nuvens onde os provedores oferecem recursos computacionais como serviços ao público em geral. Esses serviços podem ser tanto gratuitos quanto tarifados pelo modelo pago-pelo-uso. As principais desvantagens desse tipo de nuvem são: (i) a falta de informações sobre os detalhes de configuração dos serviços alugados, como, por exemplo, os provedores de IaaS geralmente não disponibilizam detalhes sobre o hardware físico onde as máquinas virtuais são executadas e também não oferencem informações sobre a largura de banda dos enlaces que conectam essas máquinas virtuais; (ii) o compartilhamento dos recursos entre vários usuários (ou organizações) da nuvem pública e (iii) os dados privados ficam, em teoria, visíveis ao provedor de serviço; como, por exemplo, os serviços de *webmails* disponibilizados na Internet. Além disso, por fornecer um controle pouco refinado dos serviços, as nuvens públicas dificultam a eficácia em vários cenários de negócios [85]. Por outro lado, (i) é fácil, barato e rápido iniciar um negócio na Internet com com o uso de nuvens públicas, pois os custos de manutenção de software, hardware e largura de banda são cobertos pelo provedor de serviços; (ii) os serviços têm alta escalabilidade, deixando a sensação que as nuvens públicas possuem "recursos infinitos"; e (iii) o usuário evita desperdícios de recursos (e dinheiro), pois os serviços são pagos pelo uso e não ficam subutilizados. Portanto, os provedores de nuvens

[24] `http://www.google.com/enterprise/apps/business/`
[25] `http://mail.google.com`
[26] `http://calendar.google.com`
[27] `http://docs.google.com`

públicas são responsáveis pela instalação, manutenção, gerenciamento e segurança dos serviços fornecidos, enquanto os usuários apenas utilizam esses serviços de acordo com a sua necessidade e pagam somente pelo uso. *Amazon EC2*, *Google App Engine* e *Google Apps for Business* são exemplos de nuvens públicas de provedores de IaaS, PaaS e SaaS, respectivamente.

***Nuvem Comunitária*:** São nuvens onde a infraestrutura é compartilhada por várias organizações (ou instituições) com interesses em comum, dos quais incluem: requisitos de segurança, políticas de uso, jurisdição, etc. [56]. Portanto, uma nuvem comunitária é constituída por um conjunto de usuários de diversas organizações que compartilham aplicações, serviços e recursos computacionais.

***Nuvem Privada*:** Também conhecida como *nuvem interna*, a nuvem privada é projetada para uso exclusivo de uma única organização, atendendo apenas um número limitado de usuários, como, por exemplo, as unidades de negócios dessa organização [56]. Dessa maneira, é possível ter um controle mais refinado dos recursos. Entretanto, alguns autores criticam esse tipo de nuvem, pois é muito semelhante a uma fazenda de servidores (do inglês, *server farm*[28]) ou uma grade privada [12, 85]. Em outras palavras, contraria a essência da computação em nuvem, porque o usuário tem que se preocupar com as peculiaridades do funcionamento dos serviços, tais como: aquisição, instalação, configuração e gerenciamento dos hardwares.

***Nuvem Híbrida*:** Uma nuvem híbrida é a combinação do uso da nuvem pública com a privada (ou comunitária), a fim de solucionar as limitações de cada abordagem. Em uma nuvem híbrida, as informações não-críticas podem ser executadas em nuvens públicas, enquanto as críticas em nuvens privadas (ou comunitárias). Portanto, esta abordagem oferece vantagens de *elasticidade* rápida e de baixo custo das nuvens públicas, e vantagens de segurança e controle mais refinado de desempenho dos serviços das nuvens não públicas.

2.2 Grafos Acíclicos Direcionados

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 categoria, por outro lado, representa os serviços que possuem

[28]Um conjunto de servidores interligados em rede, atuando como se fossem um único grande servidor

uma ordem para serem executados, pois há um fluxo de dados (comunicação entre serviços) bem-definido que deve ser respeitado. Nesta dissertação, 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 sinônimas no restante desta dissertação.

Resumindo, temos:

- $\mathcal{U}$ é o conjunto composto por $n = |\mathcal{U}|$ nós, onde cada nó $u_i \in \mathcal{U}$ representa um serviço atômico com um peso (custo de computação) $w_i \in \mathbb{R}^+$ associado e que deve ser executado em algum recurso computacional;

- $\mathcal{E}$ é o conjunto composto por $e = |\mathcal{E}|$ arcos (arestas direcionadas), onde cada arco $e_{i,j} \in \mathcal{E}$ representa uma dependência de dados entre $u_i \in \mathcal{U}$ e $u_j \in \mathcal{U}$ com um peso (custo de comunicação) $f_{i,j} \in \mathbb{R}^+$ associado. Assim, u_j não pode iniciar a sua execução antes que u_i finalize e envie as informações necessárias à u_j;

- $\mathcal{P}(u_i)$ é o conjunto de predecessores do nó $u_i \in \mathcal{U}$;

- $\mathcal{S}(u_i)$ é o conjunto de sucessores do nó $u_i \in \mathcal{U}$.

O primeiro nó do DAG é denominado *nó de entrada* e não possui nós predecessores, ou seja, $\mathcal{P}(u_{entrada}) = \varnothing$. De forma análoga, o último nó do DAG é chamado *nó de saída* e não tem nós sucessores, isto é, $\mathcal{S}(u_{saída}) = \varnothing$. Podemos assumir, sem perda de generalidade, que todo DAG possui um, e somente um, nó de entrada. Esse procedimento pode ser obtido da seguinte maneira: uma tarefa sem custo de computação $u_{entrada}$ é adicionado à $\mathcal{U}$, ou seja, $\mathcal{U} = \mathcal{U} \cup \{u_{entrada}\}$. Além disso, também são adicionados alguns arcos sem peso ao conjunto $\mathcal{E}$, interligando o novo nó $\{u_{entrada}\}$ a todos os nós do DAG sem predecessores, isto é, $\mathcal{E} = \mathcal{E} \cup \{e_{entrada,i} \; \forall \; u_i \in \mathcal{U} \mid \mathcal{P}(u_i) = \varnothing\}$. Um nó de saída único é criado de modo análogo.

O DAG da Figura 2.2 representa um aplicativo de processamento do filtro de mediana de imagens [16]. 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 somente podem iniciar suas execuções 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, esta dissertação 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 *todas* os serviços 2-13.

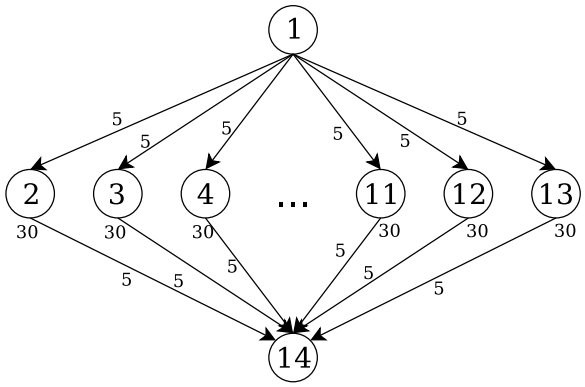


Figura 2.2: Exemplo de um *workflow* representado por um DAG com 14 nós [16].

Workflow como um DAG é um paradigma amplamente utilizado para representar processos científicos complexos que geralmente exigem alta demanda de processamento e armazenamento de dados [10, 26, 91]. Áreas da física de alta energia, física de ondas gravitacionais, geofísica, astronomia, bioinformática e ciência da computação são exemplos de comunidades científicas que utilizam esse paradigma. Em astronomia, por exemplo, os cientistas estão usando *workflows* para gerar mosaicos do céu [27], para examinar a estrutura de galáxias [68], e, em geral, para compreender a estrutura do universo. Aliás, grades computacionais são ambientes ideais para executar esses complexos *workflows científicos* em larga escala, pois oferecem vários recursos que podem computar grande volume de dados de forma rápida. *Montage* [27], *CSTEM* [28], *AIRSN* [90], *LIGO-1* [63] e *Chimera* [4] são exemplos de algumas aplicações de e-Ciência do mundo real que precisam de vários recursos interligados via rede (grades e nuvens, por exemplo) para serem solucionadas de forma rápida. Esses *workflows* estão ilustrados na Figura 2.3

Por outro lado, por ser um ambiente compartilhado, a grade, em um determinado momento, pode estar totalmente (ou parcialmente) ocupada e, com isso, os recursos disponíveis podem não ser suficientes para executar os *workflows* científicos adequadamente (execução com restrição de *deadline*[29], por exemplo). Diferentemente da computação em nuvem, o número de recursos computacionais é limitado, ou seja, os recursos da grade não são *elásticos* [52]. Portanto, essa demanda computacional pode ser suprida alugando recursos virtuais dedicados da nuvem, de acordo com com o tamanho do *workflow*. Logo, a execução de *workflow* pode ser migrada totalmente [38] (ou parcialmente [11]) da grade para a nuvem. Esta pesquisa considera somente a execução de *workflows* em nuvens.

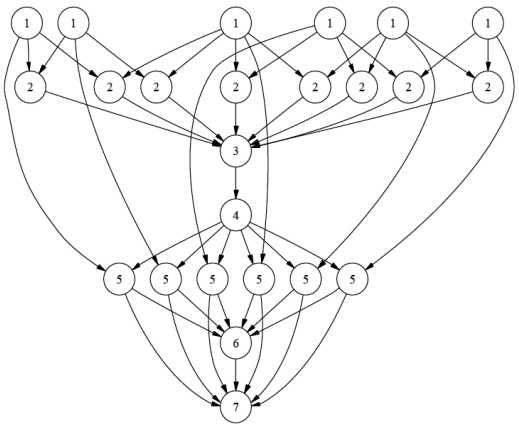
2.3 O Problema de Escalonamento

O problema de escalonamento não é novo e começou a se tornar importante no início do século XX [13]. Inicialmente usado na produção industrial, o estudo do escalonamento evoluiu com o passar do tempo e hoje é utilizado em ambientes computacionais. Pinedo em [62] definiu o termo escalonamento da seguinte maneira: *escalonamento é um processo de tomadas de decisão responsável pela alocação de recursos para execução de um conjunto de tarefas, com intuito de otimizar um ou mais objetivos.* Conhecido por ser um problema difícil de ser modelado (do ponto de vista matemático) e resolvido de modo eficiente (em termos de complexidade computacional), o escalonamento é um problema que pode ser adaptado em diversas áreas. Por exemplo, os recursos podem ser representados por pistas de um aeroporto, conjunto de operários em uma obra e unidades de processamento de um computador. As tarefas podem ser representadas por pousos e decolagens do aeroporto, etapas de um projeto de construção e execuções de programas em um computador, respectivamente.

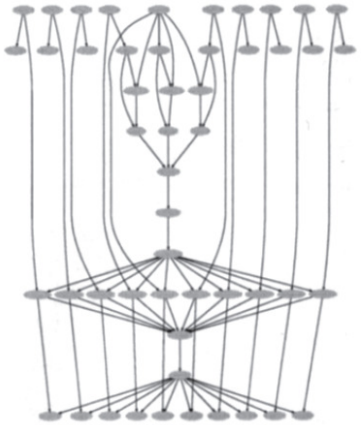
Neste trabalho, a palavra *tarefa* representa uma tarefa computacional que precisa de um serviço (geralmente *web service*) para ser executada. Desta forma, se todas as tarefas de um *workflow* de tarefas precisam de um serviço para serem executadas, então o *workflow* é denominado *workflow* de serviços, o qual iremos focar daqui em diante. Portanto, as palavras *tarefa* e *serviço*, neste capítulo, serão utilizadas como sinônimas.

Para que um conjunto de serviços (*workflow*) seja executado de modo eficiente, é necessário, então, considerar um *escalonador de serviços*. Semelhante à classificação dos serviços, o algoritmo de escalonamento também é classificado em dois tipos [13, 15]: escalonadores de serviços independentes e escalonadores de serviços dependentes. Como no primeiro tipo não há nenhuma comunicação entre serviços, cada serviço pode ser escalonado independentemente dos outros. No segundo tipo, por outro lado, o escalonamento

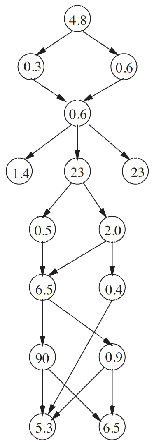
[29]Tempo de resposta



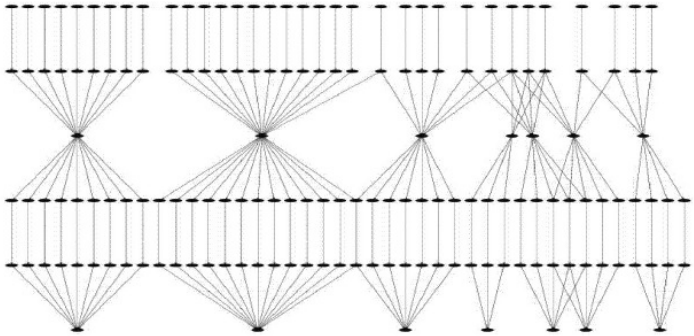
(a) Montage com 24 nós e 50 dependências



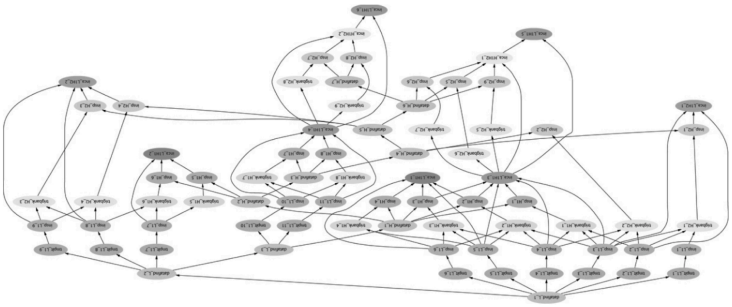
(b) AIRSN com 50 nós e 78 dependências



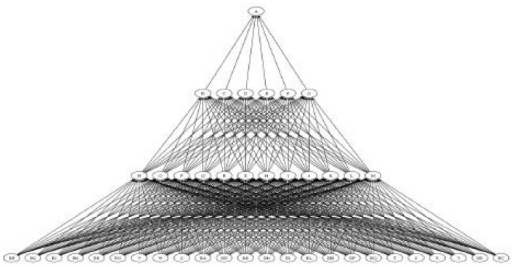
(c) CSTEM com 15 nós e 19 dependências



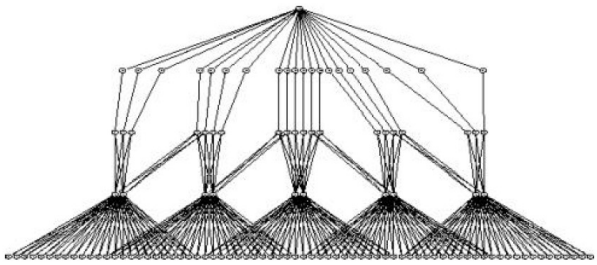
(d) LIGO-1 com 168 nós e 215 dependências



(e) LIGO-2 com 75 nós e 132 dependências



(f) Chimera-1 com 43 nós e 336 dependências



(g) Chimera-2 com 124 nós e 336 dependências

Figura 2.3: Exemplos de DAGs–*worfklows* científicos [16]

é mais complexo, pois o algoritmo deve levar em consideração a comunicação (troca de dados) entre serviços. Esta dissertação irá focar no algoritmo do segundo tipo.

Tendo estruturado o ambiente de execução, as restrições e os objetivos, podemos definir o problema de escalonamento a ser abordado, que é descrito pela tripla $\alpha \mid \beta \mid \gamma$ [13, 62]. Primeiramente, o campo α define o ambiente de processamento e contém apenas uma entrada. Na sequência, o campo β descreve as características de processamento e restrições, podendo conter nenhuma, uma ou múltiplas entradas. Para finalizar, o campo γ delineia o objetivo a ser minimizado e muitas vezes contém uma única entrada. Assim, o problema de escalonamento focado nesta dissertação considera um ambiente de processamento composto por m máquinas virtuais não-relacionadas em paralelo ($\alpha = \mathcal{R}_m$), com restrições de precedências entre serviços do *workflow* ($\beta = prec$) e o objetivo principal a ser minimizado é o *custo monetário total* da execução de cada nó $u \in \mathcal{U}$ do *workflow*, ou seja, $\gamma = \min\limits_{\forall\, u \in \mathcal{U}, \forall\, j \in \mathcal{R}_m} \left(\sum \left[p_{u,j} \times c_j \right] \right)$, onde $p_{u,j}$ é o tempo de processamento do serviço u no recurso j e c_j é o custo monetário para utilizar o recurso j, o qual é cobrado por unidade de tempo utilizado. Portanto, nosso problema de escalonamento é resumidamente modelado pela tripla:

$$\mathcal{R}_m \left| prec \right| \min_{\forall\, u \in \mathcal{U}, \forall\, j \in \mathcal{R}_m} \left(\sum \left[p_{u,j} \times c_j \right] \right)$$

O escalonador é o elemento responsável em determinar qual parte do *workflow* será executado 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 irá executar. Ordens de precedências 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*[30]) do *workflow* [14, 73], maximizar o uso de recursos computacionais privados (e minimizar a terceirização de recursos públicos) [93], minimizar o tempo de execução do escalonamento [66] ou minimizar custos monetários da execução do *workflow* [12, 38]. Existem também alguns trabalhos que consideram escalonadores multicritérios, ou seja, com múltiplas funções objetivos a serem otimizadas ao mesmo tempo [17, 43, 69, 75]. Aliás, algoritmos de escalonamento de serviços dependentes multicritério são ainda mais complexos e desafiadores de serem desenvolvidos, pois é necessário

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

encontrar o conjunto *Pareto*[31] de soluções. Enfim, para solucionar, de modo geral, um problema de escalonamento, é necessário determinar a *sequência ótima* de execução dos serviços nos recursos, em relação à função objetivo a ser otimizada.

Informações sobre o estado dos recursos são fundamentais para atingir bom desempenho no escalonamento. Assim, dependendo da maneira como essas informações são obtidas, os algoritmos de escalonamento podem ser classificados em dois tipos [13, 15]: *estático* ou *dinâmico*. De um lado, escalonadores estáticos referem-se ao escalonamento de todos os serviços utilizando informações disponíveis no momento do início do escalonamento. Por outro lado, escalonadores dinâmicos referem-se ao escalonamento dos serviços usando informações obtidas em tempo real, ou seja, como os serviços são escalonados em turnos, a cada turno essas informações são atualizadas. É importante frisar que escalonadores estáticos são, de forma geral, o ponto de partida para o desenvolvimento de escalonadores dinâmicos eficazes [13].

O problema de escalonamento é, de forma geral, NP-Completo e o problema de otimização associado é NP-Difícil [30, 62]. Isto é, não conhecemos nenhum algoritmo que gere soluções determinísticas ótimas em tempo polinomial[32]. Portanto, algoritmos ótimos para realizar escalonamento de grandes instâncias de um problema levam muito tempo para serem executados, tornando, assim, os algoritmos de aproximação [35, 89], as heurísticas [14, 16, 17, 24, 29], as meta-heurísticas [60, 80] e a programação linear [38, 50, 70] boas opções para solucionar o problema de escalonamento. Essas técnicas permitem desenvolver algoritmos de complexidade de tempo polinomial que, geralmente, produzem resultados aproximados da solução ótima.

2.4 Acordo de Nível de Serviço

Atualmente a Internet é baseada na característica do melhor esforço (*Best Effort*), ou seja, não há garantias de nível de serviço. Entretanto, os usuários geralmente precisam utilizar os serviços disponibilizados na nuvem com uma certa garantia de qualidade de serviço (QoS), a qual é especificada em um contrato conhecido como acordo de nível de serviço (SLA). Em outras palavras, SLA é um documento acordado entre duas ou mais entidades onde a descrição, a entrega e a cobrança dos serviços contratados são definidos formalmente [1, 53].

SLA inclui informações sobre a definição dos serviços contratados, tais como: de-

[31] Conjunto composto por todas as soluções viáveis para o problema de escalonamento, onde para cada solução têm pelo menos um objetivo otimizado, mantendo os demais objetivos constantes.

[32] Um algoritmo é dito ter complexidade de tempo polinomial se o seu tempo de execução no pior caso é limitado superiormente por uma expressão polinomial dos tamanhos das instâncias (entradas) do problema [65].

sempenho, gerenciamento de problemas, responsabilidade das partes, garantias, medidas emergenciais, planos alternativos, planos para soluções temporárias, relatórios de monitoramento, segurança, confidencialidade e cancelamento do contrato. Além dessas informações, SLA pode incluir especificações, como por exemplo: disponibilidade, incidência de erros e prioridades de serviço. Portanto, na prática, SLA é um documento que serve como base para garantir que ambas as partes (provedor de serviço e cliente, por exemplo) utilizarão os mesmos parâmetros para avaliar as QoSs definidas nesse contrato [2].

O provedor de SaaS pode alugar instâncias de máquinas virtuais do provedor de IaaS por meio de dois tipos de 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 utilizadas, normalmente por hora. 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. O provedor de SaaS, porém, receberá um desconto significativo sobre a tarifa das unidades de tempo utilizadas de cada instância reservada. Ambos os tipos de SLAs adotados neste trabalho são semelhantes aos tipos usados na *Amazon EC2*[33].

Portanto, do ponto de vista do cliente, SLA é um contrato que define formalmente, em termos mensuráveis, os tipos de serviços contratados e as penalidades, caso haja a quebra do contrato; isto é, ações que o provedor de serviço deverá cumprir se o serviço prestado não estiver de acordo com os parâmetros estabelecidos. De fato, é através do SLA que o cliente irá obter uma garantia de QoS dos recursos contratados. Por outro lado, do ponto de vista do prestador de serviço, SLA permite gerenciar os recursos (serviços) alugados de maneira eficaz, correta e apropriada, satisfazendo os clientes e ampliando a "fatia de mercado" (do inglês, *market-share*).

2.5 Cenário do Ambiente de Escalonamento de *Workflows* em Nuvens

Para providenciar o escalonamento de *workflows* proposto, desenvolvemos o cenário ilustrado na Figura 2.4. As principais entidades deste cenário são: clientes, provedor de SaaS e provedores de 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 existir, recursos próprios para executar esses *workflows*, de tal modo que deve obedecer os requisitos de QoS de cada *workflow* submetido. Entretanto, em picos de demanda, a infraestrutura própria pode ser insuficiente para atender todos clientes ao mesmo tempo. Assim, para garantir os requisitos de QoS das solicitações já aceitas e também não re-

[33] `http://aws.amazon.com/ec2-sla/`

cusar nenhum pedido de futuros clientes, o provedor de SaaS precisa aumentar o 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, vamos considerar 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).

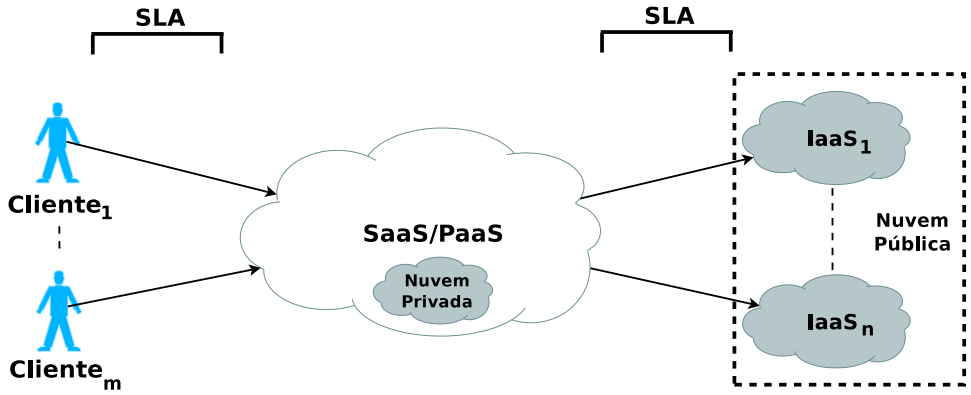


Figura 2.4: Cenário do escalonamento de *workflows* em nuvens

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 2.4, o primeiro nível contém todos os SLAs estabelecidos entre o provedor de SaaS e 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*, conforme descrito na seção anterior. 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 ambiente computacional. Em outras palavras, esses provedores de serviços não precisam, em picos de demanda, recusar pedidos de clientes quando a infraestrutura privada (ou comunitária) estiver totalmente ocupada ou for insuficiente.

Um aspecto que não será tratado neste cenário é o problema de aprisionamento em nuvens públicas (do inglês, *cloud lock-in*). Esse problema consiste na falta de portabilidade de dados (e aplicações) do usuário entre nuvens [5]. Em outras palavras, como cada

provedor de nuvem pública geralmente implementa, por exemplo, seu próprio conjunto de interface de programação de aplicativos (API), o usuário acaba ficando aprisionado na nuvem que contratou. Isto é, as APIs de cada provedor de nuvem são proprietárias e não padronizadas. Por exemplo, na *Force.com*, as aplicações devem ser desenvolvidas na linguagem proprietária do *Salesforce*, denominada *Apex*[34], que não está disponível em outros provedores de nuvens. Dessa forma, para solucionar esse problema de aprisionamento em nuvens, existem hoje algumas propostas para padronização de APIs entre nuvens, denominadas *Open-APIs*. Exemplos de propostas incluem: *Deltacloud*[35], *Zend SimpleCloud*[36] e *Open Cloud Computing Interface*[37] da *Open Grid Forum* (OPF). Essas APIs criam basicamente uma camada de abstração entre a aplicação do usuário e o provedor de nuvem, permitindo que o usuário não tenha que se preocupar com APIs proprietárias. Portanto, vamos assumir neste cenário, que o provedor de SaaS e os provedores de IaaS disponibilizam interfaces baseadas em *Open-APIs*.

2.6 Considerações Finais

Neste capítulo revisamos os conceitos básicos necessários para a compreensão deste trabalho. A computação em nuvem foi apresentada em detalhes, cobrindo tanto a definição quanto as características essencias; ambos conceitos foram publicados recentemente (setembro de 2011) pelo NIST em [56]. Em seguida, detalhamos as principais características de um *workflow*, representado por um grafo acíclico direcionado (DAG), cujo escalonamento é um problema NP-Completo. Para finalizar este capítulo, descrevemos sobre o acordo de nível de serviço(SLA) e o cenário de escalonamento de *workflows* em nuvens com dois níveis de SLAs proposto nesta dissertação.

Como pôde ser visto, a computação em nuvem oferece a *elasticidade* computacional de uma maneira rápida, simples e de baixo custo monetário. Neste contexto, apresentamos uma solução para escalonar *workflows* de serviços em máquinas virtuais alugadas de vários provedores de IaaS, mostrando ser uma solução promissora em situações onde investimentos em infraestrutura própria está fora de questão. Entretanto, devido à existência de vários tipos de máquinas virtuais para serem alugadas e consequentemente cada uma com um preço associado, temos que analisar qual o melhor conjunto de VMs que satisfaça os requisitos de QoS das execuções de *workflows* e que gere o menor custo monetário possível, conforme modelado anteriormente na tripla $\alpha \mid \beta \mid \gamma$.

O próximo capítulo apresenta uma ampla revisão bibliográfica relacionada ao pro-

[34] `http://wiki.developerforce.com/page/Apex`
[35] `http://deltacloud.apache.org/`
[36] `http://www.simplecloud.org/`
[37] `http://occi-wg.org/`

blema de escalonamento. Iniciaremos com as descrições de escalonadores desenvolvidos inicialmente para grades computacionais e, em seguida, apresentaremos a expansão (*elasticidade*) do ambiente de computação (recursos) utilizados pelos escalonadores através da computação em nuvem.

Capítulo 3

Trabalhos Relacionados

O principal problema abordado nesta dissertação é o escalonamento de *workflows* (ou serviços dependentes) representados por grafos acíclicos direcionados (DAGs). Devido à NP-Completude desse problema [62], heurísticas, algoritmos de aproximação e análise combinatória são consideradas boas estratégias para desenvolver algoritmos de escalonamento. Entretanto, a abordagem de programação linear tornou-se um dos métodos mais explorados para os problemas de escalonamento, devido à sua rigorosidade, flexibilidade e capacidade de ampla modelagem [33]. Embora o escalonamento de tarefas (ou serviços) seja amplamente estudado em grades computacionais [14,15,17,67,81], esse problema está, hoje em dia, migrando paulatinamente para as nuvens [11, 12, 16, 38, 43, 59, 61, 70, 89, 93], graças a principal característica de *elasticidade*.

Este capítulo contém uma revisão bibliográfica detalhada dos trabalhos específicos em escalonamento abordados nesta pesquisa. A Seção 3.1 apresenta diversos algoritmos de escalonamento de *workflows* existentes na literatura, e as principais vantagens e desvantagens de cada um deles são discutidas. Uma tabela resumo com as principais características desses algoritmos é apresentada na Seção 3.2, incluindo também informações a respeito do escalonador proposto nesta dissertação. Para finalizar este capítulo, a Seção 3.3 destaca algumas considerações sobre esta revisão bibliográfica.

3.1 Algoritmos de Escalonamento

O problema de escalonamento, de modo geral, é NP-Completo [30, 62], ou seja, à medida que o tamanho da entrada aumenta, o tempo de resolução (computação), no pior caso, cresce exponencialmente. Além disso, devido às decisões discretas envolvidas, tais como, alocação de recursos e distribuição de tarefas nesses recursos ao longo do tempo,

esse problema é inerentemente combinatório[1] por natureza. Isto é, existe uma quantidade muito grande de combinações de soluções entre tarefas e recursos, o que torna a busca exaustiva[2] pela *solução ótima* um método computacionalmente caro. Assim, como comentado anteriormente na Seção 2.3, heurísticas, meta-heurísticas, programação linear e algoritmos de aproximação são algumas técnicas utilizadas na tentativa de aproximar a solução ótima com algoritmos de complexidade de tempo polinomial. Nesta seção, vamos apresentar os escalonadores existentes na literatura de acordo com a infraestrutura computacional utilizada. A Seção 3.1.1 apresenta alguns escalonadores aplicados somente em grades computacionais, enquanto a Seção 3.1.2 descreve alguns escalonadores que utilizam nuvens híbridas – união da nuvem privada com a pública. Por último, a Seção 3.1.3 descreve alguns escalonamentos aplicados somente em nuvens.

3.1.1 Escalonamento em Grades Computacionais

Uma grade computacional é um sistema heterogêneo, dinâmico e amplamente distribuído, onde o problema de escalonamento é amplamente estudado. Embora escalonamento em grades não seja o foco deste trabalho, vamos apresentar, nesta seção, alguns escalonadores destinados somente à grade, a fim de enfatizar os trabalhos cujos princípios estão relacionados com esta dissertação.

Yao *et al.* propõem em [81] 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 Programa Linear Inteiro (PLI) e 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. Além disso, afirmam que todo *workflow* pode ser reduzido a um *workflow* sequencial equivalente, por meio do agrupamento dos nós paralelos em um único nó. De modo geral, o uso do *wokflow* sequencial facilita e reduz o tempo de execução do escalonamento; no entanto, diminui a qualidade da solução.

Yu e Buyya apresentam em [83] um estudo para escalonar *workflows* científicos em grades computacionais, usando a técnica de algoritmo genético (meta-heurística). O escalonador considera duas restrições de QoS aos usuários: *deadline* e orçamento (dinheiro).

[1] Ramo da matemática que estuda coleções finitas de objetos.

[2] *Busca exaustiva* ou *busca por força bruta* consiste em enumerar todos os possíveis candidatos da solução e checar cada candidato para saber se satisfaz o enunciado do problema.

No primeiro caso, o escalonador minimiza o *makespan*, enquanto no segundo caso, minimiza o custo monetário da execução do *workflow*. No entanto, durante a fase de seleção de recursos, o algoritmo de escalonamento não considera recursos com múltiplos núcleos de processamento, ao contrário das máquinas virtuais multiprocessadas disponíveis na computação em nuvem. Aliás, por utilizar algoritmo genético, o tempo de execução do escalonador em nuvens pode convergir para valores altos. Yu *et al.* propõem em [84] um algoritmo de escalonamento de *workflows* em grade computacional orientado a serviço, também chamada de grade utilitária. Nessa grade os recursos (serviços) são tarifados através do modelo *pay-per-use*. O algoritmo de escalonamento apresentado por Yu *et al.* é uma heurística que minimiza o custo monetário da execução do *workflow*, enquanto obedece o *deadline* definido no SLA entre o provedor de serviço e o usuário. Entretanto, a terceirização de recursos pelo provedor da grade utilitária não é considerada pelos autores.

Chaves *et al.* descreveram em [7, 8, 22] um programa linear inteiro para escalonar tarefas e máquinas virtuais. O *Task and Virtual Machine* (TVM) escalona, primeiramente, as máquinas virtuais nos recursos da grade computacional e, em seguida, as tarefas nessas máquinas. Em outras palavras, os autores apresentam um cenário onde o usuário acessa um *middleware* para submeter aplicação *workflow* que deseja executar em um ambiente customizado. Dessa forma, o TVM mapeia as tarefas do *workflow* nos recursos computacionais disponíveis na grade e, também, escalona a instanciação das máquinas virtuais customizadas (com requisitos de software do *workflow*) a partir do repositório de máquinas virtuais, de tal modo que, a aplicação *workflow* possa ser executada em um ambiente apropriado. A função objetivo do PLI é minimizar o *makespan* do *workflow*, com intuito dos resultados serem entregues o mais rápido possível ao usuário. Dando continuidade a esse trabalho, Chaves *et al.* substituíram a grade computacional pela computação em nuvem em [21]. Entretanto, os autores não consideram noções de SLA em nuvens e também não minimizam custos monetários com a terceirização de recursos computacionais.

3.1.2 Escalonamento em Nuvens Híbridas

Ambas as tecnologias – grade computacional e computação em nuvem – não oferecem um suporte adequado para execução de *workflows*, deixando ao usuário a responsabilidade de preparar o ambiente computacional para esse tipo de execução. Alguns autores apresentam abordagens para deslocar a execução de *workflows* da grade computacional para a computação em nuvem, pois a grade não possui a característica de *elasticidade* rápida presente na nuvem [52]. Um exemplo desse fato é quando o número de serviços em paralelo de um *workflow* se sobrepõe ao número de recursos disponíveis na grade ou quando a grade está totalmente sobrecarregada.

Hoffa *et al.* relatam em [41] *tradeoffs* entre executar *workflows* científicos em grades ou

em nuvens. Com intuito de avaliar a aplicabilidade da computação em nuvem, diferentes tamanhos do *workflow* científico *Montage* [27], que normalmente são executados em grade computacional, foram totalmente executados em nuvens. Os autores realizaram experimentos e concluíram que, em alguns casos, o poder computacional dos ambientes locais ou das grades são suficientes para executar os *workflows* científicos; porém, essa solução não é escalável. Em vista disso, quando os recursos privados/grades são insuficientes, os ambientes virtuais providos pela nuvem podem fornecer a escalabilidade necessária para a execução dos *workflows*. No entanto, o escalonamento pode sofrer pequenos atrasos gerados pela comunicação remota com ambiente virtual. Embora esse trabalho apresente vantagens e desvantagens do uso da computação em nuvem para executar *workflows* científicos, relatos sobre os custos monetários das execuções em nuvens e o número de provedores de nuvens utilizados não foram mencionados pelos autores.

Zinnen e Engel propõem em [93] estratégias para escalonar dinamicamente tarefas independentes em nuvens híbridas. Os autores estimam o tempo de execução das tarefas, representados como variáveis aleatórias, através do método *Heteroscedastic Gaussian Processes* (HGP). Bossche *et al.* apresentam em [70] uma programação linear inteira binária para escalonar tarefas independentes em nuvens híbridas, obedecendo restrições de *deadline*. Embora ambos os trabalhos não considerem execução de *workflows*, eles têm o objetivo de maximizar o uso dos recursos privados, enquanto minimizam custos monetários com terceirização de infraestrutura (nuvens públicas).

Bittencourt *et al.* apresentam em [11] uma infraestrutura composta pela união da grade com a nuvem para executar *workflow* de serviços. Assim, quando um *workflow* de serviço é executado nesse sistema híbrido, o escalonador verifica se a grade é capaz de atender a todos os serviços do *workflow*; caso contrário, solicita recursos computacionais das nuvens públicas para suprir a falta de recursos da grade. Embora os autores preparem um ambiente computacional híbrido para executar *workflows* de serviços, avaliações sobre minimização de custos monetários não são realizados. Por outro lado, Bittencourt *et al.* propõem em [12] uma estratégia (heurística) para escalonar *workflow* de serviços em nuvens híbridas. Essa estratégia determina quando e como solicitar novos recursos da nuvem pública para satisfazer *deadlines* ou obter um tempo de execução (*makespan*) razoável, enquanto minimiza os custos monetários envolvidos. Em outras palavras, mostra como maximizar o uso da infraestrutura privada (computadores locais ou grades, por exemplo), enquanto minimiza o uso da infraestrutura terceirizada (nuvens públicas). Os autores realizaram experimentos apenas com o DAG *fork-join* (ver Figura 2.2) e concluíram que, ao providenciar *elasticidade* à nuvem privada, foi possível diminuir consideravelmente o *makespan* do *workflow*, com pouco investimento em recursos terceirizados.

Uma outra estratégia (heurística) para escalonar *workflows* de serviços em nuvens híbridas foi proposta por Bittencourt e Madeira em [16]. É apresentado um algoritmo,

denominado *Hybrid Cloud Optimized Cost* (HCOC), que decide quais recursos devem ser alugados da nuvem pública para aumentar o poder de processamento da nuvem privada. Esse trabalho é um avanço do anterior ([12]), pois são realizados experimentos e simulações com oito aplicações *workflows* do mundo real: AIRSN, *Chimera-1*, *Chimera-2*, CSTEM, LIGO-1, LIGO-2, *fork-join* e *Montage*. Embora esses trabalhos apresentem abordagens para minimizar o custo monetário da execução de *workflows* em nuvens híbridas, enquanto satisfazem restrições de *deadlines*, os autores não consideram a noção de SLA nem dos clientes e nem do provedor de serviço, um aspecto muito importante no modelo de negócio proposto pelo paradigma da nuvem.

Enfim, a computação em nuvem vêm ganhando espaço em ambientes onde a elasticidade computacional é rígida (e lenta). Portanto, de uma maneira rápida e barata, é possível aumentar o poder computacional do provedor de serviços para satisfazer as exigências de seus clientes, como por exemplo, obedecer restrições de *deadlines* na execução de *workflows*. Aliás, dependendo da situação dos recursos privados (ou da grade), os *workflows* podem ser executados integralmente na nuvem, conforme mostram os trabalhos descritos na próxima seção.

3.1.3 Escalonamento em Computação em Nuvem

Os benefícios da computação em nuvem incluem custos operacionais mais baixos, computação sob demanda (*elasticidade*) e prevenção de investimentos iniciais. Essas características permitem melhorar a capacidade computacional disponível localmente, fornecendo novos recursos de computação quando necessário. Por exemplo, se os recursos privados estiverem totalmente ocupados, o provedor de serviço poderá socorrer à nuvem, alugar máquinas virtuais e atender solicitações de futuros clientes. Alguns trabalhos presentes na literatura propõem soluções para a execução de serviços independentes na computação em nuvem, mas apenas poucos consideram a relação custo-benefício do uso de recursos virtualizados (máquinas virtuais), alugados a partir de várias nuvens públicas, para oferecer serviços de execução de *workflows* de serviços focados em dois níveis de SLAs.

Li e Guo descrevem em [50] um programa linear inteiro (PLI) estocástico para escalonar recursos em computação em nuvem. É apresentada uma maneira de selecionar os recursos das nuvens públicas para executar serviços abstratos de instâncias de processos de negócio, enquanto satisfaz os custos monetários definidos nos SLAs. Para solucionar o programa estocático inteiro é aplicado a teoria da base de *Gröbner*. Wu *et al.* descrevem em [77] 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. Os serviços independentes considerados pelos autores são aplicações corporativas e não científicas. Embora esses trabalhos apresentem avanços no campo de escalonamento estático de tarefas (ou serviços) independentes em nuvens, nenhum deles considera execução de *workflows*.

Zhao *et al.* apresentam em [89] um escalonador dinâmico de tarefas independentes para computação em nuvem. Foi proposta uma heurística, baseada em algoritmo genético, no intuito das tarefas se adaptarem a um ambiente computacional heterogêneo, com diferentes configurações de computação (CPUs) e memória. Algoritmo genético, porém, requer geralmente um tempo de execução longo, o qual aumenta a probabilidade de violação de SLA nos ambientes de computação em nuvem, onde os clientes precisam ser atendidos imediatamente. No entanto, não há nenhuma menção sobre minimização de custos monetários da execução das tarefas independentes. Reig *et al.* propõem em [64] uma estratégia para ajudar os provedores de serviço (SaaS ou PaaS) a lidar com a heterogeneidade de recursos disponíveis na nuvem pública. Os provedores de serviços dedicam, de modo geral, parte de seus recursos privados para executar suas aplicações próprias, enquanto executam tarefas de seus clientes nos recursos alugados da nuvem. Dessa forma, é apresentado um sistema de previsão que determina o mínimo de recursos necessários para serem alugados pelo provedor de serviço, a fim de executar as tarefas de seus clientes antes de um determinado *deadline*; evitando, portanto, violações de SLAs. Esse sistema de previsão também ajuda os provedores de serviço a utilizarem os recursos de uma maneira eficiente, como por exemplo, descartar com antecedência as tarefas que não vão conseguir cumprir seus *deadlines*. Ao usar o sistema de previsão, é possível, então, evitar desperdícios de recursos (e dinheiro). Esses trabalhos também não consideram execução de *workflows*.

Pandey *et al.* descrevem em [60] uma otimização por enxame de partículas – *Particle Swarm Optimization* (PSO) – para escalonar estaticamente *workflows* de tarefas em nuvens, a fim de minimizar o custo monetário total da execução do *workflow*. O escalonador leva em consideração tanto os custos de computação das tarefas quanto os custos de transmissão de dados entre elas. Ao utilizar a técnica de PSO, os autores conseguiram reduzir os custos monetários em até 3 vezes, quando comparado aos resultados obtidos pelo algoritmo de "seleção do melhor recurso" – *Best Resource Selection* (BRS). Por outro lado, Wu *et al.* também propõem em [80] um algoritmo de escalonamento estático de *workflows* de tarefas para computação em nuvem. O algoritmo é desenvolvido através de uma versão melhorada do PSO, conhecido como otimização por enxame de partículas discreto e revisado – *Revised Discrete Particle Swarm Optimization* (RDPSO). Além de minimizar o tempo de execução (*makespan*), os autores também minimizam o custo monetário da execução do *workflow*. Para realizar o escalonamento, a formulação também leva em consideração os custos de computação das tarefas e os custos de transmissão de dados entre elas. Enfim, ao utilizar a técnica de RDPSO para escalonar aplicações *workfows* de

larga escala em nuvens, os autores conseguiram melhores reduções nos custos monetários, quando comparado aos escalonamentos providos pelos algoritmos PSO e BRS. É importante frisar que ambos os métodos de otimização – PSO e RDPSO – são classificados como meta-heurística. Um algoritmo de escalonamento estático de *workflows* de serviços com múltiplos critérios para a nuvem é proposto por Juhnke *et al.* em [43]. É apresentado um escalonador formulado através de algoritmos genéticos e baseado na linguagem *Business Process Execution Language* (BPEL). Os autores desejam minimizar não só o *makespan*, mas também o custo monetário da execução do *workflow*. Embora esses trabalhos apresentem avanços no campo de escalonamento de *workflows* em nuvens, minimizando custos monetários, nenhum deles considera a noção de SLA em ambos lados do serviço contratado (do cliente e do provedor), que é um aspecto muito importante no modelo de negócio proposto pelo paradigma da nuvem.

Mohammadi *et al.* propõem em [32] um algoritmo de escalonamento bi-critério de *workflows* de tarefas para ambientes distribuídos com fins comerciais, como a computação em nuvem, por exemplo. Baseado na teoria dos jogos, esse algoritmo minimiza tanto o *makespan*, quanto o custo monetário da execução do *workflow*. Simulações mostraram que o escalonamento obtido é aproximadamente ótimo de *Pareto*. Os autores consideram que o provedor de serviço (SaaS) tem apenas seus recursos computacionais próprios e, além disso, não consideram a terceirização de recursos com os provedores de infraestrutura (IaaS). Aliás, noções de SLA também não são mencionados pelos autores.

Na próxima seção vamos apresentar uma tabela resumo comparando os trabalhos anteriormente descritos, incluindo os aspectos do escalonador proposto nesta dissertação.

3.2 Tabela Resumo

A Tabela 3.1 apresenta um resumo das principais características dos algoritmos de escalonamento em computação em nuvem expostos na seção anterior. Essa tabela também inclui, na última linha, as informações acerca do escalonador proposto neste trabalho.

Devido à NP-Completude do problema de escalonamento, a coluna *técnica* refere-se à técnica contida no escalonador para tentar aproximar à solução ótima com algoritmos de complexidade de tempo polinomial. Heurísticas e meta-heurísticas são as abordagens mais frequentemente encontradas na literatura de escalonamento de tarefas (ou serviços), pois possuem baixa complexidade e execução rápida. Entretanto, fornecem soluções sem um limite formal de qualidade, tipicamente avaliado empiricamente em termos da qualidade das soluções. Por outro lado, a abordagem de programação linear, se possível, garante uma solução ótima para o problema de escalonamento. Porém, o tempo de execução depende, de modo geral, da quantidade de variáveis de decisão, do tamanho da entrada e das restrições do problema. Quando todas variáveis do programa linear assumem somente

valores discretos, o problema é dito linear inteiro (PLI), mas se algumas forem inteiras e as restantes contínuas, é denominado Programa Inteiro Misto (PIM). Devido a alguns parâmetros discretos envolvidos, muitos problemas de escalonamento podem ser formulados como um PLI (ou PIM), cuja solução é geralmente encontrada através da relaxação das variáveis inteiras do problema.

A coluna *classificação* faz referência às duas classes de escalonadores descritos no capítulo anterior: estático ou dinâmico. Enquanto escalonadores estáticos promovem o escalonamento de todas as tarefas/serviços usando informações disponíveis no momento do início do escalonamento, escalonadores dinâmicos escalonam as tarefas em turnos, atualizando as informações sobre os recursos a cada turno de escalonamento. A maioria desses escalonadores são estáticos. Mesmo não utilizando informações sobre os recursos em tempo real, estes são as soluções mais presentes na literatura, já que escalonamento dinâmico em nuvens ainda impõe desafios consideráveis. Aliás, um bom algoritmo de escalonamento estático é primordial para obter escalonamentos dinâmicos eficientes [13].

Para escalonar *workflows* em nuvens, a fim de *minimizar custos monetários*, o algoritmo de escalonamento pode considerar *múltiplos provedores de IaaS*, pois cada provedor disponibiliza vários tipos de máquinas virtuais, sendo que cada uma com um preço associado. Assim, é necessário verificar, com cada provedor de IaaS, a relação custo benefício de cada máquina virtual. Além disso, o modelo de negócio na nuvem é geralmente baseado em acordos (SLA) para garantir a qualidade dos serviços (QoS). Então, para se adaptar ao paradigma da nuvem, é desejável que o algoritmo de escalonamento seja *orientado a SLA*. Entretanto, a maioria dos escalonadores de *workflows* em nuvem apresentados neste capítulo não são orientados a SLA e, além disso, não definem parâmetros de QoS ao usuário do provedor de SaaS e ao próprio provedor de serviço (SaaS). Portanto, a área de escalonamento de *workflows* em nuvens carece desse tipo de algoritmo.

Esta dissertação considera um provedor de SaaS que fornece um serviço de execução de *workflows* aos seus clientes. Assim, com intuito de minimizar investimentos em recursos computacionais, o cliente executa seus *workflows* através desse serviço disponibilizado na nuvem. Além disso, pode estipular um prazo (*deadline*), definido em um SLA estabelecido com o provedor de SaaS, para a execução do seu *workflow*. O provedor de SaaS, por sua vez, pode executar os *workflows* de seus clientes em máquinas virtuais alugadas, também via SLA, em vários provedores de IaaS, caso seus recursos próprios sejam insuficientes ou estejam ocupados. Portanto, além de respeitar os *deadlines* dos *workflows*, o provedor de SaaS tem o objetivo de minimizar custos monetários com máquinas virtuais e, por consequência, a maximização de lucros.

O parágrafo anterior descreve o principal cenário desta dissertação. Assim, para minimizar os custos monetários e satisfizer os *deadlines*, consideramos dois níveis de SLAs. O primeiro nível inclui todos SLAs estabelecidos entre o provedor de SaaS e cada um dos

seus clientes, enquanto o segundo nível contém todos SLAs acordados entre o provedor de SaaS e cada provedor de IaaS, onde as máquinas virtuais forem alugadas. Dessa forma, em relação aos *parâmetros de QoS* definidos nos SLAs, este trabalho considera que o *deadline* (tempo de resposta) é um requisito que o provedor de SaaS deve garantir na execução do *workflow* do seu cliente, enquanto o *tipo da VM*, *preço da VM*, *número de VMs* e *duração do SLA* são parâmetros que o provedor de IaaS deve satisfazer ao provedor de SaaS.

Trabalhos Relacionados	Técnica	Classificação	*Workflows*	Minimiza custos monetários	Múltiplos Provedores de IaaS	1 ou 2 Níveis de SLA	Parâmetros de QoS	
							para o cliente	para o provedor de SaaS
Li e Guo [50]	PLI	Estático	Não	Sim	Não	1	Orçamento Latência Vazão	Não se aplica
Wu *et al.* [77]	Heurística	Estático	Não	Sim	Sim	2	Tipo do produto Tipo da conta Duração do SLA Tempo de resposta	Tipo da VM Preço da VM
Zhao *et al.* [89]	Heurística	Dinâmico	Não	Não	Sim	–	Não se aplica	
Reig *et al.* [64]	Heurística	Dinâmico	Não	Sim	Não	1	*Deadline*	Não se aplica
Pandey *et al.* [60]	Meta-heurística	Estático	Sim	Sim	Sim	–	Não se aplica	
Wu *et al.* [80]	Meta-heurística	Estático	Sim	Sim	Sim	–	Não se aplica	
Juhnke *et al.* [43]	Heurística	Dinâmico	Sim	Sim	Sim	–	Não se aplica	
Mohammadi *et al.* [32]	Heurística	Estático	Sim	Sim	Não	–	Não se aplica	
Genez *et al.* (trabalho proposto)	PLI	Estático	Sim	Sim	Sim	2	*Deadline*	Tipo da VM Preço da VM nºde VMs Duração do SLA

Tabela 3.1: Resumo das características dos algoritmos de escalonamento de tarefas em computação em nuvem

Enfim, quando a infraestrutura computacional está inteiramente ocupada, o provedor de SaaS pode terceirizar recursos a fim de não recusar nenhuma solicitação de (futuros) clientes. Portanto, neste trabalho, o escalonamento de *workflow* em nuvem consiste basicamente em um processo de decisão cujo principal objetivo é (i) determinar quais e quantas máquinas virtuais serão necessárias para satisfazer o *deadline* do *workflow* e (ii) verificar quais provedores de IaaS deve ser utilizado com intuito de minimizar os custos monetários totais da execução do *workflow*. Aliás, como podemos observar, poucos trabalhos consideram noções de SLAs em dois níveis, um aspecto muito importante no modelo de negócio proposto pelo paradigma da computação em nuvem.

3.3 Considerações Finais

Neste capítulo, foi feita uma revisão bibliográfica sobre os algoritmos de escalonamento de tarefas (ou serviços), enfatizando os escalonadores de *workflows* em computação em nuvem, o qual é uma área de grande interesse atualmente. Os principais conceitos envolvidos nesta dissertação foram apresentados, comentando os trabalhos mais relevantes sobre o assunto em questão.

Como foi mostrado nas seções anteriores, existem vários algoritmos para escalonar *workflows* na literatura, os quais podem ser executados em diversas infraestruturas computacionais, tais como: grades, nuvens híbridas e, a ultimamente mais utilizadas, infraestruturas de computação em nuvem. Um dos principais interesses desses algoritmos pela computação em nuvem é a presença da característica de *elasticidade*, onde os recursos são facilmente expandidos/contraídos de forma rápida e barata. No entanto, independentemente do recurso usado, o problema de escalonamento é NP-Completo e, com isso, são utilizadas algumas técnicas para tentar encontrar uma solução que seja aproximada da ótima. Os algoritmos baseados nas técnicas de heurísticas e meta-heurísticas produzem, de modo geral, resultados satisfatórios e aproximados em tempo de execução aceitável, mas não é possível justificar, em termo estritamente lógico, a validade do resultado. Por outro lado, o uso da abordagem da programação linear permite alcançar soluções ótimas, mas o tempo de execução do escalonador pode ser alto (não-aceitável).

No capítulo seguinte é apresentado o mecanismo de escalonamento de *workflows* em nuvens proposto neste trabalho. É formulado como uma programação linear inteira cuja função objetivo é minimizar os custos monetários da execução do DAG, enquanto satisfaz o *deadline* estipulado no SLA. Além disso, também são apresentados duas heurísticas para solucionar a versão relaxada do PLI proposto, com o intuito de extrair uma solução inteira da fracionária (relaxada).

Capítulo 4

Um Modelo Linear para Escalonar Workflow em Nuvens

Computação em nuvem é uma tecnologia emergente que permite aos usuários usufruir da computação sob demanda de qualquer lugar do mundo. Entretanto, os provedores de serviços (IaaS, PaaS e SaaS) geralmente cobram pelo uso desses serviços, através do modelo de tarifação *pago-pelo-uso*. Aplicações complexas de e-Ciência e *e-Business*, modeladas como *workflows* e em forma de DAGs, são executadas em nuvens a fim de minimizar investimentos em infraestrutura computacional. Isto é, devido à insuficiência de recursos próprios, os usuários finais recorrem aos serviços de execução de *workflows* disponibilizados na nuvem pelos provedores de SaaS e de PaaS. Estes, por sua vez, alugam máquinas virtuais dos provedores de IaaS a fim de atender picos de demanda, pois seus recursos próprios podem não ser suficientes para executar os *workflows* de seus clientes. Além disso, os serviços são geralmente contratados através do acordo de nível de serviço (SLA), onde os requisitos de qualidade de serviços (QoS) são definidos.

Neste capítulo, formulamos um escalonador como um programa linear inteiro (PLI) cuja função objetivo é minimizar os custos monetários da execução do DAG, enquanto satisfaz o *deadline* estipulado no SLA do usuário. Esse escalonador determina quais provedores de IaaS e quais tipos de máquinas virtuais devem ser alugadas para o provedor de SaaS/PaaS minimizar seus custos monetários ao executar os *workflows* de seus clientes dentro do *deadline* estipulado. O restante deste capítulo está organizado como segue. A Seção 4.1 apresenta as notações matemáticas e a modelagem do problema de escalonamento nuvens. Na sequência, a Seção 4.2 introduz a formulação do PLI propriamente dito, incluindo as variáveis binárias e restrições lineares do problema. Na Seção 4.3 são descritas as abordagens para solucionar o programa linear inteiro, através das técnicas de heurísticas, relaxação linear e discretização da linha do tempo. Por fim, na Seção 4.4, são apresentadas algumas considerações finais deste capítulo.

4.1 Notações e Modelagem do Problema

Antes do problema de escalonamento ser modelado, temos que descrever algumas notações que são intrínsecas à formulação matemática do programa linear inteiro. Como visto anteriormente, na Seção 2.2, representamos um *workflow* como um grafo acíclico direcionado (DAG) $\mathcal{G} = \{\mathcal{U}, \mathcal{E}\}$, onde $\mathcal{U}$ é o conjunto de serviços atômicos e $\mathcal{E}$ representa as dependências de dados entre esses serviços. Seja $\mathcal{G}$ um *workflow* submetido ao provedor de SaaS por um de seus clientes. Assim, o principal problema abordado nesta dissertação é escalonar $\mathcal{G}$ em máquinas virtuais alugadas pelo provedor de SaaS de vários provedores de IaaS. Além disso, o custo monetário do escalonamento de $\mathcal{G}$ deve ser mínimo e o *makespan* de $\mathcal{G}$ deve ser menor ou igual ao *deadline* estipulado pelo cliente.

A seguir vamos definir as notações matemáticas utilizadas para modelar o problema de escalonamento de $\mathcal{G}$ como uma programação linear inteira [38].

- $n = |\mathcal{U}|$: 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 dados a serem transmitidos entre os nós $u_i \in \mathcal{U}$ e $u_j \in \mathcal{U}$ ($f_{i,j} \in \mathbb{R}^+$);
- $\mathcal{H}(j) = \left\{ ij : i < j, \text{ existe um arco do vértice } i \text{ ao vértice } j \text{ no DAG } \mathcal{G} \right\}$: 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 ($\delta_i \in \mathbb{N}$).

Como dito anteriormente na Seção 2.5, nosso cenário é composto por dois níveis de SLA. O primeiro contém todos os SLAs onde os *deadlines* $\mathcal{D}$ são definidos. Por outro lado, o segundo é composto por todos os SLAs onde as características de cada máquina virtual alugada são definidas, como por exemplo, número de núcleos de processamento, poder de processamento de cada núcleo, quantidade de memória RAM e preços. Seja $\mathcal{S}_i$ o conjunto composto por todos os SLAs assinados entre o provedor de SaaS e cada provedor de IaaS $i \in \mathcal{I}$. Esse conjunto é formado pela união de dois subconjuntos disjuntos, como

mostra a equação (4.1), onde o subconjunto σ_i^r inclui os SLAs destinados às máquinas virtuais reservadas e que estão prontamente disponíveis, e o subconjunto σ_i^o inclui apenas os SLAs para as máquinas virtuais alugadas sob demanda (*on-the-fly*).

$$\mathcal{S}_i = \sigma_i^r \bigcup \sigma_i^o \text{ e } \sigma_i^r \bigcap \sigma_i^o = \varnothing \tag{4.1}$$

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. Aliás, 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 solicitados 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 [78], a restrição em (4.2) deve ser rigorosamente obedecida.

$$\sum_{s \in \mathcal{S}_i} \alpha_s \leq \delta_i, \quad \forall\ i \in \mathcal{I} \text{ e para qualquer instante de tempo } t \tag{4.2}$$

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

$$\mathcal{V} = \left\{ \bigcup_{i=1}^{m} \mathcal{V}_i \quad \forall\ i \in \mathcal{I} : \mathcal{S}_i \neq \varnothing \right\} \tag{4.3}$$

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 $\mathcal{L}_{v_g,v_h} \in \mathbb{R}^+$. Aliás, 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 demandas. 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 citado 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

O programa linear inteiro soluciona o problema de escalonamento considerado neste trabalho 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 [33]. 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* [66]. Portanto, a formulação deste programa linear inteiro considera apenas intervalos discretos de tempo. A Seção 4.3.4 apresenta melhores detalhes sobre a representação da linha do tempo no escalonador proposto.

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 programa linear inteiro é formulado da seguinte maneira [38]:

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

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

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

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

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

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

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

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

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

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

As restrições em (C1) especificam que toda tarefa deve ser executada 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 todas 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.

4.3 Solucionando o Programa Linear Inteiro

Devido à NP-Completude do problema de escalonamento, precisamos utilizar algumas técnicas para desenvolver algoritmos de complexidade de tempo polinomial, pois essas técnicas podem, de modo geral, produzir resultados aproximados da solução ótima. Portanto, esta seção apresenta os métodos utilizados para solucionar o programa linear inteiro proposto na Seção 4.2. A Seção 4.3.1 descreve brevemente o método de enumeração *Branch & Cut*, um dos tradicionais algoritmos para solucionar um PLI, enquanto a Seção 4.3.2 introduz três abordagens desenvolvidas neste trabalho para extrair uma solução

(ótima ou não) do *solver*[1]. Na sequência, a Seção 4.3.3 apresenta duas heurísticas para obter soluções inteiras da versão relaxada do programa linear proposto. Por fim, a Seção 4.3.4 descreve o uso de diferentes níveis de granularidade da linha do tempo na PLI, a fim de (i) reduzir o tempo de execução do escalonador e (ii) encontrar soluções viáveis ao DAGs com grande número de nós e dependências de forma rápida.

4.3.1 O Método de Enumereção *Branch & Cut*

O *IBM ILOG CPLEX Optimizer*[2], frequentemente referenciado apenas como *CPLEX*, foi o software utilizado neste trabalho para solucionar o programa linear inteiro descrito anteriormente. Esse *solver* é um dos pacotes comercias amplamente utilizados em ambientes acadêmicos para resolver vários problemas matemáticos, como por exemplo, programação linear (PL), programação linear inteira (PLI) e programação quadrática (PQ). É importante notar que outros *solvers* poderia ser utilizado para solucionar o PLI proposto, tais como: *GNU Linear Programming Kit* (GLPK)[3], *Solving Constraint Integer Programs* (SCIP)[4], Gurobi[5] e Xpress[6]; dos quais os três primeiros são de código aberto (não comerciais) e os dois últimos de código fechado (comerciais). Embora o*CPLEX* também seja um aplicativo comercial, a IBM liberou o seu o uso de forma gratuita apenas para iniciativas acadêmicas. Portanto, escolhemos o *CPLEX* para solucionar o PLI proposto pelos seguintes motivos: (i) uso gratuito para o desenvolvimento desta pesquisa e (ii) teve um dos melhores desempenhos no *benchmark* realizado em [47].

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* [39], um dos tradicionais algoritmos para solucionar problemas de PLI. Aliás, esse método é uma combinação de outros dois, são eles: *Branch & Bound* [48] e *Planos de Corte* [23]. 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

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

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

[3] `http://www.gnu.org/software/glpk/`

[4] `http://scip.zib.de/`

[5] `http://www.gurobi.com/`

[6] `http://www.solver.com/xpress-solver-engine`

obtida em cada subproblema produzido no processo de remificação [25]. 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 (chamadas 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*.

Antes de iniciar o *Branch & Cut*, o *CPLEX* aplica algumas heurísticas na raiz da árvore de busca, com intuito de encontrar, rapidamente, algumas soluções viáveis ao programa linear inteiro. No entanto, essas soluções não têm garantias teóricas, isto é, sem provas de otimalidade. Em outras palavras, as heurísticas podem encontrar uma ou mais soluções viáveis, satisfazendo todas restrições e condições de integralidade do problema inicial, mas não garante se a solução é ótima. Assim, o *CPLEX* poderá possuir de antemão algumas soluções viáveis para o problema inicial e, portanto, poderá apresentá-las em situações onde a solução ótima demora (devido à complexidade alta) ou não é encontrada. Além disso, as soluções disponibilizadas pelas heurísticas podem servir como um estopim para o método de enumeração, ou seja, iniciar a busca na raiz da árvore com alguma solução viável para o problema inicial. Periodicamente, o *CPLEX* também pode aplicar heurísticas durante a execução do *Branch & Cut*, a fim de calcular alguma solução inteira a partir das informações disponíveis no momento (por exemplo, a solução relaxada do nó atual). Em outras palavras, a execução dessa heurística não substitui os passos de ramificação do *Branch & Cut*, mas as vezes é capaz de encontrar, com baixo custo computacional, uma nova solução viável inteira antes do processo de ramificação. Note que, as soluções encontradas via heurística são tratadas da mesma maneira que as soluções viáveis encontradas via *Branch & Cut*.

Depois de aplicar as heurísticas, o *Branch & Cut* é iniciado. A busca pela solução ótima do problema começa, então, na resolução do *programa linear relaxado*[9] representado pelo nó raiz[10] da árvore de enumeração. Se uma solução ótima é encontrada e as variáveis reais

[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.

[9] O programa linear relaxado é o programa linear inteiro sem as restrições de integralidade de cada variável inteira. Esse procedimento é conhecido como *relaxação linear* de uma formulação inteira.

[10] Note que, o nó raiz representa todo o espaço de soluções do problema inicial, e os demais nós da

receberem valores inteiros, então ela é guardada pois é a melhor solução até o momento. Caso contrário, as variáveis receberam valores não-inteiros (fracionários) e, com isso, o algoritmo *Planos de Corte* é utilizado para encontrar novas restrições lineares (planos de corte) que satisfazem todos os pontos viáveis inteiros do espaço de soluções e não satisfazem a solução ótima fracionária atual. Se tais restrições são encontradas, elas são adicionadas ao programa linear, o qual é solucionado novamente na esperança de encontrar uma solução ótima onde as variáveis recebam valores "menos fracionados" ou inteiros.

O processo de adicionar novos planos de cortes e, em seguida, resolver o programa linear é repetido até que (i) uma solução ótima cujas variáveis contínuas recebam somente valores inteiros é encontrada, ou (ii) quando não há mais planos de cortes a serem adicionados à formulação. Quando o primeiro caso ocorre, ou a solução é guardada, se nenhuma solução ótima foi encontrada até o momento; ou é comparada com a melhor solução inteira obtida até agora. Dessa forma, se na comparação ela for melhor, então torna-se a melhor solução; caso contrário, é descartada. Quando o segundo caso ocorre, as variáveis continuam recebendo valores fracionários e, portanto, nenhuma solução inteira é encontrada para o problema inicial.

Todas as tentativas de encontrar uma solução na raiz terminaram. Assim, para criar os nós do próximo nível da árvore de busca, é executado o procedimento de ramificação do *Branch & Bound*. Resumidamente, esse método restringe o programa linear relaxado da raiz (por exemplo, através da técnica de fixação[11] de variável) e cria vários problemas mais restritos, sendo que cada um será representado por um novo nó da árvore de enumeração. Portanto, para cada nó criado, os mesmos procedimentos feitos na raiz são aplicados, e a busca termina quando a solução ótima (global) seja encontrada ou o *solver* declara que o problema inicial não tem solução viável.

Devido à NP-Completude dos programas lineares inteiros (PLIs), a execução do *Branch & Bound* pode demorar muito, já que esses problemas utilizam tempo exponencial quanto ao tamanho da entrada. Então, de modo geral, o tempo de execução é limitado para um certo parâmetro, e a solução encontrada nesse limite de tempo pode não ser ótima, embora muitas vezes seja boa.

4.3.2 Abordagens para Solucionar o Programa Linear Inteiro

Conforme descrito na Seção 2.3, o problema de escalonamento é geralmente NP-Completo e, com isso, o tempo gasto para encontrar uma solução ótima inteira para o PLI aumenta exponencialmente com o tamanho da entrada (instâncias). Isto é, pode ser computaci-

árvore, que são gerados em tempo de execução, representam programas lineares com espaço de solução reduzido.

[11]Por exemplo, no caso de uma variável binária, fixa o seu valor para 0 em um nó da árvore de enumeração e para 1 em um outro nó.

onalmente caro encontrar uma solução ótima para entradas maiores, devido ao grande número de combinações possíveis entre as variáveis do problema de escalonamento; em nosso caso, combinações entre serviços (nós do DAG) e máquinas virtuais. Assim, se não estipularmos um limite para o tempo de resolução, o *CPLEX* pode, dependendo do tamanho da entrada do problema, demorar dias (ou até semanas) para encontrar uma solução ótima inteira. Por essa razão, definimos, a seguir, três abordagens para (tentar) obter rapidamente uma solução (ótima) inteira para o programa linear inteiro:

a) **Abordagem ótima**: O *solver* tenta encontrar uma ou mais soluções ótimas inteiras dentro de um intervalo de tempo. Se encontrado, retorna a melhor solução obtida até o momento; caso contrário, relata que a solução do escalonamento é inviável;

b) **Abordagem da raiz (primeira solução)**: O *solver* deve retornar a solução ótima inteira obtida na raiz da árvore de enumeração, dentro de um intervalo de tempo. Se alcaçado, retorna essa solução; caso contrário, relata que o escalonamento não tem uma solução viável;

c) **Abordagem relaxada**: Primeiramente, o *solver* aplica a relaxação linear no programa linear inteiro, simplesmente alterando o conjunto inteiro $\{0, 1\}$ das restrições em (C8) e (C9), descritas na Seção 4.2, para o intervalo real $[0, 1]$. Depois de remover as restrições de integridade das variáveis inteiras, ele soluciona o programa linear resultante (relaxado). Em seguida, aplicamos heurísticas na solução fracionária obtida, a fim de encontrar uma solução inteira. Ambas atividades devem ser feitas dentro de um intervalo de tempo. Se nenhuma solução inteira é encontrada através das heurísticas, então o escalonamento não possui uma solução viável.

As abordagens (a) e (b) são diretamente obtidas através da modificação de alguns parâmetros do *solver*. Além disso, o *CPLEX* cessa a execução quando (i) o intervalo de tempo é esgotado ou quando (ii) a solução ótima inteira é encontrada pela abordagem (a) ou a primeira solução ótima inteira é obtida pela abordagem (b). Em ambos os casos, se o tempo limite é atingido, o *solver* deve retornar a melhor solução obtida até o momento ou relatar que a solução da instância do problema de escalonamento é inviável. A abordagem (c), por sua vez, precisa de heurísticas para encontrar soluções inteiras na saída do *CPLEX*, pois as variáveis x e y, neste caso, recebem valores reais ($\mathbb{R}^+$). É necessário informar o *solver* para aplicar a relaxação linear no PLI antes de iniciar a execução.

4.3.3 Programa Linear Relaxado

Conforme descrito anteriormente, o relaxamento do programa inteiro linear consiste em remover a restrição de integralidade das variáveis, em nosso caso, das variáveis x e y.

Assim, temos que modificar apenas as restrições (C8) e (C9) da formulação do programa linear inteiro:

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

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

Dessa forma, as variáveis $x_{u,t,v}$ e $y_{t,v}$ receberão valores reais ($\mathbb{R}^+$) após a resolução do programa linear relaxado, se o escalonamento for viável. A variável real $x_{u,t,v}$ irá representar a melhor maneira de dividir um nó do DAG (serviço) entre as máquinas virtuais do conjunto $\mathcal{V}$. Além disso, mais de uma opção de máquina virtual pode aparecer para executar cada porção de um nó. No entanto, na Seção 2.2, assumimos que cada nó do DAG é indivisível. Por outro lado, a variável real $y_{t,v}$ irá representar a melhor maneira de dividir o uso de uma VM de $\mathcal{V}$ em cada instante de tempo $t \in \mathcal{T}$. Entretanto, para um determinado instante tempo, a máquina virtual está sendo utilizada ou não, isto é, uma decisão binária. Nesta seção descrevemos duas heurísticas que utilizam um método iterativo para extrair uma solução (ótima) inteira da solução fracionária do PLI proposto. Ambas heurísticas são baseadas na variável $x_{u,t,v}$, que providencia o identificador do nó u em questão, o tempo de término t da sua execução e a máquina virtual v que o executará.

O método iterativo funciona da seguinte maneira: Dado um DAG $\mathcal{G}$, um *deadline* $\mathcal{D}_\mathcal{G}$ e um conjunto de máquinas virtuais $\mathcal{V}$, o primeiro passo é relaxar o programa linear inteiro e, em seguida, chamar o *solver* para obter uma solução relaxada da instância inicial do problema. Dessa forma, obtemos valores reais para cada variável $x_{u,t,v}$. Com base nessa solução fracionária, selecionamos uma saída $x_{u,t,v}$ para torná-la inteira, simplesmente adicionando uma nova restrição[12] $x_{u,t,v} = 1$ no conjunto de restrições do PLI original. Note que, quando definimos que uma variável $x_{u,t,v}$ será igual a 1, estamos realmente decidindo que o nó u do DAG $\mathcal{G}$ será escalonado para ser executado na máquina virtual v e irá terminar a sua execução no tempo t. Depois de adicionar a nova restrição, chamamos o *solver* novamente para resolver o "novo" programa linear relaxado. Portanto, essas etapas são repetidas $k \leq |\mathcal{U}| = n$ vezes, ou seja, até que todos os nós $u \in \mathcal{U}$ tenham sido escalonados. É importante frisar que as duas heurísticas propostas diferem apenas na maneira de como selecionar, a cada iteração, uma variável $x_{u,t,v}$ da solução relaxada e ajustar o seu valor para 1. O algoritmo iterativo geral é mostrado no Algoritmo 4.1.

A primeira heurística inicialmente joga uma moeda para decidir se ela deve começar a partir do primeiro ou do último nó do DAG. Se o primeiro nó (u_0) for selecionado, então a heurística cria uma nova restrição $x_{u_0,t_a,v_x} = 1$, de tal modo que t_a seja *mínimo*. Isto é, estipula que o nó u_0 será executado na máquina virtual v_x e terminará a sua execução no

[12] Essa restrição consiste simplesmente em igualar o limite inferior e superior da variável $x_{u,t,v}$ a 1.

Algoritmo 4.1: Método iterativo para obter uma solução inteira do resultado do PLI relaxado.

Entrada: $\mathcal{G} = \{\mathcal{U}, \mathcal{E}\}$, deadline $\mathcal{D}_\mathcal{G}$, conjunto de VMs $\mathcal{V}$
Saída: Escalonamento de $\mathcal{G}$ em $\mathcal{V}$
inicio
  Aplique a relaxação linear no PLI
  Chame o *solver* para solucionar o PLI relaxado
  enquanto $\exists\ u \in \mathcal{U}$ *tal que* $\sum_{t \in \mathcal{T}} \sum_{v \in \mathcal{V}} x_{u,t,v} \neq 1$ **faça**
    Escolha um nó $u_i \in \mathcal{U}$ de acordo com a heurística
    Escolha uma saída x_{u_i,t_a,v_x} de acordo com a heurística
    Adiciona a nova restrição $x_{u_i,t_a,v_x} = 1$ no conjunto de restrições do PLI relaxado
    Chama o *solver* para resolver o "novo" PLI relaxado
  fim enqto
fin

tempo mínimo t_a. Por outro lado, se o último nó (u_n) for selecionado, então a heurística constrói a restrição $x_{u_n,t_b,v_z} = 1$, de tal modo que t_b seja *máximo*; estipulando, portanto, que o nó u_n será executado na máquina virtual v_z e terminará a sua execução no tempo máximo t_b. Depois dessa primeira etapa, a heurística continua selecionando, em turnos, os nós não escalonados a partir do início e do fim do DAG, até que uma solução inteira seja encontrada ou uma solução inviável seja alcançada. Caso haja empate no parâmetro *tempo*, a quebra é feita selecionando a variável x cujo valor seja maior; e se o empate ainda persistir, escolhemos a variável x aleatoriamente. Nós chamamos essa heurística de *Begin-Minimum End-Maximum Time* (BMEMT).

A segunda heurística seleciona aleatoriamente, em cada iteração, um nó pronto u_r com todos os seus antecessores já escalonados, e define a restrição $x_{u_r,t,v} = 1$, de tal modo que t seja *mínimo*. Empates são quebrados, em primeiro lugar, selecionando a máquina virtual v que tenha o menor custo, e em segundo lugar, a variável x cujo valor seja maior. Se o empate ainda persistir, a escolha de x é feita aleatoriamente. Portanto, esse processo continua até que uma solução inteira seja encontrada ou uma solução inviável seja alcançada. Denominamos essa heurística de *Begin-Minimum Time* (BMT).

4.3.4 Representação da Linha do Tempo

O problema de escalonamento é conhecido por ser um problema difícil de ser modelado e resolvido de modo eficiente [13, 62, 66]. 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 [33]: 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. Dessa forma, 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 $[1, deadline]$. Embora a linha temporal contínua aumente a precisão do escalonador, ou seja, a linha do tempo apresenta uma granularidade mais fina, o modelo matemático torna-se mais complexo [82], assim como torna-se mais complexa a obtenção de soluções ótimas inteiras para o programa linear inteiro.

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 [66]. 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-per-use* discutido na Seção 2.1. Aliás, as unidades de tempo parcialmente consumidas são geralmente cobradas como se fossem unidades de tempo completas[13]. Portanto, devido à cobrança das máquinas virtuais serem feitas por unidades de tempo inteiras, o escalonador apresentado neste capítulo utiliza apenas tempo discreto para representar a execução do *workflow*. No entanto, dependendo do nível de granularidade da linha do tempo e do tamanho do *workflow*, 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. Assim, neste trabalho vamos avaliar o uso de diferentes níveis de fragmentação do tempo discretizado no escalonamento de diferentes tipos de DAGs.

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 no conjunto $\mathcal{T}$ usado no programa linear inteiro. Por exemplo, o aumento do *deadline* (e/ou do grafo) implica diretamente no aumento do tamanho do conjunto $\mathcal{T}$. Essas situações podem aumentar o espaço de busca das soluções e, consequentemente, aumentar o tempo de execução do escalonador. Em outras palavras, o escalonamento torna-se um problema computacionalmente caro de ser resolvido ou até mesmo em um problema sem solução em tempo hábil [33]. Portanto, definimos λ como um *fator multiplicativo* que determina a granularidade do tempo discreto no problema de escalonamento.

Assim, redefinimos o conjunto $\mathcal{T}$ da PLI, descrito na Seção 4.2, no seguinte conjunto [37]:

$$\mathcal{T} = \left\{ \lambda, 2\lambda, 3\lambda, 4\lambda, \ldots, \Lambda \right\} \quad \middle| \quad \Lambda \leq \mathcal{D}_{\mathcal{G}} \quad \text{e} \quad \lambda \in \mathbb{N}^{+} \tag{4.4}$$

Por outro lado, o uso de intervalos de tempo longos (granularidade grossa) pode di-

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

minuir a quantidade de intervalos de tempo, mas o escalonamento pode continuar sendo inviável; pois, neste caso, não terá unidades de tempo suficientes para representar todas as dependências dos nós do DAG. Além disso, outra consequência do uso da granularidade grossa são soluções encontradas com custos mais altos, pois a redução do conjunto $\mathcal{T}$ implicará no aluguel de máquinas virtuais mais rápidas (e consequentemente mais caras) e, com isso, alguns núcleos de processamento não são utilizados; resultando, então, em desperdício de recursos (ou dinheiro). Assim, 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. Portanto, em nossos experimentos, variamos o fator multiplicativo λ para diminuir o tempo de execução no escalonamento de *workflows* com uma grande quantidade de nós e dependências, de modo que o escalonamento seja viável e, ainda, mantendo os custos monetários baixos. É importante ressaltar que, utilizamos essa abordagem sem aplicar o relaxamento linear no PLI; ou seja, as heurísticas não foram usadas nesse contexto.

4.4 Considerações Finais

Neste capítulo foi apresentado um modelo linear para escalonar *workflows* em nuvens, uma solução baseada na programação linear inteira cuja função objetivo é minimizar custos monetários com aluguéis de máquinas virtuais. A modelagem desse problema foi feita para o cenário descrito na Seção 2.5. Além de satisfazer o *deadline* estipulado no SLA do usuário, esse escalonador também 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. Assim, ao considerar essas variáveis, o escalonador é capaz de decidir qual é o melhor recurso para executar cada nó do DAG. Portanto, o programa linear proposto 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.

Como o tempo de escalonamento aumenta exponencialmente com o tamanho da entrada, adotamos algumas abordagens para encontrar rapidamente uma solução inteira do PLI. Abordagem ótima e raiz (primeira solução) são facilmente obtidas através da manipulação dos parâmetros do *CPLEX*, o qual retorna, dentro de um intervalo de tempo, a melhor solução inteira e a primeira solução inteira, respectivamente. Na abordagem relaxada, é aplicado o relaxamento linear no PLI e, em seguida, o programa linear relaxado é solucionado de acordo com Algoritmo 4.1 e as heurísticas BMEMT e BMT, com intuito de extrair solução inteira da fracionária. Para finalizar, variamos o nível de discretização da linha do tempo no PLI a fim de encontrar soluções viáveis em tempo hábil para DAGs

com grande número de nós e dependências.

No capítulo seguinte, o escalonador proposto é avaliado através de simulações e os resultados obtidos são discutidos. Além disso, as métricas utilizadas, o cenário e o ambiente das simulações são apresentados.

Capítulo 5

Avaliação de Desempenho do Escalonador Proposto

Neste capítulo é apresentada a avaliação de desempenho do escalonador de *workflows* em nuvens, incluindo as heurísticas BMEMT e BMT, descritas na Seção 4.3.2, e a discretização da linha do tempo, apresentada na Seção 4.3.4. O programa linear inteiro foi implementado em *Java*[1] e as simulações foram realizadas por meio da biblioteca *IBM ILOG CPLEX Optimizer*[2] com as configurações internas padrão. Nas simulações foram utilizados apenas grafos que representam aplicações do mundo real, tais como *fork-join* (Figura 2.2), *Montage* (Figura 2.3(a)) e *LIGO-1* (Figura 2.3(d)), e três provedores de IaaS distintos. Enfim, o objetivo destas simulações é demonstrar o funcionamento do escalonador proposto em ambientes cuja infraestrutura computacional é composta, principalmente, por máquinas virtuais alugadas de diferentes provedores de IaaS.

O restante deste capítulo está organizado como segue. Na Seção 5.1, são apresentadas as configurações e os parâmetros utilizados durante o processo de simulação. Na sequência, as análises dos resultados obtidos são descritas na Seção 5.2. Por fim, na Seção 5.3, são apresentadas as considerações finais obtidas durante a avaliação de desempenho do escalonador de *workflows* proposto.

5.1 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 5.1, 5.2 e 5.3 mostram, respectivamente, as opções de máquinas virtuais dos provedores de IaaS A, B e C [12]. Além disso, a Tabela 5.4 mostra todos os SLAs

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

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

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 da formulação do programa linear inteiro estão descritos na Seção 4.1.

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 5.1: Provedor de IaaS A

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 5.2: Provedor de IaaS B

As nuvens públicas geralmente não fornecem informações sobre a qualidade de serviço dos enlaces internos (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 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 neste capítulo.

5.2 Resultados

As simulações foram executadas em um processador Intel® Xeon $X5650$ (com 6 núcleos e 12 *threads*) CPU $2.67GHz$ com $16GB$ de memória RAM. Nossa avaliação é composta

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 5.3: Provedor de IaaS C

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

Tabela 5.4: Máquinas virtuais reservadas para o provedor de SaaS

por simulações de DAGs que representam aplicações do mundo real, tais como *Montage*, *LIGO-1*, *CSTEM* e *fork-join*; todos apresentados na Seção 2.2. 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 $[1, 3]$. Além disso, para as simulações onde não variamos a discretização da linha do tempo, realizamos 50 simulações para cada DAG, enquanto para as demais onde variamos o valor de λ, fizemos 30 para cada DAG. Note que, todas as simulações foram realizadas com as configurações dos provedores de IaaS e dos SLAs presentes nas Tabelas 5.1 a 5.4. 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). Segundo o nosso conhecimento, nenhum trabalho presente na literatura considera o escalonamento de *workflows* no cenário descrito na Seção 2.5 e, portanto, os resultados descritos aqui não foram comparados com outros algoritmos.

Para representar os possíveis valores dos *deadlines* solicitados pelos usuários, executamos as simulações com $\mathcal{D}_{\mathcal{G}}$ 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, provavelmente, 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.

Aliás, também verificamos o consumo de memória RAM utilizado pelo *CPLEX* para escalonar os DAGs. Esse processo de capturar o consumo de memória foi realizado da seguinte maneira: imediatamente antes de iniciar o *solver*, uma *Thread* é criada para ficar capturando o consumo de memória RAM da Máquina Virtual Java – *Java Virtual Machine* (JVM) a cada segundo do escalonamento; em seguida, imediatamente após o término do *solver*, essa *Thread* é destruída. Entretanto, tanto o gerenciamento de memória quanto o coletor de lixo (do inglês, *garbage collection*) são feitos internamente pela JVM, ou seja, a própria linguagem é responsável por liberar espaços de memória (lixo) que não estão sendo utilizados, mas em algum momento foram alocados. O núcleo do *CPLEX*, por sua vez, é implementado na linguagem C, os quais são chamados (ou invocados) pela aplicação *Java* por meio da *Java Native Interface* (JNI)[3]. Dessa forma, o gerenciador de memória e o coletor de lixo do *Java* não controlam a memória alocada dentro das rotinas essenciais do núcleo do *CPLEX*. Portanto, por ser um código fechado e ainda ser executado indiretamente via JNI, torna-se difícil a verificação do consumo exato de memória RAM utilizado pelo núcleo do *CPLEX*. No entanto, para se ter uma visão mais ampla das soluções, vamos apresentar apenas o consumo de memória RAM da JVM em alguns casos, quando, dado um determinado DAG $\mathcal{G}$, tanto as heurísticas BMEMT e BMEMT quanto as abordagens ótima (OPT) e primeira solução (PS) encontraram alguma solução viável para o escalonamento de $\mathcal{G}$.

As seções a seguir apresentam os resultados obtidos durante as simulações. Na Seção 5.2.1 estão descritos os resultados obtidos sem aplicar o aumento da discretização do tempo, ou seja, mantemos o valor de λ constante e igual a 1 ($\lambda = 1$), enquanto que na Seção 5.2.2 são apresentados os resultados onde a técnica da discretização do tempo é aplicada para diminuir o tempo de execução do escalonador. Note que, os gráficos a seguir foram plotados de acordo com a média das 30 simulações de cada DAG, com um intervalo de confiança de 95%. Em alguns pontos, o intervalo de confiança é pequeno e, consequentemente, a sua visualização torna-se imperceptível.

5.2.1 Simulações com λ constante e igual a 1

Nesta seção, vamos apresentar os resultados das simulações das abordagens descritas na Seção 4.3.2, as quais são: *ótima*, *primeira solução (raiz)* e *relaxada*. Lembrando que na abordagem relaxada é executado o Algoritmo 4.1 de acordo com a heurística escolhida, isto é, ou com a heurística *Begin-Minimum End-Maximum Time* (BMEMT) ou com a heurística *Begin-Minimum Time* (BMT). No entanto, como o aumento da discretização da linha do tempo não foi aplicado nestas simulações, a linha temporal considerada foi:

[3]A JNI permite que um código escrito em *Java* utilize a implementação de bibliotecas desenvolvidas em outras linguagens de programação, como, por exemplo, *C/C++* e *Assembler*.

$\mathcal{T} = \{1, 2, 3, \ldots, \mathcal{D}_\mathcal{G}\}$. Além disso, é importante frisar que na abordagem ótima o *CPLEX* utiliza todas as *threads* disponíveis no processador. Isto é, devido à integralidade das variáveis inteiras, o *CPLEX* tenta acelerar a procura da solução ótima colocando cada ramo da árvore de enumeração em uma *thread* disponível no processador. Os DAGs utilizados nestas simulações foram: *fork-join* com 10, 20 e 30 nós, *CSTEM* e *Montage*. Cada simulação foi realizada na seguinte ordem:

1. Foi executada a abordagem relaxada com a heurística BMEMT e com o tempo limite de 1800 segundos em cada iteração do *solver*. Em seguida, foi armazenado o tempo de execução do *solver* na variável RT_{BMEMT};

2. Foi executada a abordagem relaxada com a heurística BMT e com o tempo limite de 1800 segundos em cada iteração do *solver*. Em seguida, foi armazenado o tempo de execução do *solver* na variável RT_{BMT};

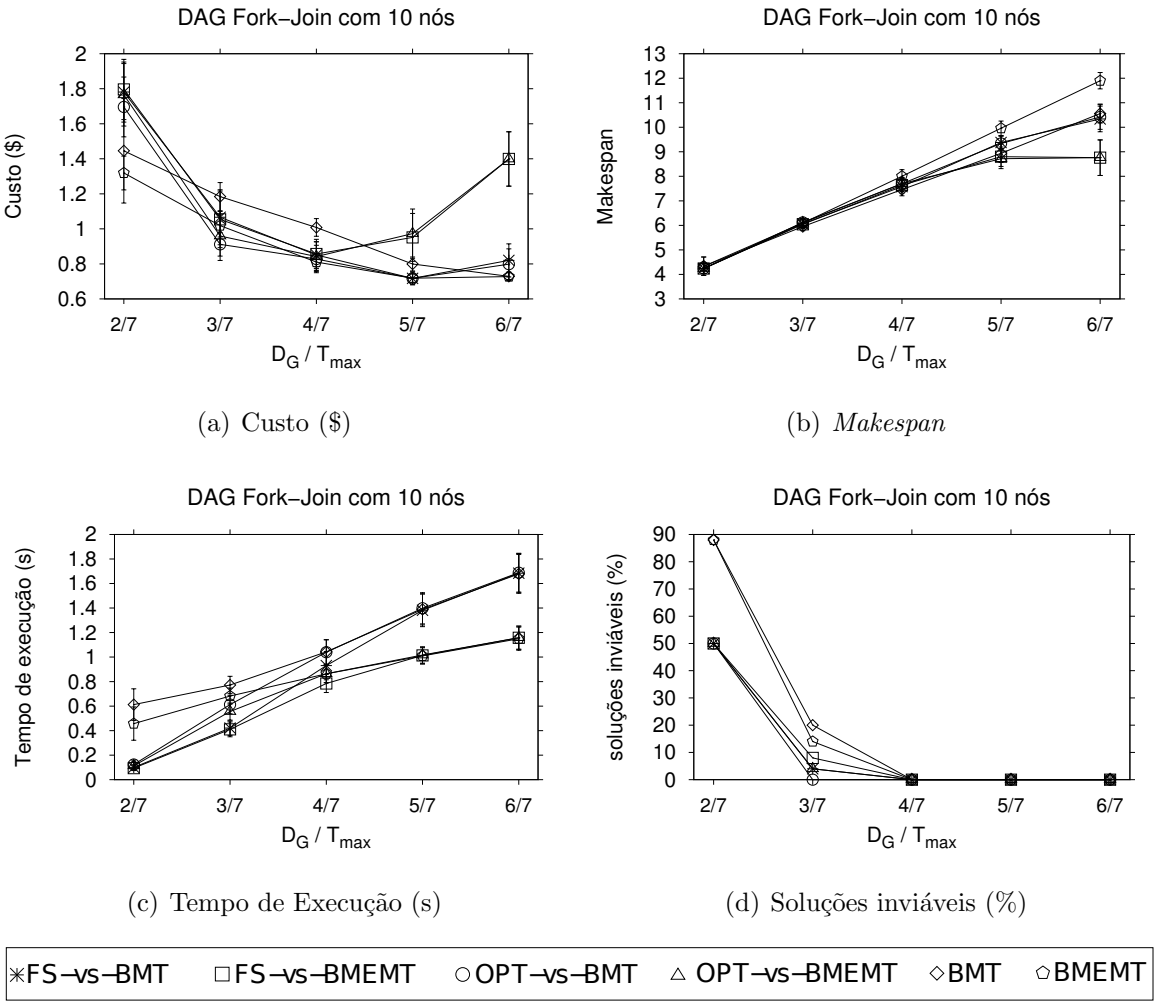
3. Foram executadas as abordagens *ótima* (OPT) e *primeira solução* (PS) com o tempo limite igual a RT_{BMEMT} (curvas $OPT\text{-}vs\text{-}BMEMT$ e $PS\text{-}vs\text{-}BMEMT$ nos gráficos);

4. Foram executadas as abordagens *ótima* (OPT) e *primeira solução* (PS) com o tempo limite igual a RT_{BMT} (curvas $OPT\text{-}vs\text{-}BMT$ e $PS\text{-}vs\text{-}BMT$ nos gráficos).

Resumindo, para uma determinada entrada na simulação, o *solver* é executado 6 vezes e na seguinte ordem: BMEMT, BMT, $OPT\text{-}vs\text{-}BMEMT$, $PS\text{-}vs\text{-}BMEMT$, $OPT\text{-}vs\text{-}BMT$ e $PS\text{-}vs\text{-}BMT$.

DAG *fork-join* com 10 nós

A Figura 5.1 mostra resultados das simulações para o DAG *fork-join* com 10 nós. Para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 2/7$, ambas heurísticas BMEMT e BMT encontraram poucas soluções viáveis, apenas 10% das simulações, como mostra a Figura 5.1(d). Além disso, 50% das soluções das abordagens OPT e PS também foram inviáveis. Portanto, devido ao *deadline* curto, os números de soluções inviáveis foram altos nestes casos, pois o *solver* não teve intervalos de tempo suficientes para escalonar todos os nós do DAG. Por outro lado, para as soluções viáveis, o *solver* teve que utilizar máquinas virtuais rápidas (e consequentemente caras) para escalonar todos os nós do DAG *fork-join* e, portanto, os custos monetários dos poucos escalonamentos obtidos para esse *deadline* foram altos, como mostra o gráfico ilustrado na Figura 5.1(a). Embora os custos monetários gerados pelas heurísticas BMEMT e BMT apresentem valores menores que os custos produzidos pelas abordagens OPT e PS, a comparação, neste caso, foi feita através da média dos 10% das soluções viáveis de



(a) Custo ($)

(b) *Makespan*

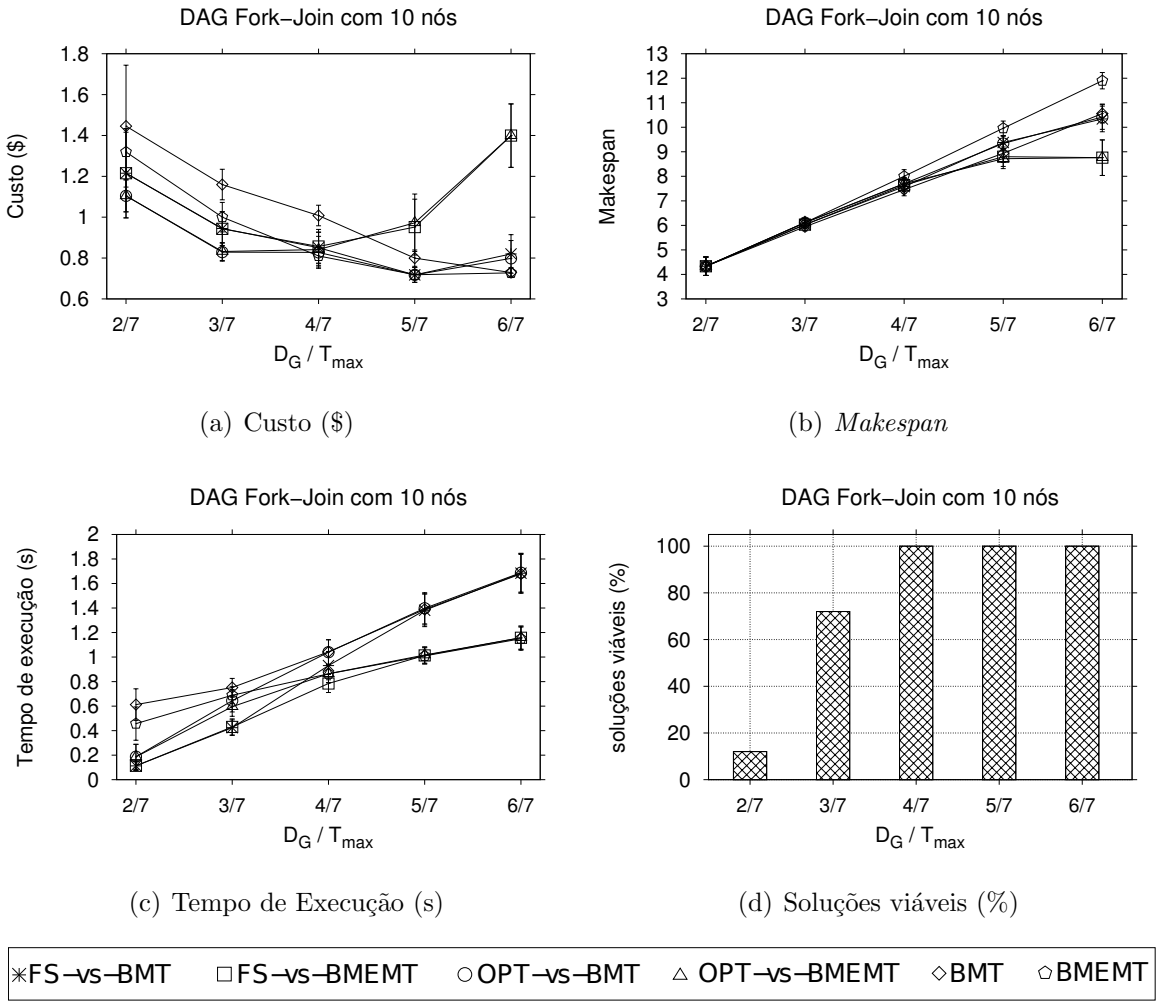
(c) Tempo de Execução (s)

(d) Soluções inviáveis (%)

Figura 5.1: Resultados para o DAG *fork-join* com 10 nós

BMEMT e BMT contra os 50% das soluções viáveis de OPT e PS. A comparação entre as soluções viáveis será discutida a frente.

Aumentando o *deadline* para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 3/7$, OPT conseguiu encontrar soluções viáveis antes de atingir o tempo limite estipulado pelas variáveis RT_{BMEMT} e RT_{BMT}, ou seja, as soluções encontradas foram ótimas e, com isso, os custos monetários foram mínimos. Embora PS também tenha conseguido encontrar soluções viáveis antes de atingir os limites de tempo RT_{BMEMT} e RT_{BMT}, o custo monetário médio foi aproximadamente 20% maior em relação à OPT. Todavia, esse comportamento já era esperado, pois a abordagem PS procura soluções que sejam somente nó raiz da árvore de enumeração, conforme descrito na Seção 4.3.1. Aliás, BMEMT conseguiu encontrar soluções com custos monetários semelhantes em relação à abordagem PS e custos 15% maiores em relação à



(a) Custo ($)

(b) *Makespan*

(c) Tempo de Execução (s)

(d) Soluções viáveis (%)

Figura 5.2: Comparação entre as soluções viáveis do DAG *fork-join* com 10 nós

OPT. Note que, o número de soluções inviáveis de todas abordagens foi pequeno para esse *deadline* e, à medida que esse aumenta, o *solver* conseguiu encontrar soluções para todas abordagens, ou seja, 0% de soluções inviáveis.

O desempenho da heurística BMEMT também tornou-se melhor com aumento do *deadline*, como, por exemplo, BMEMT foi capaz de encontrar soluções com custos 35% menores que $OPT\text{-}vs\text{-}BMEMT$ e $PS\text{-}vs\text{-}BMEMT$ quando $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 5/7$ e custos 45% menores quanto $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$, ambos com tempos de execução equivalentes. O mesmo fato ocorreu para BMT, com custos 15% menores que $OPT\text{-}vs\text{-}BMT$ e $PS\text{-}vs\text{-}BMT$ quando $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$. Portanto, ao considerar o mesmo tempo de execução do *solver*, as heurísticas conseguiram encontrar soluções com custos monetários menores que OPT e PS para *deadlines* altos.

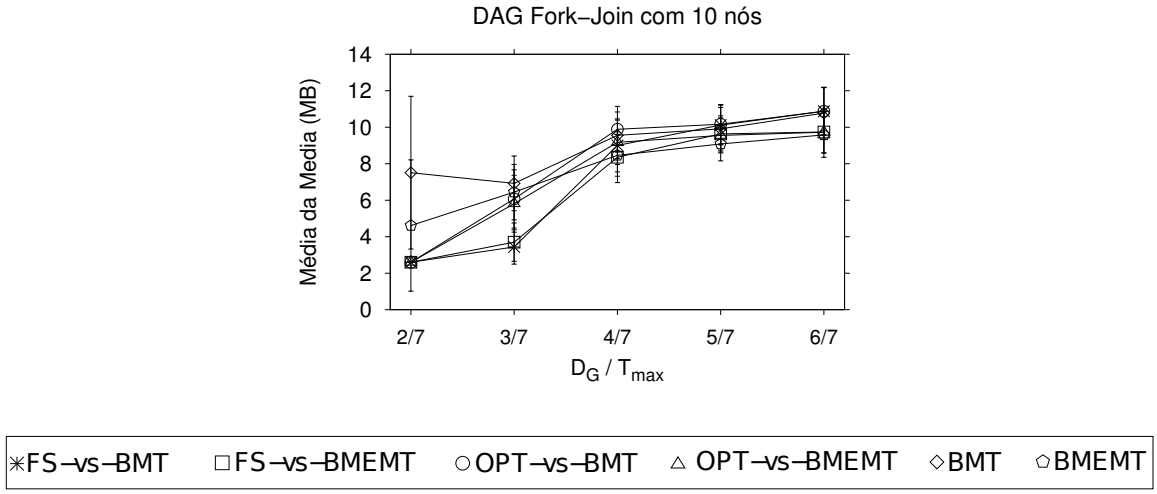


Figura 5.3: Consumo de memória RAM nas simulações DAG *fork-join* com 10 nós

O aumento do custo monetário das curvas $OPT\text{-}vs\text{-}BMEMT$ e $PS\text{-}vs\text{-}BMEMT$ de $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 5/7$ para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$ ocorreu porque o *solver* manteve o valor do *makespan*, como ilustra a Figura 5.1(b). Em outras palavras, ao aumentar o *deadline* de $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 5/7$ para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$, aumentamos não apenas o número de elementos do conjunto $\mathcal{T}$, mas também o número de possibilidades de soluções para essa configuração. Devido ao limite de tempo estipulado pela variável RT_{BMEMT} ser curto, o *solver* não conseguiu encontrar uma solução melhor para esse *deadline* em tempo viável; isto é, o *solver* não teve tempo suficiente para analisar todas as possibilidades do conjunto de soluções para minimizar o custo monetário do escalonamento. Por exemplo, como $RT_{BMEMT} \leq RT_{BMT}$ em $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$, o *solver* teve 35% a mais de tempo para encontrar soluções melhores, ou seja, aproveitar o *deadline* alto para prolongar o *makespan* e diminuir o custo monetário do escalonamento em 41%. Esses detalhes podem ser observados na comparação entre as curvas $OPT\text{-}vs\text{-}BMT$ e $OPT\text{-}vs\text{-}BMEMT$ e entre $PS\text{-}vs\text{-}BMEMT$ e $PS\text{-}vs\text{-}BMT$ nas Figuras 5.1(a) e 5.1(b) em $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$.

Para $\mathcal{D}_\mathcal{G} \leq \mathcal{T}_{max} \times 4/7$, o *makespan* do DAG manteve-se semelhante para todas as execuções; mas para $\mathcal{D}_\mathcal{G} > \mathcal{T}_{max} \times 4/7$, somente BMEMT gerou *makespans* altos. Conforme descrito na Seção 4.3.3, esse comportamento já era esperado, pois BMEMT tende, respeitando um *deadline*, a maximizar o tempo disponível para escalonar todos os nós do DAG. É importante frisar que, neste trabalho, a única restrição do *makespan* era que fosse menor ou igual ao *deadline* do escalonamento e, portanto, vamos fazer poucos comentários sobre ele nos próximos DAGs.

A Figura 5.2 mostra apenas a comparação das soluções viáveis obtidas por todas execuções do *solver* para o DAG *fork-join* com 10 nós. Isto é, para uma determinada

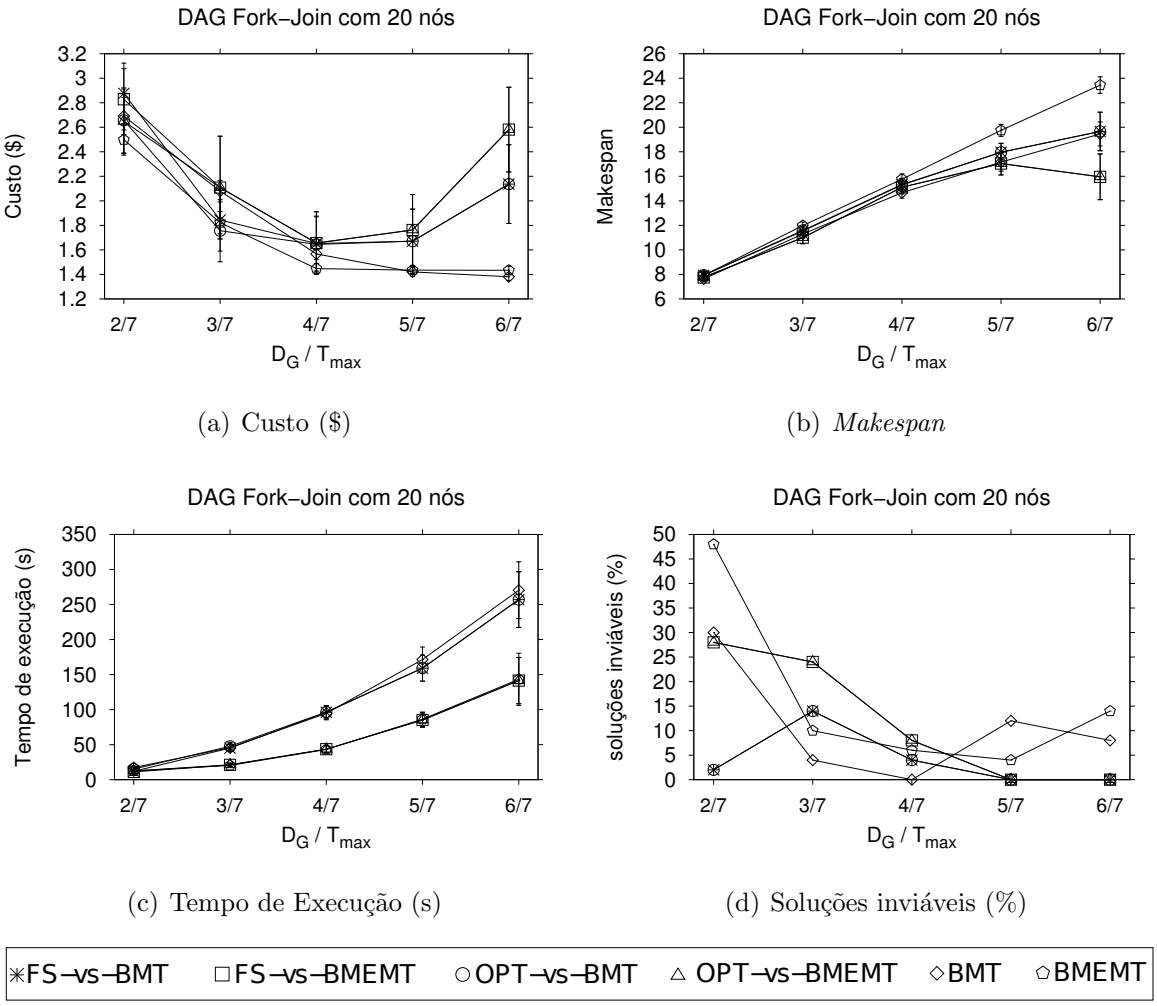
entrada, as 6 execuções do *solver* encontraram uma solução viável para o escalonamento do DAG. Como podemos observar, para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 2/7$, OPT e PS encontraram rapidamente soluções viáveis e, além disso, com custos monetários menores que BMEMT e BMT. Aliás, as soluções encontradas por OPT atingiram o custo mínimo (solução ótima) e as soluções da raiz da árvore de enumeração (PS) custaram 10% a mais que OPT. Entretanto, as soluções viáveis de todas as 6 execuções com esse *deadline* representam apenas 12% do conjunto de simulações. A heurística BMEMT alcançou soluções com custos monetários próximos das soluções encontradas por PS, como por exemplo, os custos monetários médios de BMEMT foram de 5% ($\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 3/7$) a 9% ($\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 2/7$) maiores que PS (curva $PS\text{-}vs\text{-}BMEMT$), sendo que PS encontrou soluções mais rapidamente que BMEMT. A heurística BMT também não teve um bom desempenho para *deadlines* curtos. Portanto, para um mesmo tempo de execução do *solver*, as heurísticas tendem a encontrar soluções com custos monetários baixos quando o *deadline* é longo.

A Figura 5.3 mostra a *média da média* do consumo de memória das simulações da Figura 5.2 da seguinte maneira: devido a captura do uso de memória ser feita a cada segundo da simulação, calculamos primeiro a média de cada simulação e, em seguida, a média das 50 simulações. Para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 2/7$, as heurísticas consumiram mais memória que OPT e PS; BMEMT consumiu, em média, 77% a mais de memória que OPT e PS, enquanto BMT consumiu 288% a mais na média. Por outro lado, para *deadlines* maiores, ambas heurísticas passaram a consumir a mesma quantidade de memória RAM que OPT e PS e, além disso, com custos monetários menores. É importante frisar que, devido ao *garbage colletor* ser controlado pela JVM, o consumo de memória pode sofrer variações de simulação para simulação.

DAG *fork-join* com 20 nós

Quando aumentamos o número de nós do DAG *fork-join* de 10 para 20, o desempenho das heurísticas tornou-se ainda melhor com *deadlines* maiores, como mostra a Figura 5.4. Por exemplo, BMEMT foi capaz de encontrar soluções com custos médios de 22% (para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 3/7$) a 46% (para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$) menores que $OPT\text{-}vs\text{ -}BMEMT$ e $PS\text{-}vs\text{-}BMEMT$, enquanto BMT encontrou soluções com custos monetários médios de 7% (para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 3/7$) a 42% (para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$) menores que $OPT\text{-}vs\text{-}BMT$ e $PS\text{-}vs\text{-}BMT$; comparação realizada considerando o mesmo tempo de execução.

A propósito, para todos os *deadlines*, OPT e PS não encontraram soluções antes do limite de tempo estipulado pelas variáveis RT_{BMEMT} e RT_{BMT}; logo, as soluções encontradas por essas abordagens não foram ótimas (ver também Figura 5.5). Além disso, o aumento do custo monetário médio das curvas $OPT\text{-}vs\text{-}BMEMT$ e $PS\text{-}vs\text{-}BMEMT$ em $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$ ocorreu pelo mesmo motivo do DAG anterior, redução do *makespan* com o uso de máquinas virtuais rápidas e caras. Em outras palavras, com um limite



(a) Custo ($)

(b) *Makespan*

(c) Tempo de Execução (s)

(d) Soluções inviáveis (%)

Figura 5.4: Resultados para o DAG *fork-join* com 20 nós

de tempo curto (armazenado em RT_{BMEMT}) para solucionar o escalonamento, o *solver* retornou a melhor solução que tinha até o momento, pois não teve tempo suficiente para encontrar uma solução melhor com custos menores. Note que, de modo geral, quanto maior o *deadline*, maior o conjunto $\mathcal{T}$ e, consequentemente, maior o tempo necessário para o *solver* conseguir encontrar a solução ótima.

Por outro lado, como podemos analisar, RT_{BMT} foi 77% maior que RT_{BMEMT} na média, então o *solver* teve mais tempo para analisar melhor o conjunto de soluções para reduzir os custos monetários. Por exemplo, ainda em $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$, os custos monetários de ambas curvas *OPT-vs-BMT* e *PS-vs-BMT* foram, respectivamente, 20% menores que os custos das curvas *OPT-vs-BMEMT* e *PS-vs-BMEMT*. É importante frisar que, para um mesmo tempo de execução do *solver*, se o custo monetário de OPT é

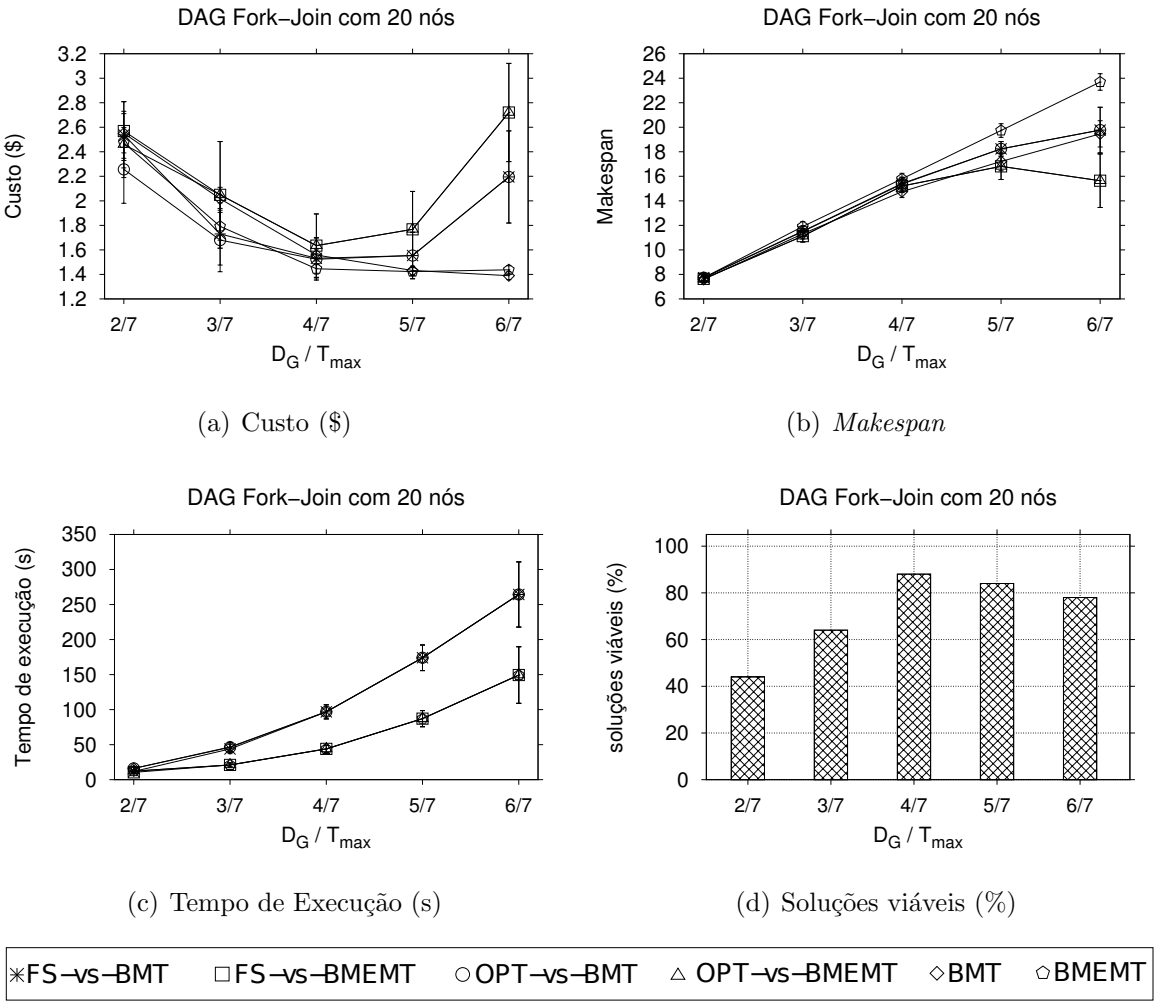
menor ou igual ao de PS, então a busca pela solução com uso da abordagem OPT permaneceu apenas na raiz da árvore de enumeração. Entretanto, tanto o custo monetário médio quanto o *makespan* médio das curvas *OPT-vs-BMT* e *PS-vs-BMT* aumentaram de $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 5/7$ para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$. Isto é, mesmo RT_{BMT} sendo 77% maior que RT_{BMEMT}, as soluções encontradas pelas curvas *OPT-vs-BMT* e *PS-vs-BMT* foram piores que as curvas das heurísticas BMEMT e BMT. Em outras palavras, nem sempre o aumento do custo monetário está relacionado à redução do *makespan*. Por exemplo, para um mesmo tempo de execução do *solver*, tanto BMT quanto *OPT-vs-BMT* encontraram soluções com valores semelhantes para o *makespan* (na média, *OPT-vs-BMT* foi 1.5% maior que BMT), porém os custos monetários de BMT foram 36.5% menores que os custos de *OPT-vs-BMT*. Portanto, para um mesmo tempo de execução, BMT conseguiu escolher máquinas virtuais melhores que *OPT-vs-BMT*, pois continuou reduzindo o custo monetário enquanto o *deadline* aumentava.

A comparação das soluções viáveis obtidas por todas as execuções do *solver* para o DAG *fork-join* com 20 nós é mostrada na Figura 5.5. Note que, dado uma entrada, as 6 execuções do *solver* juntas não conseguiram encontrar soluções viáveis para todas as 50 simulações e para todos os *deadlines*, como mostra a Figura 5.5(d). É importante ressaltar que, de modo geral, as heurísticas encontraram soluções com custos monetários semelhantes (para *deadlines* curtos) e até menores (para *deadlines* longos) que as soluções encontradas pelas abordagens OPT e PS; comparação feita considerando o mesmo tempo de execução do *solver*.

O consumo médio de memória RAM das simulações da Figura 5.5 é apresentado na Figura 5.6. Como podemos analisar, mesmo com aumento do número de nós, as heurísticas BMEMT e BMT consumiram, em média, quantidades semelhantes de memória RAM que as abordagens OPT e PS para todos os *deadlines*. Embora não haja um padrão explícito entre o consumo de memória RAM e os *deadlines* (devido aos detalhes explicados anteriormente), a execução do *solver* com as heurísticas não tem tendência em consumir mais memória que com as execuções com as abordagens OPT e PS, lembrando que ambas heurísticas são baseadas em um método iterativo no número de nós do DAG.

DAG *fork-join* com 30 nós

A principal característica do DAG *fork-join* é que a maioria dos nós pode ser escalonada em paralelo, ou seja, os nós que não são *fork* e *join* são geralmente independentes. Isso implica no aumento do número de possibilidades do escalonamento e no tempo de execução do *solver*; como pode ser visto na Figura 5.7, que apresenta os resultados das simulações para o DAG *fork-join* com 30 nós. Como podemos observar na Figura 5.7(c), o tempo de execução de todas as abordagens foram, na maioria das simulações, bem maiores do que para os DAGs anteriores. Por exemplo, para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$, o *solver* chegou a demorar



(a) Custo ($)

(b) *Makespan*

(c) Tempo de Execução (s)

(d) Soluções viáveis (%)

Figura 5.5: Comparação entre as soluções viáveis do DAG *fork-join* com 20 nós

1.8, 300 e 4500 segundos para encontrar alguma solução para o DAG *fork-join* com 10, 20 e 30 nós, respectivamente. Em outras palavras, quando aumentamos o número de nós do DAG *fork-join* de 10 para 20 e de 20 para 30, 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*. Portanto, devido à NP-Completude do problema de escalonamento, encontrar uma solução ótima para esse DAG de 30 nós pode levar horas ou até mesmo dias.

Para $\mathcal{D}_\mathcal{G} \geq \mathcal{T}_{max} \times 3/7$, as heurísticas BMEMT e BMT conseguiram encontrar soluções com custos monetários mais baixos do que as abordagens OPT e PS. Por exemplo, BMEMT foi capaz de achar soluções com custos médios de 55% (para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 3/7$) a 60% (para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$) menores que $OPT\text{-}vs\text{-}BMEMT$ e $PS\text{-}vs\text{-}BMEMT$, con-

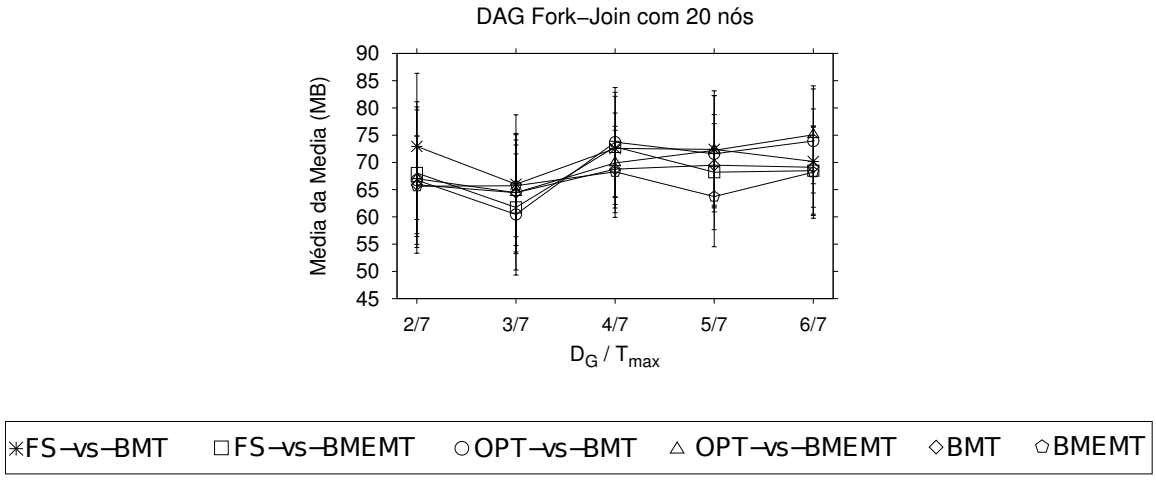


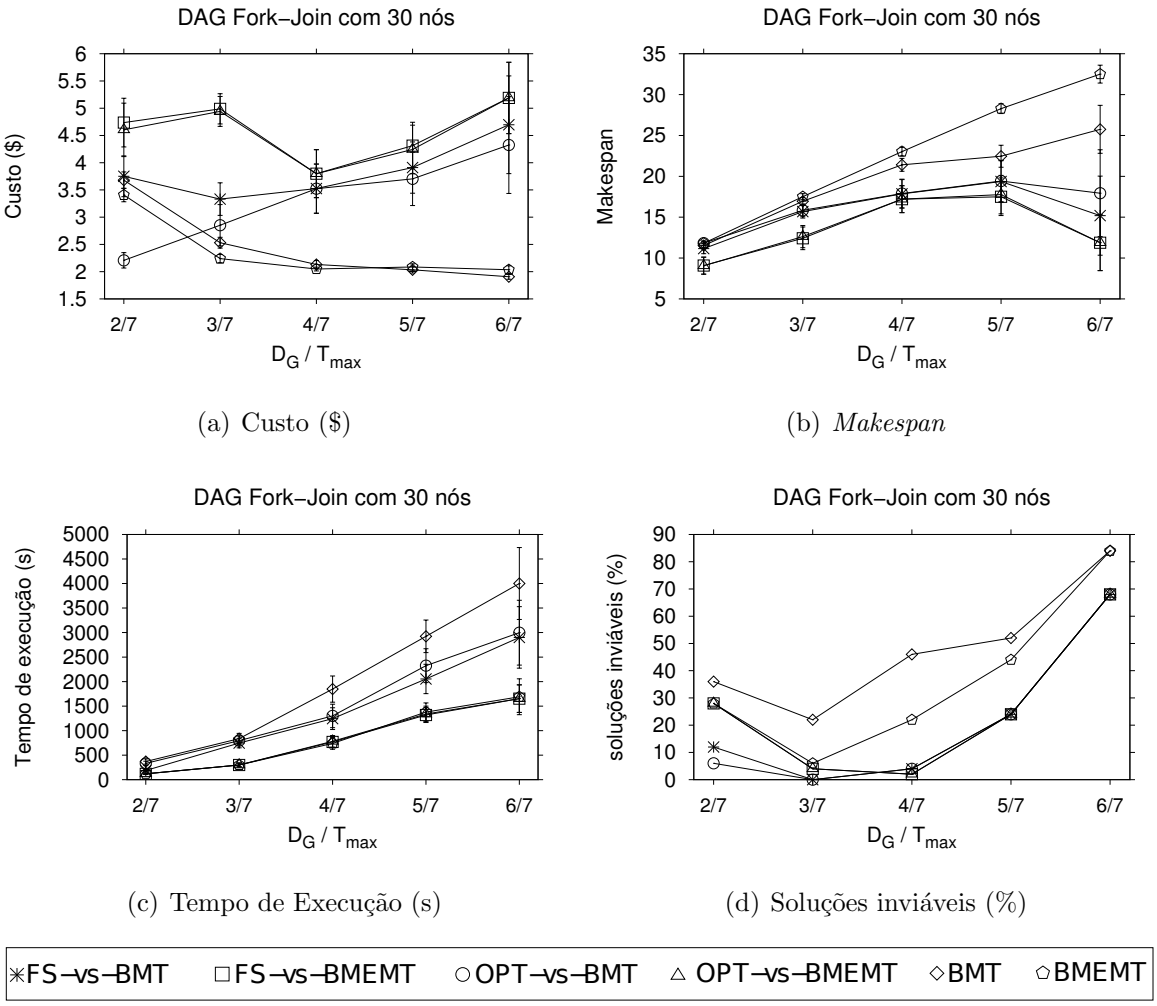
Figura 5.6: Consumo de memória RAM nas simulações DAG *fork-join* com 20 nós

siderando o mesmo tempo de execução. Em contrapartida, BMT encontrou soluções com custos monetários médios de 15% (para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 3/7$) a 55% (para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$) menores que $OPT\text{-}vs\text{-}BMT$ e $PS\text{-}vs\text{-}BMT$, comparação também realizada considerando o mesmo tempo de execução. Por outro lado, na maioria dos casos, as porcentagens das soluções inviáveis de BMEMT e BMT foram maiores que as porcentagens de OPT e PS. Conforme explicado anteriormente, houve um aumento do custo monetário para as abordagens OPT e PS em $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$ porque houve uma redução do *makespan* médio.

Dado uma entrada, a Figura 5.8 mostra apenas a comparação das soluções viáveis obtidas pelas 6 abordagens. Como podemos analisar, para *deadlines* curtos ($\mathcal{D}_\mathcal{G} \leq \mathcal{T}_{max} \times 3/7$), os custos monetários médios foram maiores, pois o *solver* teve que utilizar máquinas virtuais caras e rápidas. Além disso, como $RT_{BMEMT} \leq RT_{BMT}$ para todos os *deadlines*, então os custos monetários representados pelas curvas $OPT\text{-}vs\text{-}BMEMT$ e $PS\text{-}vs\text{-}BMEMT$ são maiores que os custos monetários das curvas $OPT\text{-}vs\text{-}BMT$ e $PS\text{-}vs\text{-}BMT$ no gráfico da Figura 5.10(a).

Para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 4/7$, todas as abordagens reduziram o custo monetário médio do escalonamento, mas para *deadlines* altos ($\mathcal{D}_\mathcal{G} \geq \mathcal{T}_{max} \times 6/7$), PS e OPT aumentaram os custos monetários, pois o *solver* não teve tempo suficiente para encontrar soluções melhores. As heurísticas, por sua vez, tiveram resultados semelhantes quando comparados aos DAGs anteriores, ou seja, encontraram soluções com custos monetários menores para *deadlines* longos, comparação realizada com as abordagens OPT e PS e considerando o mesmo tempo de execução. Aliás, os valores do *makespan* tiveram comportamentos semelhantes aos DAGs anteriores.

A Figura 5.9 mostra a média da média do uso de memória RAM para simular os



(a) Custo ($)

(b) *Makespan*

(c) Tempo de Execução (s)

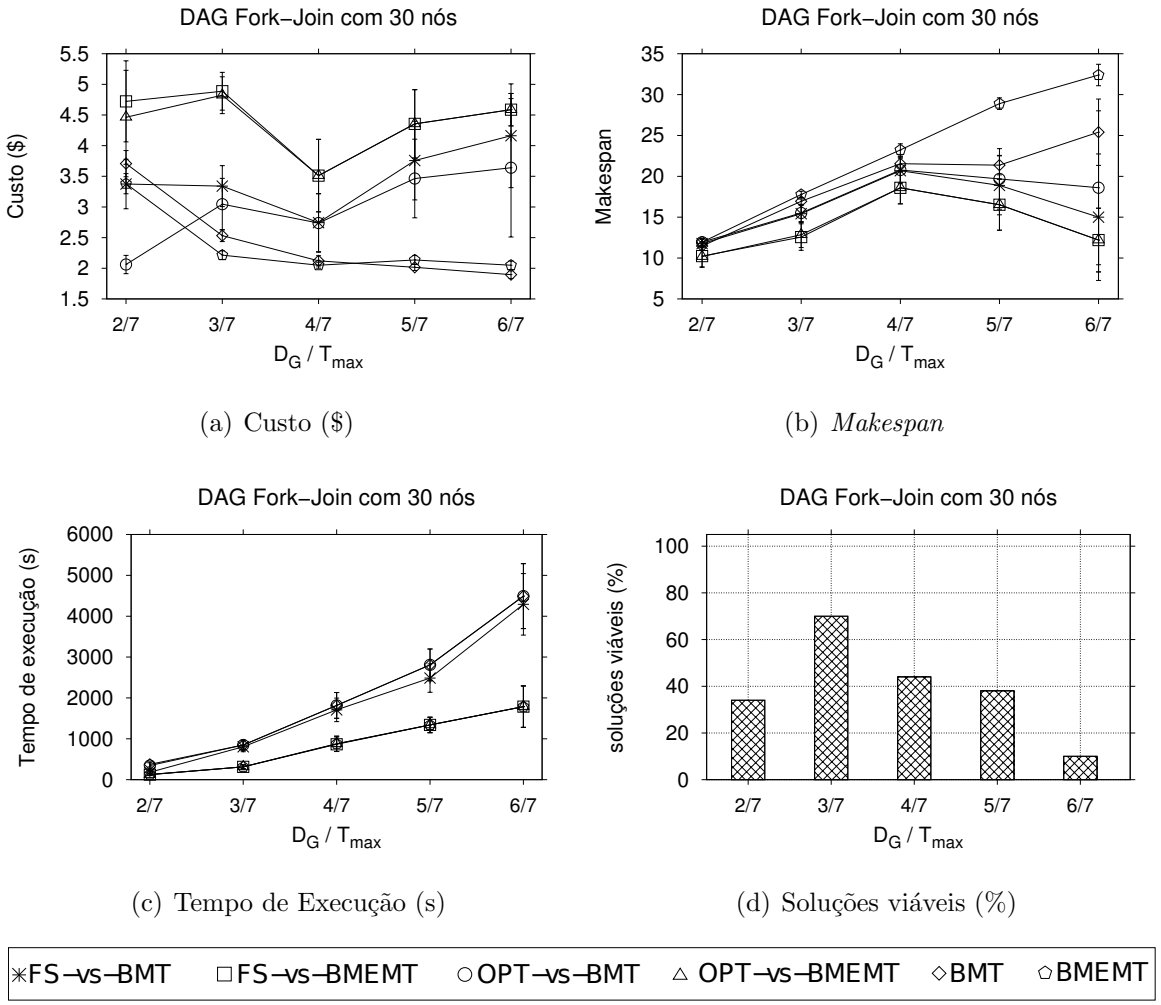
(d) Soluções inviáveis (%)

Figura 5.7: Resultados para o DAG *fork-join* com 30 nós

DAGs *fork-join* com 30 nós da Figura 5.8. Com o aumento do número de nós de 20 para 30, as heurísticas também não gastaram mais memória RAM do que as abordagens OPT e PS. Além disso, conforme mostra os custos monetários no gráfico da Figura 5.8(a) para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$, ambas heurísticas conseguiram reduzir consideravelmente o custo monetário utilizando quantidades semelhantes de memória RAM que OPT e PS.

DAG *CSTEM*

A Figura 5.10 ilustra os resultados da simulação do DAG *CSTEM*. Em comparação com o *fork-join*, o DAG *CSTEM* possui vários níveis de dependências entre os nós (ver Figura 2.3(c)) e, com isso, pode implicar que a solução seja inviável para um *deadline* curto



(a) Custo ($)

(b) *Makespan*

(c) Tempo de Execução (s)

(d) Soluções viáveis (%)

Figura 5.8: Comparação entre as soluções viáveis do DAG *fork-join* com 30 nós

(ou um conjunto de tempo $\mathcal{T}$ pequeno). Em outras palavras, o *solver* não encontrará intervalos de tempo suficientes para todas as dependências do DAG. Por exemplo, como mostra o gráfico da Figura 5.10(d), nenhuma abordagem conseguiu encontrar soluções viáveis para $\mathcal{D}_\mathcal{G} \leq \mathcal{T}_{max} \times 3/7$. Dessa forma, o custo monetário médio para esses *deadlines* foi zero, pois não houve uso de máquinas virtuais.

A propósito, é importante frisar que, para todos os *deadlines*, OPT e PS encontraram soluções antes do limite de tempo estipulado pelas variáveis RT_{BMEMT} e RT_{BMT}, logo as soluções encontradas pela abordagem OPT foram ótimas, ou seja, com custo monetário mínimo. Dessa forma, o tempo de execução do *solver* utilizando a abordagem OPT é maior ou igual ao tempo de execução usando a abordagem PS, como mostram as curvas *OPT-vs-BMEMT*, *PS-vs-BMEMT*, *OPT-vs-BMT* e *PS-vs-BMT* na Figura 5.10(c). Note que,

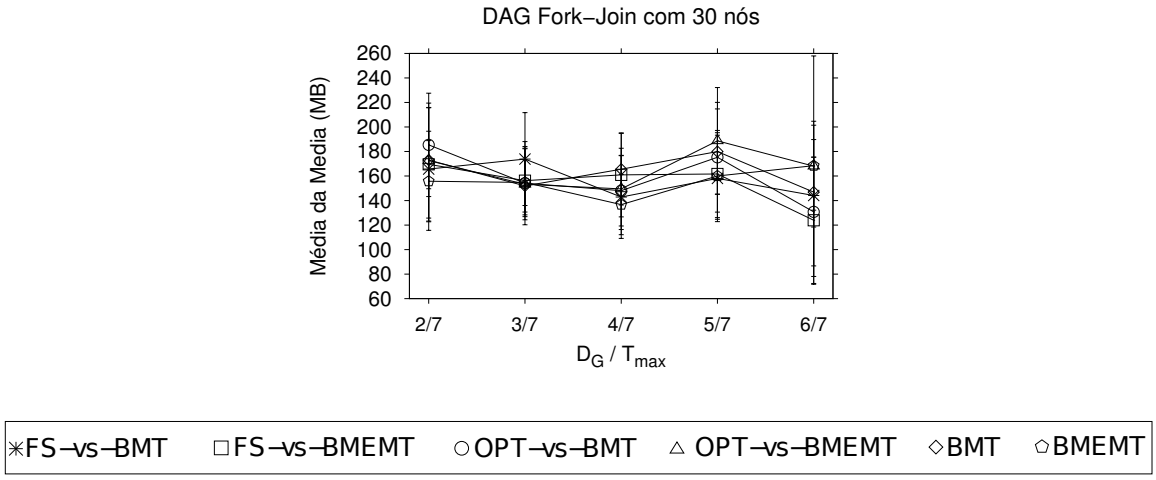


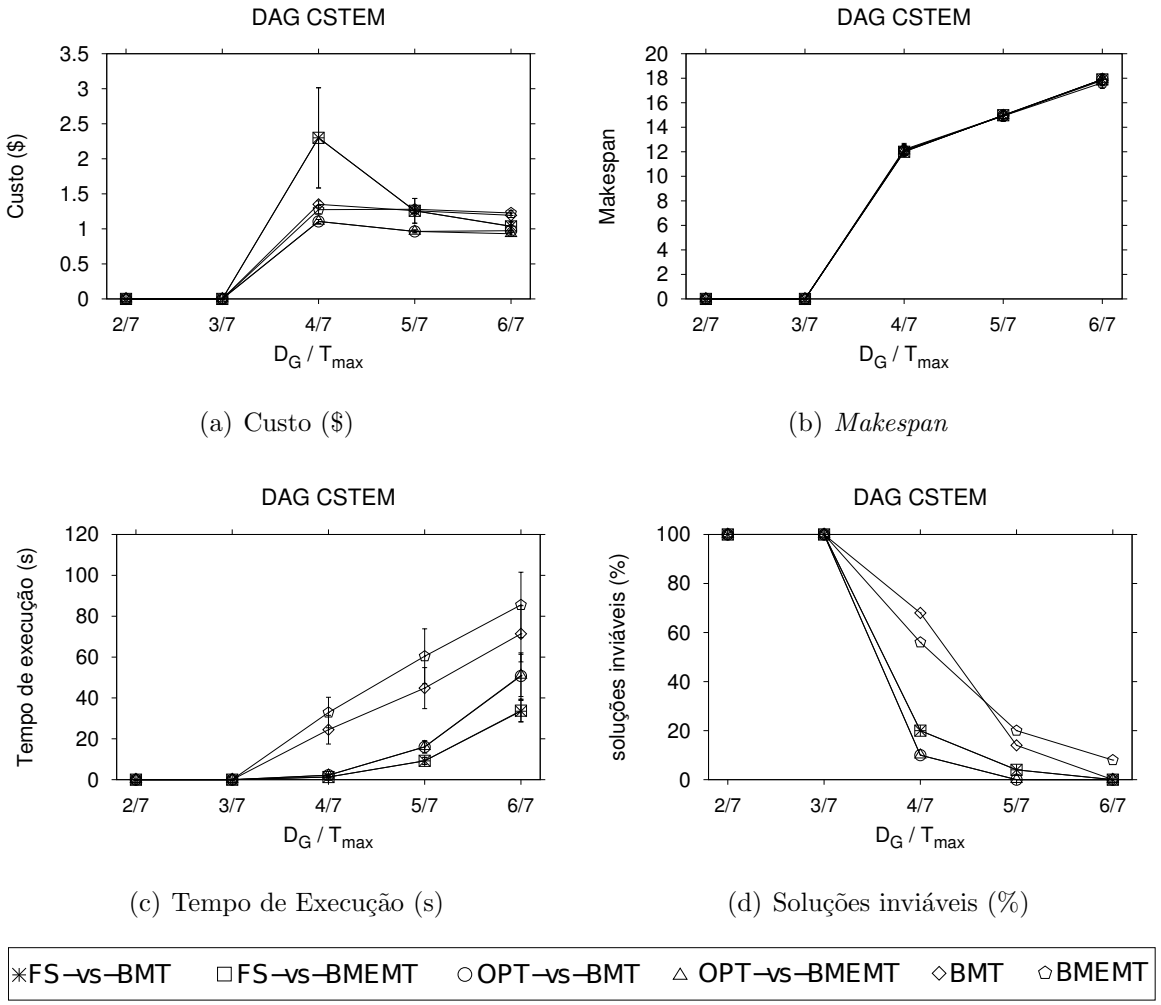
Figura 5.9: Consumo de memória RAM nas simulações DAG *fork-join* com 30 nós

como o DAG *CSTEM* possui vários níveis de dependências e a abordagem OPT encontrou rapidamente as soluções ótimas (o *solver* levou 40 segundos para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$), então podemos deduzir que o conjunto de soluções para esse escalonamento era pequeno para o conjunto de infraestrutura representado pelas Tabelas 5.1 a 5.4 .

Para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 4/7$, BMEMT e BMT foram capazes de encontrar soluções com 55% do custo de PS e aproximadamente 15% mais caro que OPT. Aumentando o *deadline* para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$, a diferença dos custos monetários é aumentada, sendo que as heurísticas conseguiram encontrar soluções com custos médios aproximadamente 25% maior que OPT e PS. É importante lembrar que, para esse DAG, as heurísticas não tiveram um bom desempenho, pois a solução ótima foi facilmente encontrada pela abordagem OPT.

A comparação das soluções viáveis encontradas para cada entrada e por todas as 6 execuções do *solver* está presente na Figura 5.11. Como podemos observar, as curvas $OPT\text{-}vs\text{-}BMEMT$ e $OPT\text{-}vs\text{-}BMT$ permaneceram abaixo de todas as outras curvas na Figura 5.11(a), indicando o custo monetário mínimo encontrado para todos os *deadlines*. Entretanto, para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 4/7$, BMEMT e BMT encontraram soluções com custos monetários maiores que as curvas $PS\text{-}vs\text{-}BMEMT$ e $PS\text{-}vs\text{-}BMT$ (8% e 16%, respectivamente), e 40% mais caros que as curvas da solução ótima ($OPT\text{-}vs\text{-}BMEMT$ e $OPT\text{-}vs\text{-}BMT$, respectivamente). Conforme dito anteriormente, para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$, as soluções das heurísticas foram aproximadamente 25% maiores que OPT e PS. Aliás, houve um aumento do número de soluções viáveis para esse *deadline*, pois o número de elementos do conjunto $\mathcal{T}$ era maior e o conjunto de soluções era menor.

A Figura 5.12 mostra a média da média do uso de memória RAM para simular os DAGs *CSTEM* da Figura 5.11. Para $\mathcal{D}_\mathcal{G} \leq \mathcal{T}_{max} \times 3/7$, o consumo de memória foi praticamente zero, pois o *solver* descobriu rapidamente que as soluções para esses *deadlines*



(a) Custo ($)

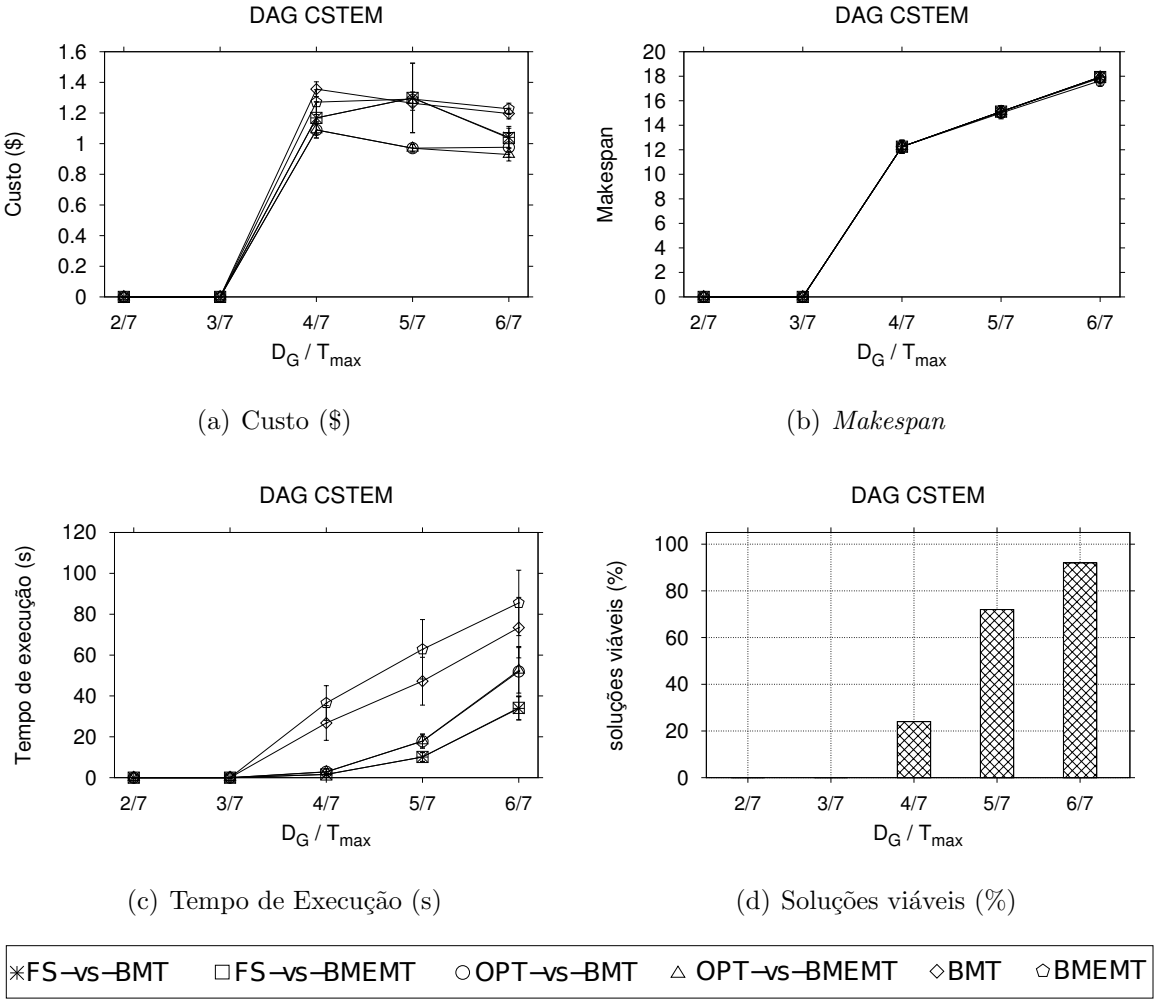
(b) *Makespan*

(c) Tempo de Execução (s)

(d) Soluções inviáveis (%)

Figura 5.10: Resultados para o DAG *CSTEM* com 15 nós

eram inviáveis. Aumentando o *deadline* para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 4/7$, BMEMT e BMT consumiram, em média, mais memória RAM que as abordagens OPT e PS; aproximadamente 58% e 43%, respectivamente. Entretanto, para *deadlines* maiores ($\mathcal{D}_\mathcal{G} \geq \mathcal{T}_{max} \times 5/7$), a execução do *solver* com ambas heurísticas passaram a consumir quantidades semelhantes com as execuções com OPT e PS na média. Portanto, mesmo encontrando soluções com custos monetários altos para esse DAG (pois a solução ótima foi facilmente encontrada pela abordagem OPT), as heurísticas não passaram a consumir mais memória que OPT e PS para *deadlines* longos.



(a) Custo ($)

(b) *Makespan*

(c) Tempo de Execução (s)

(d) Soluções viáveis (%)

Figura 5.11: Comparação entre as soluções viáveis do DAG *CSTEM* com 15 nós

DAG *Montage*

Os resultados para o DAG *Montage* são mostrados na Figura 5.13 e sua topologia é ilustrada na Figura 2.3(a). Para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 2/7$, o número de soluções inviáveis foi alto, como por exemplo, BMEMT teve 94% de soluções inviáveis, enquanto BMT obteve 98%. Por outro lado, OPT e PS também tiveram um número alto de soluções inviáveis, 72% e 76%, respectivamente. Como dito anteriormente, o número de elementos do conjunto $\mathcal{T}$ é pequeno para *deadlines* curtos e, portanto, o *solver* teve dificuldades em encontrar um escalonamento viável com poucos instantes de tempo. Note que, as abordagens OPT e PS encontraram as soluções antes de RT_{BMEMT} e RT_{BMT}, logo os custos médios das curvas $OPT\text{-}vs\text{-}BMEMT$ e $OPT\text{-}vs\text{-}BMT$ são menores ou iguais aos das curvas $PS\text{-}vs\text{-}$

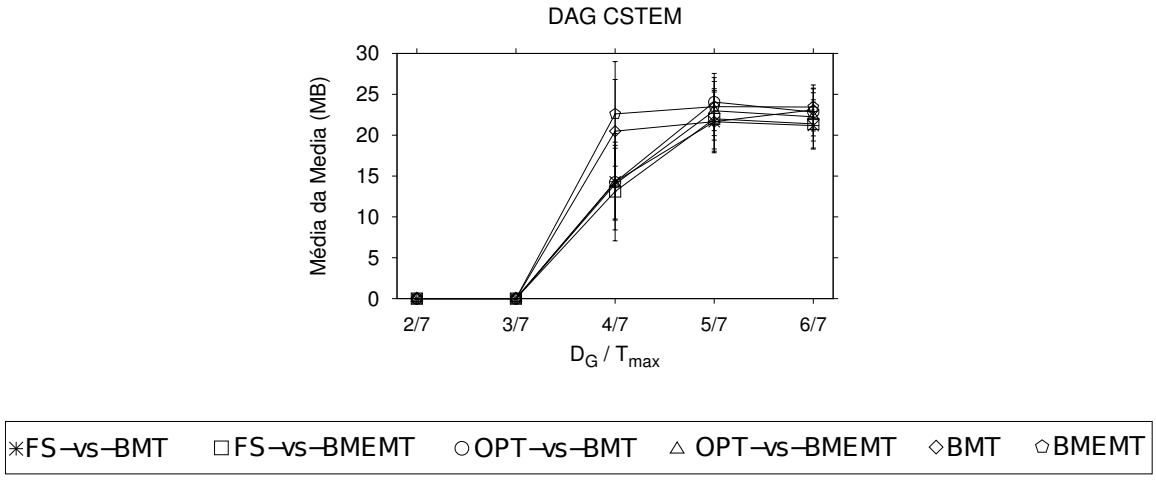


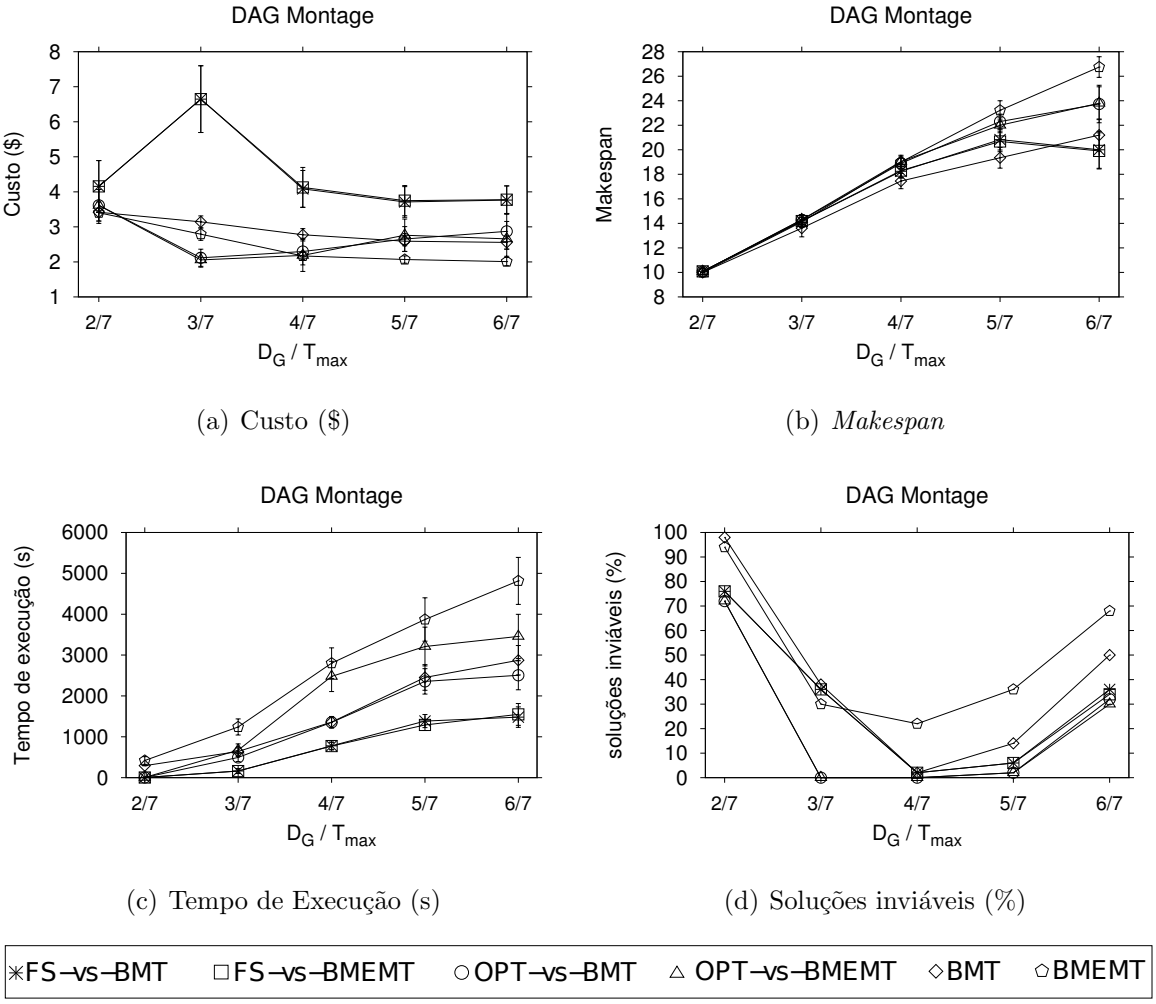
Figura 5.12: Consumo de memória RAM nas simulações DAG *CSTEM* com 15 nós

BMEMT e *PS-vs-BMT*, respectivamente; como ilustra a Figura 5.14(a).

Aumentando o *deadline* para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 3/7$, 100% das soluções encontradas por OPT foram viáveis, enquanto 36% das execuções de PS encontraram soluções inviáveis. Além disso, OPT encontrou soluções antes de RT_{BMEMT} e RT_{BMT}, logo as curvas *OPT-vs-BMEMT* e *OPT-vs-BMT* da Figura 5.14(a) apresentam o custo monetário mínimo para esse *deadline*. A abordagem PS, por sua vez, também encontrou soluções antes do limite de tempo estipulado pelas variáveis RT_{BMEMT} e RT_{BMT} para esse *deadline*, mas o *solver* não conseguiu encontrar, em 36% das simulações, uma solução viável na raiz da árvore de enumeração. Note que, os 64% das simulações restantes de PS encontraram soluções viáveis na raiz da árvore de enumeração com custos monetários altos. A propósito, BMEMT e BMT encontraram soluções com custos monetários 54% mais baratos que PS e 36% mais caros que OPT. Em relação ao *makespan*, todas as abordagens encontraram valores aproximados.

O desempenho das heurísticas tornou-se ainda melhor quando aumentamos o *deadline* para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 4/7$. Assim, as soluções encontradas por BMEMT foram, em média, 0.7% mais caras que as soluções de *OPT-vs -BMEMT* e 47% mais baratas que as soluções de *PS-vs -BMEMT*. Por outro lado, BMT foi capaz de encontrar soluções com custos monetários médios 21% maiores que as soluções de *OPT-vs -BMT* e 44% menores que as soluções de *PS-vs -BMT*. As porcentagens de soluções inviáveis para todas abordagens foram baixas, com exceção de BMT cujo valor foi de 22%. Note que, o tempo de execução das heurísticas não foi alto somente para esse *deadline*, como também para $\mathcal{D}_\mathcal{G} > \mathcal{T}_{max} \times 4/7$.

Para detalhar os *deadlines* $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 5/7$ e $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$, vamos utilizar a Figura 5.14, que apresenta somente os resultados das simulações onde, dada uma entrada, todas



(a) Custo ($)

(b) *Makespan*

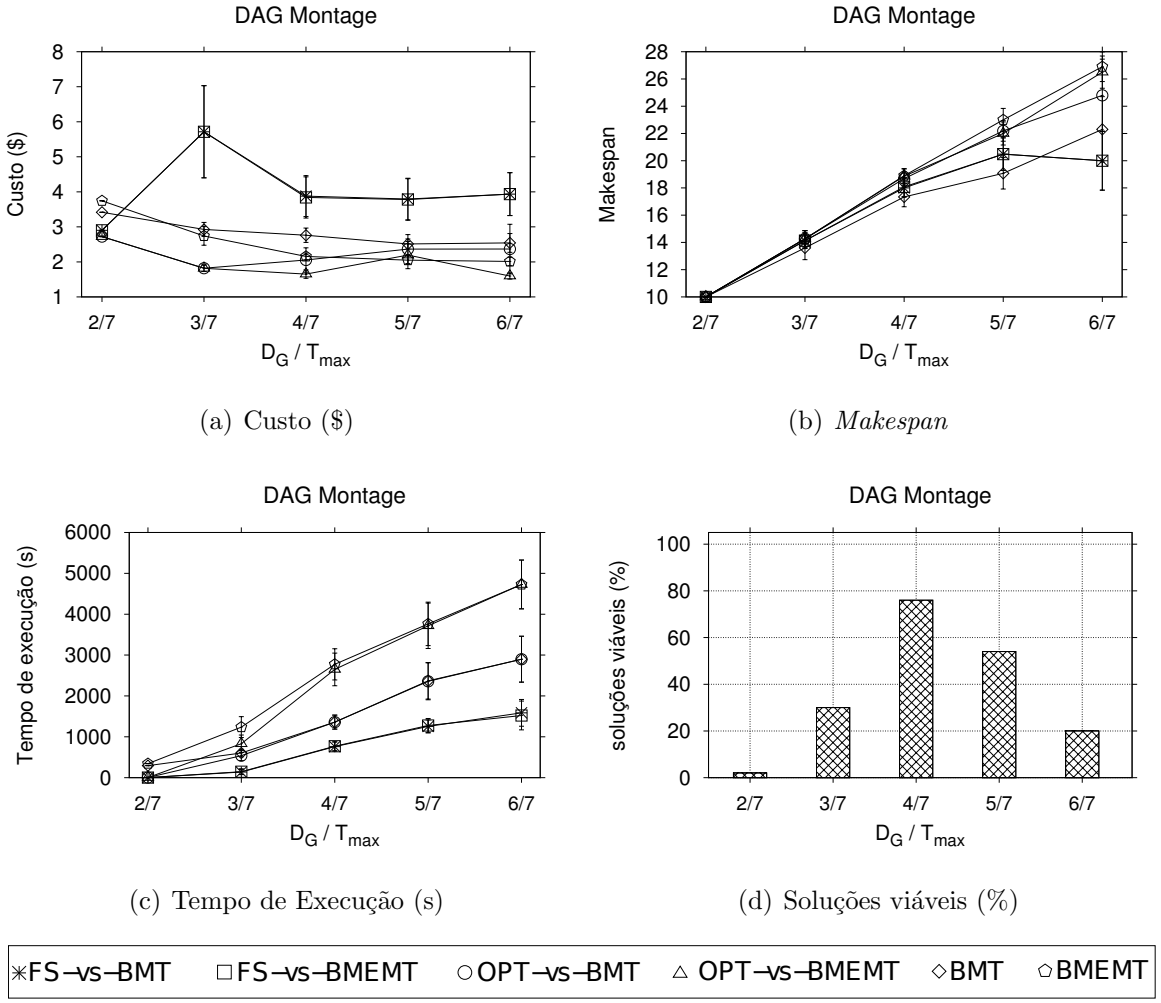
(c) Tempo de Execução (s)

(d) Soluções inviáveis (%)

Figura 5.13: Resultados para o DAG *Montage* com 24 nós

as abordagens encontraram alguma solução viável. Embora o tempo de execução das heurísticas tenha sido alto, elas conseguiram encontrar soluções com custos monetários próximos da abordagem OPT. Por exemplo, para o mesmo tempo de execução, BMEMT foi capaz de encontrar soluções com custos médios 6.5% (para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 5/7$) menores e 26% (para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$) maiores que $OPT\text{-}vs\text{-}BMEMT$.

Por outro lado, as soluções encontras por BMT foram 6% (para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 5/7$) e 7% (para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$) mais caras que $OPT\text{-}vs\text{-}BMT$; considerando também o mesmo tempo de execução. No entanto, ambas heurísticas encontraram soluções mais baratas que PS, como por exemplo, BMEMT foi capaz de encontrar soluções 49% mais baratas que $PS\text{-}vs\text{-}BMEMT$, enquanto que a heurística BMT encontrou soluções 35% mais baratas que $PS\text{-}vs\text{-}BMT$. Portanto, para *deadlines* longos, as heurísticas também



(a) Custo ($)

(b) *Makespan*

(c) Tempo de Execução (s)

(d) Soluções viáveis (%)

Figura 5.14: Comparação entre as soluções viáveis do DAG *Montage* com 24 nós

conseguiram encontrar soluções com custos monetários reduzidos.

O consumo médio de memória RAM utilizada pelo *solver* para simular os DAGs da Figura 5.14 está ilustrado na Figura 5.15. Para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 2/7$, o *solver* consumiu mais memória RAM usando as heurísticas do que usando as abordagens OPT e PS. Para os valores de *deadlines* entre $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 3/7$ a $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 5/7$, as heurísticas consumiram quantidades de memória semelhantes a OPT e PS. No entanto, para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$, BMT consumiu 12% a menos de memória que *PS-vs-BMT* e 41% a menos que *OPT-vs-BMT* na média.

Como podemos observar nas Figuras 5.13(d) e 5.14(d), de modo geral, o *solver* teve dificuldades para encontrar soluções viáveis para *deadlines* curtos e longos. No primeiro caso, o conjunto de tempo $\mathcal{T}$ era pequeno demais para escalonar todos os nós do DAG

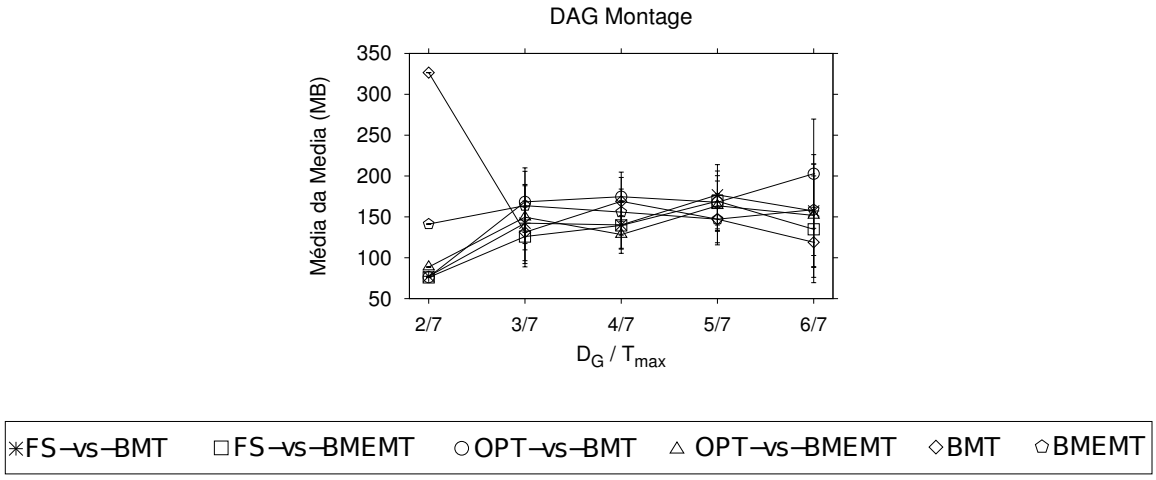


Figura 5.15: Consumo de memória RAM nas simulações DAG *Montage* com 24 nós

Montage e, portanto, o *solver* descobriu rapidamente que a maioria dos escalonamentos eram inviáveis. Já no segundo caso, o conjunto de tempo $\mathcal{T}$ era grande o suficiente para o *solver* analisar todas as possibilidades de escalonamento e escolher a melhor solução possível em tempo hábil; assim, o tempo de execução do escalonamento foi alto. Esse segundo caso é tratado na próxima seção.

Análise Geral dos Resultados

Os resultados apresentados nesta seção sugerem que as heurísticas BMEMT e BMT podem ser eficazes para reduzir os custos monetários dos escalonamentos com *deadlines* longos, quando se tem o mesmo tempo de execução do *solver* para a abordagem ótima (OPT). Em outras palavras, para *deadlines* longos, as heurísticas tiveram um bom desempenho quando comparados aos resultados de OPT e PS. Assim, quando a execução via abordagem OPT ou PS tende a demorar, podemos utilizar as heurísticas para acelerar o processo de escalonamento e, além disso, minimizar os custos monetários.

Uma característica muito interessante da heurística BMEMT foi encontrar soluções viáveis com valores altos para o *makespan* do escalonamento. De acordo com a sua definição, BMEMT tenta maximizar, respeitando um *deadline*, o espaço de tempo disponível para escalonar todos os nós do DAG. Aliás, do ponto vista do provedor de SaaS, não é interessante utilizar máquinas virtuais caras e rápidas para executar DAGs cujos valores dos *deadlines* sejam altos. Dessa forma, por exemplo, o provedor de SaaS pode utilizar a heurística BMEMT para escalonar DAGs com *deadines* longos e/ou priorizar as execuções dos DAGs com *deadlines* curtos com a abordagem OPT. Além disso, de modo geral, custos mais elevados não implicam em *makespans* proporcionalmente mais baixos.

Em outras palavras, escolher máquinas virtuais erradas pode encarecer o escalonamento sem melhorar o *makespan*; como mostram, por exemplo, a comparação dos resultados da heurística BMT com OPT e PS nas Figuras 5.4(a) e 5.4(b). Portanto, uma sugestão para o provedor de SaaS é que execute as heurísticas BMEMT e BMT e a abordagem OPT ao mesmo tempo, em seguida, utilize a primeira solução viável obtida.

Enfim, as abordagens apresentadas nesta seção podem fornecer uma base para o provedor de SaaS poder negociar SLA com seus clientes. Entretanto, *deadlines* longos aumentam não apenas o número de intervalos discretos do conjunto $\mathcal{T} = \{1, \ldots, \mathcal{D}_\mathcal{G}\}$, como também a complexidade do escalonamento; portanto, dificultando o *solver* de encontrar uma boa solução viável inteira em tempo hábil. Assim, simulações com um aumento da granularidade da discretização da linha do tempo foram realizadas apenas na abordagem ótima, cujos resultados são mostrados na próxima seção.

5.2.2 Simulações com λ variável

Nesta seção, vamos apresentar os resultados das simulações da abordagem descrita na Seção 4.3.4: o uso de diferentes níveis de discretização da linha do tempo no programa linear inteiro. Assim, em cada simulação, 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, usando apenas *abordagem ótima* (ver Seção 4.3.2); ou seja, sem relaxar as variáveis binárias x e y. Os DAGs utilizados nestas simulações foram: *fork-join* com 30 e 50, *Montage* e *LIGO-1*. É importante frisar que não aplicamos a variação de λ nos DAGs *CSTEM* e *fork-join* com 10 e 20 nós, pois, conforme apresentado na seção anterior, o tempo de execução do *solver* para solucionar esses DAGs foi curto.

O parâmetro (ou dimensão) *tempo* é o principal gargalo que impede a escalabilidade do nosso escalonador. 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 $\mathcal{T}$ cresce rapidamente quando tentamos escalonar DAGs com grande número de nós e dependências. Em outras palavras, a principal variável do programa linear inteiro, descrito na Seção 4.2, 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 será 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, aumentando a discretização da linha do tempo, podemos reduzir o número de elementos do conjunto de tempo $\mathcal{T}$ e, consequentemente, a dimensão tempo

da matriz binária. Portanto, com um conjunto $\mathcal{T}$ de tamanho reduzido, o tempo de execução do *solver* tende a diminuir.

Por outro lado, como o número de elementos de $\mathcal{T}$ foi reduzido, 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 para atender o *deadline*. Isto é, a redução do tamanho do conjunto $\mathcal{T}$ gera uma penalidade no escalonamento, pois quando aumentamos a discretização da linha do tempo, podemos desperdiçar núcleos virtualizados (alugados ou não) 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 uma unidade de tempo em $\lambda = 2$. Em outras palavras, haverá o custo monetário adicional de uma unidade de tempo não utilizada na máquina virtual em que o nó será escalonado em $\lambda = 2$; aumentando, então, o custo monetário do escalonamento. Portanto, para cada valor de λ, o *solver* dará uma estimativa[4] do valor monetário do escalonamento.

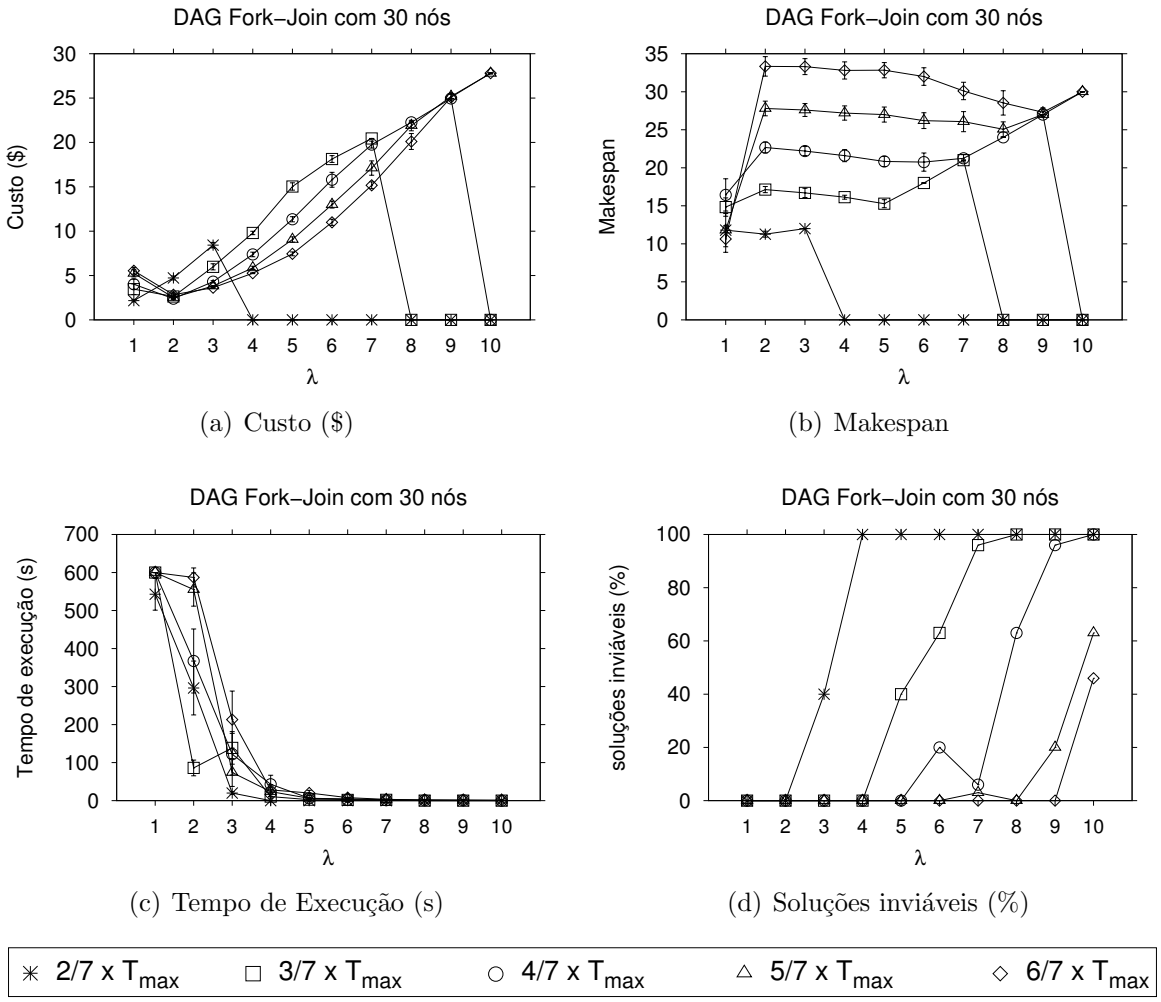
Conforme dito na Seção 4.3.4, há um claro *trade-off* entre utilizar intervalos curtos de tempo para obter um escalonamento mais preciso ou usar intervalos de tempo longos para diminuir o tempo de execução do escalonamento. Os gráficos a seguir foram plotados de acordo com a média das 30 simulações de cada DAG e em 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.

DAG *fork-join* com 30 nós

A Figura 5.16 mostra resultados das simulações para o DAG *fork-join* com 30 nós. Para $\lambda = 1$ e $\lambda = 2$, o *solver* encontrou soluções para todos os $\mathcal{D}_\mathcal{G}$s, ou seja, 0% de soluções inviáveis. Para $\lambda = 2$ e $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 2/7$, o escalonamento foi 1.83 vezes mais rápido, mas gerou um aumento de duas vezes no custo monetário, quando comparado às soluções para o mesmo $\mathcal{D}_\mathcal{G}$ e $\lambda = 1$. No entanto, mantendo $\lambda = 2$ e aumentando o *deadline*, o escalonamento tornou-se mais rápido 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 quando comparado às soluções para o mesmo $\mathcal{D}_\mathcal{G}$ e $\lambda = 1$.

Note que, para $\lambda = 1$, a solução com menor *deadline* ($\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 2/7$) teve o menor

[4]Esta estimativa contradiz o modelo de tarifação *pay-as-you-go* da computação em nuvem (descrito na Seção 2.1.1), 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 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 contabilidadas no custo monetário do escalonamento, para que possamos descontar esse custo adicional. No entanto, essa tarefa está como trabalhos futuros.



(a) Custo ($)

(b) Makespan

(c) Tempo de Execução (s)

(d) Soluções inviáveis (%)

Figura 5.16: Resultados para o DAG *fork-join* com 30 nós

custo monetário médio, pois o escalonamento terminou antes do limite de tempo de 10 minutos; ou seja, o *solver* conseguiu, para algumas entradas, alcançar a solução ótima, e para as demais, alcançar níveis mais profundos da árvore de enumeração e encontrar soluções viáveis com custos monetários baixos. No entanto, todas as soluções encontradas para os demais *deadlines* em $\lambda = 1$, o *solver* procurou apenas na raiz da árvore de enumeração. Aliás, para $\mathcal{D}_\mathcal{G} \geq \mathcal{T}_{max} \times 5/7$, algumas das soluções foram encontradas pelas heurísticas internas[5] do *CPLEX*, antes mesmo de iniciar o método de enumeração *Branch & Cut*. Isso explica o porquê da solução dos demais *deadlines* ser mais cara que a solução de $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 2/7$. Em outras palavras, 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*

[5]Heurísticas de código fechado.

conseguiu avançar na árvore.

Por outro lado, para $\lambda = 2$, o *solver* conseguiu achar soluções antes do limite de tempo para a maioria dos *deadlines* e, como esperado, o custo monetário médio das soluções de $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 2/7$ aumentaram (devido aos desperdícios de recursos não utilizados). Entretanto, para os demais *deadlines*, o custo monetário médio reduziu. Essa redução do custo 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 a procura na árvore; porém, em $\lambda = 2$, o *solver* conseguiu alcançar níveis mais profundos da árvore de enumeração e, portanto, encontrar 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 com custos menores e sem desperdício de recursos, como explicado anteriormente. Note que colocamos a expressão *dar continuidade* entre aspas, pois a árvore de $\lambda = 1$ pode não ser a mesma de $\lambda = 2$.

É importante frisar que uma conjectura tirada dos resultados de todas as simulações onde variamos o fator λ é: seja um DAG $\mathcal{G}$ qualquer 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 $\lambda_\mathcal{B}$, tal que $\lambda_\mathcal{A} < \lambda_\mathcal{B}$, $\lambda_\mathcal{A} \neq \lambda_\mathcal{B}$ e $\lambda_\mathcal{A}, \lambda_\mathcal{B} \in \mathbb{N}^+$, 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 $\lambda_\mathcal{B}$. Aliás, essa conjectura era esperada antes mesmo de iniciarmos estas simulações.

Quando aumentamos a discretização do tempo para $\lambda = 3$, o *solver* começou a desperdiçar recursos para todos os escalonamentos com $\mathcal{D}_\mathcal{G} \geq \mathcal{T}_{max} \times 2/7$, implicando no aumento do custo monetário médio para todos os *deadlines*, como mostra a Figura 5.16(a). Lembrando que para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 2/7$ esse aumento começou 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.

Além disso, o aumento de λ, acompanhado de redução de $\mathcal{D}_\mathcal{G}$, diminui a linha temporal do escalonamento devido ao aumento da granularidade da discretização do tempo e, portanto, aumenta 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; observação também válida para os

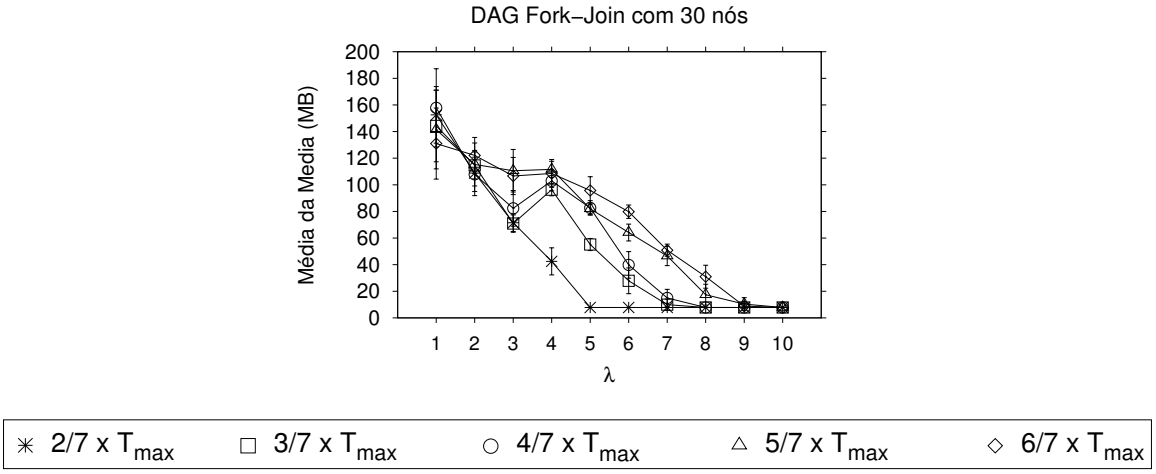


Figura 5.17: Consumo de memória RAM nas simulações do DAG *fork-join* com 30 nós

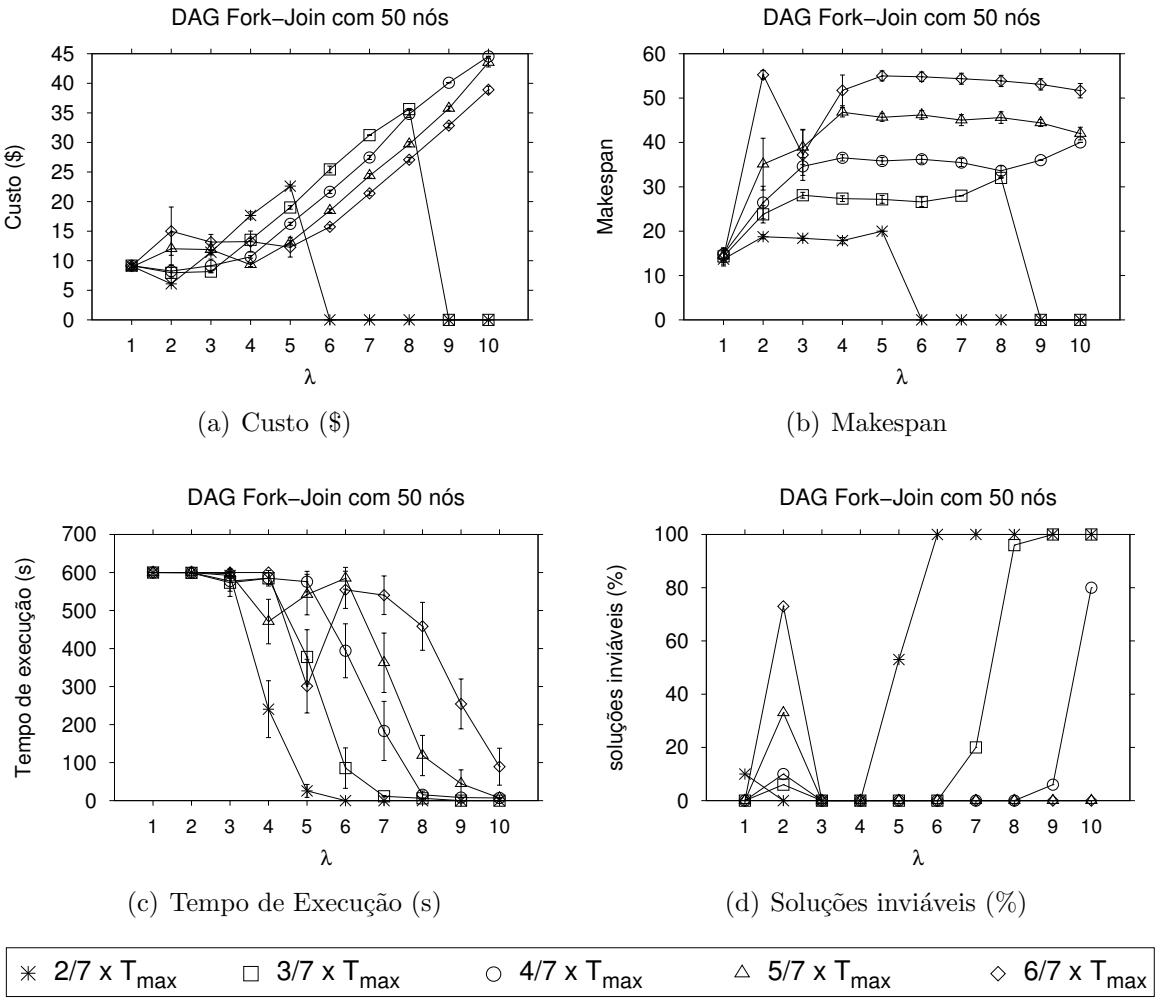
resultados dos demais DAGs.

Conforme ilustra a Figura 5.16(d), o aumento da granularidade da discretização do tempo também gerou um aumento no *makespan* na maioria dos casos. Para *deadlines* curtos, o *makespan* não teve um aumento significativo. Por exemplo, para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 2/7$, o *makespan* teve pouca alteração com a variação de λ, mas para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 3/7$, o *makespan* aumentou 13% de $\lambda = 1$ para $\lambda = 2$ e 11% de $\lambda = 1$ para $\lambda = 3$. Por outro lado, para *deadlines* longos, o *makespan* aumentou consideravelmente, como por exemplo, para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$, o valor do *makespan* em $\lambda = 2$ foi três vezes maior que $\lambda = 1$. Esse aumento, porém, é acentuado apenas no primeiro aumento da granularidade do tempo, ou seja, com poucas variações nos demais níveis de discretização ($\lambda \geq 3$).

A Figura 5.17 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 $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 2/7$ em $\lambda = 5$.

DAG *fork-join* com 50 nós

Ao aumentar o número de nós do DAG *fork-join* para 50 nós, o tempo de execução do *solver* ficou cada vez menor com o aumento da discretização do tempo, como mostra a Figura 5.18. É importante lembrar que, antes de solucionar o programa linear inteiro, o



(a) Custo ($)

(b) Makespan

(c) Tempo de Execução (s)

(d) Soluções inviáveis (%)

Figura 5.18: Resultados para o DAG *fork-join* com 50 nós

CPLEX executa inicialmente heurísticas próprias cujos resultados podem ser utilizados para dar início ao método *Branch & Cut*. Dessa maneira, em todas as simulações para todos os *deadlines* em $\lambda \leq 2$, essas heurísticas gastaram praticamente os 10 minutos para encontrar alguma solução viável. No entanto, em alguns casos, sobraram alguns segundos para o *solver* iniciar o *Branch & Cut*, dos quais poucos conseguiram finalizar a busca na raiz da árvore de enumeração. Em outras palavras, nenhuma solução ótima foi encontrada em $\lambda \leq 2$, pois os resultados encontrados pelo *solver* vieram ora das heurísticas do *CPLEX* ora da busca (finalizada ou não) na raiz da árvore de enumeração. Note que, em $\lambda = 2$, a porcentagem de soluções inviáveis foi alta para *deadlines* altos, pois as heurísticas proprietárias gastaram muito tempo para encontrar (e na maioria dos casos não encontraram) alguma solução viável. Como consequência, as heurísticas do *CPLEX*

não conseguiram passar informações suficientes para que o processo de *Branch & Cut* fosse iniciado com um ponto de partida (ou seja, uma direção) na busca da solução ótima no conjunto de soluções. Portanto, devido às heurísticas serem de código fechado, fica difícil de concluirmos concretamente o porquê do aumento ou da redução do custo monetário das soluções encontradas de $\lambda = 1$ para $\lambda = 2$ de todos os *deadlines*. Provavelmente, se tivéssemos aumentado o tempo limite de execução do *solver*, então a maioria das soluções em $\lambda \leq 2$ seriam encontradas na árvore de enumeração. Entretanto, o objetivo desta seção é avaliar a variação do valor de λ com intuito de acelerar o *Branch & Cut*, como mostra a Figura 5.18(c) em $\lambda \geq 3$.

Assim, em $\lambda = 3$, as heurísticas do *CPLEX* conseguiram encontrar rapidamente uma solução viável para todos os *deadlines* e, portanto, a busca na árvore de enumeração foi inicializada mais cedo que nas simulações em $\lambda = 2$. Aliás, o *solver* encontrou 100% de soluções viáveis para todos os *deadlines*, como mostra a Figura 5.18(d). Embora o *solver* também tenha conseguido alcançar níveis mais profundos da árvore para as simulações de todos os *deadlines* em $\lambda = 3$, o tempo de execução manteve-se alto na média. Além disso, houve um aumento no custo monetário nas soluções encontradas para os *deadlines* curtos ($\mathcal{D}_\mathcal{G} \leq \mathcal{T}_{max} \times 4/7$), devido à influência do aumento da granularidade da linha do tempo no desperdício de recursos.

A redução do tempo de execução do *solver* começou a ocorrer somente quando $\lambda \geq 4$. Por exemplo, para $\lambda = 7$, o tempo de execução médio teve uma redução de 70% para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 4/7$, 40% para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 5/7$ e 10% para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$. Aumentando a granularidade do tempo para $\lambda = 8$, essas reduções ficaram ainda maiores na média; 97.5% para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 4/7$, 80% para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 5/7$ e 24% para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$. Por outro lado, os custos monetários médios das soluções encontradas para essas configurações ficaram mais caros; uma média de 3 vezes maior em $\lambda = 7$ e 3.5 em $\lambda = 8$ para cada *deadline* comparado acima. Além disso, o número de soluções inviáveis manteve-se baixo para valores altos de λ, como mostra a Figura 5.18(d). Para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 2/7$, o aumento da discretização do tempo foi possível até $\lambda \leq 5$, pois o tamanho do conjunto $\mathcal{T}$ reduziu de tal forma que o escalonamento não era mais possível. Note que, o aumento da porcentagem de soluções inviáveis ocorreu primeiro para os *deadlines* curtos e paulatinamente foi ocorrendo, em ordem, para os demais *deadlines*.

A Figura 5.19 mostra o consumo de memória RAM usada pelo *solver* para solucionar o escalonamento dos DAGs *fork-join* com 50 nós. Como podemos analisar, houve uma leve redução no uso de memória RAM com aumento da granularidade da linha do tempo. Além disso, como comentado anteriormente, a maioria das soluções em $\lambda = 2$ foram encontradas por meio das heurísticas do *CPLEX* cujas execuções parecem consumir mais memória RAM. De modo geral, para todos os *deadlines*, as heurísticas privadas do *CPLEX* encontraram, em $\lambda = 1$, soluções com custos monetários e *makespans* baixos e, além disso,

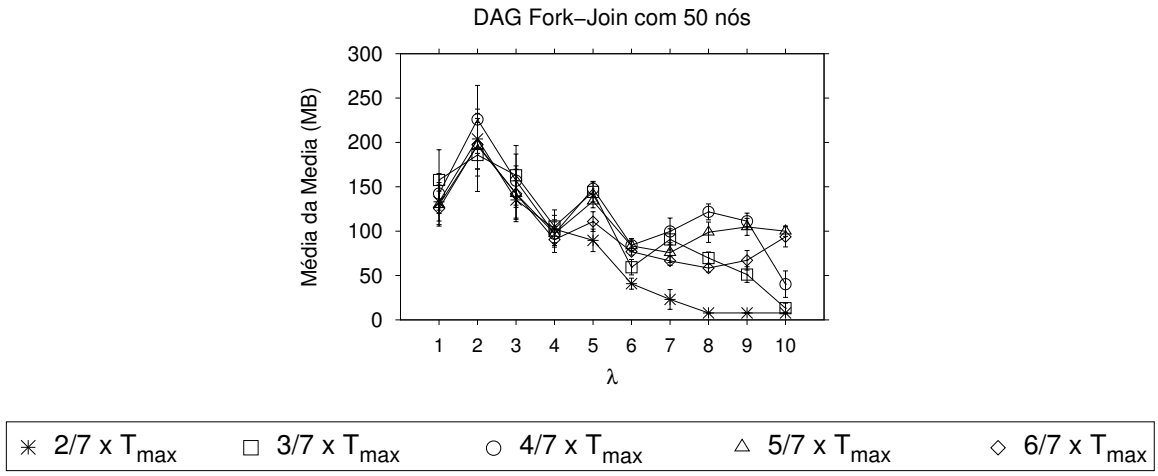


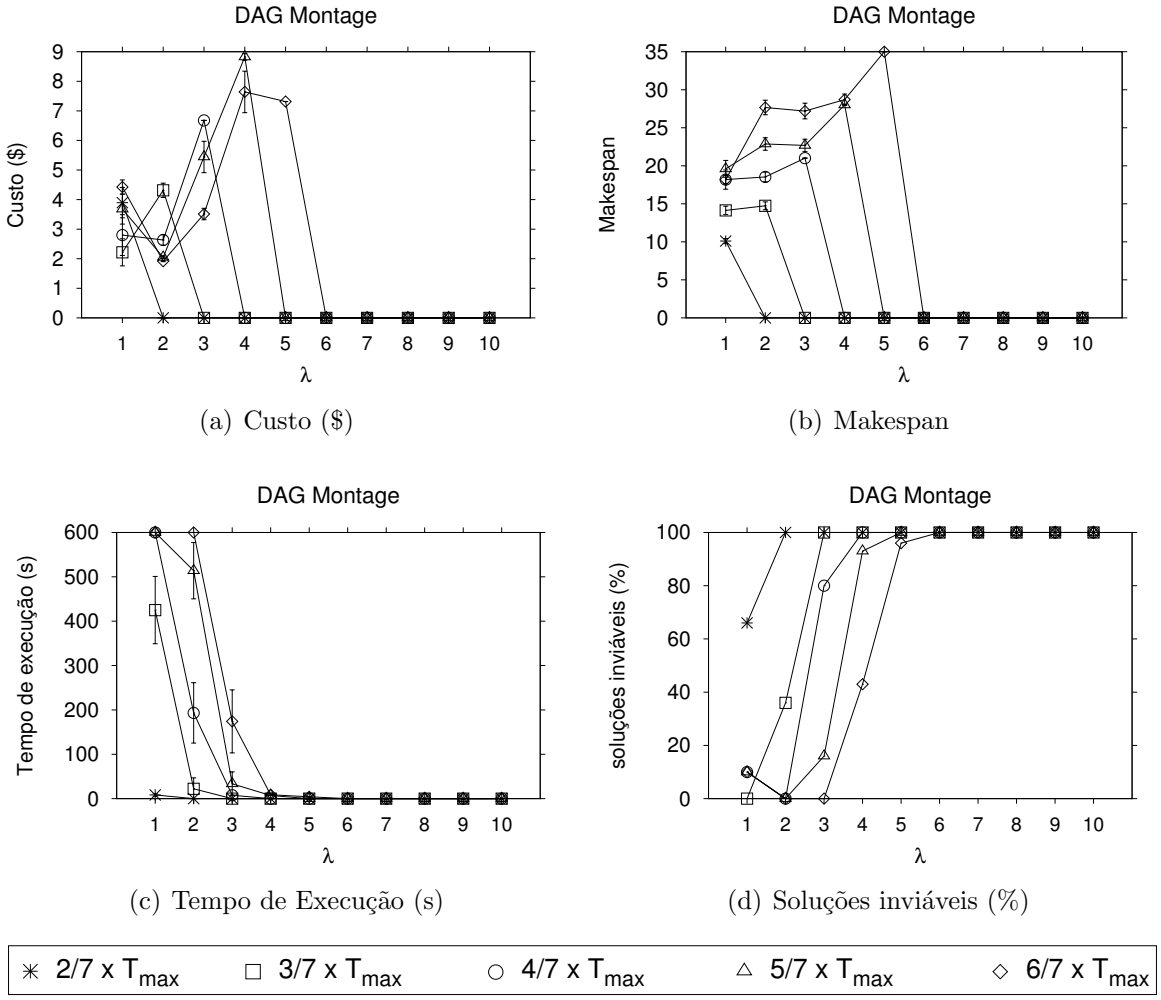
Figura 5.19: Consumo de memória RAM nas simulações do DAG *fork-join* com 50 nós

com 0% de soluções inviáveis. Entretanto, mesmo encontrando soluções piores (com custos e *makespans* elevados), conseguimos diminuir consideravelmente o tempo de execução do *solver* com aumento de λ.

DAG *Montage*

Os resultados das simulações para o DAG *Montage* são mostrados na Figura 5.20. Para $\lambda = 1$, o *solver* conseguiu encontrar soluções viáveis para as 30 simulações somente quando $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 3/7$; em outras palavras, esse *deadline* teve 0% de soluções inviáveis em $\lambda = 1$, como mostra a Figura 5.20(d). No entanto, entre as soluções viáveis, 80% foram ótimas (com custos mínimos), pois foram encontradas antes do limite de tempo de 10 minutos. Os 20% restantes, por sua vez, foram encontrados em níveis menos profundos da árvore de enumeração, pois a execução do *solver* teve que ser 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 5.20(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* fosse escalonado com sucesso. Aliás, como consequência 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 5.20(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,



(a) Custo ($)

(b) Makespan

(c) Tempo de Execução (s)

(d) Soluções inviáveis (%)

Figura 5.20: Resultados para o DAG *Montage* com 24 nós

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 $\mathcal{D}_\mathcal{G}$ e $\lambda = 1$. 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 re-

dução 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 um 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 enumeração, 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 outras palavras, ao solucionar um programa linear inteiro, as heurísticas do *CPLEX* podem produzir uma ou mais soluções viáveis, satisfazendo todas as restrições e todas as condições de integralidade, mas sem alguma indicação sobre ter encontrado a melhor solução possível. 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 $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$. Aliás, 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.

O consumo médio de memória RAM usado pelo *solver* para simular o DAG *Montage* está ilustrado na Figura 5.21. 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

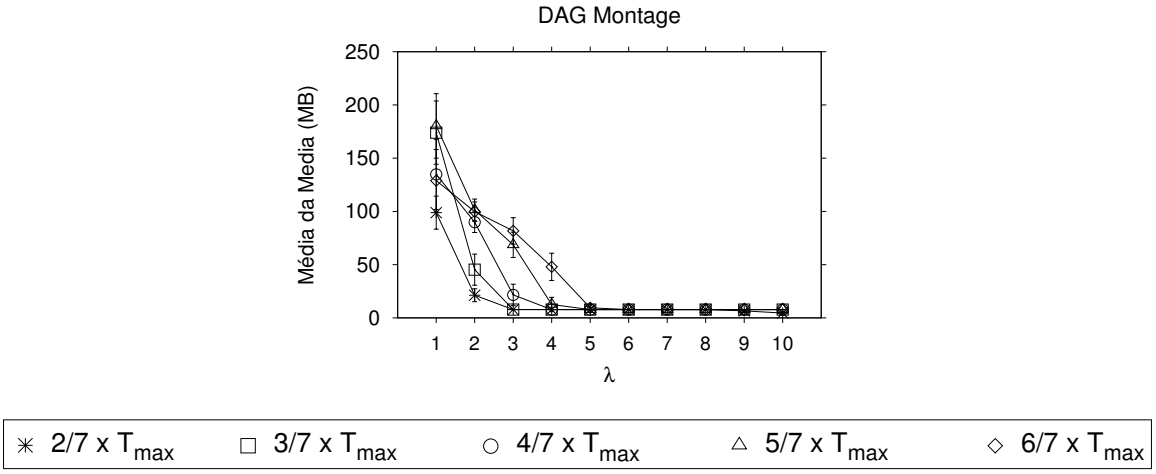


Figura 5.21: Consumo de memória RAM nas simulações do DAG *Montage* com 24 nós

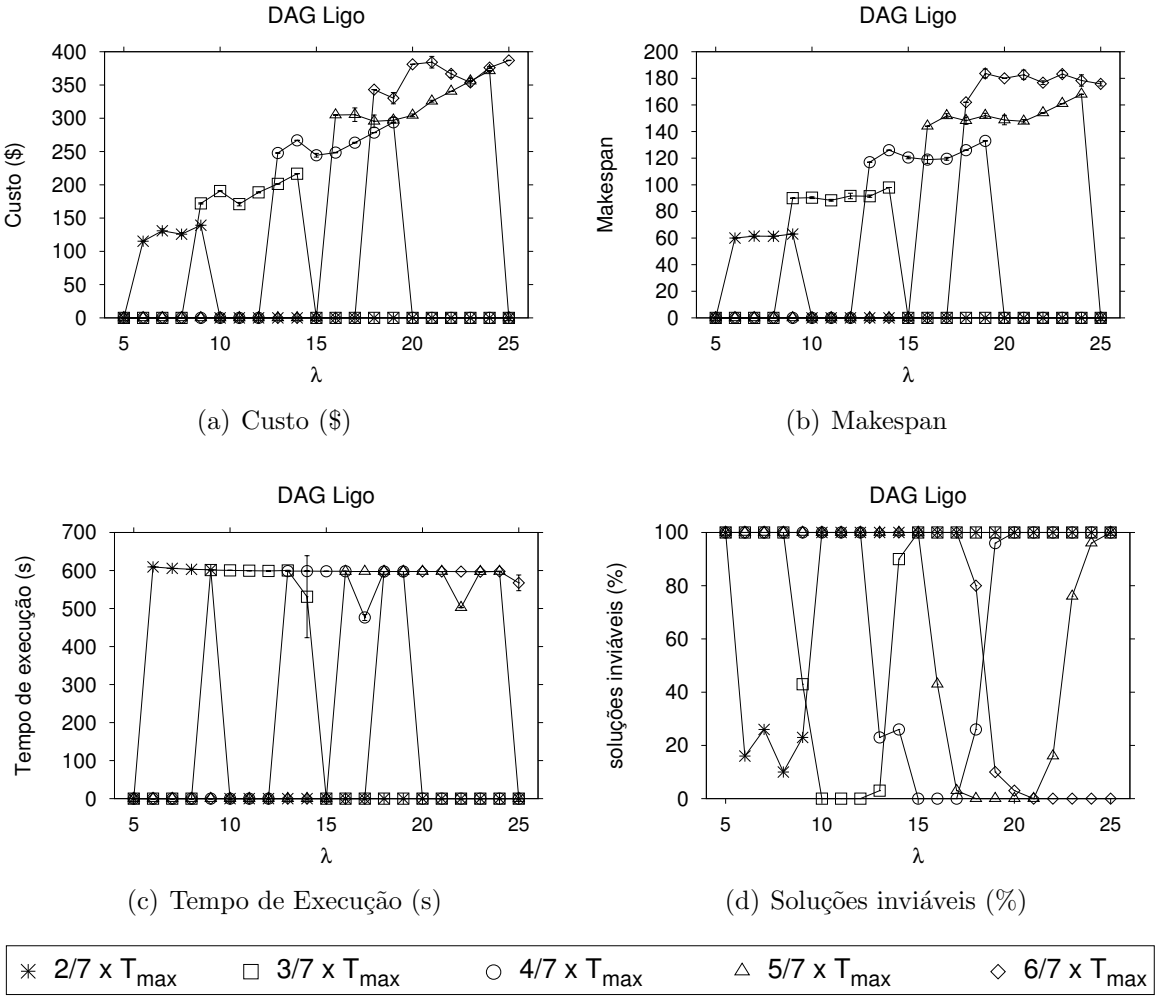
$\lambda = 2$, o uso de memória do *solver* foi reduzido, em média, 22.5%, enquanto de $\lambda = 2$ para $\lambda = 2$ essa redução foi de 18% em média.

DAG *LIGO-1*

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-1* na Figura 5.22, cuja topologia está ilustrada na Figura 2.3(d). Entretanto, devido a grande quantidade de nós (168) e dependência entre nós, tivemos que variar o valor de λ de acordo com o valor de $\mathcal{D}_\mathcal{G}$, pois com o conjunto $\lambda = \{1, 2 \ldots, 9, 10\}$, o *solver* teve dificuldades de encontrar soluções viáveis em 10 minutos de simulação para todos os *deadlines*. Assim, simulamos o DAG *LIGO-1* 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 os seguintes valores inteiros para λ: $\{5, 6, \ldots, 14, 15\}$;
- Para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 2/7$, usamos os seguintes valores inteiros para λ: $\{10, 11, \ldots, 19, 20\}$;
- Para $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 5/7$ e $\mathcal{D}_\mathcal{G} = \mathcal{T}_{max} \times 6/7$, usamos os seguintes valores inteiros para λ: $\{15, 16, \ldots, 24, 25\}$

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-1* e, como podemos observar na Figura 5.22(d), nenhum *deadline* teve soluções viáveis para todos os



(a) Custo ($)

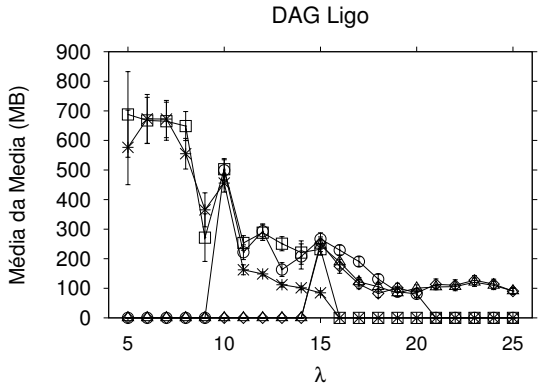
(b) Makespan

(c) Tempo de Execução (s)

(d) Soluções inviáveis (%)

Figura 5.22: Resultados para o DAG *LIGO-1* com 168 nós

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 outras palavras, as soluções viáveis para esses tipos de DAGs 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* teve que utilizar os 10 minutos para achar alguma solução viável. 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.



(a) Consumo médio

Figura 5.23: Consumo de memória RAM nas simulações do DAG *LIGO-1* com 168 nós

Contendo 168 nós, o *LIGO-1* é o maior DAG simulado neste tabalho; contra os 10, 20 e 30 nós do *fork-join*, os 15 do *CSTEM* e os 24 do *Montage*. Dessa forma, em comparação aos DAGs menores, o aumento do multiplicador de $\mathcal{T}_{max}$ no *LIGO-1*, 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-1*, 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 pode 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 5.23 apresenta o consumo médio das simulações do DAG *LIGO-1*. 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 (em *gigabytes*).

Análise Geral dos Resultados

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 a complexidade computacional do problema pode crescer exponencialmente. Dessa forma, o *solver* pode não encontrar uma solução viável em tempo hábil. Portanto, os resultados apresentados nesta seção 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 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, junto com as heurísticas BMEMT e BMT, a abordagem do aumento da granularidade da linha temporal também pode auxiliar o provedor de SaaS a negociar os SLAs com seus clientes.

Como dito anteriormente, haveria 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, conseguiríamos corrigir esse erro inserido e visualizar os instantes de tempo não utilizados em $\lambda = 1$, mas usados $\lambda = 5$. Além disso, poderíamos utilizar esses instantes de tempo não utilizados para escalonar nós de outros DAGs, tornando o escalonador em um de múltiplos DAGs. Entretanto, essa heurística está como trabalhos futuros.

5.3 Considerações Finais

Neste capítulo foi apresentado o processo de simulação do escalonador de *workflows* em nuvens proposto, bem como as várias simulações realizadas utilizando o *IBM ILOG CPLEX Optimizer*, a fim de avaliar o comportamento da programação linear inteira desenvolvida neste trabalho. Realizamos simulações com vários DAGs que representam aplicações do mundo real, tais como: *CSTEM*, *Montage*, *LIGO-1* e DAG *fork-join* com 10, 20 e 30 nós. Primeiramente, apresentamos os resultados das heurísticas BMEMT e BMT, as quais foram desenvolvidas com o intuito de reduzir o tempo de execução do *solver*. Isto é, através das heurísticas, tentamos obter uma solução inteira da resolução do programa linear relaxado, cuja execução tende a ser mais rápida que a execução do próprio programa linear

inteiro. Além das heurísticas, executamos outras duas abordagens, ótima (OPT) e do primeiro nó (PS), para que pudéssemos avaliar os resultados encontrados pelas heurísticas. No entanto, para que essa avaliação fosse de forma justa, estipulamos o tempo de execução de OPT e PS com o tempo de execução utilizado em cada heurística. Dessa forma, os resultados apresentados sugerem que as heurísticas BMEMT e BMT podem ser eficazes para reduzir os custos monetários dos escalonamentos com *deadlines* longos.

Além das heurísticas, apresentamos, em seguida, os resultados da 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. Para avaliar somente a variação do fator λ no conjunto de tempo $\mathcal{T}$, utilizamos apenas a abordagem ótima; ou seja, a abordagem PS e as heurísticas BMEMT e BMT não foram utilizadas nessas simulações. Os resultados mostraram que, para DAGs grandes e *deadlines* altos, a estratégia de aumentar a granularidade da discretização do tempo permite ao *solver* encontrar rapidamente soluções viáveis ao escalonamento. Entretanto, quando aumentamos λ, erros (desperdício de recursos e dinheiro) são inseridos no resultado do escalonamento, os quais podem ser controlados através de novas heurísticas que pretendemos desenvolver nos trabalhos futuros. Embora experimentos de execução de *workflows* em ambientes reais não foram realizados neste trabalho, supomos que o tempo de execução do *solver* (tempo para achar uma solução viável) seja suficientemente menor que o tempo da execução do *workflow* propriamente dito. Reproduzir as simulações da avaliação de desempenho do escalonador através de experimentos em ambientes reais também se encontra como trabalhos futuros.

Capítulo 6

Conclusão

Aplicações complexas modeladas como *workflows* e representadas por DAGs estão, hoje em dia, exigindo um poder computacional cada vez maior. Esse aumento de demanda é, por exemplo, destacado no campo da e-Ciência, com aplicações representadas por DAGs em diversas áreas do conhecimento, tais como: física, biologia, astronomia e ciência da computação. Entretanto, a demanda computacional pode ser sazonal e, dessa maneira, investimento em recursos próprios pode implicar em altos custos de manutenção e subutilização em momentos fora do pico de demanda. Assim, com intuito de promover *elasticidade* ao poder de computação local (ou privado), a computação em nuvem está sendo muito utilizada por instituições (universidades, empresas de TI, bancos, empresas *start-ups*, etc.) para executar tais aplicações complexas com um investimento baixo. Portanto, a computação em nuvem é um paradigma que fornece aos usuários recursos de computação sob demanda, o qual é tarifado pelo modelo "pago-pelo-uso".

Por outro lado, para determinar qual a melhor estratégia para executar *workflows* nos recursos computacionais alugados da nuvem, isto é, extrair de modo eficiente o poder computacional com custos monetários baixos, um estudo de escalonamento de *workflows* representados por DAGs em nuvens é de suma importância. Assim, é necessário levar em consideração algumas características inerentes às nuvens, das quais se destacam: (i) a heterogeneidade de máquinas virtuais com um poder de processamento e uma tarifa associada; e (ii) o tempo de comunicação entre essas máquinas virtuais. Enquanto a heterogeneidade implica em uma complexidade na seleção de qual provedor de nuvem utilizar para minimizar o custo monetário, o tempo de comunicação pode atrasar o escalonamento e, consequentemente, romper os requisitos de QoS (*deadine*, por exemplo) estabelecidos nos SLAs.

Nesta dissertação apresentamos uma abordagem para escalonar *workflows* em computação em nuvem. Consideramos um cenário realístico com dois níveis de SLAs, onde o provedor de SaaS aluga máquinas virtuais de vários provedores de IaaS para executar

os *workflows* de seus clientes. Entretanto, devido à existência de múltiplos provedores de IaaS e a exigência do tempo de resposta (*deadline*) da execução do *workflow* estipulada pelo cliente, o provedor de SaaS precisa minimizar o custo monetário dessa execução, respeitar o *deadline* e, ao mesmo tempo, tentar maximizar o seu lucro. Dessa forma, através dos provedores de infraestrutura, o provedor de SaaS evita investimentos e custos de manutenção em recursos computacionais extras e, além disso, pode utilizar seus recursos privados para executar aplicações próprias. As contribuições deste trabalho foram:

- Desenvolvimento de um programa linear inteiro para o problema de escalonamento de *workflows* em nuvens, considerando dois níveis de SLA;
- Relaxamento das variáveis binárias x e y do PLI, a fim de diminuir o tempo de execução do escalonador;
- Desenvolvimento de duas heurísticas (BMEMT e BMT) para encontrar uma solução inteira da solução fracionada (relaxada);
- Uso de diferentes níveis de discretização da linha do tempo no PLI, também com intuito de reduzir o tempo de execução do escalonador;
- Implementação do programa linear inteiro em Java usando o solucionador matemático *IBM ILOG CPLEX Optimizer*;
- Avaliação do desempenho do escalonador com DAGs que representam aplicações do mundo real.

Descrevemos um programa linear inteiro (PLI) para solucionar o problema de escalonamento de *workflows* em nuvens. Esse PLI, cuja função objetivo é minimizar custos monetários com aluguel de máquinas virtuais, é composto por 2 variáveis binárias e 9 restrições lineares. No entanto, programas lineares inteiros são problemas NP-Difícil e, com isso, aplicamos a técnica de relaxação linear nas variáveis binárias inteiras a fim de diminuir o tempo de execução do *solver*. Assim, duas heurísticas, nomeadas BMEMT e BMT, foram desenvolvidas para encontrar possíveis soluções inteiras sobre o PLI relaxado. Simulações mostraram que, quando comparados aos resultados encontrados pelas abordagens ótima (OPT) e primeiro nó (PS) da raiz da árvore de enumeração, ambas heurísticas foram eficazes na redução dos custos monetários dos escalonamentos com *deadlines* longos. De modo geral, quando a execução via abordagem OPT ou PS tende a demorar, podemos utilizar as heurísticas para acelerar o processo de escalonamento e, além disso, minimizar os custos monetários envolvidos. Note que todas as comparações das heurísticas com OPT e PS foram feitas considerando o mesmo tempo de execução

do *solver*. Além disso, é importante frisar que todas as simulações de escalonamento de DAGs foram realizadas utilizando o *solver IBM ILOG CPLEX Optimizer*.

Outra alternativa para diminuir o tempo de execução do escalonador foi o aumento do nível da discretização da linha do tempo na formulação do PLI. Essa técnica reduz consideravelmente o tamanho do conjunto de intervalos discretos $\mathcal{T}$ e, portanto, reduz também o conjunto de soluções do escalonamento, que implica diretamente na redução do tempo de execução do *solver*. Dessa forma, foi possível realizar escalonamento de DAG maiores (com grande números de nós e dependências) em nosso escalonador. Os resultados mostraram que, para DAGs grandes e *deadlines* altos, a estratégia de aumentar a granularidade da discretização do tempo permite ao *solver* encontrar rapidamente soluções viáveis ao escalonamento. A fim de avaliar somente a variação do nível de discretização do tempo, usamos apenas a abordagem OPT nessas simulações, as quais mostraram que é possível encontrar soluções viáveis e de baixo custo monetário para o escalonamento de DAGs grandes em nuvens com um nível de discretização alto ($\lambda \geq 1$).

Portanto, as abordagens de escalonamento apresentadas nesta dissertação podem fornecer uma base para o provedor de SaaS poder negociar SLAs com seus clientes e, além disso, poder escolher melhor as máquinas virtuais para minimizar os custos monetários a fim de maximizar seu lucro.

Trabalhos Futuros

O escalonamento de DAGs em nuvens é um tema que ainda pode ser abordado de diferentes maneiras. Este trabalho abre espaço para uma diversidade de trabalhos futuros:

- Desenvolvimento de heurísticas não iterativas para encontrar soluções viáveis inteiras sobre a execução do PLI relaxado. Portanto, evitaria que, para cada entrada do escalonamento, o *solver* fosse executado $k \leq |\mathcal{U}| - 1 = n - 1$ vezes, reduzindo o tempo de execução do *solver*;

- Desenvolvimento de heurísticas (iterativas ou não) nas execuções do programa linear inteiro com o aumento da granularidade da linha do tempo (fator λ);

- Desenvolvimento de heurísticas 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 a necessidade de reescalonar o DAG;

- Realizar uma avaliação do escalonamento de DAGs em condições de incertezas. Para que a execução seja eficiente, o escalonador deve distribuir os nós do DAG nos recur-

sos alugados, utilizando informações prévias sobre as capacidades de processamento das máquinas virtuais e largura de banda entre máquinas virtuais, assim como também dados sobre a duração e comunicação dos nós (serviços) componentes do DAG. Entretanto, tais informações podem ser imprecisas, isto é, não refletindo o que de fato é encontrado durante a execução. Além disso, dados sobre a infraestrutura interna de um provedor de IaaS público não são divulgados. Portanto, essa avaliação trará melhor credibilidade aos resultados encontrados pelas simulações;

- Realizar uma avaliação do escalonamento de DAGs em máquinas virtuais que compartilham núcleo de processamento da máquina física;
- Também espera-se ser possível reproduzir as simulações da avaliação de desempenho do escalonador através de experimentos em ambiente real, com o intuito de validar os resultados obtidos nesta dissertação.

A área de escalonamento de múltiplos DAGs tem sido pouco estudada. Assim, outro trabalho futuro que podemos enfatizar é o desenvolvimento de uma estratégia de escalonamento de múltiplos DAGs em nuvens. O escalonador desenvolvido nesta dissertação considera apenas um DAG por vez, e a área de escalonamento de múltiplos DAGs em nuvens carece desse tipo de algoritmo. Assim, um trabalho já em andamento delimita estratégias de como realizar escalonamento de múltiplos DAGs no cenário da Figura 2.4 – um provedor de SaaS com dois níveis de SLAs. Em outras palavras, no escalonamento de único DAG, podem surgir espaços entre serviços dependentes. Esses espaços são oriundos do tempo de transmissão de dados entre serviços que estão em recursos diferentes. Assim, podemos, por exemplo, preenchê-los com serviços de outros DAGs, permitindo otimizar (e acelerar) a fila de DAGs prontos para serem escalonados. Embora seja necessário cumprir o requisito de QoS na execução de cada DAG, uma justiça para acessar os recursos deve ser estabelecida, pois o escalonamento de um DAG não deve prejudicar (atrasar, por exemplo) outros escalonamentos. Portanto, em comparação com o escalonamento de único DAG, o escalonamento de múltiplos DAGs é um problema intrinsecamente mais complexo e desafiador.

Referências Bibliográficas

[1] M. Alhamad, T. Dillon, and E. Chang. Conceptual SLA framework for cloud computing. In *Proceedings of the 4th IEEE International Conference on Digital Ecosystems and Technologies*, DEST '10, pages 606 –610, april 2010.

[2] M. Alhamad, T. Dillon., and E. Chang. SLA-based trust model for cloud computing. In *Proceedings of the 13th International Conference on Network-Based Information Systems*, NBiS '10, pages 321 –324, sept. 2010.

[3] M. Alhamad, T. Dillon, C. Wu, and E. Chang. Response time for cloud computing providers. In *Proceedings of the 12th International Conference on Information Integration and Web-based Applications Services*, iiWAS '10, pages 603–606, New York, NY, USA, 2010. ACM.

[4] J. Annis, Y. Zhao, J. Voeckler, M. Wilde, S. Kent, and I. Foster. Applying chimera virtual data concepts to cluster finding in the sloan sky survey. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Supercomputing '02, pages 1–14, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[5] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Journal of Communication ACM*, 53:50–58, apr 2010.

[6] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[7] D. Batista, C. Chaves, and N. da Fonseca. Embedding software requirements in grid scheduling. In *2011 IEEE International Conference on Communications (ICC)*, ICC'11, pages 1 –6, june 2011.

[8] D. M. Batista and N. L. da Fonseca. Robust scheduler for grid networks under uncertainties of both application demands and resource availability. *Journal of Computer Networks*, 55(1):3 – 19, 2011.

[9] M.-E. Begin. An egee comparative study: Grids and clouds – evolution or revolution. Technical report, CERN - Engeneering and Equipment Data Management Service, jun 2008.

[10] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi. Characterization of scientific workflows. In *Proceedings of the 3rd Workshop on Workflows in Support of Large-Scale Science*, WORKS '08, pages 1 –10, nov. 2008.

[11] L. Bittencourt, C. Senna, and E. Madeira. Enabling execution of service workflows in grid/cloud hybrid systems. In *Proceedings of the 2010 IEEE/IFIP Network Operations and Management Symposium Workshops*, NOMS Wksps '10, pages 343 –349, april 2010.

[12] L. Bittencourt, C. Senna, and E. Madeira. Scheduling service workflows for cost optimization in hybrid clouds. In *Proceedings of the 2010 International Conference on Network and Service Management*, CNSM '10, pages 394 –397, oct. 2010.

[13] L. F. Bittencourt. *Algoritmos para Escalonamento de Tarefas Dependentes Representadas por Grafos Acíclicos Direcionados em Grades Computacionais.* Tese de doutorado, Instituto de Computação – Universidade Estadual de Campinas, Brasil, 2010.

[14] L. F. Bittencourt and E. R. M. Madeira. Fulfilling task dependence gaps for workflow scheduling on grids. In *Proceedings of the 3rd International IEEE Conference on Signal-Image Technologies and Internet-Based System*, SITIS '07, pages 468–475, Washington, DC, USA, 2007. IEEE Computer Society.

[15] L. F. Bittencourt and E. R. M. Madeira. Towards the scheduling of multiple workflows on computational grids. *Journal of Grid Computing*, 8:419–441, 2010.

[16] 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.

[17] L. F. Bittencourt, C. Senna, and E. R. M. Madeira. Bicriteria service scheduling with dynamic instantiation for workflow execution on grids. In N. Abdennadher and D. Petcu, editors, *Advances in Grid and Pervasive Computing*, volume 5529 of

Lecture Notes in Computer Science, pages 177–188. Springer Berlin / Heidelberg, 2009.

[18] R. Buyya, C. S. Yeo, and S. Venugopal. Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. In *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications*, HPCC '08, pages 5 –13, sept. 2008.

[19] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Journal of Future Generation Computer Systems*, 25(6):599–616, jun 2009.

[20] D. Cerbelaud, S. Garg, and J. Huylebroeck. Opening the clouds: qualitative overview of the state-of-the-art open source vm-based cloud management platforms. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '09, pages 22:1–22:8, New York, NY, USA, 2009. Springer-Verlag New York, Inc.

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

[22] C. Chaves, D. Batista, and N. da Fonseca. Scheduling grid applications with software requirements. *Journal of IEEE Latin America Transactions*, 9(4):578 –585, july 2011.

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

[24] R. Correa, A. Ferreira, and P. Rebreyend. Integrating list heuristics into genetic algorithms for multiprocessor scheduling. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing*, SPDP '96, pages 462 –469, oct 1996.

[25] R. M. Cury. *Uma abordagem difusa para o problema de flow-shop scheduling*. Tese de doutorado, Programa de Pós-Graduação em Engenharia de Produção – Universidade Federal de Santa Catarina, Brasil, 1999.

[26] E. Deelman, D. Gannon, M. Shields, and I. Taylor. Workflows and e-Science: An overview of workflow system features and capabilities. *Journal of Future Generation Computer Systems*, 25(5):528 – 540, 2009.

[27] E. Deelman, G. Singh, M.-H. 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.

[28] A. Doğan and F. Özgüner. Biobjective scheduling algorithms for execution time-reliability trade-off in heterogeneous computing systems. *Journal of The Computer Journal*, 48:300–314, May 2005.

[29] B. Duran and F. Xhafa. The effects of two replacement strategies on a genetic algorithm for scheduling jobs on computational grids. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 960–961, New York, NY, USA, 2006. ACM.

[30] H. El-Rewini, H. Ali, and T. Lewis. Task scheduling in multiprocessing systems. *Journal of Computer*, 28(12):27 –37, dec 1995.

[31] H. Fard, R. Prodan, G. Moser, and T. Fahringer. A bi-criteria truthful mechanism for scheduling of workflows in clouds. In *Proceedings of the 3rd IEEE International Conference on Cloud Computing Technology and Science*, CloudCom '11, pages 599 –605, 29 2011-dec. 1 2011.

[32] H. M. Fard, R. Prodan, G. Moser, and T. Fahringer. A bi-criteria truthful mechanism for scheduling of workflows in clouds. In *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, CLOUDCOM '11, pages 599–605, Washington, DC, USA, 2011. IEEE Computer Society.

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

[34] M. Fouquet, H. Niedermayer, and G. Carle. Cloud computing for the masses. In *Proceedings of the 1st ACM workshop on User-provided networking: challenges and opportunities*, U-NET '09, pages 31–36, New York, NY, USA, 2009. ACM.

[35] N. Fujimoto and K. Hagihara. Near-optimal dynamic task scheduling of precedence constrained coarse-grained tasks onto a computational grid. In *Proceedings of the Second international conference on Parallel and distributed computing*, ISPDC'03, pages 80–87, Washington, DC, USA, 2003. IEEE Computer Society.

[36] S. Garg, C. Vecchiola, and R. Buyya. Mandi: A market exchange for trading utility and cloud computing services. *Journal of Supercomputing*, pages 1–22, 2011.

[37] T. A. L. Genez, L. F. Bittencourt, and E. R. M. Madeira. Discretização do tempo na utilização de programação linear para o problema de escalonamento de workflows em múltiplos provedores de nuvem. In *XXX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, SBRC '12, abril 2012.

[38] 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.

[39] R. E. Gomory. Outline of an algorithm for integer solutions to linear programs and an algorithm for the mixed integer problem. In M. Jünger, T. M. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleyblank, G. Reinelt, G. Rinaldi, and L. A. Wolsey, editors, *Proceedings of the 50 Years of Integer Programming 1958-2008*, pages 77–103. Springer Berlin Heidelberg, 2010.

[40] C. J. Guo, W. Sun, Y. Huang, Z. H. Wang, and B. Gao. A framework for native multi-tenancy application development and management. In *Proceedings of the 9th IEEE International Conference on E-Commerce Technology and in proceedings of the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services*, CEC/EEE '07, pages 551 –558, july 2007.

[41] C. Hoffa, G. Mehta, T. Freeman, E. Deelman, K. Keahey, B. Berriman, and J. Good. On the use of cloud computing for scientific workflows. In *Proceedings of 4th IEEE International Conference on eScience*, eScience '08, pages 640 –645, dec. 2008.

[42] C. Höfer and G. Karagiannis. Cloud computing services: taxonomy and comparison. *Journal of Internet Services and Applications*, 2:81–94, 2011.

[43] E. Juhnke, T. Dornemann, D. Bock, and B. Freisleben. Multi-objective scheduling of bpel workflows in geographically distributed clouds. In *Proceedings of the 2011 IEEE International Conference on Cloud Computing*, CLOUD '11, pages 412 –419, july 2011.

[44] K. H. Kim and R. Buyya. Fair resource sharing in hierarchical virtual organizations for global grids. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, GRID '07, pages 50 –57, sept. 2007.

[45] L. Kleinrock. An Internet vision: The invisible global infrastructure. *Journal of AdHoc Networks*, 1(1):3–11, July 2003.

[46] L. Kleinrock. A vision for the Internet. *ST Journal for Research*, 2(1):4–5, November 2005.

[47] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. M. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D. E. Steffy, and K. Wolter. MIPLIB 2010 – mixed integer programming library version 5. *Mathematical Programming Computation*, 3(2):103–163, 2011.

[48] A. H. Land and A. G. Doig. An automatic method for solving discrete programming problems. In M. Jünger, T. M. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleyblank, G. Reinelt, G. Rinaldi, and L. A. Wolsey, editors, *Proceedings of the 50 Years of Integer Programming 1958-2008*, pages 105–132. Springer Berlin Heidelberg, 2010.

[49] A. Lenk, M. Klems, J. Nimis, S. Tai, and T. Sandholm. What's inside the cloud? an architectural map of the cloud landscape. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, CLOUD '09, pages 23–31, Washington, DC, USA, 2009. IEEE Computer Society.

[50] Q. Li and Y. Guo. Optimization of resource scheduling in cloud computing. In *Proceedings of the 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, SYNASC '10, pages 315–320, sep. 2010.

[51] Q. Li, Q. Hao, L. Xiao, and Z. Li. Adaptive management of virtualized resources in cloud computing using feedback control. In *Proceedings of the 1st International Conference on Information Science and Engineering*, ICISE '09, pages 99 –102, 26-28 2009.

[52] C. Lin and S. Lu. Scheduling scientific workflows elastically for cloud computing. In *Proceedings of the IEEE International Conference on Cloud Computing*, CLOUD '11, pages 746 –747, july 2011.

[53] E. Marilly, O. Martinot, H. Papini, and D. Goderis. Service Level Agreements: a main challenge for next generation networks. In *Proceedings of the 2nd European Conference on Universal Multiservice Networks*, ECUMN'02, pages 297 – 304, 2002.

[54] D. A. Menascé. Virtualization: Concepts, applications, and performance modeling. In *Proceedings of the 31th International Computer Measurement Group Conference, December 4-9, 2005, Orlando, Florida, USA, Proceedings*, pages 407–414. Computer Measurement Group, 2005.

[55] S. Nepal, S. Chen, J. Yao, and D. Thilakanathan. DIaaS: Data integrity as a service in the cloud. In *Proceedings of the 2011 IEEE International Conference on Cloud Computing*, CLOUD'11, pages 308 –315, july 2011.

[56] NIST. The nist definition of cloud computing, sep. 2011. Disponível em http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf. Acessado em dezembro de 2011.

[57] S. Pandey. *Scheduling and Management of Data Intensive Application Workflows in Grid and Cloud Computing Environments*. PhD thesis, Department of Computer Science and Software Engineering – University of Melbourne, Australia, 2010.

[58] S. Pandey, D. Karunamoorthy, and R. Buyya. *Workflow Engine for Clouds*, pages 321–344. John Wiley & Sons, Inc., 2011.

[59] S. Pandey, L. Wu, S. Guru, and R. Buyya. A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments. In *Proceedings of the 24th IEEE International Conference on Advanced Information Networking and Applications*, AINA '10, pages 400 –407, april 2010.

[60] S. Pandey, L. Wu, S. M. Guru, and R. Buyya. A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments. In *Proceedings of the 2010 24th IEEE International Conference on Advanced Information Networking and Applications*, AINA'10, pages 400–407, Washington, DC, USA, 2010. IEEE Computer Society.

[61] G. Papuzzo and G. Spezzano. Autonomic management of workflows on hybrid grid-cloud infrastructure. In *Proceedings of the 7th International Conference on Network and Service Management*, CNSM '11, pages 1 –4, oct. 2011.

[62] M. L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer Publishing Company, Incorporated, 3rd edition, 2008.

[63] 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.

[64] G. Reig, J. Alonso, and J. Guitart. Prediction of job resource requirements for deadline schedulers to manage high-level slas on the cloud. In *Proceedings of the 9th IEEE International Symposium on Network Computing and Applications*, NCA '10, pages 162 –167, july 2010.

[65] M. Sipser. *Introduction to the theory of computation*. Thomson Course Technology, 2006.

[66] 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.

[67] M. Tao, S. Dong, and K. He. A new replication scheduling strategy for grid workflow applications. In *Proceedings of the 6th Annual Chinagrid Conference*, ChinaGrid '11, pages 74 –80, aug. 2011.

[68] I. Taylor, M. Shields, and I. Wang. Distributed P2P computing within triana: a galaxy visualization test case. In *Proceedings of the 2003 International Parallel and Distributed Processing Symposium*, IPDPS '03, page 8 pp., april 2003.

[69] V. T'kindt and J.-C. Billaut. Some guidelines to solve multicriteria scheduling problems. In *Proceedings of the 1999 IEEE International Conference on Systems, Man, and Cybernetics*, volume 6 of *SMC '99*, pages 463 –468 vol.6, 1999.

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

[71] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Newsletter of ACM Computer Communication Review*, 39:50–55, Dec. 2008.

[72] L. Wang, G. von Laszewski, A. Younge, X. He, M. Kunze, J. Tao, and C. Fu. Cloud computing: a perspective study. *Journal of New Generation Computing*, 28:137–146, 2010.

[73] 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.

[74] A. Weiss. Computing in the clouds. *netWorker Magazine – Cloud computing: PC functions move onto the web*, 11:16–25, Dec. 2007.

[75] M. Wieczorek, S. Podlipnig, R. Prodan, and T. Fahringer. Bi-criteria scheduling of scientific workflows for the grid. In *Proceedings of 8th IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '08, pages 9 –16, may 2008.

[76] D. Wischik, M. Handley, and M. B. Braun. The resource pooling principle. *Newsletter of ACM SIGCOMM Computer Communication Review*, 38:47–52, September 2008.

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

[78] L. Wu, S. K. Garg, and R. Buyya. SLA-based admission control for a software-as-a-service providerin cloud computing environments. Technical Report CLOUDS-TR-2010-7, Cloud Computing and Distributed Systems Laboratory, The University of Melbourne, Australia, sep. 2010.

[79] Z. Wu, X. Liu, Z. Ni, D. Yuan, and Y. Yang. A market-oriented hierarchical scheduling strategy in cloud workflow systems. *The Journal of Supercomputing*, pages 1–38, 2011.

[80] Z. Wu, Z. Ni, L. Gu, and X. Liu. A revised discrete particle swarm optimization for cloud workflow scheduling. In *Proceedings of the 2010 International Conference on Computational Intelligence and Security*, CIS '10, pages 184 –188, dec. 2010.

[81] 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.

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

[83] J. Yu and R. Buyya. Scheduling scientific workflow applications with deadline and budget constraints using genetic algorithms. *Journal Scientific Programming - Scientific Workflows*, 14:217–230, December 2006.

[84] J. Yu, R. Buyya, and C. K. Tham. Cost-based scheduling of scientific workflow application on utility grids. In *Proceedings of the 1st International Conference on e-Science and Grid Computing*, E-SCIENCE '05, pages 140–147, Washington, DC, USA, 2005. IEEE Computer Society.

[85] 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.

[86] Q. Zhang, Q. Zhu, and R. Boutaba. Dynamic resource allocation for spot markets in cloud computing environments. In *Proceedings of the 4th IEEE International Conference on Utility and Cloud Computing*, UCC '11, pages 178 –185, dec. 2011.

[87] S. Zhang, S. Zhang, X. Chen, and X. Huo. Cloud computing research and development trend. In *Proceedings of the 2010 Second International Conference on Future Networks*, ICFN '10, pages 93–97, Washington, DC, USA, 2010. IEEE Computer Society.

[88] Y. Zhang, B. Zhang, and Y. Liu. A method of SaaS multi-tenant model recommendation based on graph matching. In *Proceedings of the 3rd International Conference on Data Mining and Intelligent Information Technology Applications*, ICMiA '11, pages 135 –140, oct. 2011.

[89] C. Zhao, S. Zhang, Q. Liu, J. Xie, and J. Hu. Independent tasks scheduling based on genetic algorithm in cloud computing. In *Proceedings of the 5th International Conference on Wireless Communications, Networking and Mobile Computing*, WiCom '09, sep. 2009.

[90] Y. Zhao, J. Dobson, I. Foster, L. Moreau, and M. Wilde. A notation and system for expressing and executing cleanly typed workflows on messy scientific data. *Newsletter of ACM SIGMOD Record*, 34(3):37–43, sep 2005.

[91] Y. Zhao, I. Raicu, and I. Foster. Scientific workflow systems for 21st century, new bottle or new wine? In *Proceedings of the IEEE Congress on Services - Part I*, SERVICES '08, pages 467 –471, july 2008.

[92] M. Zhou, R. Zhang, D. Zeng, and W. Qian. Services in the cloud computing era: A survey. In *Proceedings of the 4th International Universal Communication Symposium*, IUCS '10, pages 40 –46, oct. 2010.

[93] A. Zinnen and T. Engel. Deadline constrained scheduling in hybrid clouds with gaussian processes. In *Proceedings of the 2011 International Conference onHigh Performance Computing and Simulation*, HPCS '11, pages 294 –300, july 2011.