Algoritmos para Escalonamento de Tarefas Dependentes Representadas por Grafos Acíclicos Direcionados em Grades Computacionais

Este exemplar corresponde à redação final da Tese/Dissertação devidamente corrigida e defendida por: Luiz Fernando Bittencourt

e aprovada pela Banca Examinadora.
Campinas, 18 de junho de 2010

COORDENADOR DE PÓS-GRADUAÇÃO
Prof. Dr. Julio César Lopez Hernández
Coord. Subst. de Pós-Graduação
Instituto de Computação/Unicamp
Matr. 28.620-1

Este exemplar corresponde à redação final da Tese devidamente corrigida e defendida por Luiz Fernando Bittencourt e aprovada pela Banca Examinadora.

Campinas, 19 de maio de 2010.

Edmundo R. M. Madeira

Edmundo Roberto Mauro Madeira
Instituto de Computação - UNICAMP
(Orientador)

Tese apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Crisllene Queiroz Custódio – CRB8 / 7966

Bittencourt, Luiz Fernando
B548a Algoritmos para escalonamento de tarefas dependentes representadas por grafos acíclicos direcionados em grades computacionais / Luiz Fernando Bittencourt -- Campinas, [S.P. : s.n.], 2010.

Orientador : Edmundo Roberto Mauro Madeira
Tese (Doutorado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Sistemas distribuídos. 2. Computação em grade (Sistema de computador). 3. Escalonamento. 4. Fluxo de trabalho. I. Madeira, Edmundo Roberto Mauro. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Scheduling algorithms for dependent tasks represented by directed acyclic graphs on computational grids.

Palavras-chave em inglês (Keywords): 1. Distributed systems. 2. Computational grids (Computer systems). 3. Scheduling. 4. Workflow.

Área de concentração: Sistemas de Computação

Titulação: Doutor em Ciência da Computação

Banca examinadora: Prof. Dr. Edmundo Roberto Mauro Madeira (IC-UNICAMP)
Prof. Dr. Alfredo Goldman vel Lejbman (IME-USP)
Prof. Dr. Eugene Francis Vinod Rebello (IC-UFF)
Prof. Dr. Nelson Luis Saldanha da Fonseca (IC-UNICAMP)
Prof. Dr. Luiz Eduardo Buzato (IC-UNICAMP)

Data da defesa: 22/03/2010

Programa de Pós-Graduação: Doutorado em Ciência da Computação

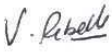
TERMO DE APROVAÇÃO

Tese Defendida e Aprovada em 22 de março de 2010, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Luiz Eduardo Buzato
IC / UNICAMP

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

Prof. Dr. Alfredo Goldman vel Lejbman
IME / USP



Prof. Dr. Eugene Francis Vinod Rebello
IC / UFF

Prof. Dr. Edmundo Roberto Mauro Madeira
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Algoritmos para Escalonamento de Tarefas Dependentes Representadas por Grafos Acíclicos Direcionados em Grades Computacionais

Luiz Fernando Bittencourt[1]

Junho de 2010

Banca Examinadora:

- Edmundo Roberto Mauro Madeira
  Instituto de Computação - UNICAMP (Orientador)
- Alfredo Goldman vel Lejbman
  Instituto de Matemática e Estatística - USP
- Eugene Francis Vinod Rebello
  Instituto de Computação - UFF
- Nelson Luis Saldanha da Fonseca
  Instituto de Computação - UNICAMP
- Luiz Eduardo Buzato
  Instituto de Computação - UNICAMP
- Bruno Richard Schulze (Suplente)
  Laboratório Nacional de Computação Científica
- Islene Calciolari Garcia (Suplente)
  Instituto de Computação - UNICAMP
- Maria Beatriz Felgar de Toledo (Suplente)
  Instituto de Computação - UNICAMP

[1]Suporte financeiro: FAPESP (2005/50796-3), CAPES (3248-08-9)

Resumo

Grades computacionais são sistemas distribuídos compartilhados potencialmente grandes compostos por recursos heterogêneos que são ligados através de uma rede com enlaces heterogêneos. Esses sistemas tornaram-se ambientes largamente difundidos para execução de tarefas que demandam grande capacidade de processamento. Por serem sistemas compartilhados, a submissão de tarefas nas grades é oriunda de diversos usuários independentemente, o que gera uma demanda concorrente pelos recursos computacionais que deve ser gerenciada pelo *middleware* da grade. O escalonador é o componente responsável por decidir de que forma a distribuição dessas tarefas será realizada, devendo tratar das peculiaridades desse ambiente, tais como a heterogeneidade e o comportamento dinâmico dos recursos que o compõem, com variações tanto em quantidade quanto em qualidade. A função objetivo mais comum encontrada no escalonamento de tarefas é a minimização do *makespan*, ou seja, o tempo de término das tarefas que estão sendo escalonadas. Dentre os possíveis tipos de tarefas executadas em grades podemos destacar as tarefas independentes, que executam sem comunicação entre si, e as tarefas dependentes, que possuem dependências de dados que geram precedências de execução e são freqüentemente modeladas como grafos acíclicos direcionados (DAGs - do inglês *directed acyclic graphs*). Dentre as aplicações compostas por tarefas dependentes, os DAGs de e-Ciência se sobressaem pela complexidade e necessidade crescente de recursos computacionais. Adicionalmente, o problema de escalonamento de tarefas, em sua forma geral, é NP-Completo. Dessa forma, o estudo do escalonamento de DAGs em grades computacionais é importante para o aprimoramento da execução de aplicações científicas utilizadas em diversas áreas do conhecimento.

Nesta tese apresentamos algoritmos para quatro tipos de problema relacionados ao escalonamento de DAGs em grades: escalonamento estático de DAGs, escalonamento dinâmico de DAGs, escalonamento bi-critério e escalonamento de múltiplos DAGs. Apresentamos avaliações do *makespan* gerado pelos algoritmos após o escalonamento inicial e após a execução das tarefas com carga externa simulada nos recursos.

Abstract

Computational grids are potentially large distributed systems composed of heterogeneous resources connected by a network with heterogeneous links. These systems became largely used in the execution of tasks which require large processing capacities. Because they are shared systems, task submission in grids independently originate from a number of users, leading to a concurrent demand over the computational resources, which must be managed by the grid middleware. The scheduler is the component responsible for deciding how the distribution of such tasks will occur, and it must deal with peculiarities of this environment, such as the heterogeneity and dynamic behavior of the resources, with variations in both quality and quantity. The objective function usually adopted in task scheduling is makespan minimization, which means that the scheduler tries to minimize the finish time of the tasks being scheduled. Among the tasks executed in grids we can find independent tasks, which execute without communication among them, and dependent tasks, which have data dependencies that yield in precedence constraints and are frequently modeled as directed acyclic graphs (DAGs). Among the applications composed of dependent tasks, e-Science DAGs are distinguished because of their complexity and increasing demand for computational resources. Additionally, the task scheduling problem, in its general form, is NP-Complete. Therefore, the study of scheduling of dependent tasks represented by directed acyclic graphs in computational grids is important to improve the execution of scientific applications in many areas of knowledge.

In this thesis we present algorithms for four types of problems related to the DAG scheduling in grids: static scheduling of DAGs, dynamic scheduling of DAGs, bi-criteria scheduling, and scheduling of multiple DAGs. We present evaluations of the makespan generated by the algorithms after the initial scheduling and after the execution of the tasks with simulated external load in the resources.

Agradecimentos

Agradeço primeiramente à minha família, principalmente aos meus pais, Luiz Carlos e Maria do Rocio, pelo apoio incondicional às decisões que tomei até aqui, além do carinho e dedicação comigo e com meus irmãos, Cíntia e Emerson.

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

À Flávia, minha namorada, pelo apoio, pela motivação, pelo carinho, pelos bons momentos de descontração, pelas risadas, pelos filmes, pelas novelas, pela correção do texto final da tese e por todas as outras coisas indescritíveis que podem ser atribuídas e derivadas desse tempo juntos.

Ao André Vignatti, amigo e companheiro de ensino médio, graduação, mestrado e doutorado. Ele foi quem acompanhou mais de perto e quem durante mais tempo participou dessa fase de pós-graudação, tanto estudando quanto em festas e em papos nerds (e não nerds) de computeiros.

A todos os amigos e amigas de Campinas e de Curitiba que de alguma forma fizeram parte da minha vida nesses últimos 4 anos e que tornaram a minha vida agradabilíssima, aos quais devo parte considerável da minha felicidade.

Ao Rizos Sakellariou, que foi meu orientador durante o período de doutorado sanduíche em Manchester, e aos amigos que me receberam lá.

Aos funcionários e professores que colaboraram com a minha formação e se esforçaram para que eu pudesse desenvolver meu trabalho com o melhor apoio possível.

À FAPESP pela bolsa de doutorado e à CAPES pela bolsa de doutorado sanduíche.

Sumário

Lista de Figuras

Capítulo 1

Introdução

Durante a evolução da computação, observamos a mudança do paradigma centralizado para o paradigma distribuído, onde a execução de tarefas e o armazenamento de dados exploram recursos computacionais dispersos. Essa mudança de paradigma foi possível inicialmente graças ao avanço das redes de computadores, notadamente pelo aumento sensível da largura de banda disponível nos enlaces das redes *ethernet*.

O surgimento de *clusters* de computadores foi o início da difusão da mudança para o paradigma de processamento distribuído. Um *cluster* computacional é um conjunto de computadores homogêneos normalmente interligados por uma rede local. Tal conjunto de computadores é geralmente utilizado para ampliar a capacidade de processamento e armazenamento em relação a computadores ou supercomputadores, ainda que com um custo mais baixo [4].

O aprimoramento do suporte à computação distribuída através do desenvolvimento de *middlewares* em conjunto com avanços nos protocolos de rede, em especial aqueles relativos à Internet, culminou na concepção das *grades computacionais* [47, 45]. Grades gerenciam o compartilhamento coordenado de recursos heterogêneos, geograficamente distantes, em larga escala. Esses recursos podem ser variados, tais como processadores, armazenamento em disco, hardware especializado e *softwares*. A heterogeneidade dos recursos computacionais e a distribuição geográfica, englobando recursos de diferentes organizações ou pessoas, são as diferenças básicas das grades em relação aos *clusters*. Assim, podemos citar a interoperabilidade entre recursos heterogêneos como resultado do avanço de *software*, enquanto avanços na área de rede, como a redução de latência e o aumento de largura de banda, tornaram possível a distribuição da execução de tarefas computacionais sobre as grades sem comprometimento de desempenho.

Para que seja possível explorar a capacidade de processamento disponível em uma grade computacional, um componente específico do *middleware* é de suma importância: o escalonador. Esse componente é responsável pela escolha de recursos para as tarefas,

ou seja, ele determina qual tarefa será executada em qual recurso computacional. Conseqüentemente, a duração da execução de cada tarefa é fortemente dependente da escolha feita pelo escalonador, pois a grade é composta por recursos heterogêneos com diferentes capacidades de processamento.

Grafos acíclicos direcionados (DAGs - do inglês *directed acyclic graphs*) são freqüentemente utilizados para capturar as restrições de precedência de várias aplicações. DAGs são *workflows* em uma forma mais restrita, podendo representar aplicações com tarefas que não possuam ciclos de dependência. Quando tais aplicações são executadas em recursos computacionais em paralelo, a ordem de execução dessas tarefas nos recursos e como a distribuição é feita são fatores decisivos que determinam o desempenho da execução. Em sua forma geral, o escalonamento de tarefas dependentes em um conjunto de processadores é um problema NP-Completo [76]. Em razão disso, durante os anos, uma variedade de heurísticas têm sido desenvolvidas na tentativa de se obter um bom balanceamento entre tempo de execução, complexidade e qualidade do escalonamento [64].

Grades computacionais estão sendo usadas atualmente como um ambiente colaborativo para a e-Ciência. Existe uma grande gama de aplicações em e-Ciência que são compostas por tarefas acopladas, as quais necessitam comunicação entre si, como por exemplo Montage [37], Chimera [3], AIRSN [108] e LIGO [36]. Nesse contexto, aplicações compostas por um *workflow* de tarefas com dependências de dados são geralmente representadas como grafos acíclicos direcionados [93]. As necessidades dessas aplicações são compatíveis com o que uma grade computacional oferece, como capacidades de processamento e armazenamento elevadas. Assim, o desenvolvimento de algoritmos para escalonamento de DAGs, considerando suas dependências e as características das grades, são de grande interesse [27, 103].

Para aperfeiçoar a execução de tarefas dependentes representadas por DAGs em grades devemos considerar algumas dificuldades, das quais se destacam a heterogeneidade e o comportamento dinâmico dos recursos. Enquanto a heterogeneidade implica complexidade na seleção de recursos para execução das tarefas, o comportamento dinâmico (variação no desempenho e disponibilidade dos recursos) implica em complexidade tanto na predição de desempenho quanto no reescalonamento. Escalonamento, predição de desempenho e reescalonamento têm recebido atenção substancial de pesquisadores em grades atualmente [7, 29, 77, 92, 52, 53, 104].

Recentemente houve um aumento no interesse em heurísticas de escalonamento de DAGs em recursos heterogêneos. Além do crescente interesse em aplicações que podem ser modeladas por DAGs, como os *workflows* [93], a motivação para esse interesse também tem origem no surgimento de plataformas de processamento heterogêneas, principalmente as grades computacionais e, mais recentemente, a computação em nuvem.

Nesta tese, abordamos o problema de escalonamento de tarefas dependentes repre-

sentadas por grafos acíclicos direcionados em grades computacionais, levando em consideração características inerentes a esses ambientes. As contribuições apresentadas nesta tese são algoritmos para escalonamento de tarefas dependentes representadas por *workflows* que incluem: (i) um método para escalonamento dinâmico e adaptativo; (ii) um algoritmo de escalonamento bi-critério; (iii) um algoritmo estático para escalonamento de um único *workflow*; (iv) algoritmos para escalonamento de múltiplos *workflows*; e (v) simulações dos algoritmos descritos em sistemas heterogêneos compartilhados com a existência de carga externa que visam posicionar tais algoritmos dentro da literatura de escalonamento em grades. Apresentamos também outras contribuições como trabalhos adicionais, que incluem o desenvolvimento de uma infra-estrutura para grades com resultados experimentais que englobam a avaliação de diferentes formas de distribuir as tarefas em uma grade real, a avaliação de máquinas virtuais como recursos computacionais na grade e o desenvolvimento e avaliação de algoritmos para balanceamento de carga em sistemas heterogêneos utilizando teoria dos jogos.

Esta tese está organizada da seguinte forma: no Capítulo 2, introduzimos definições e conceitos básicos relativos às grades computacionais, grafos acíclicos direcionados e o problema de escalonamento. Em seguida, no Capítulo 3, descrevemos trabalhos relacionados nas áreas de escalonamento estático e dinâmico, escalonamento bi-critério e escalonamento de múltiplos DAGs. No Capítulo 4, apresentamos as heurísticas utilizadas no desenvolvimento dos algoritmos e nas simulações, bem como parâmetros e configurações gerais das simulações realizadas. Os algoritmos propostos são descritos e avaliados no Capítulo 5, onde também são descritos resumos dos trabalhos adicionais. No Capítulo 6, apresentamos a conclusão e potenciais trabalhos futuros.

Capítulo 2

Definições e Conceitos Básicos

Neste capítulo apresentamos os conceitos básicos e definições gerais utilizadas nesta tese. Aqui definimos o modelo de grade computacional considerado, com sua arquitetura e possíveis variações, além de definirmos o problema de escalonamento e qual a forma de representação dos *workflows*. Definições e notações específicas para os algoritmos propostos são introduzidas quando os mesmos são apresentados.

2.1 Grades Computacionais

Sistemas distribuídos vêm substituindo os supercomputadores na execução de aplicações que demandam alto poder de processamento e grande quantidade de memória. O desenvolvimento da tecnologia de redes locais permitiu a criação de *clusters* de computadores homogêneos, que tornaram-se uma alternativa de baixo custo e alto desempenho. As constantes evoluções da tecnologia de redes e de *software*, como sistemas operacionais e linguagens de programação, permitiram a criação de sistemas heterogêneos, transformando qualquer recurso computacional em um potencial colaborador em um sistema distribuído. A partir da premissa de que qualquer recurso computacional pode ser agregado a um sistema heterogêneo, surgiu o conceito de *grades computacionais* [47, 45].

Uma grade computacional é um sistema heterogêneo colaborativo, geograficamente distribuído, multi-institucional e dinâmico, onde qualquer recurso computacional ligado a uma rede, local ou não, é um potencial colaborador. Uma grade geralmente é um ambiente compartilhado, o que leva a variações na carga do sistema independentes da gerência da grade. Esse ambiente pode fornecer recursos a usuários que demandam grandes capacidades de processamento e armazenamento. Para oferecer esses recursos, a grade pode disponibilizá-los de maneira que o usuário submeta suas tarefas à grade, que as executa e devolve os resultados através de componentes que interagem entre si. Uma grade também pode disponibilizar aos usuários seus recursos através de interfaces de serviços. Em uma

grade orientada a serviços, a execução depende da disponibilidade dos serviços necessários oferecidos pela grade.

Devido à sua capacidade de agregar recursos, as grades computacionais têm potencialmente um grande poder de processamento. Um *middleware* para grades computacionais deve fornecer suporte à execução de aplicações de tipos e tamanhos variados em um ambiente heterogêneo e dinâmico. Oferecer confiabilidade nesse ambiente não é de forma alguma trivial, já que o desempenho e a disponibilidade dos recursos podem variar imprevisivelmente. Alguns serviços imprescindíveis para o funcionamento de uma grade computacional são os serviços de escalonamento, monitoramento de recursos, gerência de recursos, controle de acesso e gerência de processos [23, 47]. O tamanho de uma grade pode variar muito em pouco tempo, então esses serviços e o *middleware* como um todo devem ser escaláveis, além de desejavelmente independentes de plataforma.

Grades computacionais são atualmente um grande foco de estudos relacionados à execução de aplicações paralelas, tanto aquelas que demandam grande poder computacional quanto aquelas que se adaptam bem a ambientes distribuídos [45, 30, 51, 65, 44, 83, 88, 20, 66, 70, 28]. Como os recursos de uma grade podem pertencer a vários domínios administrativos diferentes com políticas de acesso diferenciadas, cada recurso tem autonomia para participar ou deixar de participar da grade em qualquer momento. Essa característica dinâmica e a heterogeneidade tornam o escalonamento de aplicações, a gerência de recursos e a tolerância a falhas grandes desafios nesses sistemas.

O ambiente computacional considerado neste trabalho é um conjunto de recursos heterogêneos $\mathcal{R} = \{r_1, r_2, ..., r_k\}$, com capacidades de processamento associadas $p_{r_i} \in \mathbb{R}^+$, conectados por enlaces heterogêneos. Cada recurso r_i possui um conjunto de enlaces (*links*) $\mathcal{L}_i = \{l_{i,1}, l_{i,2}, ..., l_{i,m}\}$, $1 \leq m \leq k$, onde $l_{i,j} \in \mathbb{R}^+$ é a largura de banda disponível no enlace entre os recursos r_i e r_j, com $l_{i,i} = \infty$. Esses recursos interconectados por uma rede são capazes de transmitir e processar dados simultaneamente, porém só podem executar uma tarefa da grade de cada vez. Como $l_{i,i} = \infty$, o custo de comunicação entre duas tarefas que executam no mesmo recurso é considerado nulo.

A arquitetura da infra-estrutura computacional considerada é uma grade composta de grupos de recursos. Recursos dentro de um mesmo grupo têm enlaces com mesma largura de banda entre si. Ainda, um grupo pode ter apenas um recurso, caracterizando um computador pessoal ou qualquer tipo de recurso que é de alguma forma independente. A Figura 2.1 ilustra essa arquitetura, onde a espessura das arestas representa a largura de banda, o diâmetro dos recursos representa o poder de processamento e os tons de cinza dos recursos computacionais representam diferentes sistemas operacionais.

Consideramos que cada recurso possui uma fila de tarefas a serem executadas. O escalonador define a ordem das tarefas nessas filas, e, uma vez que uma tarefa inicia sua execução, ela executa sozinha no recurso até finalizar, ou seja, não sofre preempção. Nesse

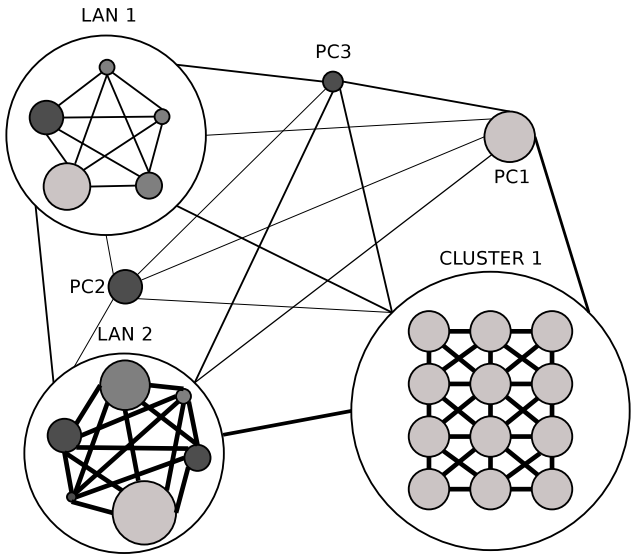


Figura 2.1: Exemplo da infra-estrutura computacional.

contexto a palavra *sozinha* significa que a tarefa não divide o recurso com outra tarefa submetida na grade. Entretanto, ela pode dividir o recurso com tarefas não gerenciadas pelo *middleware* da grade, o que caracterizaria uma carga externa. Então, não há execução simultânea de tarefas gerenciadas pela grade no mesmo recurso.

2.2 Grafos Acíclicos Direcionados

Tarefas submetidas para execução em uma grade podem ser, de forma geral, divididas em duas classes: tarefas independentes e tarefas dependentes. Enquanto tarefas independentes não realizam comunicação entre si, tarefas dependentes têm um fluxo de dados em que cada tarefa depende de dados computados por outras tarefas. Um conjunto de tarefas dependentes pode ser chamada de *workflow*: uma coleção de tarefas que possuem dependências entre si. Esse paradigma tem sido usado amplamente na representação de processos científicos [93]. Com o surgimento das grades computacionais, o escalonamento de *workflows*, particularmente aqueles de aplicações de e-Ciência, têm recebido atenção substancial [5, 40, 6, 80, 12, 77, 97, 27]. Algumas aplicações conhecidas que usam esse paradigma são Montage [37], AIRSN [108], LIGO [36], Chimera [3] e CSTEM [38], incluindo aplicações em química, biologia, física e ciência da computação.

Em um *workflow* a execução de cada tarefa deve respeitar as precedências impostas por tais dependências. Então, o escalonador deve tratar das dependências de dados, custos de comunicação entre tarefas e custos de computação das tarefas componentes do processo.

Em um sistema heterogêneo e dinâmico como uma grade computacional, esse problema adquire novas variáveis e torna-se ainda mais complexo.

Um *workflow* (ou processo) composto por tarefas dependentes é tipicamente modelado como um grafo acíclico direcionado (directed acyclic graph - DAG). Para cada nó n de um DAG não existe qualquer caminho direcionado que inicia e termina em n. Um processo é representado por um DAG $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ com n nós (ou tarefas), onde $t_a \in \mathcal{V}$ é uma tarefa do *workflow* com um custo de computação (peso) $w_{t_a} \in \mathbb{R}^+$ associado, e $e_{a,b} \in \mathcal{E}$, é uma dependência de dados entre t_a e t_b com custos de comunicação $c_{a,b} \in \mathbb{R}^+$ associados. Assim:

- $\mathcal{V}$ é o conjunto de $n = |\mathcal{V}|$ nós, e $t_i \in \mathcal{V}$ representa uma tarefa atômica que deve ser executada em um recurso;

- $\mathcal{E}$ é o conjunto de $e = |\mathcal{E}|$ arestas direcionadas (arcos), onde $e_{i,j} \in \mathcal{E}$ representa dependência de dados entre t_i e t_j. Isso implica que t_j não pode iniciar sua execução antes que t_i termine e envie os dados necessários à t_j;

Um nó sem predecessores é chamado de *nó de entrada* e um nó sem sucessores é chamado de *nó de saída*. Podemos assumir, sem perda de generalidade, que todo DAG possui um, e somente um, nó de saída. Isso pode ser alcançado adicionando uma tarefa sem custo t_{saida} e adicionando um conjunto de arcos sem custo $\{e_{a,saida} \; \forall \; t_a \in \mathcal{V} \mid t_a$ não possui sucessores$\}$. De forma análoga pode-se criar um nó de entrada único. Cada nó (ou tarefa) do grafo é rotulado com o custo de computação e cada arco é rotulado com o custo de comunicação. Uma tarefa é dita *pronta* para ser escalonada se todos os seus predecessores estão escalonados, isto é, t_i é uma tarefa *pronta* se $\{\forall \; t_h \in pred(t_i)\} \Rightarrow \{t_h$ está escalonada$\}$.

Essa representação permite a visualização dos componentes que formam a aplicação e como esses componentes interagem entre si. A Figura 2.2 mostra um exemplo de *workflow* representado por um grafo acíclico direcionado. As tarefas são representadas pelos nós e as dependências entre tarefas são representadas pelos arcos. Além da numeração, cada nó possui um rótulo com seu custo computacional. Por exemplo, a tarefa 1 tem custo computacional 3500. Os rótulos dos arcos representam o custo de comunicação entre as tarefas. Por exemplo, o custo de comunicação entre os nós 1 e 2 é 200. Como o nó 2 depende dos dados enviados pelo nó 1, o nó 2 só pode iniciar sua execução após o término da execução do nó 1. No caso do nó 11, as dependências de dados podem ser interpretadas de maneiras diferentes. Uma maneira é interpretar a condição para que a tarefa 11 seja executada como um E lógico, isto é, a tarefa 11 só pode iniciar sua execução após o término da execução e recebimento dos dados das tarefas 10, 3 *e* 9. Essa é a forma mais comum de interpretação, e a grande maioria dos algoritmos de escalonamento utiliza

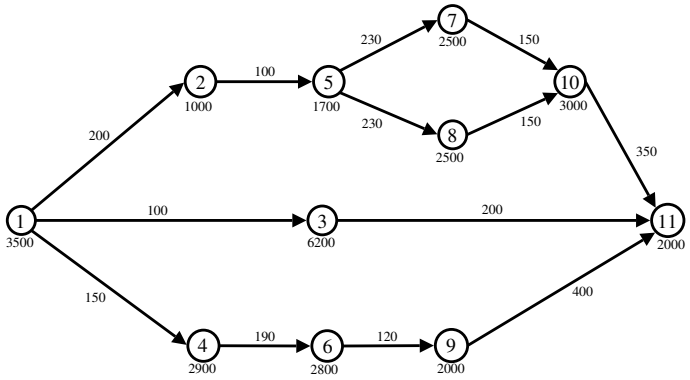


Figura 2.2: Exemplo de DAG representando um processo de *workflow*.

somente esse tipo de interpretação. Existem outras maneiras de interpretar essa condição de dependência, como a apresentada por Cicerre e outros em [30], onde uma linguagem baseada em BPEL permite que as dependências sejam interpretadas como um *OU* lógico. Dessa forma, a tarefa 11 pode iniciar sua execução assim que uma das três tarefas, 10 *ou* 3 *ou* 9, termine sua execução e envie os dados. Essa condição também pode ser uma composição de *Es* e *OUs* lógicos. Por exemplo, o início da tarefa 11 está sujeito à condição lógica (10 *OU* 3 *E* 9). Grafos de processos interpretados dessa maneira são chamados de *grafos condicionais*, que não são considerados pelos algoritmos aqui apresentados.

DAGs tornaram-se o padrão na representação de processos, com suas tarefas dependentes e fluxos de execução. Com a evolução da e-Ciência, esses *workflows* tornaram-se maiores e passaram a demandar mais poder computacional. Essas demandas dos *workflows* científicos podem ser supridas utilizando grades computacionais em conjunto com escalonadores eficientes. Além disso, grades são compartilhadas e distribuídas, então podem receber mais de uma requisição de execução de *workflow* no mesmo conjunto de recursos ao mesmo tempo. Dessa forma, o aprimoramento constante de algoritmos e heurísticas é fundamental para melhorar o desempenho na execução de *workflows* em grades [93]. As palavras *processo* e *workflow* usadas durante o texto referem-se ao DAG, que por sua vez representa uma aplicação paralela composta por tarefas dependentes.

2.3 O Problema de Escalonamento

O paradigma de grade computacional tornou-se amplamente utilizado no processamento de grande quantidade de dados sobre um sistema geograficamente distribuído. Grades computacionais estão recebendo atenção substancial da comunidade de pesquisa desde meados dos anos 90 [47]. Uma grade computacional é dinâmica, heterogênea e compar-

tilhada, o que traz novas características e novos problemas a serem tratados. Problemas relativos à escalabilidade e gerência de recursos incluem novas variáveis a serem consideradas, apresentando novos desafios. Entre esses desafios, o escalonamento de tarefas é de grande importância.

O estudo do escalonamento começou a tornar-se importante no início do século XX, sendo Henry Laurence Gantt um dos principais pioneiros na área. O problema abordado inicialmente foi o escalonamento na produção industrial, evoluindo durante os anos através da formalização dada por formulações em programação dinâmica e programação inteira, e pela hierarquização da complexidade dos problemas de escalonamento [76]. Mais tarde, com a popularização dos computadores, o desenvolvimento de sistemas computacionais de escalonamento passou a ser estudado, sendo um foco de pesquisa atualmente. Portanto, o escalonamento é um problema genérico que é encontrado nas mais variadas áreas, sendo definido de maneiras diferentes de acordo com o problema abordado. A definição dada por Pinedo em [76] é a seguinte:

Scheduling is a decision-making process that is used on a regular basis in many manufacturing and services industries. It deals with the allocation of resources to tasks over given time periods and its goal is to optimize one or more objectives.

Em um sistema distribuído, o escalonador tem como função escolher qual tarefa será executada em qual recurso computacional, distribuindo tarefas para execução em paralelo. Assim, o tempo de execução de um conjunto de tarefas em um sistema distribuído é intrínseco ao escalonamento que lhe for atribuído. O escalonador distribui as tarefas nos recursos de acordo com uma função objetivo a ser otimizada. A função objetivo mais encontrada na literatura é a minimização do tempo de execução (*makespan*) do conjunto de tarefas. Outros critérios a serem otimizados, como por exemplo maximizar o uso dos recursos ou minimizar a comunicação, também podem ser considerados.

A partir do problema a ser considerado, do ambiente alvo e dos objetivos a serem alcançados, definimos o problema de escalonamento abordado. O problema de escalonamento é descrito como uma tripla $\alpha \mid \beta \mid \gamma$ [76]. O campo α descreve o ambiente de processamento e contém apenas uma entrada. O campo β fornece detalhes e características de processamento e restrições e pode não conter entradas, conter uma única entrada ou múltiplas entradas. O campo γ descreve o objetivo a ser minimizado e freqüentemente contém uma única entrada. Nesta tese focamos no problema em que o ambiente de processamento é composto por m máquinas não-relacionadas em paralelo ($\alpha = R_m$), onde existem restrições de precedência entre tarefas ($\beta = pred$), e o objetivo principal a ser minimizado é o *makespan* ($\gamma = C_{max}$). Dessa forma, o problema geral abordado nesta tese é o escalonamento $R_m \mid pred \mid C_{max}$.

O escalonamento é fundamental para atingir bom desempenho na execução de processos em sistemas distribuídos, e em um sistema heterogêneo como as grades, novas variáveis devem ser consideradas. Nesses sistemas heterogêneos, cada tarefa pode ter tempos de computação e comunicação diferentes em cada recurso, já que cada recurso pode ter capacidade computacional singular e os enlaces podem ter diferentes larguras de banda. Sendo assim, recentemente houve um crescente interesse em algoritmos para escalonamento de DAGs em recursos heterogêneos. Parte da motivação desses trabalhos vem da difusão de plataformas de processamento heterogêneas, como as grades, e o crescente interesse em aplicações que podem ser modeladas por DAGs, como *workflows* científicos [35, 93].

Para otimizar a função objetivo desejada, o escalonador pode utilizar informações sobre o estado atual do sistema, o que guia o processo de decisão sobre o recurso em que cada tarefa será executada. Exemplos de informações sobre os recursos que podem ser utilizadas incluem o poder de processamento, quantidade de memória volátil e não volátil disponíveis, largura de banda, contenção na rede e proximidade de tarefas acopladas. O peso de cada uma dessas informações na decisão de escalonamento pode depender da aplicação e das condições atuais dos recursos da grade.

Algoritmos de escalonamento podem ser classificados em escalonadores para tarefas independentes e escalonadores para tarefas dependentes. Escalonamento de tarefas independentes não necessita de tratamento de comunicação entre tarefas, então cada tarefa pode ser escalonada independentemente. Já no escalonamento de tarefas dependentes, que é o foco deste trabalho, o algoritmo deve levar em consideração a comunicação originada pelas dependências de dados entre tarefas. Esses custos estão implícitos quando as tarefas são distribuídas para execução em um conjunto de recursos distribuídos, com a heterogeneidade dos enlaces tendo um papel importante no tempo de comunicação.

Outra classificação dos algoritmos de escalonamento é que estes podem ser estáticos ou dinâmicos. Escalonadores estáticos promovem o escalonamento de todas as tarefas usando informações disponíveis no momento do início do escalonamento, enquanto escalonadores dinâmicos escalonam as tarefas em turnos, atualizando as informações sobre os recursos a cada turno de escalonamento. É importante notar que algoritmos de escalonamentos estáticos são, de forma geral, o ponto de partida para algoritmos de escalonamento dinâmicos. Dessa forma, um bom algoritmo de escalonamento estático é primordial para a obtenção de escalonamentos dinâmicos eficientes.

Sendo o escalonamento de tarefas um problema NP-Completo [76], o problema de otimização associado é NP-Difícil e não possui algoritmo conhecido que gere soluções ótimas em tempo polinomial de forma determinística. Com isso, algumas técnicas são utilizadas na tentativa de aproximar a solução ótima com algoritmos de complexidade de tempo polinomial. Algumas técnicas e suas características são listadas abaixo:

- **Heurísticas**: podem apresentar baixa complexidade e execução rápida, porém geralmente não fornecem garantias em relação à qualidade da solução obtida;

- **Meta-heurísticas**: tempo de execução depende da condição de parada (por exemplo, número de iterações) imposta pelo programador. Para alcançar soluções de boa qualidade, geralmente precisam de um tempo de execução maior que as heurísticas. Também não fornecem garantias sobre a qualidade da solução: ótimos locais são freqüentemente a solução final;

- **Programação linear**: tempo de execução e qualidade da solução dependem do relaxamento de restrições e da redução do número de variáveis;

- **Algoritmos de aproximação**: a obtenção de algoritmos de aproximação com baixa complexidade e com fator de aproximação razoável é difícil para problemas genéricos, envolvendo o uso de ferramentas do desenvolvimento de algoritmos exatos e a obtenção de limitantes justos [96]. Algoritmos de aproximação fornecem garantia quanto à qualidade da solução, apresentando limitantes em relação à solução ótima. Quanto mais geral o problema, mais difícil é a obtenção de uma aproximação satisfatória.

A abordagem mais freqüentemente encontrada na literatura de escalonamento de DAGs é o uso de heurísticas, que combinam soluções com boa qualidade, de forma geral, e baixa complexidade. Em conseqüência dessas características, os algoritmos que desenvolvemos são baseados na utilização de heurísticas.

Capítulo 3

Trabalhos Relacionados

O problema geral abordado nesta tese é o escalonamento de tarefas dependentes, ou *workflows*, representadas por grafos acíclicos direcionados (DAGs). Este capítulo apresenta uma visão geral dos trabalhos específicos em escalonamento abordados nesta tese, introduzindo tais problemas e discutindo trabalhos relacionados encontrados na literatura. Assim, a estrutura deste capítulo está baseada nos problemas descritos, que são:

- Escalonamento estático de DAGs;
- Escalonamento dinâmico de DAGs;
- Escalonamento bi-critério;
- Escalonamento de múltiplos DAGs.

3.1 Escalonamento Estático de DAGs

Atualmente, as grades computacionais são ambientes largamente difundidos, sendo encontradas em muitas instituições de pesquisa, universidades, órgãos governamentais e empresas espalhados por todos os continentes. Exemplos de grades incluem: InteGrade [1], CoreGrid [2], OurGrid [3], Grid5000 [4], LHC Computing Grid [5], World Community Grid [6], Open Science Grid [7], entre outros. Com o advento da e-Ciência, a utilização desses

[1] http://ccsl.ime.usp.br/integrade/
[2] http://www.coregrid.net/
[3] http://www.ourgrid.org/
[4] https://www.grid5000.fr/
[5] http://lcg.web.cern.ch/LCG/
[6] http://www.worldcommunitygrid.org/
[7] http://www.opensciencegrid.org/

ambientes para a execução de *workflows* é crescente. Alguns exemplos de aplicações de *workflows* executadas em grades que necessitam de grande capacidade de processamento e/ou armazenamento são Montage [37], AIRSN [108], LIGO [36] e Chimera [3]. Como conseqüência, durante os anos várias heurísticas foram desenvolvidas na tentativa de alcançar um bom balanceamento entre tempo de execução, complexidade e qualidade do escalonamento [64].

Sendo o escalonamento de tarefas um problema NP-Completo [76], a maior parte dos esforços em algoritmos de escalonamento estão direcionados às heurísticas e metaheurísticas [40, 58, 32, 54]. Há alguns trabalhos que utilizam algoritmos de aproximação [75, 49] e outros que utilizam abordagens conhecidas no campo da combinatória [6].

Mais especificamente, a área de escalonamento de tarefas em sistemas distribuídos vem sendo bastante explorada. Tanto sistemas distribuídos homogêneos [56, 62, 102, 41, 64, 101] quanto heterogêneos [95, 16, 29, 40, 80, 26, 21, 55, 5, 63, 82, 9] possuem vasta bibliografia no escalonamento de tarefas dependentes. Descrições extensas sobre escalonamento em grades computacionais são feitas em [39] e [60]. No escalonamento de tarefas em sistemas distribuídos, o objetivo a ser otimizado mais encontrado na literatura é a minimização do *makespan*, ou seja, a busca por um escalonamento que resulte em um tempo de execução mínimo para as tarefas.

Uma técnica bastante utilizada em heurísticas para escalonamento de *workflows* é o *list scheduling*, em que cada tarefa a ser escalonada recebe uma prioridade calculada de acordo com o algoritmo utilizado. As tarefas são escalonadas seguindo a ordem de prioridade, com o recurso sendo selecionado de acordo com critérios estabelecidos pelo algoritmo de escalonamento. Um exemplo de algoritmo que utiliza essa técnica é o *Heterogeneous Earliest Finish Time* (HEFT) [95]. O HEFT é um algoritmo de escalonamento de tarefas dependentes em sistemas heterogêneos notadamente reconhecido por sua simplicidade e baixa complexidade. Esses dois fatores aliados aos bons escalonamentos gerados nos mais variados cenários tornaram o HEFT um dos algoritmos mais populares no escalonamento estático de *workflows* em sistemas heterogêneos. O algoritmo *Critical Path on a Processor* (CPOP) [95] é similar ao HEFT. A principal diferença é que no CPOP todos os nós no caminho crítico do DAG são escalonados no mesmo recurso.

Outra técnica encontrada em heurísticas de escalonamento de tarefas dependentes é o *clustering*. Essa técnica consiste em criar grupos de tarefas (chamados de *clusters* de tarefas) que têm dependência de dados entre si. Essas tarefas são escalonadas no mesmo recurso, almejando a redução dos custos de comunicação, conseqüentemente minimizando o tempo total de execução do *workflow*. O algoritmo *Path Clustering Heuristic* [17] utiliza uma combinação da técnica de *clustering* com a técnica de *list scheduling*.

A técnica de *task duplication* também é utilizada em heurísticas de escalonamento de tarefas [74, 55]. Essa técnica consiste em duplicar tarefas em mais de um recurso. Dessa

forma, cada tarefa pode ser escalonada em dois ou mais recursos distintos e o resultado da instância que finalizar primeiro pode ser utilizado. Além disso, a duplicação das tarefas insere um certo nível de tolerância a falhas na execução do *workflow*. Por outro lado, essa técnica utiliza mais recursos que o estritamente necessário, já que a mesma tarefa poderá ocupar mais de um recurso, reduzindo a disponibilidade para outras tarefas e, conseqüentemente, podendo aumentar o tempo de execução global dos *workflows*.

Condor é um conhecido gerenciador de recursos entre domínios, que é utilizado em grades pelo Condor-G [94]. Um meta-escalonador de DAGs para grades que utiliza o Condor para despachar as tarefas é o DAGMan [48]. Sua proposta é realizar uma pré-seleção de tarefas prontas para serem executadas e, de acordo com essa seleção, enviar as tarefas de forma independente ao escalonador. Então, o escalonador realiza o procedimento de escalonamento das tarefas sem considerar as dependências entre elas, o que pode acarretar altos custos de comunicação

Cooper e Dasgupta [31] propuseram uma estratégia de escalonamento para o projeto GraDS, incluindo uma abordagem para escalonamento de *workflows* e reescalonamento de aplicações. O escalonamento de *workflows* no projeto GraDS usa uma estratégia de classificar cada recurso de acordo com sua afinidade com cada componente do processo, em que quanto menor a classificação, melhor a afinidade do recurso com os componentes do processo. O escalonador coloca essas informações em uma matriz de desempenho em que o elemento $e_{i,j}$ da matriz mostra a afinidade do componente (tarefa) i com o recurso j. Finalmente, três heurísticas são executadas sobre a matriz para determinar o escalonamento de cada componente.

Um algoritmo de aproximação para escalonamento de tarefas dependentes é apresentado por Fujimoto e Hagihara em [49]. O algoritmo usa a utilização dos recursos como critério de desempenho, não focando na minimização do *makespan*. Pelo fato de algoritmos de aproximação gerais serem difíceis de se obter, os autores fazem algumas suposições para tornar o problema mais específico, como por exemplo considerar que todas as tarefas do *workflow* têm o mesmo peso.

Uma avaliação comparativa de 20 diferentes heurísticas pode ser encontrada em [26]. Entre essas heurísticas, a *Heterogeneous Earliest Finish Time* (HEFT) , que é uma das mais freqüentemente citadas e utilizadas, tem a vantagem de ser simples e produzir escalonamentos de boa qualidade com um *makespan* menor na maioria dos cenários.

Nesta tese, na área de escalonamento estático de um único DAG e com apenas um critério de otimização, apresentamos modificações no algoritmo HEFT, utilizando a técnica de *lookahead* na seleção de recursos, aumentando o espaço de busca do algoritmo para alcançar melhores resultados.

3.2 Escalonamento Dinâmico de DAGs

Grades computacionais são ambientes compartilhados e, em geral, não dedicados. Essa característica leva os recursos que compõem a grade a terem variações na carga, o que pode interferir imprevisivelmente no tempo de execução das tarefas controladas pela grade. Assim, algoritmos de escalonamento dinâmicos são desejáveis em grades computacionais.

Prodan e Fahringer apresentam um estudo de caso em escalonamento dinâmico para *workflows* científicos em grades em [77]. É proposto um reescalonamento dinâmico com iterações sobre o *workflow*, gerando um DAG com tarefas que devem ser re-escalonadas em cada iteração. Após isso, o DAG gerado é escalonado, conseqüentemente realizando um reescalonamento de suas tarefas.

Um escalonador dinâmico em dois níveis para *wide area networks* (WANs) é proposto por Chen e Maheswaran em [29], onde um escalonador de nível macro consulta escalonadores no segundo nível para selecionar em que LAN o processo será executado. Não há mecanismo para dividir o processo para execução em mais de uma LAN, o que pode ocasionar tempos de execução maiores. Ainda, todos os escalonadores no segundo nível devem responder a todas as requisições de escalonamento.

Maheswaran e Siegel [71] realizam escalonamento dinâmico baseado em um remapeamento híbrido que utiliza informações obtidas em tempo de execução em conjunto com valores estimados. As tarefas do DAG são separadas em blocos de acordo com seu nível no DAG, fornecendo uma quantidade fixa de avaliações a serem feitas durante o processo de escalonamento. Os autores apresentam resultados que mostram melhoras em relação ao escalonamento estático.

Um escalonador de tarefas dinâmico pode usar, de forma geral, duas estratégias: escalonar todas as tarefas estaticamente e então reescalonar tarefas quando necessário, ou escaloná-las e executá-las gradualmente, à medida que outras tarefas terminam. É importante salientar que escalonar uma tarefa não significa despachá-la para o recurso que a executará, mas apenas que existe um escalonamento já realizado. Neste trabalho apresentamos uma abordagem que utiliza a primeira estratégia, onde um escalonamento inicial é realizado, porém as tarefas não são todas despachadas para execução. O envio de tarefas para execução é baseado no desempenho dos recursos escolhidos, sendo adaptado de acordo com o desempenho nas execuções passadas.

3.3 Escalonamento Bi-Critério

No problema de escalonamento tradicional em sistemas distribuídos, que considera um único critério a ser otimizado, geralmente o objetivo é minimizar o tempo de execução das tarefas. No caso do escalonamento de *workflows*, isso tem como conseqüência a min-

imização do tempo de execução total do *workflow*. No escalonamento multi-critério, o objetivo é otimizar mais de um critério ao mesmo tempo, em geral considerando o tempo de execução como o critério mais importante [98]. Quando um algoritmo de escalonamento realiza a otimização de dois critérios, chamamos esse algoritmo de algoritmo de escalonamento bi-critério. Por exemplo, com o surgimento da *utility computing* e da computação em nuvem, é desejável a minimização dos custos econômicos em conjunto com a minimização do tempo de execução das tarefas [103].

Em um problema de otimização multi-critério a definição de otimalidade depende de como duas possíveis soluções são comparadas. Na verdade, mais de uma solução ótima pode existir se considerarmos que duas soluções não podem ser diretamente comparadas quando uma solução é melhor que outra em um conjunto de critérios, porém pior em outro conjunto. Nesse sentido, quando otimizamos uma função objetivo multi-critério, desejamos encontrar o *conjunto Pareto-ótimo* de soluções, que é composto por todas as soluções não-dominadas. Dizemos que uma solução x_1 domina uma solução x_2 quando existem W critérios de otimização se as duas condições a seguir são satisfeitas:

1. x_1 não é pior que x_2 em nenhum dos critérios, ou seja, $f_w(x_1) \leq f_w(x_2)\ \forall\ w \in \{1, ..., W\}$.

2. x_1 é estritamente melhor que x_2 em pelo menos um critério, ou seja, $\exists\ w \in \{1, ..., W\}$ tal que $f_w(x_1) < f_w(x_2)$.

Uma técnica para tratar problemas de otimização bi-critério é manter um critério entre limiares fixos enquanto otimiza-se o outro critério, diminuindo o espaço de busca [103]. A utilização de uma função objetivo agregada (*aggregate objective function* - AOF) é outra técnica, onde ambos os objetivos são combinados em uma única função a ser otimizada [98]. Uma AOF comum é realizar uma soma linear ponderada dos objetivos: $f(obj_1, obj_2) = \alpha \times obj_1 + (1 - \alpha) \times obj_2,\ \alpha \in [0, 1]$. Outra técnica é a otimização multi-objetivo usando algoritmos evolucionários (*multiobjective optimization evolutionary algorithms* - MOEA).

Na área de grades computacionais, muitos *middlewares* focam em *workflows* de tarefas. Alguns outros fornecem mecanismos para execução de *workflows* compostos por tarefas que executam em serviços da grade [88]. A execução de *workflows* em serviços da grade pode ser melhorada através da instanciação dinâmica de serviços.

O trabalho descrito em [78] foca em mecanismos para provisão de instanciação dinâmica de serviços baseada em uma infra-estrutura de instanciação dinâmica de alta disponibilidade. Entretanto, não há menção à necessidade de um escalonador de serviços para escolher os melhores recursos para a instanciação dinâmica.

O DynaGrid [24] é um arcabouço para a estruturação de grades de larga escala para aplicações compatíveis com WSRF (*web services resource framework*). O DynaGrid provê

mecanismos para instanciação dinâmica de serviços, porém não cita suporte a *workflows*.

No campo de escalonamento há trabalhos que tratam de escalonamento bi-critério, mas nenhum deles aborda o problema específico de instanciação dinâmica de serviços. Wieczorek e outros tratam em [98] do escalonamento bi-critério de *workflows* em grades através da modelagem do problema como uma extensão do problema da mochila de múltipla escolha, apresentando bons resultados quando comparado com algoritmos que consideram *makespan* e restrições de orçamento como critérios. Zeng e outros abordam a otimização multi-critério usando programação linear inteira em [105], focando no preço, duração, disponibilidade e taxa de sucesso na execução das tarefas. Robustez e *makespan* são os dois critérios tratados por Canon e Jeannot em [25], enquanto Dogan e Özgüner [38] tratam do *trade-off* entre tempo de execução e confiabilidade. Ainda, alguns trabalhos focam em custos econômicos [103], o que pode ser estendido ao trabalho apresentado nesta tese se considerarmos que a instanciação de um serviço acarreta custos monetários ao usuário. Adicionalmente, em [103] não há menção a abordagens para esse problema. Dessa forma, o problema de minimização do número de serviços criados em conjunto com o *makespan* do *workflow* é abordado nesta tese de forma original.

Nesta tese, abordamos o problema de escalonamento bi-critério em grades orientadas a serviço. Nesse contexto, a execução de *workflows* está fortemente ligada ao conjunto de recursos em que os serviços estão previamente publicados. Se há um serviço muito requisitado, isso pode levar a uma sobrecarga, o que limita o uso dos recursos da grade de acordo com a freqüência de submissões de *workflows* que usam um ou outro tipo de serviço. Dessa forma, recursos menos potentes podem ser mais utilizados que recursos mais potentes, resultando em um maior tempo de execução dos *workflows*. Para tratar essa limitação, nesta tese apresentamos um algoritmo que escalona serviços de acordo com os *workflows* submetidos. O algoritmo proposto se apóia na existência de uma infraestrutura que forneça tanto a execução de *workflows* de serviços [88], como a instanciação dinâmica de serviços da grade [78]. O algoritmo considera os custos de instanciação dos serviços ao selecionar os recursos para execução do *workflow*. Assim, caracterizamos esse escalonador como bi-critério, tendo como objetivos a minimização do tempo de execução do *workflow* e a minimização do número de novos serviços criados. Nós abordamos esse problema com uma heurística que usa o conceito de *ALAP - as late as possible* [89].

3.4 Escalonamento de Múltiplos DAGs

Considerando que grades são ambientes compartilhados, eventualmente elas precisarão tratar de mais de um *workflow* ao mesmo tempo, assim como seus recursos poderão executar tarefas provenientes de usuários de fora da grade. Com isso, é necessário o desenvolvimento de *middlewares* de grades que gerenciem mais de um *workflow* em seu

conjunto de recursos. Além disso, esses algoritmos devem ter bom desempenho quando existe carga externa nos recursos e, como parte importante do *middleware*, o escalonador também deve ser capaz de realizar um escalonamento eficiente nesse cenário. Apesar de existir uma grande gama de trabalhos no escalonamento de *workflows*, o escalonamento de múltiplos *workflows* ainda é um problema em aberto, apenas com estudos iniciais disponíveis na literatura [107]. O aperfeiçoamento da execução de múltiplos *workflows* pode acelerar a geração de resultados quando processos são submetidos para execução na grade, contribuindo com o desenvolvimento da e-Ciência.

Como citado anteriormente, o objetivo mais comum no escalonamento de tarefas é a minimização do *makespan*, que é o responsável direto pelo tempo de execução do *workflow*. Quando múltiplos *workflows* compartilham o mesmo ambiente de execução, além da minimização do *makespan*, há conflitos que devem ser observados e tratados para garantir a eficiência do sistema. Por exemplo, como escalonar *workflows* de maneira que nenhum deles seja alocado em recursos com desvantagens em termos de tempo de execução quando comparado aos outros *workflows* na grade. Se existem múltiplos *workflows* a serem escalonados e o algoritmo considera apenas um por vez, independentemente, o primeiro *workflow* estará em vantagem e seu *makespan* será minimizado mais intensamente que o do último *workflow* a ser escalonado. Dessa forma, um algoritmo de escalonamento deve considerar a justiça para compartilhar os recursos de forma igual entre os *workflows* submetidos.

Zhao e Sakellariou analisam alguns métodos de composição de *workflows* para realizar o escalonamento de múltiplos *workflows* [107], além de apresentar duas políticas para obter justiça entre os *workflows* escalonados. Os autores analisam o comportamento de duas heurísticas de escalonamento para um único *workflow*, utilizando os métodos e políticas propostas para que os múltiplos *workflows* sejam escalonados por essas heurísticas. A conclusão encontrada é que é possível atingir um bom nível de justiça entre os *workflows* sem comprometer o *makespan* global quando comparado ao *makespan* atingido utilizando-se as heurísticas para escalonamento de um único *workflow*. Ainda, os autores deixam claro em suas conclusões que este é apenas um estudo inicial em heurísticas para o escalonamento de múltiplos *workflows* e que trabalhos futuros são necessários.

Considerando a importância em distribuir as tarefas do *workflow* nos recursos de forma que a capacidade de processamento seja compartilhada de forma justa entre os *workflows*, além da importância em se obter resultados de execução rapidamente, nesta tese descrevemos quatro abordagens diferentes para escalonamento de múltiplos *workflows* e analisamos como cada estratégia se comporta em relação à justiça e *makespan*. Além disso, avaliamos qual o impacto da existência de carga externa no escalonamento resultante de cada uma das quatro abordagens apresentadas.

Capítulo 4

Heurísticas Utilizadas e Configurações Gerais de Simulação

Nesta tese, utilizamos duas heurísticas nas simulações dos algoritmos apresentados: PCH e HEFT. O algoritmo PCH foi utilizado no desenvolvimento dos algoritmos de escalonamento dinâmico e nos algoritmos de escalonamento de múltiplos DAGs. No algoritmo que propõe a utilização de *lookahead* durante o escalonamento, utilizamos a heurística HEFT. Descrevemos nesta seção esses dois algoritmos, além dos parâmetros utilizados nas simulações.

4.1 PCH

O algoritmo *Path Clustering Heuristic* (PCH) [16], foi desenvolvido durante o mestrado para ser utilizado no *middleware* Xavantes [30]. O PCH é um algoritmo de escalonamento de tarefas dependentes que utiliza as técnicas de *list scheduling* e *clustering*. Para reduzir a comunicação entre tarefas, o PCH agrupa caminhos do DAG, criando *clusters* de tarefas. Após a criação desses agrupamentos baseados em caminhos do grafo, o algoritmo escalona as tarefas que estão no mesmo grupo em um mesmo recurso. Dessa forma, a comunicação entre tarefas do mesmo grupo é suprimida.

4.1.1 Definições

O algoritmo PCH utiliza atributos calculados para cada tarefa (ou nó) do DAG. Esses atributos são computados da seguinte maneira:

• **Custo de computação:**

$$w_{i,r} = \frac{instrucões_i}{p_r}$$

$w_{i,r}$ representa o custo de computação do nó i no recurso r, e p_r é a capacidade de processamento do recurso r, em instruções por segundo.

• **Custo de comunicação:**

$$c_{i,j} = \frac{dados_{i,j}}{l_{r,p}}$$

$c_{i,j}$ representa o custo de comunicação entre os nós i e j, utilizando o enlace l entre os recursos r e p. Se $r = p$, então $c_{i,j} = 0$.

• **Prioridade:**

$$\mathcal{P}_i = \begin{cases} w_{i,r}, & \text{se } i \text{ é nó de saída} \\ w_{i,r} + \max\limits_{\forall\ t_j \in\ suc(t_i)} (c_{i,j} + \mathcal{P}_j), & \text{caso contrário} \end{cases}$$

$\mathcal{P}_i$ é o nível de prioridade do nó i em um dado momento do processo de escalonamento.

• $\mathbf{suc(t_i)}$ é o conjunto de sucessores imediatos do nó t_i no grafo acíclico direcionado.

• $\mathbf{pred(t_i)}$ é o conjunto de predecessores imediatos do nó t_i no grafo acíclico direcionado.

• **Earliest Start Time:**

$$EST(t_i, r_k) = \begin{cases} Tempo(r_k), & \text{se } i = 1 \\ \max\{Tempo(r_k), \max\limits_{\forall\ t_h \in\ pred(t_i)} (EST(t_h, r_k) + w_{h,k} + c_{h,i})\}, & \text{caso contrário} \end{cases}$$

$EST(t_i, r_k)$ representa o menor tempo de início possível para o nó i no recurso k em um dado momento do processo de escalonamento. $Tempo(r_k)$ é o tempo em que o recurso k está disponível para execução da tarefa.

• **Estimated Finish Time:**

$$EFT(t_i, r_k) = EST(t_i, r_k) + w_{i,k}$$

$EFT(t_i, r_k)$ representa o tempo estimado de término do nó i no recurso k.

O valor inicial para esses atributos é calculado atribuindo-se cada tarefa a um processador diferente em um sistema homogêneo virtual com um número ilimitado de processadores. Esses processadores têm capacidade de processamento iguais à do melhor recurso disponível no sistema real, e todos os enlaces possuem largura de banda igual à mais alta disponível no sistema real.

Após computar esses atributos, enquanto existirem nós não escalonados, o algoritmo cria um grupo (*cluster*) de tarefas e seleciona um recurso para ele. As informações necessárias para o cálculo dos atributos são fornecidas pelo *middleware* e pela especificação do DAG. Como essa informação é obtida e fornecida está fora do escopo deste trabalho. Como na maioria dos trabalhos em escalonamento de DAGs disponíveis na literatura, consideramos que o modelo de programação e/ou o *middleware* podem fornecer informações como o tamanho de cada tarefa, capacidade de processamento disponível e largura de banda para o escalonador. Por exemplo, o tamanho de cada tarefa pode ser obtido através de *benchmarks* de aplicações, histórico de execuções, históricos de entradas de dados, estimativas dadas pelo programador ou estimativas de acordo com execuções passadas e tamanho dos dados de entrada. Uma visão geral desses passos é dada pelo Algoritmo 1, com cada passo sendo detalhado nas seções seguintes.

Algoritmo 1 Algoritmo PCH

1: Atribui o DAG $\mathcal{G}$ ao sistema homogêneo virtual
2: Computa os atributos iniciais das tarefas de $\mathcal{G}$
3: **while** há nós não escalonados **do**
4: $cluster \Leftarrow$ próximo_agrupamento($\mathcal{G}$)
5: $recurso \Leftarrow$ seleciona_melhor_recurso($cluster$)
6: Escalona $cluster$ em $recurso$
7: Recalcula pesos, ESTs e EFTs.
8: **end while**

Inicialmente, o algoritmo atribui o DAG ao sistema homogêneo virtual para o cálculo inicial dos atributos (linhas 1 e 2). Em seguida, o algoritmo itera até que todas as tarefas tenham sido escalonadas. A cada laço da iteração um *cluster* de tarefas é criado (linha 4) e um recurso é selecionado para esse agrupamento (linha 5). Após escalonar o agrupamento criado no recurso selecionado, os atributos são recalculados para que na próxima iteração o algoritmo possa realizar esses mesmos passos com os atributos atualizados de acordo com o escalonamento atualizado.

4.1.2 Seleção de Tarefas e Agrupamento

Neste passo do algoritmo, a heurística seleciona tarefas para compor os grupos de tarefas. Tarefas que ficam no mesmo grupo serão inicialmente escalonadas no mesmo recurso. A

primeira tarefa t_i a ser selecionada para compor o grupo cls_k é a tarefa não escalonada com a maior prioridade. Ela é adicionada ao grupo cls_k, e então o algoritmo inicia uma busca em profundidade no DAG, começando pela tarefa t_i, sempre selecionando $t_s \in suc(t_i)$ com o maior $\mathcal{P}_s + EST_s$ e adicionando-a a cls_k, até que t_s não possua sucessores não escalonados. O Algoritmo 2 ilustra essa estratégia.

Algoritmo 2 próximo_agrupamento($\mathcal{G}$)

1: $t \Leftarrow$ nó não escalonado de $\mathcal{G}$ com maior prioridade.
2: $cluster \Leftarrow cluster \cup t$
3: **while** (t tem sucessores não escalonados) **do**
4: $\quad t_{suc} \Leftarrow sucessor_i$ de t com maior $\mathcal{P}_i + EST_i$
5: $\quad cluster \Leftarrow cluster \cup t_{suc}$
6: $\quad t \Leftarrow t_{suc}$
7: **end while**
8: **retorna** *cluster*

Na primeira linha o algoritmo que cria o próximo agrupamento seleciona o nó t não escalonado com a maior prioridade no grafo. Note que o nó não escalonado de maior prioridade sempre será um nó pronto, isto é, com todos os seus predecessores previamente escalonados. Em seguida, na linha 2, t é adicionada ao agrupamento que está sendo criado. Então, o algoritmo inicia um laço, adicionando na linha 4 o sucessor t_i de t que tenha maior $\mathcal{P}_i + EST_i$. O sucessor que tem essa característica, t_{suc}, é adicionado ao *cluster* na linha 5. Essa iteração se repete, agora com t_{suc} no lugar de t (linha 6), até que o nó atual t não possua sucessores não escalonados.

4.1.3 Seleção de Processadores

O passo de seleção de processadores é executado após cada passo de agrupamento de tarefas. O algoritmo de escalonamento cria um grupo de tarefas, seleciona um processador (ou recurso) para o grupo recebido e recalcula os atributos das tarefas, repetindo esses passos até que todas as tarefas estejam escalonadas.

Seja t_{n_k} a última tarefa do grupo cls_k. Podemos selecionar um processador para um grupo de tarefas com o objetivo de minimizar o tempo estimado de término de t_{n_k} (EFT_{n_k}). Outra maneira de selecionar um recurso para um grupo é considerar também o sucessor de t_{n_k}, pois se considerarmos somente o EFT de t_{n_k} estaríamos desprezando os custos de comunicação entre esta tarefa e seu sucessor, o que poderia atrasar a execução. Note que ao escalonar qualquer grupo de tarefas cls_m, os sucessores de t_{n_m} (a última tarefa do grupo) já estarão escalonados. Dessa forma, adotamos como critério para seleção de recurso para um grupo cls_k a minimização do EST do sucessor de t_{n_k} com maior prioridade.

Para cumprir essa tarefa, o algoritmo deve calcular o EFT de cada tarefa do grupo em cada recurso disponível. Se o recurso já possuir tarefas do mesmo DAG em sua fila, as tarefas são ordenadas de forma não crescente de prioridade para que respeitem as restrições de precedência, evitando *deadlocks*.

O próximo passo é calcular o EST do sucessor da última tarefa do agrupamento que está sendo escalonado. O primeiro grupo de tarefas de um DAG não tem predecessores nem sucessores, pois sua primeira tarefa é a tarefa de entrada e sua última tarefa é a tarefa de saída do DAG. Assim, não havendo sucessores, o agrupamento é escalonado no recurso que proporcionar o menor EFT para sua última tarefa. Todos os outros grupos de tarefas têm seus sucessores já escalonados pois, por construção, o último nó do grupo não tem sucessores não escalonados. Um *cluster* cls_k é escalonado no recurso que proporcionar o menor EST para o sucessor de cls_k. Após o escalonamento de cada *cluster*, os pesos, ESTs e EFTs são recomputados. O Algoritmo 3 mostra como a seleção de recursos é feita.

Algoritmo 3 seleciona_melhor_recurso(*cluster*)

1: **for all** $r \in \mathcal{R}$ **do**
2: $\quad escalonamento \Leftarrow$ insere *cluster* em $escalonamento_r$
3: $\quad$ computa_EFT(*escalonamento*);
4: $\quad tempo_r \Leftarrow$ computa_EST(sucessor(*cluster*))
5: **end for**
6: **retorna** recurso r com o menor $tempo_r$.

O algoritmo de seleção de recursos insere, para cada recurso r no conjunto de recursos $\mathcal{R}$, o agrupamento a ser escalonado no escalonamento de r (linha 2). A seguir, o algoritmo calcula o EFT do escalonamento atual (linha 3) para em seguida calcular o EST do sucessor do agrupamento escalonado (linha 4). O recurso selecionado é aquele que resultar no menor EST calculado na linha 4.

Por exemplo, considere 3 recursos em 2 grupos. Em um grupo, recursos 0 e 1 com capacidade de processamento de 133 e 130, respectivamente. No outro grupo, o recurso 2 com capacidade de processamento de 118. A largura de banda entre dois recursos do mesmo grupo é 10, enquanto para pares de recursos em grupos distintos é 5. Para o grafo da Figura 2.2, a primeira tarefa adicionada ao primeiro grupo, cls_0, é t_1 (prioridade de t_1: $\mathcal{P}_1 = 206$). Em seguida são adicionadas as tarefas t_2 ($\mathcal{P}_2 = 159,7$), t_5 ($\mathcal{P}_5 = 142,2$), t_7 ($\mathcal{P}_7 = 106,4$), t_{10} ($\mathcal{P}_{10} = 72,6$) e t_{11} ($\mathcal{P}_{11} = 15$). Então cls_0, que contém o caminho crítico do grafo inicial, é escalonado no recurso 0, que proporciona menor EFT para a tarefa t_{11}, e os ESTs, EFTs e pesos são recalculados. Após isso, a tarefa não escalonada com maior prioridade é t_4 ($\mathcal{P}_4 = 143,9$), que é adicionada ao grupo cls_1. Em seguida são adicionadas a cls_1 as tarefas t_6 ($\mathcal{P}_6 = 103,1$) e t_9 ($\mathcal{P}_9 = 70,1$). Esse grupo é escalonado no recurso 1, que é o que proporciona menor EST para o seu sucessor, no caso a tarefa t_{11}. O

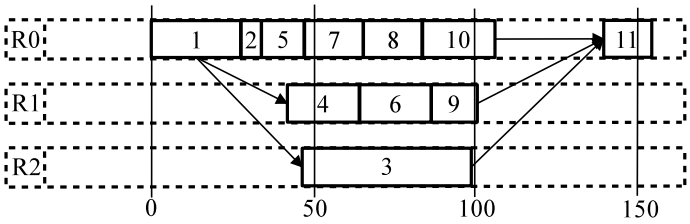


Figura 4.1: Escalonamento obtido para o grafo da Figura 2.2.

próximo grupo de tarefas formado é $cls_2 = \{t_8\}$, escalonado no recurso 0 antes da tarefa t_{10}, obedecendo as precedências do DAG. Finalmente, $cls_3 = \{t_3\}$ é criado e escalonado no recurso 2. O escalonamento resultante para o exemplo é mostrado na Figura 4.1.

4.2 HEFT

Nesta seção, apresentamos uma breve descrição do algoritmo *Heterogeneous Earliest Finisht Time* (HEFT) [95]. O HEFT é um algoritmo de escalonamento de tarefas dependentes em sistemas heterogêneos notadamente reconhecido por sua simplicidade e baixa complexidade. Esse algoritmo utiliza a técnica de *list scheduling*, criando uma lista de prioridades para os nós a serem escalonados e selecionando os recursos para cada nó de acordo com essa lista.

O HEFT primeiramente atribui um peso para cada nó e arco do DAG. O peso de cada nó é calculado como sendo o custo médio de computação da tarefa em cada máquina, enquanto o peso de cada arco é a média dos custos de comunicação para o arco sobre todos os pares de recursos. Em seguida, um valor de classificação, $rank_u$, é computado atravessando o DAG de trás para frente. O atributo de prioridade $rank_u$ para cada tarefa t_i é definido da seguinte forma:

$$rank_u(t_i) = \overline{w_i} + \max_{t_j \in suc(t_i)} (\overline{c_{i,j}} + rank_u(t_j)), \tag{4.1}$$

onde $\overline{w_i}$ é a média do tempo de execução de t_i sobre todos os recursos, $suc(t_i)$ é o conjunto de sucessores imediatos de t_i, e $\overline{c_{i,j}}$ é a média dos custos de comunicação entre t_i e t_j. Dessa forma, o $rank_u$ de uma tarefa t é o peso de t mais o valor máximo resultando do peso de cada filho de t adicionado ao peso do arco que conecta esse filho a t. Então, as tarefas são escalonadas uma a uma em ordem não crescente de $rank_u$ no recurso que resultar o menor tempo estimado de término (EFT) considerando que tarefas podem ser inseridas em espaços no escalonamento.

4.3 Configurações de simulação

Nas simulações realizadas, utilizamos tanto grafos gerados aleatoriamente quanto grafos de aplicações reais. Os grafos que foram tomados aleatoriamente possuíam de 7 a 82 nós e foram gerados utilizando o procedimento descrito no Apêndice B. Os seis grafos de aplicações reais utilizados foram Montage (Figura 4.2(a), de [37]); AIRSN (Figura 4.2(b), de [108]); CSTEM (Figura 4.2(d), de [38]); LIGO (Figura 4.2(c), de [36]); e, Chimera-1 e Chimera-2 (Figuras 4.2(e) e 4.2(f), de [3]).

Os algoritmos foram executados com valores aleatórios de custos de computação e comunicação nos DAGs utilizados. Comunicação média significa que os custos de comunicação e computação foram gerados aleatoriamente dentro do mesmo intervalo. Comunicação alta significa que todos os custos de comunicação foram maiores que todos os custos de computação. Os experimentos foram executados simulando uma topologia de grupo, como descrito na Seção 2.1. Variamos a quantidade de grupos de 2 a 25, enquanto a quantidade de recursos por grupo foi gerada aleatoriamente entre 1 e 10. Os custos das tarefas do grafo CSTEM foram gerados aleatoriamente, mas com valores proporcionais aos encontrados no DAG original [38]. Todos os números aleatórios foram obtidos de uma distribuição uniforme, e diferentes intervalos foram utilizados nas diferentes simulações. Detalhes sobre os intervalos utilizados em cada simulação são apresentados nas seções de resultados.

Para avaliar os escalonamentos gerados, as principais métricas usadas na literatura, além do *makespan*, são o *Schedule Length Ratio* (SLR) e o *speedup*, definidos como:

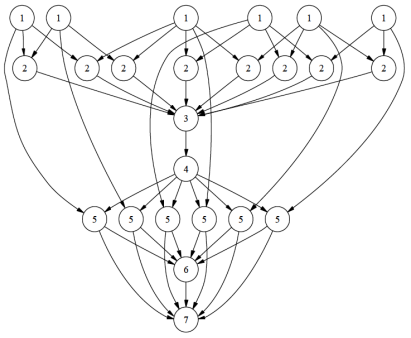
$$SLR = \frac{makespan}{\sum_{t_i \in CC} \frac{instrucões_{t_i}}{p_{melhor}}}$$

A soma no denominador representa o custo de computação do caminho crítico no melhor recurso disponível.

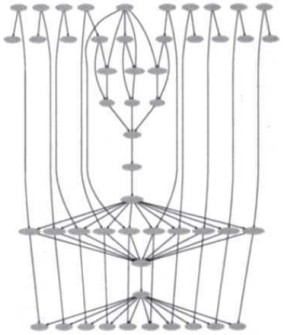
$$Speedup = \frac{\sum_{t_i \in V} \frac{instrucões_{t_i}}{p_{melhor}}}{makespan}$$

A soma no numerador representa o tempo de execução de todas as tarefas seqüencialmente no melhor recurso disponível.

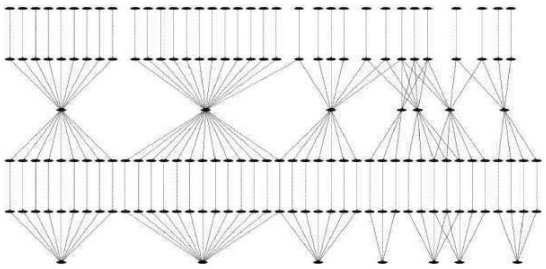
Assim, o SLR mostra quantas vezes o escalonamento é maior que a execução do caminho crítico do DAG no melhor recurso disponível, enquanto o *speedup* mostra quantas vezes mais rápida é a execução com o escalonamento dado do que a execução seqüencial no melhor recurso. O número de vezes que um algoritmo gera melhores escalonamentos que outro também é uma métrica encontrada na literatura.



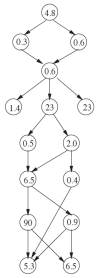
(a) Montage



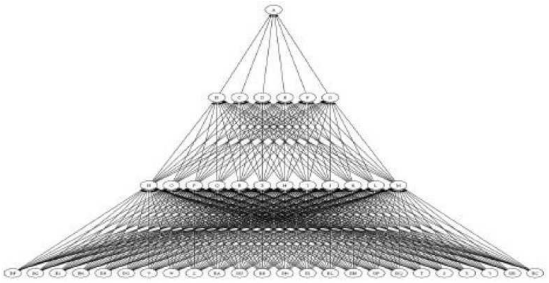
(b) AIRSN



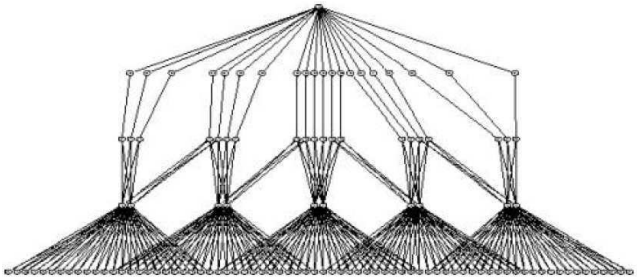
(c) LIGO



(d) CSTEM



(e) Chimera-1



(f) Chimera-2

Figura 4.2: Aplicações de *workflow* utilizadas nas simulações.

Quando consideramos a existência de múltiplos *workflows*, executamos simulações com o escalonamento de 2 a 10 DAGs. Nesse contexto, para avaliar o *makespan*, além do *SLR* e do *speedup* de cada processo P_k, utilizamos o *makespan médio*, o *slowdown* e o *makespan global* dos processos, definidos como:

Makespan médio($average_{M_N}$): média de *makespan* para os N primeiros processos após escalonar todos os DAGs considerados de acordo com a quantidade N, $1 \leq N \leq 10$, definido como:

$$average_{M_N} = \frac{1}{N} \sum_{k=0}^{N-1} makespan_{P_k}$$

Note que o *makespan* médio considera a média do *makespan* sobre os N primeiros *workflows* escalonados, mas com todos os DAGs que necessitavam ser escalonados já escalonados. Por exemplo, se há 10 DAGs a serem escalonados e $N = 3$, o *makespan* médio é a média de P_0, P_1 e P_2, mas com todos os 10 DAGs já escalonados.

Slowdown: *slowdown* de cada *processo* após escalonar todos os DAGs. O *slowdown* de um processo P_k representa quantas vezes o *makespan* de P_k é maior que o *makespan* de P_k no seu escalonamento inicial se fosse escalonado sozinho no mesmo conjunto de recursos. Essa métrica permite comparar os algoritmos de modo a avaliar a justiça na distribuição dos DAGs nos recursos.

Makespan global ($overall_M$): o *makespan* global do escalonamento, calculado com todos os DAGs escalonados, definido como:

$$overall_M = \max_{P_k \in DAGs} makespan_{P_k}$$

Em uma grade os recursos podem ser compartilhados, o que leva a variações de desempenho. Realizamos simulações onde pode haver perda de desempenho em conseqüência de carga externa executando nos recursos. Nessas simulações, utilizamos dois tipos de carga externa para determinar o desempenho dos recursos utilizados na simulação e suas variações de desempenho. No primeiro tipo, a simulação da carga externa é gerada a partir de um padrão de desempenho baseado em medições realizadas em computadores do Laboratório de Redes de Computadores do Instituto de Computação da UNICAMP [1]. Utilizando esse padrão, no início da simulação o algoritmo gera as capacidades de processamento de cada recurso. Dessa forma os algoritmos utilizam as mesmas capacidades de processamento para cada recurso em seus escalonamentos. O segundo tipo de carga externa é baseada em um modelo mais geral. Nesse modelo, supomos que a chegada

[1] Detalhes sobre as medições são mostrados no Apêndice A.

de tarefas independentes da grade segue uma distribuição de Poisson, com cada tarefa tendo um tempo de vida de $2/x$, onde x é um número aleatório no intervalo $(0, 1]$. Essa suposição segue o comportamento observado em aplicações reais [2].

No contexto de existência de carga externa, realizamos simulações de execução do DAG sem reescalonamento, onde todas as tarefas foram executadas nos recursos dados pelo escalonamento inicial, não importando o desempenho durante a execução, e simulações com reescalonamento, em que tarefas são reescalonadas dependendo do desempenho do recurso na execução das tarefas anteriores.

Os enlaces também podem sofrer de perda de desempenho se houver um aumento no tráfego. Nos algoritmos apresentados aqui, não há mecanismos para tratar esse problema. Supomos que os dados são fornecidos pelo *middleware* e que são uma estimativa confiável, como a maioria dos trabalhos em escalonamento de tarefas dependentes. Por exemplo, o *middleware* pode utilizar o Network Weather Service (NWS) [99] para tal estimativa. É importante ressaltar que as estimativas de desempenho são fornecidas pelo *middleware*. Dessa forma, quando o escalonador determina que certo recurso teve desempenho aquém do esperado, o *middleware* pode atualizar seus repositórios para que o escalonador acesse informações atualizadas sobre o desempenho de tal recurso. Além disso, o *middleware* deve possuir mecanismos independentes para determinar o desempenho atual de um recurso que não é utilizado há algum tempo, mantendo informações condizentes com o estado atual do recurso para serem utilizadas pelo escalonador.

Capítulo 5

Algoritmos Desenvolvidos

Neste capítulo, descrevemos os algoritmos desenvolvidos e apresentamos simulações e avaliações de desempenho. Primeiramente apresentamos o algoritmo PCH dinâmico, que realiza escalonamento de DAGs utilizando informações atualizadas sobre os recursos. Em seguida, apresentamos uma extensão adaptativa para esse algoritmo, que realiza ajustes na dinamicidade de acordo com o desempenho dos recursos. O segundo algoritmo apresentado é o HEFT com *lookahead*, que utiliza informações adicionais no escalonamento, tentando prever como a escolha atual afetará as escolhas futuras, a fim de melhorar o escalonamento alcançado. O terceiro algoritmo é um algoritmo bi-critério que trata do escalonamento de DAGs em grades orientadas a serviço. Esse algoritmo tem como função objetivo a minimização do *makespan* e do número de novos serviços criados para execução do *workflow*. Em seguida, descrevemos algoritmos para escalonamento de múltiplos DAGs, o que se torna um problema importante quando temos ambientes compartilhados como as grades. Em cada seção os algoritmos são avaliados através de simulações, onde são mostrados gráficos e análises dos resultados.

5.1 PCH Dinâmico

No capítulo anterior descrevemos o algoritmo PCH estático, que realiza o escalonamento de todas as tarefas com informações obtidas no início do processo de escalonamento. Em um ambiente dinâmico é desejável que um algoritmo de escalonamento seja capaz de utilizar informações atualizadas para realizar o escalonamento das tarefas do *workflow*. Portanto, o desenvolvimento de algoritmos de escalonamento dinâmicos encaixa-se com as características apresentadas por ambientes como as grades computacionais e as nuvens.

Apresentamos nesta seção o algoritmo PCH dinâmico, que realiza o escalonamento do DAG de forma parcial e contínua, acompanhando a execução das tarefas e o desempenho dos recursos utilizados. Nesse sentido, o algoritmo dinâmico tenta minimizar os efeitos

da perda de desempenho nos recursos selecionados. Essa perda é normalmente originada por processos externos à grade e que, portanto, não são controlados pelo *middleware* da grade.

Um escalonador de tarefas dinâmico pode usar, em geral, duas estratégias:

- Escalonar todas as tarefas estaticamente e então reescaloná-las quando necessário. Dessa forma, o escalonador trata de um DAG de cada vez.

- Escalonar as tarefas gradualmente à medida que outras tarefas terminam. Aqui poderá ser necessário tratar mais de um DAG, considerando que a grade é um ambiente compartilhado e recebe requisições em paralelo de usuários diferentes. Dessa forma, outros DAGs podem ser recebidos pelo escalonador antes do término do processo de escalonamento do DAG recebido anteriormente.

O PCH dinâmico é baseado na primeira estratégia. Isso significa que o primeiro DAG a ser recebido terá todas as suas tarefas escalonadas antes dos próximos DAGs, dessa forma o primeiro *workflow* terá prioridade no escalonamento. Por outro lado, a segunda estratégia deve se preocupar com a prioridade dos processos de forma a não atrasar de forma indefinida a execução de um DAG quando há muitos *workflows* a serem escalonados. Por exemplo, se há muitos DAGs sendo submetidos e o escalonador não considerar quando um DAG iniciou sua execução ao selecionar o próximo DAG a ter tarefas escalonadas, a vantagem de distribuir suas tarefas pode desaparecer, sendo vantajoso executá-lo em um recurso local e não na grade. O PCH realiza o escalonamento inicial e utiliza uma abordagem baseada em turnos, ou *rounds*, que é aplicada para reescalonar tarefas quando necessário.

Desenvolvemos uma abordagem dinâmica adaptativa para escalonar tarefas em grades computacionais, onde utilizamos um esquema de execução em quatro passos, como mostrado na Figura 5.1. No primeiro passo é utilizado o algoritmo *Path Clustering Heuristic* (PCH), que realiza o escalonamento inicial dos processos. A técnica de execução baseada em turnos, descrita a seguir, é utilizada no segundo passo, onde o algoritmo decide dinamicamente em cada turno quais tarefas escalonadas serão enviadas para execução. Dessa forma o algoritmo obtém informações de desempenho de cada recurso para tomar decisões de reescalonamento. O módulo dinâmico decide quais tarefas serão executadas em cada turno amparado pelo módulo adaptativo, apresentado na Seção 5.1.2 , que regula o tamanho dos turnos em cada recurso. Essa regulagem adaptativa do tamanho dos turnos tenta evitar o envio de muitas tarefas para execução em recursos que têm histórico de alta variação de desempenho. O tamanho dos turnos é relativo ao tempo de execução das tarefas em cada recurso, dependendo do desempenho do recurso em questão. A execução é o terceiro passo, e o reescalonamento, o quarto.

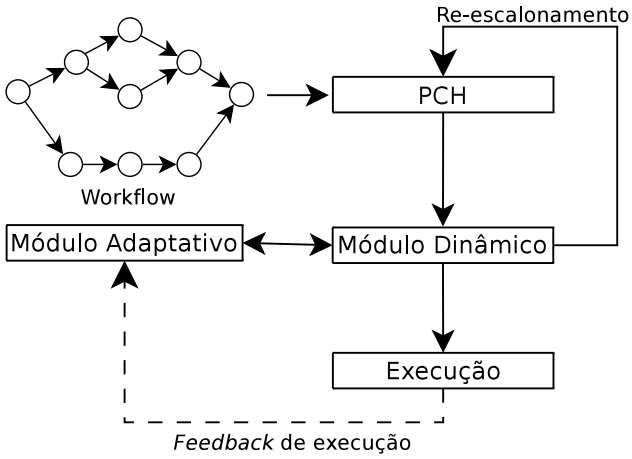


Figura 5.1: Esquema de execução em quatro passos.

Para realizar o escalonamento dinâmico de tarefas, em cada turno algumas tarefas são selecionadas baseando-se em um critério pré-estabelecido e são enviadas para execução. Então, o escalonador verifica o desempenho obtido em cada recurso utilizado no turno. Se o desempenho de determinado recurso estiver aquém do esperado, o algoritmo reescalona as tarefas que foram escalonadas naquele recurso mas que ainda não foram enviadas para execução. É importante destacar que as tarefas podem ser enviadas para execução somente quando definido que realmente executarão em determinado recurso. Dessa forma, reescalonar uma tarefa é uma ação local e interna do escalonador, sem a necessidade de cancelamento de execuções, reenvio de código ou alteração de filas remotas. Uma visão geral dessa estratégia é mostrada no Algoritmo 4.

Algoritmo 4 Visão geral da abordagem dinâmica

1: Escalona o DAG $\mathcal{G}$ de forma estática.
2: **while** existem tarefas de $\mathcal{G}$ não executadas **do**
3: Seleciona um conjunto de tarefas de $\mathcal{G}$ de acordo com uma política pré-definida.
4: Envia o conjunto selecionado para execução.
5: Avalia o desempenho dos recursos utilizados na execução das tarefas enviadas para execução.
6: Reescalona as tarefas não executadas, se necessário.
7: **end while**

Cada iteração do Algoritmo 4 é chamada de *round*. Em cada *round*, ou turno, algumas tarefas são selecionadas baseando-se em alguma política pré-estabelecida (linha 3), e então são enviadas para execução (linha 4). Em seguida o escalonador verifica o desempenho obtido pelos recursos durante a execução das tarefas (linha 5). Se o desempenho estiver

aquém do esperado, o algoritmo reescalona as tarefas ainda não executadas (linha 6).

5.1.1 Turnos e Fase Dinâmica

Após realizar o escalonamento inicial, a fase dinâmica inicia-se. Primeiramente, baseando-se no escalonamento inicial, o algoritmo seleciona somente parte do DAG para ser enviada para execução. Em nosso algoritmo, uma tarefa t_n é enviada para execução no turno k (de um total de T turnos) se t_n ainda não iniciou sua execução e se $EFT_n \leq \frac{makespan}{T/k}$ [1], com $1 \leq k \leq T$, ou se a tarefa é a primeira tarefa no escalonamento (fila) do recurso. À medida que as tarefas executam, o escalonador pode comparar os tempos reais de execução das tarefas terminadas com os tempos previamente estimados pelos atributos utilizados no escalonamento. Na tentativa de não deixar os recursos ociosos, a avaliação de desempenho é feita considerando a execução da penúltima tarefa que foi enviada. Dessa forma, o algoritmo pode realizar o reescalonamento, quando necessário, antes do término da última tarefa, evitando espaços de tempo livres desnecessários na execução do *workflow*. Em seguida, o algoritmo compara o tempo real de execução com o tempo esperado para determinar se é necessário reescalonar as tarefas do recurso em questão. Se existir uma perda de desempenho maior que determinado limiar (por exemplo, 10%) em determinado recurso, as tarefas que ainda não iniciaram sua execução e que estão na fila daquele recurso são reescalonadas utilizando o algoritmo PCH. O PCH dinâmico é mostrado no Algoritmo 5.

5.1.2 Extensão Adaptativa

No esquema de quatro passos apresentado na Figura 5.1, o algoritmo adaptativo ajusta o número de turnos para DAGs de tamanhos diferentes e recursos com desempenho variável. O objetivo dessa extensão adaptativa é variar o tamanho dos turnos de acordo com o peso das tarefas, o tamanho do grafo e o desempenho dos recursos. Assim, entre as características desejáveis em um algoritmo dessa classe estão:

1. Adaptar o tamanho dos turnos de acordo com o peso das tarefas;

2. Adaptar o tamanho de cada turno de acordo com o desempenho dos recursos;

3. Considerar o histórico de desempenho de cada recurso ao decidir o tamanho dos turnos.

[1] O *makespan* é o tamanho do escalonamento em um dado instante. Assim, antes do término da execução, o *makespan* é igual ao tempo estimado de término (EFT) da última tarefa do processo. Após a finalização do processo, o *makespan* é o tempo real de execução do *workflow*.

Algoritmo 5 Algoritmo PCH Dinâmico

1: Escalona DAG $\mathcal{G}$ usando o algoritmo PCH.
2: $T \Leftarrow$ número de *rounds*
3: **for all** $r \in \mathcal{R}$ que possui um nó de $\mathcal{G}$ no seu escalonamento **do**
4: $round_r \Leftarrow 1$ //*Primeiro turno em todos os recursos*
5: $N_r \Leftarrow$ tarefas de $\mathcal{G}$ não executadas e que estão no escalonamento de r tal que $EFT \leq \frac{makespan}{T/round_r}$
6: Envia tarefas de N_r para execução em r.
7: **end for**
8: **while** not(todos os nós de $\mathcal{G}$ terminaram) **do**
9: $t \Leftarrow$ aguarda_próxima_tarefa_terminar() //*espera blocante*
10: $r \Leftarrow$ recurso de t
11: $tempo\_total\_exec_r \Leftarrow tempo\_total\_exec_r + tempo\_exec_t$
12: **if** t é a penúltima tarefa de $\mathcal{G}$ na fila de execução de r **then**
13: **if** $tempo\_total\_exec_r > t\_esp\_exec_r + limiar$ **then**
14: Reescalona tarefas não iniciadas de r com o PCH
15: $r \Leftarrow$ novo recurso selecionado pelo PCH no passo anterior para tarefas de N_r ainda não executadas
16: **end if**
17: $round_r \Leftarrow round_r + 1$ //*próximo turno*
18: $N_r \Leftarrow$ tarefas de $\mathcal{G}$ não executadas e que estão no escalonamento de r tal que $EFT \leq \frac{makespan}{T/round_r}$
19: Envia tarefas de N_r para execução em r.
20: **end if**
21: **end while**

Para obter essas características, desenvolvemos uma formulação matemática a ser adotada pelo algoritmo de escalonamento. Sejam:

- $\mathcal{G}$ o DAG do processo a ser escalonado.
- $\mathcal{R}$ o conjunto de recursos selecionados ao escalonar $\mathcal{G}$.
- $M_{r,k}$ o conjunto de tarefas enviadas para execução no recurso r no turno k.
- S_r o escalonamento (fila) do recurso r.
- $t\_esp\_exec_{M_{r,k}} = \sum_{t \in M_{r,k}} (EFT_{t,r} - EST_{t,r})$ o tempo esperado de execução das tarefas em $M_{r,k}$.
- $d_{r,k} = \dfrac{t\_esp\_exec_{M_{r,k}}}{tempo\_real\_exec_{M_{r,k}}}$ o desempenho do recurso r no turno k.

A partir dessas definições, apresentamos qual deverá ser a duração do turno $k+1$ no recurso r, $ET_{r,k+1}$, como segue:

$$ET_{r,k+1} = \begin{cases} \dfrac{\sum_{t \in G} peso(t)}{|G|}, & \text{se } k = 0 \\ ET_{r,k} \times d_{r,k} + \lfloor d_{r,k} \rfloor \times f\_recompensa \times t\_esp\_exec_{M_{r,k}}, & \text{se } k \geq 1 \end{cases} \tag{5.1}$$

A equação 5.1 define $ET_{r,k}$ de forma que ele represente o tamanho (tempo de execução) do turno k no recurso r. O algoritmo seleciona as tarefas a serem enviadas para execução em cada turno baseando-se nessa definição, obedecendo $\sum_{t \in M_{r,k}} peso(t) \leq ET_{r,k}$. Dessa forma, $M_{r,k}$ será o conjunto de tarefas de $\mathcal{G}$ escalonadas em r tal que a soma de seus pesos seja menor ou igual a $ET_{r,k}$. Caso o peso da primeira tarefa enviada para execução seja maior que $ET_{r,k}$, ela será a única tarefa enviada para execução no recurso r no turno k ($|M_{r,k}| = 1$).

A multiplicação $\lfloor d_{r,k} \rfloor \times f\_recompensa \times t\_esp\_exec_{M_{r,k}}$ funciona como uma "recompensa", aumentando o tamanho do próximo turno cada vez que o recurso alcançar o desempenho previsto. Essa recompensa é dada em função do tempo de execução esperado para o último turno executado no recurso. Quando $0 < d_{r,k} < 1$, a multiplicação é 0, então não existe recompensa e a multiplicação $ET_{r,k} \times d_{r,k}$ diminui o tamanho do turno. Por outro lado, quando $d_{r,k} > 1$, a multiplicação $ET_{r,k} \times d_{r,k}$ aumenta o tamanho do turno, existindo recompensa. No caso particular em que $d_{r,k} = 1$, o tamanho do turno é aumentado somente pelo fator de recompensa.

Fazendo $avrg\_w(G) = \frac{\sum_{t \in G} peso(t)}{|G|}$, e expandindo a recorrência (5.1), temos:

$$ET_{r,k+1} = avrg\_w(G) \times \prod_{i=1}^{k} d_{r,i} + \sum_{i=1}^{k} \Big(\lfloor d_{r,i} \rfloor \times f\_recompensa \times t\_esp\_exec_{M_{r,i}} \times \prod_{j=i+1}^{k} d_{r,j} \Big) \tag{5.2}$$

Analisando (5.2), podemos identificar as propriedades enumeradas anteriormente. A primeira propriedade (adaptar o tamanho dos turnos de acordo com o peso das tarefas) é dada pelo fator $avrg\_w(G)$. O produto de valores de desempenho é como um histórico de desempenho do recurso, fornecendo uma visão do desempenho alcançado nas execuções passadas e, em conjunto com o fator de recompensa, automaticamente adapta o tamanho do turno em cada recurso que está executando tarefas de $\mathcal{G}$. Note que um aumento no tamanho do turno tende a diminuir a quantidade total de turnos, enquanto uma diminuição no tamanho dos turnos tende a aumentar a quantidade de turnos. Os Algoritmos 6 e 7 apresentam a estratégia de escalonamento dinâmico adaptativo.

Na linha 1 do Algoritmo 6, o DAG é escalonado usando o algoritmo PCH estático. Após isso, nas linhas 2 a 7, o algoritmo gera o conjunto $M_{r,1}$ para todos os recursos que tenham tarefas de $\mathcal{G}$ em seu escalonamento. O primeiro turno de todos os recursos terá o tamanho do peso médio dos nós de $\mathcal{G}$ (linha 4). Na linha 5 o conjunto $M_{r,1}$ é gerado para cada recurso, com esses conjuntos sendo enviados para execução na linha 6. A iteração da linha 8 se repete até que a execução do *workflow* termine. Na linha 11 o tempo real de execução das tarefas finalizadas é acumulado, enquanto na linha 12 o algoritmo verifica se a tarefa que terminou é a penúltima tarefa do turno naquele recurso. Caso afirmativo, o algoritmo passa para o próximo turno naquele recurso. Na linha 15 o algoritmo verifica se há necessidade de reescalonamento comparando os tempos reais de execução acumulados na linha 11 com o tempo esperado, calculado a partir dos atributos do DAG. O algoritmo gera o novo $M_{novo\_r,round_{novo\_r}}$ (linha 19) e o envia para execução no recurso $novo\_r$ (linha 20). Note que $novo\_r$ pode ser o mesmo recurso que r em dois casos: primeiro se r teve o desempenho esperado; segundo se o reescalonamento resultar no mesmo escalonamento anterior (ou seja, não há recurso melhor para executar as tarefas restantes).

O algoritmo 7 é a extensão adaptativa que gera o conjunto $M_{r,k}$ de acordo com o desempenho dos recursos em cada turno. Nas linhas 2 a 4 o algoritmo calcula o desempenho do recurso no último turno (linha 2) e atribui o tamanho do próximo turno à variável $próximo\_tempo\_total\_exec_r$ (linha 3). Na linha 4 as tarefas já executadas são excluídas da fila. Se é o primeiro turno, as linhas 2 a 4 não devem ser executadas (linha 1), pois não há desempenho a ser avaliado e não há tarefas que já foram executadas no recurso. Então, o algoritmo atribui a primeira tarefa não executada à variável $M_{r,round_r}$. Enquanto a soma dos pesos das tarefas em $M_{r,round_r}$ for menor que o tamanho do turno atual (linha 9), o algoritmo continua escolhendo a primeira tarefa da fila (linha 10) e, se há espaço para a tarefa (linha 11), ela é adicionada a $M_{r,round_r}$ (linha 12), sempre acumulando o peso total

Algoritmo 6 Algoritmo PCH Dinâmico Adaptativo

1: Escalona DAG $\mathcal{G}$ usando o algoritmo PCH
2: **for all** $r \in \mathcal{R}$ **do**
3: $round_r \Leftarrow 1$ //*Primeiro turno em todos os recursos*
4: $próximo\_tempo\_total\_exec_r \Leftarrow avrg\_w(G)$
5: $M_{r,round_r} \Leftarrow$ próximas_tarefas_a_executar(M, r, $round_r$, $escalonamento_r$, $tempo\_exec_r$, $t\_esp\_exec_r$, $próximo\_tempo\_total\_exec_r$)
6: Envia tarefas de $M_{r,round_r}$ para execução em r.
7: **end for**
8: **while** not(todos os nós de $\mathcal{G}$ terminaram) **do**
9: $t \Leftarrow$ aguarda_próxima_tarefa_terminar() //*espera blocante*
10: $r \Leftarrow$ recurso de t
11: $tempo\_exec_r \Leftarrow tempo\_exec_r + tempo\_real\_exec_t$
12: **if** t é a penúltima tarefa de $\mathcal{G}$ na fila de execução de r **then**
13: $round_r \Leftarrow round_r + 1$ //*próximo turno*
14: $novo\_r \Leftarrow r$
15: **if** $tempo\_exec_r > t\_esp\_exec_r + limiar$ **then**
16: Reescalona tarefas não iniciadas de r com o PCH
17: $novo\_r \Leftarrow$ novo recurso selecionado pelo PCH para executar as tarefas de $escalonamento_r$
18: **end if**
19: $M_{novo\_r,round_{novo\_r}} \Leftarrow$ próxima_tarefa_a_executar(M, $novo\_r$, $round_r$, $escalonamento_{novo\_r}$, $tempo\_exec_r$, $tempo\_esperado_r$, $próximo\_tempo\_total\_exec_r$)
20: Envia tarefas de $M_{novo\_r,round_{novo\_r}}$ para executar em $novo\_r$.
21: **end if**
22: **end while**

Algoritmo 7 próxima_tarefa_a_executar(M, r, $round_r$, $escalonamento_r$, $tempo\_exec_r$, $t\_esp\_exec_r$, $próximo\_tempo\_total\_exec_r$)

1: **if** $round_r > 1$ **then**
2: $d_r \Leftarrow \dfrac{t\_esp\_exec_r}{tempo\_real\_exec_r}$
3: $próximo\_tempo\_total\_exec_r \Leftarrow próximo\_tempo\_total\_exec_r \times d_r + \lfloor d_r \rfloor \times f\_recompensa \times t\_esp\_exec_r$
4: $tarefas\_a\_executar \Leftarrow escalonamento_r - tarefas\_iniciadas_r$ //subtração de conjuntos
5: **end if**
6: $M_{r,round_r} \Leftarrow$ primeira_tarefa_de($task\_to\_execute_r$) //$M_{r,round_r}$ tem somente a primeira tarefa da fila
7: $tarefas\_a\_executar \Leftarrow tarefas\_a\_executar - M_{r,round_r}$ //subtração de conjuntos
8: $tempo\_exec_{M_{r,round_r}} \Leftarrow EFT_t - EST_t$
9: **while** $tempo\_exec_{M_{r,round_r}} < próximo\_tempo\_total\_exec_r$ **do**
10: $t \Leftarrow$ primeira_tarefa_de($tarefas\_a\_executar_r$)
11: **if** $tempo\_exec_{M_{r,round_r}} + (EFT_t - EST_t) \leq próximo\_tempo\_total\_exec_r$ **then**
12: $M_{r,round_r} \Leftarrow M_r \cup t$
13: $tarefas\_a\_executar \Leftarrow tarefas\_a\_executar - t$
14: $tempo\_exec_{M_{r,round_r}} \Leftarrow tempo\_exec_{M_{r,round_r}} + (EFT_t - EST_t)$
15: **end if**
16: **end while**
17: retorna $M_{r,round_r}$

na variável $tempo\_exec_{M_{r,round_r}}$ (linha 14) e retirando as tarefas da fila (linha 13).

5.1.3 Resultados Experimentais

Em uma grade os recursos podem ser compartilhados, o que leva a variações de desempenho. Nesta seção comparamos o PCH dinâmico (com reescalonamento) com o PCH estático (sem reescalonamento) em diferentes cenários de comunicação e com diferentes números de turnos. Em seguida, apresentamos comparações entre o algoritmo dinâmico com e sem a extensão adaptativa. Nas simulações de execução das tarefas, as capacidades dos recursos variavam ao longo do tempo obedecendo as medições apresentadas no Apêndice A, simulando carga externa.

Avaliação do PCH Dinâmico

Para as simulações desta seção utilizamos 15 grafos tomados aleatoriamente com número de nós entre 7 e 82. Comunicação média significa que os custos de comunicação e computação foram gerados aleatoriamente dentro do mesmo intervalo (500 a 1100 unidades de tempo). Comunicação alta significa que todos os custos de comunicação (de 1100 a 1600 unidades de tempo) foram maiores que todos os custos de computação (de 500 a 1100 unidades de tempo). Os experimentos foram executados simulando uma topologia de grupo, como descrito na Seção 2.1. Para cada quantidade de grupos, variando de 2 a 25, cada grafo foi escalonado 1000 vezes. Em cada grupo, a quantidade de recursos foi escolhida aleatoriamente entre 1 e 5. Note que no reescalonamento não existe realocação de tarefas, portanto não existe *overhead* de movimentação de tarefas, pois o reescalonamento é feito antes que as tarefas sejam enviadas para execução.

Os resultados das execuções com um total de 2, 3 e 4 turnos com comunicação média e variação no desempenho dos recursos são mostradas na Figura 5.2. A diferença entre resultados com e sem reescalonamento aumenta com o incremento do número de turnos. Por exemplo, com 10 grupos e comunicação média, a diferença em valores absolutos entre o SLR médio com e sem reescalonamento é $0,33$, $0,69$ e $0,93$ para 2, 3 e 4 turnos, respectivamente. No caso do *speedup* essas diferenças são de $0,06$, $0,13$ e $0,18$. Já no caso da quantidade de melhores escalonamentos, essas diferenças são de 2.381, 4.158 e 4.993.

As médias de SLR e *speedup* para comunicação alta são mostradas na Figura 5.3. Podemos notar que com maiores custos de comunicação em relação aos de computação, o desempenho do PCH dinâmico se mantém estável em relação ao desempenho do PCH estático, com os gráficos apresentando o mesmo padrão de comportamento com o aumento da quantidade de grupos. No cenário de comunicação alta, as diferenças na quantidade de melhores escalonamentos são de 2.317, 4.624 e 5.493, para 2, 3 e 4 turnos.

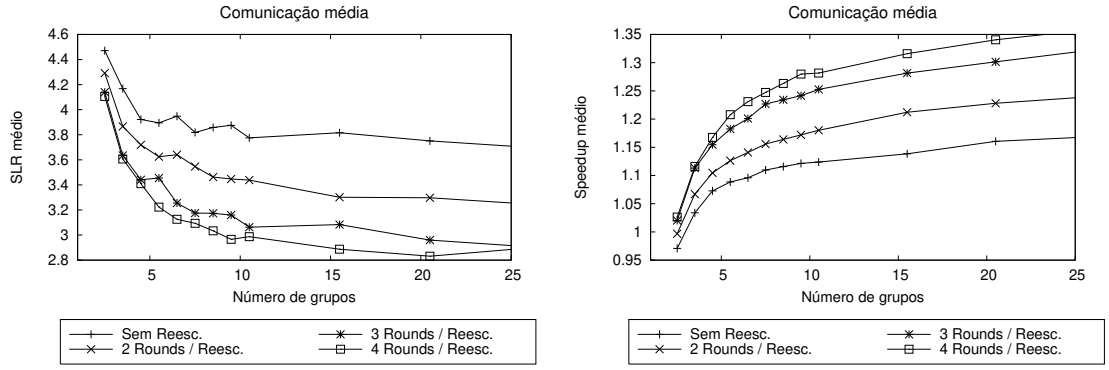


Figura 5.2: SLR e *speedup* médios com reescalonamento e comunicação média.

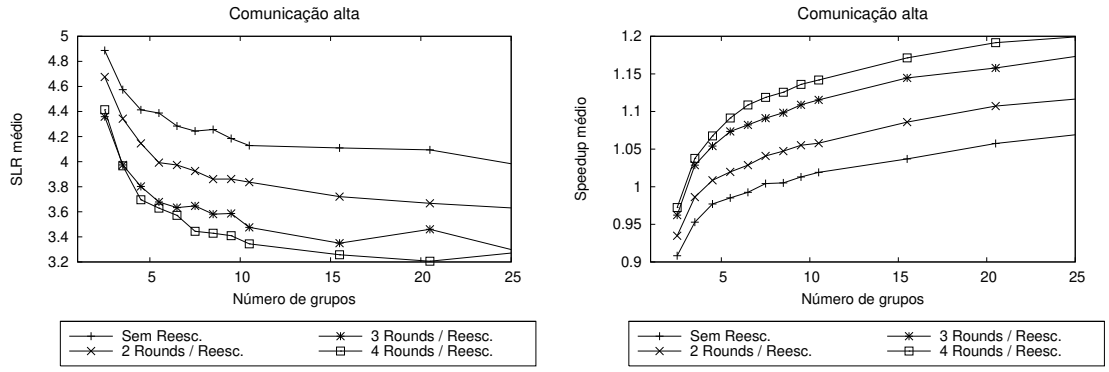


Figura 5.3: SLR e *speedup* médios com reescalonamento e comunicação alta.

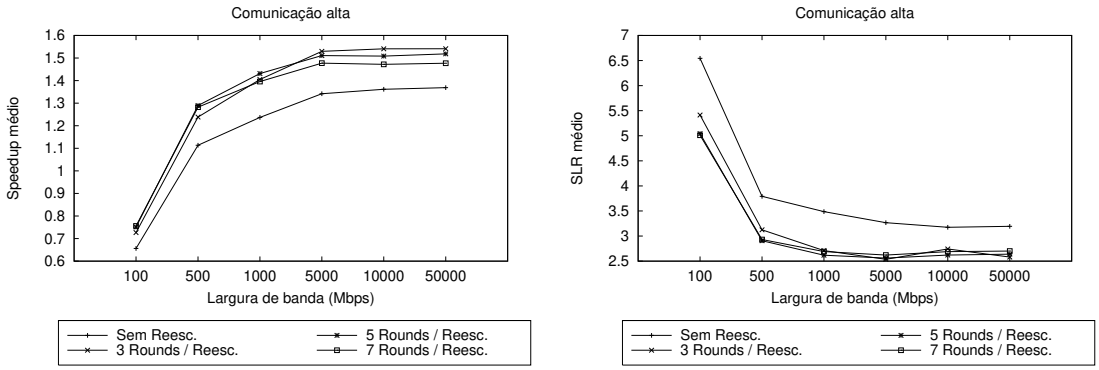


Figura 5.4: *Speedup* e SLR médios.

PCH dinâmico em enlaces de alta velocidade

Uma grade com enlaces de alta velocidade é um ambiente poderoso para execução de processos com tarefas dependentes, pois uma alta largura de banda minimiza o tempo de transmissão de dados entre tarefas em recursos diferentes. Com isso, as dependências de dados têm menor impacto no *makespan* final quando este é comparado a ambientes com enlaces de menor capacidade.

Nesta seção avaliamos o comportamento do PCH dinâmico em relação ao PCH estático em ambientes com diferentes capacidades de transmissão de dados [10]. O propósito dessa avaliação é verificar a evolução do PCH dinâmico quando os enlaces variam de Mbps para Gbps. Nesse experimento os custos de computação (número de instruções) das tarefas foram gerados aleatoriamente dentro do intervalo $(10.000, 310.000)$ milhões de instruções. Os custos de comunicação entre tarefas foram gerados entre 250 e 750 *megabytes*. Utilizamos 5 grupos de recursos, escolhendo quantidade de recursos em cada grupo aleatoriamente entre 1 e 7. Cada grupo de recursos possuía conexões internas entre 100% e 40% da capacidade máxima, gerados aleatoriamente, simulando uma rede não dedicada. A conexão entre grupos possuía menor largura de banda, variando entre 85% e 20% da capacidade máxima da largura de banda considerada. O poder de processamento de cada recurso foi gerado entre 4.000 e 10.000 MIPS. Para cada tupla (largura de banda, turnos) cada grafo foi escalonado 1000 vezes. A Figura 5.4 mostra as médias de *speedup* e SLR para execuções com 3, 5 e 7 turnos. A largura de banda varia de 100 Mbps a 50 Gbps.

Os resultados indicam que a abordagem dinâmica apresentada mantém um bom desempenho com o aumento das capacidades de transmissão. Os resultados de *speedup* mostram uma diferença absoluta maior entre o PCH dinâmico e estático com 50 Gbps do que com 100 Mbps, sendo proporcionalmente equivalentes. No caso dos resultados de

SLR a diferença é menor no cenário de 50 Gbps quando comparado ao cenário de 100 Mbps, com a proporção se mantendo. Podemos notar uma estabilização em torno de 5 Gbps, onde os custos de comunicação se tornam pequenos em comparação aos custos de computação, tendo pouco impacto no *makespan* do *workflow*.

Avaliação da Extensão Adaptativa

Após fornecer uma avaliação que compara o PCH dinâmico com o PCH estático, apresentamos uma avaliação do comportamento da extensão adaptativa no mesmo ambiente, utilizando as mesmas características nas simulações. Na Figura 5.5 mostramos os resultados da comparação entre o PCH dinâmico (com reescalonamento) com o PCH estático (sem reescalonamento) e com o PCH dinâmico com extensão adaptativa para simulações com comunicação média. Como nas simulações anteriores, executamos simulações com o PCH considerando a variação de desempenho dos recursos, com e sem reescalonamento. A quantidade de turnos também foi mantida entre 2 e 4 para as execuções do PCH dinâmico sem a extensão adaptativa, e essas execuções foram comparadas com as execuções com a extensão adaptativa. Note que o algoritmo adaptativo não tem um número pré-definido de turnos. Nos gráficos dos resultados, a legenda *T rounds/Adapt.* possui o número de turnos T para indicar que a curva com T turnos da extensão adaptativa deve ser comparada com a curva do PCH dinâmico também com T turnos, pois é a simulação que utilizou os mesmos vetores de desempenho dos recursos. O algoritmo adaptativo foi executado com $f\_recompensa = 0,05$. Simulações com diferentes valores para o fator de recompensa mostraram que se esse fator é muito alto ($> 0,2$), o desempenho pode piorar, pois os turnos rapidamente se tornam maiores, reduzindo o número de turnos de maneira não desejável. Considerando isso, o fator de recompensa pode ser ajustado de acordo com as expectativas relativas ao desempenho de cada recurso.

A quantidade de melhores escalonamentos com o algoritmo adaptativo fica entre 4.000, para simulações com 2 grupos, e 8.000, para simulações com 25 grupos. Com o reescalonamento dinâmico não adaptativo, esses números ficam entre 2.000 e 1.000, respectivamente. Já para o PCH sem reescalonamento, os números estão entre 2.500 e 3.400. O número de melhores escalonamentos mostra que o PCH com extensão adaptativa fornece uma maior quantidade de melhores escalonamentos que o PCH com ou sem reescalonamento, o que sugere que o algoritmo adaptativo consegue melhorar os resultados apresentados pelo PCH dinâmico.

Discussão

Em um ambiente com recursos compartilhados de desempenho variado, como uma rede local de um laboratório, a abordagem dinâmica melhorou os resultados sobre o PCH

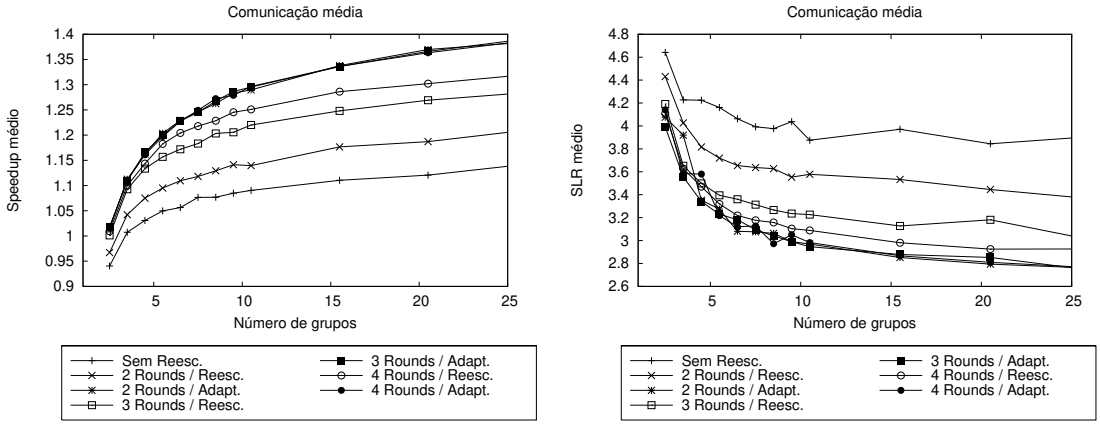


Figura 5.5: *Speedup* e SLR médios com o algoritmo adaptativo.

estático. Os resultados melhoram à medida que o número de turnos aumenta, pois para cada turno o algoritmo realiza uma nova avaliação dos recursos através dos tempos de execução medidos, obtendo uma nova visão da capacidade atual dos recursos e fornecendo uma oportunidade de reescalonamento caso necessário. Numa breve análise em ambientes com enlaces de alta velocidade, há indícios que a estratégia dinâmica não perde eficiência em um ambiente de grades onde os recursos são interligados por redes com largura de banda alta.

Com a inclusão da extensão adaptativa, os resultados de *speedup* e SLR após a execução das tarefas mostram que essa extensão pode fornecer execuções mais rápidas na média quando comparada ao PCH dinâmico sem a extensão. Como esperado, quanto maior o número de *rounds*, mais o PCH dinâmico com reescalonamento se aproxima dos resultados com a extensão adaptativa, pois o aumento do número de turnos proporciona melhores resultados ao PCH dinâmico com reescalonamento. Ainda, podemos notar que quanto maior o número de grupos, melhor o desempenho do algoritmo adaptativo. Isso é explicado pelo fato de que, havendo mais grupos, existem também mais recursos disponíveis, aumentando as opções ao reescalonar as tarefas.

5.2 HEFT com *lookahead*

Entre as numerosas heurísticas para escalonamento de DAGs, o algoritmo *Heterogeneous Earliest Finish Time* (HEFT) é conhecido por alcançar bons resultados com baixa complexidade. Nesta seção, propomos uma melhoria no HEFT, onde decisões ótimas locais não baseiam-se apenas em uma tarefa, mas também olham adiante no escalonamento, levando em conta informações sobre o impacto da decisão local também nas tarefas suces-

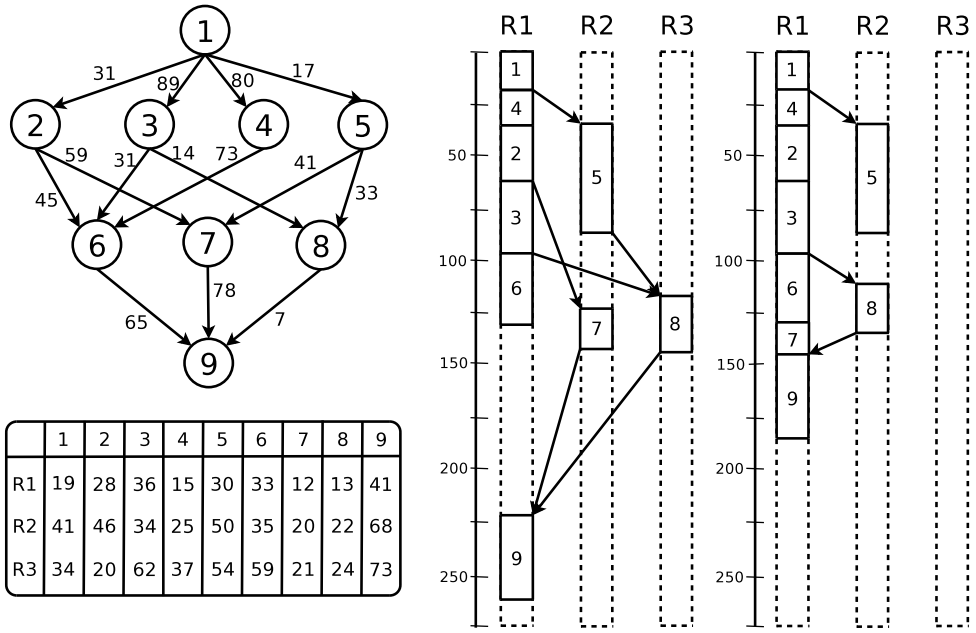
soras da tarefa que está sendo escalonada [14]. Através dessa melhoria no algoritmo, obtivemos resultados que mostram um sensível ganho na qualidade do escalonamento, minimizando o *makespan* dos DAGs escalonados sem tornar o tempo de execução proibitivamente alto.

HEFT é essencialmente uma heurística de *list scheduling*: uma lista de tarefas é organizada por prioridade, com o algoritmo selecionando tarefas dessa lista e realizando decisões baseadas em ótimos locais considerando o tempo estimado de término da tarefa. Entretanto, decisões baseadas em ótimos locais podem não ser necessariamente uma boa decisão para os descendentes dessa tarefa no grafo. A idéia motivacional do algoritmo HEFT com *lookahead* é aperfeiçoar o processo de escalonamento de tarefas no HEFT através da avaliação de como uma decisão local afeta o escalonamento futuro, isto é, verificar se a decisão de escalonar uma tarefa em certo recurso será boa para seus sucessores no DAG. Apesar das perspectivas de melhora no resultado final, tal avaliação tem o potencial de aumentar o tempo de execução da heurística. Entretanto, o HEFT já é uma heurística simples e rápida, o que deixa espaço para tal aumento. Dessa forma, a contribuição do algoritmo proposto é a sugestão de melhorias específicas que permitem o HEFT tomar decisões melhores aplicando a noção de *lookahead* para obter informações extras antes de escalonar uma tarefa.

De forma similar ao HEFT, muitas outras heurísticas para escalonamento de DAGs selecionam recursos baseando-se somente em atributos calculados para uma única tarefa, ignorando conseqüências adversas dessa decisão em seus sucessores. Uma exceção a essa regra é o algoritmo k-DLA [100], em que um atributo computado estaticamente leva em conta o número de vizinhos de um recurso na rede em relação ao número de filhos da tarefa sendo escalonada. Nosso algoritmo vai adiante, levando em consideração o impacto do escalonamento de cada tarefa separadamente através da previsão de impacto durante a geração do escalonamento.

Para ilustrar os possíveis benefícios da melhoria proposta, considere o DAG apresentado na Figura 5.6. O custo para executar cada tarefa nos três diferentes (heterogêneos) recursos é dado pela tabela abaixo do grafo na mesma figura. Supomos que os recursos estão conectados por enlaces de mesma capacidade para simplificar o exemplo. Dessa forma, os requisitos de comunicação entre tarefas são determinados somente pela quantidade de dados enviada, que é mostrada ao lado de cada arco do DAG. Dois escalonamentos são mostrados do lado direito da figura: o primeiro ilustra o escalonamento produzido pela versão padrão do HEFT, enquanto o segundo mostra o escalonamento obtido utilizando a versão proposta, com a utilização de *lookahead* para avaliar o impacto de cada decisão de escalonamento. O segundo escalonamento possui um *makespan* aproximadamente 30% menor que *makespan* produzido pelo HEFT original (184 contra 260 unidades de tempo).

Seguindo a classificação das tarefas dada pelo $rank_u$, a lista de prioridade das tarefas



	1	2	3	4	5	6	7	8	9
R1	19	28	36	15	30	33	12	13	41
R2	41	46	34	25	50	35	20	22	68
R3	34	20	62	37	54	59	21	24	73

Figura 5.6: Um exemplo ilustrando os benefícios do uso de informação de *lookahead*.

usada em nosso exemplo é $\{1, 4, 2, 3, 5, 6, 7, 8, 9\}$. A melhora no escalonamento é alcançada quando a tarefa 7 é escalonada. Sem a informação de *lookahead*, o menor EFT para a tarefa 7 é 141 no recurso $R2$. Em seguida, a tarefa 8 tem menor EFT em $R3$ (143), e a tarefa 9 tem o menor EFT em $R1$ (260). Quando o algoritmo utiliza a informação de *lookahead*, a tarefa 7 é escalonada no recurso que resulta no menor EFT para seus sucessores, no caso somente a tarefa 9. Essa seleção acontece da seguinte maneira. A tarefa 7 é atribuída a cada recurso, e para cada recurso onde a tarefa 7 é atribuída, seu filho (ou seja, a tarefa 9) é escalonada usando HEFT. Quando a tarefa 7 é atribuída ao recurso $R2$ seu EFT é 141, e o melhor recurso para a tarefa 9 seria $R1$ (260). Por outro lado, quando a tarefa 7 é atribuída a $R1$, ainda que ela fique com um EFT pior (143), o EFT de seu filho seria 184, o que é melhor que os 260 obtidos com a tarefa 7 em $R2$. Dessa forma, o algoritmo decide escalonar a tarefa 7 no recurso que proporciona um EFT pior para ela, mas que pode reduzir o EFT de sua sucessora no grafo. Após isso, a tarefa 8 é escalonada em $R2$, que resulta no menor EFT para a tarefa 9 em $R1$ (184). Finalmente, a tarefa 9 é escalonada no recurso que lhe propicia o menor EFT, pois ela não possui filhos.

Claramente, nossa proposta aumenta a complexidade de tempo do HEFT original. Entretanto, os benefícios em potencial resultantes desse aumento (cerca de 30% no exemplo acima, e até 20% nas simulações mostradas adiante) são justificadas na prática pois: (i) HEFT é uma heurística simples e rápida; (ii) existem heurísticas com complexidade de

tempo maior que HEFT que, entretanto, não geram necessariamente *makespans* menores que aqueles gerados pelo HEFT, como mostrado em [26].

Ao realizar o *lookahead*, o algoritmo pode levar em consideração de diferentes maneiras os EFTs dos filhos da tarefa sendo escalonada. Apresentamos quatro diferentes possibilidades para acrescentar ao HEFT a informação de *lookahead*. As quatro diferentes versões correspondem a todas as combinações de: (i) duas formas diferentes de selecionar um recurso para cada tarefa (baseado em duas abordagens diferentes para calcular o EFT dos filhos da tarefa); e (ii) duas maneiras diferentes de selecionar qual tarefa será escalonada primeiro (usando ou estritamente o atributo $rank_u$ do HEFT para classificar as tarefas ou parcialmente relaxando essa classificação, motivado por observações em [82, 106]). As quatro versões estudadas são como segue:

1. *Lookahead*: essa é a versão de *lookahead* como descrito no exemplo, onde o recurso selecionado para a tarefa t é aquele que minimiza o EFT máximo dentre todos os filhos de t em todos os recursos onde t é tentada;

2. *Lookahead com média ponderada de EFT*: o recurso selecionado para t é aquele que minimiza a média ponderada dos EFTs de todos os filhos de t, onde a média é ponderada usando o $rank_u$ de cada tarefa;

3. *Alteração da lista de prioridades*: dado que as primeiras duas tarefas não escalonadas e prontas (como classificadas pelo HEFT) são independentes, elas são individualmente consideradas, em turnos, para alocação fazendo uso de informação do *lookahead* dos filhos de ambas. A tarefa que foi selecionada primeiro quando o EFT máximo foi minimizado é a que será escalonada;

4. *Alteração da lista de prioridades com média ponderada de EFT*: o mesmo que a alteração de lista de prioridades acima, mas usando a média ponderada de EFTs dos filhos em detrimento do EFT simples.

5.2.1 Lookahead

Para implementar a versão com *lookahead* utilizamos a versão padrão do HEFT, incluindo o atributo de prioridade $rank_u$ para cada tarefa t_i. A versão com *lookahead* traz alguns problemas que precisam ser tratados. Seja t a tarefa com maior prioridade pronta para ser escalonada. A tarefa t é tentada em cada recurso e seus filhos são escalonados usando HEFT para calcular seus tempos estimados de término de acordo com o recurso onde t está escalonado. Entretanto, alguns (ou todos) os filhos de t podem não ficar prontos imediatamente após o escalonamento de t (como resultado de dependência de outro predecessor ainda não escalonado). Dessa forma, para calcular o EFT dos filhos

de t consideramos apenas os predecessores escalonados e t, ignorando outros atrasos que possam surgir devido a algum predecessor não escalonado. Então, a estimativa de tempo de término computada é, de certa forma, otimista.

Outro ponto é que existem várias maneiras de selecionar um recurso para uma tarefa t baseando-se em estimativas dos tempos de término de seus filhos. Nesta tese consideramos e avaliamos duas delas:

1. Selecionar o recurso que minimize o máximo EFT dos filhos de t dentre os recursos onde t é tentada. Em outras palavras, isso significa que t será escalonada no recurso que resultar no menor tempo de execução para todos os filhos de t, que foram escalonados usando HEFT;

2. Selecionar o recurso que minimize o EFT médio ponderado, $Wavg$, para os filhos de t. A razão para utilização do peso ponderado é que o EFT de filhos com baixa classificação dada pelo $rank_u$ (isto é, filhos que espera-se que sejam escalonados mais tarde entre todos) recebem um peso baixo. O EFT médio ponderado $Wavg$ de uma tarefa t é computado da seguinte maneira:

$$Wavg_t = \frac{\sum_{t_j \in L} (rank_u(t_j) \times EFT_{t_j})}{\sum_{t_j \in L} rank_u(t_j)},$$

   onde L é o *conjunto de lookahead*, contendo as tarefas que são usadas no *lookahead*. Na versão de *lookahead* descrita aqui, L contém todos os filhos de t. Quando o conjunto L é vazio, t é escalonada no recurso que propiciar o menor EFT para si própria.

A versão baseada na minimização do EFT máximo é mostrada no Algoritmo 8. A versão que utiliza a média ponderada requer uma modificação na linha 8 para atribuir $Wavg_t$ para EFT_{r_i}. Notamos que a complexidade de tempo do algoritmo com *lookahead* aumenta a complexidade de tempo do HEFT por um fator de $r \times c$, onde r é o número de recursos e c é o número médio de filhos por tarefa. Realizar um *lookahead* recursivo, alcançando um maior nível de *lookahead*, é factível, porém leva a uma complexidade de tempo maior e maior tempo de execução. Simulações mostraram que não houve uma melhora significativa nos resultados com o algoritmo apresentado se utilizamos *lookaheads* com profundidade 2 ou 3.

5.2.2 Alteração na Lista de Prioridades

Motivado pelas observações feitas em [82, 106], onde foi mostrado que algumas vezes relaxar a ordem de prioridade das tarefas obtidas na classificação pode resultar em escalon-

Algoritmo 8 HEFT com *lookahead*

```
1: Ordenar tarefas usando o rank_u
2: while há tarefas não escalonadas do
3:    t ← tarefa não escalonada com o maior rank_u
4:    L ← sucessores de t
5:    for all recurso r_i ∈ R do
6:       Escalona t em r_i
7:       Escalona todas as tarefas de L usando HEFT
8:       EFT_{r_i} ← máximo EFT entre as tarefas em L
9:       Retorna o escalonamento ao estado anterior ao início deste laço
10:   end for
11:   Escalona t em r_i tal que EFT_{r_i} <= EFT_{r_k} ∀r_k ∈ R
12: end while
```

amentos melhores, desenvolvemos uma versão que pode alterar a ordem de escalonamento da primeira tarefa não escalonada, t, com seu sucessor na lista de prioridades, t_1, dependendo da saída das informações de *lookahead*. Supomos que essas duas tarefas, t e t_1, estão prontas (isto é, todos os seus predecessores já estão escalonados), e que, portanto, elas são independentes entre si (isto é, sua ordem de execução não está ligada a uma restrição de dependência). A idéia dessa versão é realizar também um *lookahead* na lista de prioridades, além do *lookahead* no EFT dos filhos descrito até aqui. Essa versão trabalha de maneira similar à apresentada no Algoritmo 8, mas com duas diferenças:

- Além do filho da primeira tarefa pronta, t, o conjunto L inclui a segunda tarefa pronta com a prioridade mais alta, t_1, e os filhos de t_1;

- a iteração interna no algoritmo de *lookahead* (linhas 5-10) é executada duas vezes: primeiro considerando t como a primeira tarefa a ser escalonada, e depois considerando t_1.

Então, a ordem de escalonamento que apresentar um resultado melhor é selecionada. Uma versão desse algoritmo, que minimiza o tempo de término de todas as tarefas de L, é ilustrado no Algoritmo 9. A linha 10 pode ser mudada para usar a abordagem de média ponderada. Novamente, realizar um *lookahead* mais profundo resultaria em maior complexidade de tempo e maior tempo de execução. Por exemplo, inserir mais uma tarefa da lista de prioridades no conjunto L resultaria em 6 combinações distintas a serem tentadas, tendo como conseqüência um tempo de execução demasiadamente alto.

Algoritmo 9 HEFT com *lookahead* e alteração de prioridades

1: Ordenar tarefas usando o $rank_u$
2: **while** há tarefas não escalonadas **do**
3: $t \leftarrow$ tarefa pronta não escalonada com maior $rank_u$
4: $t_1 \leftarrow$ tarefa pronta não escalonada com o segundo maior $rank_u$
5: **for** $(i = 0; i < 2; i++)$ **do**
6: $L \leftarrow (suc(t)) \cup t_1 \cup (suc(t_1))$
7: **for all** recurso $r_i \in \mathcal{R}$ **do**
8: Escalona t em r_i
9: Escalona todas as tarefas de L usando HEFT
10: $EFT_{t,r_i} \leftarrow$ máximo EFT entra as tarefas em L
11: Retorna o escalonamento ao estado anterior ao início deste laço
12: **end for**
13: $recurso_t \leftarrow r_i$ tal que $EFT_{t,r_i} <= EFT_{t,r_k} \forall r_k \in \mathcal{R}$
14: $EFT_t \leftarrow \min_{r_i \in R} EFT_{t,r_i}$
15: Retorna o escalonamento ao estado anterior ao início deste laço
16: $aux \leftarrow t$; $t \leftarrow t_1$; $t_1 \leftarrow aux$
17: **end for**
18: Seleciona t que resultou no menor EFT_t
19: Escalona t em $recurso_t$
20: **end while**

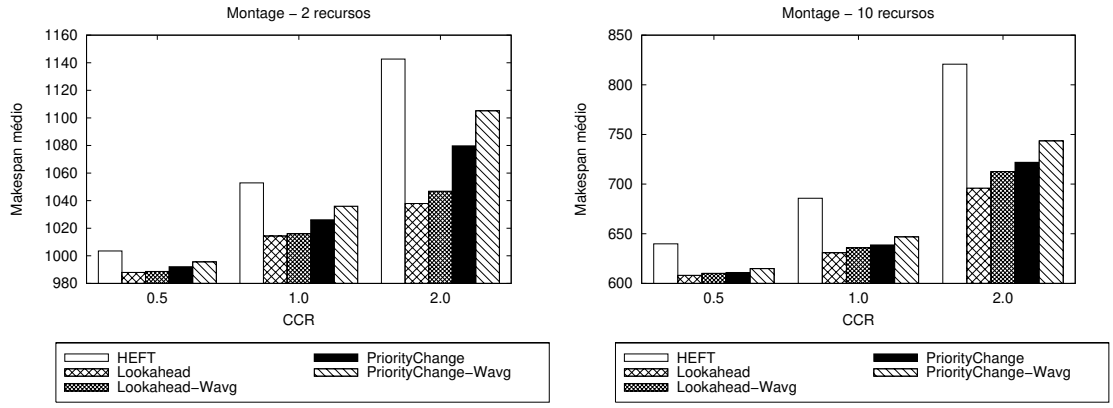


Figura 5.7: *Makespan* médio para DAGs Montage usando 2 e 10 recursos.

5.2.3 Resultados Experimentais

Nas simulações realizadas, avaliamos o *makespan* médio para o algoritmo com *lookahead* e para o algoritmo com alteração da lista de prioridades. Ambos foram avaliados com a função objetivo de minimizar o EFT máximo e com a função objetivo de minimizar o EFT médio ponderado dos filhos das tarefas sendo escalonadas. Adicionalmente, medimos o tempo de execução de cada algoritmo para compará-lo com o HEFT original. As simulações foram executadas usando 2 e 10 recursos heterogêneos no ambiente alvo, cada um com capacidade de processamento escolhida aleatoriamente dentro do intervalo $(10, 100)$. Os recursos formavam um grafo completo, conectados por uma rede heterogênea. Cada enlace teve sua largura de banda escolhida aleatoriamente do intervalo $(10, 100)$.

As simulações foram realizadas para grafos das aplicações Montage, AIRSN, LIGO, Chimera-1 e Chimera-2. O custo de computação de cada nó foi escolhido aleatoriamente no intervalo $(500, 4000)$. Para cada tipo de DAG realizamos simulações utilizando relações entre comunicação e computação (*communication to computation ratios* - CCR) de $0,5$, $1,0$ e $2,0$. O CCR é definido como a taxa entre a quantidade de comunicação e a quantidade de computação realizados durante a execução do DAG. Os resultados apresentados são médias sobre 500 execuções.

***Montage**:* Resultados mostrados na Figura 5.7. A maior melhoria no *makespan* sobre o HEFT foi obtida pelo algoritmo de *lookahead* usando a abordagem de minimização do EFT máximo de todos os filhos. As melhorias variaram de $1,55\%$, com 2 recursos e $CCR = 0,5$, a $15,2\%$, com 10 recursos e $CCR = 2,0$.

***AIRSN**:* Resultados mostrados na Figura 5.8. O algoritmo de alteração da lista de prioridades utilizando a abordagem da média ponderada apresentou os melhores resultados para o DAG AIRSN com 2 recursos e $CCR = 0,5$ (melhora de $2,85\%$) ou $CCR = 1,0$

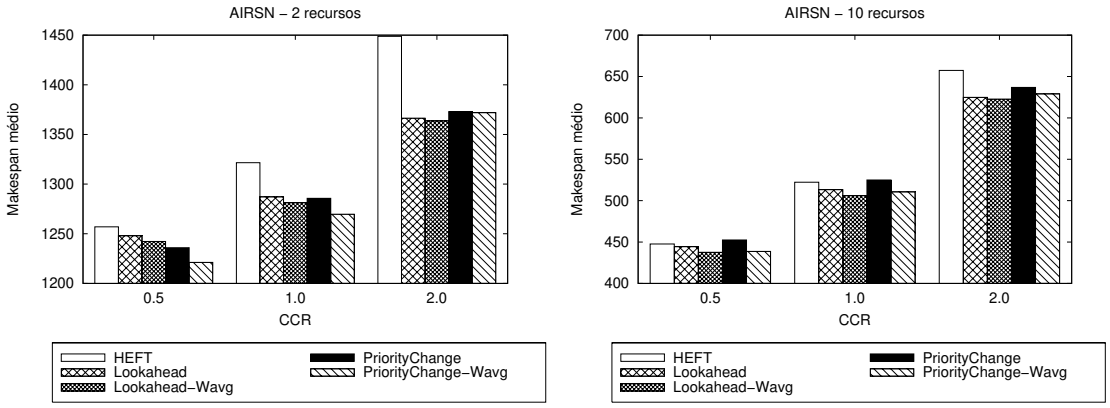


Figura 5.8: *Makespan* médio for para DAGs AIRSN usando 2 e 10 recursos.

(melhora de $3,93\%$). Com 2 recursos e $CCR = 2,0$, o algoritmo com *lookahead* foi um pouco melhor que o de alteração de prioridades, melhorando o *makespan* em $5,88\%$. Com 10 recursos os resultados são similares para os algoritmos de *lookahead* e alteração de lista de prioridades quando ambos estão usando a abordagem da média ponderada, com melhorias em torno de 6%.

***LIGO**:* Resultados mostrados na Figura 5.9. O *makespan* é similar para todos os algoritmos quando consideramos 2 recursos, com os algoritmos que utilizam a média ponderada saindo-se ligeiramente melhores quando $CCR = 2,0$. Para simulações com 10 recursos, quando $CCR = 0,5$ a alteração da lista de prioridade usando média ponderada foi um pouco melhor, melhorando o escalonamento do HEFT em $0,77\%$. Com $CCR = 1,0$ ambos os algoritmos com média ponderada geraram resultados similares, com melhorias em torno de 2%, enquanto com $CCR = 2,0$ o *lookahead* com média ponderada teve melhor desempenho que os outros algoritmos, resultando em um *makespan* médio $7,2\%$ menor que aqueles gerados pelo HEFT.

***Chimera-1**:* Resultados mostrados na Figura 5.10. Com o CCR de $0,5$ ou $1,0$ os resultados são similares para os quatro algoritmos propostos. Com 2 recursos e $CCR = 0,5$ a melhoria em comparação ao HEFT é em torno de $1,7\%$, enquanto com $CCR = 1,0$ a melhoria é em torno de $3,5\%$. Resultados para 10 recursos mostram uma melhoria em torno de $5,2\%$ com $CCR = 0,5$ e 11% com $CCR = 1,0$. Quando o CCR é maior, a melhoria no *makespan* com o algoritmo de *lookahead* com média ponderada foi em torno de 20%.

***Chimera-2**:* Resultados mostrados na Figura 5.11. Resultados para o DAG Chimera-2 são similares aos do Chimera-1. A maior melhoria com 2 recursos é dada pelo lookahead com média ponderada quando $CCR = 2,0$ $(2,6\%)$. Com 10 recursos a melhora máxima foi em torno de $14,4\%$ com $CCR = 2,0$.

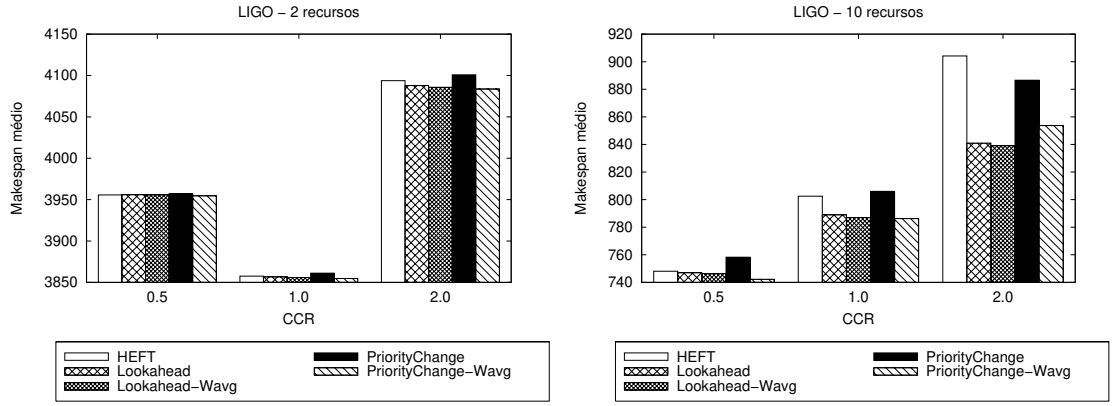


Figura 5.9: *Makespan* médio para DAGs LIGO usando 2 e 10 recursos.

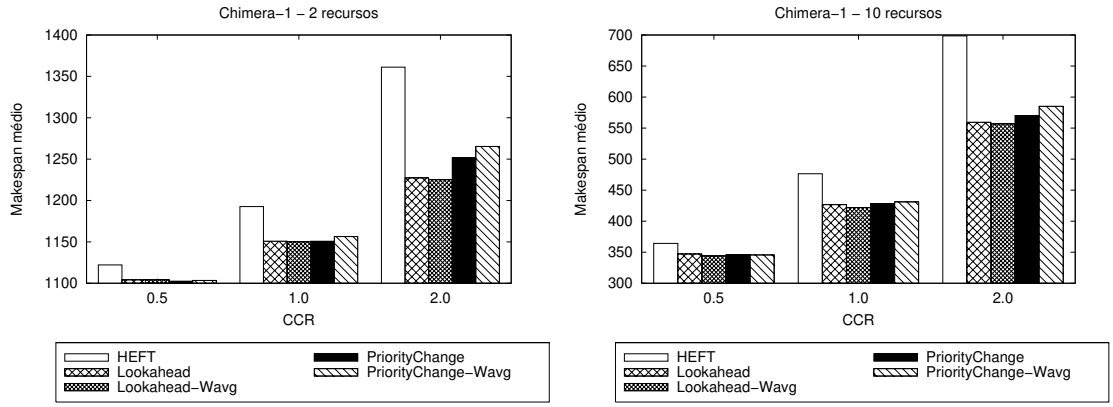


Figura 5.10: *Makespan* médio para DAGs Chimera-1 usando 2 e 10 recursos.

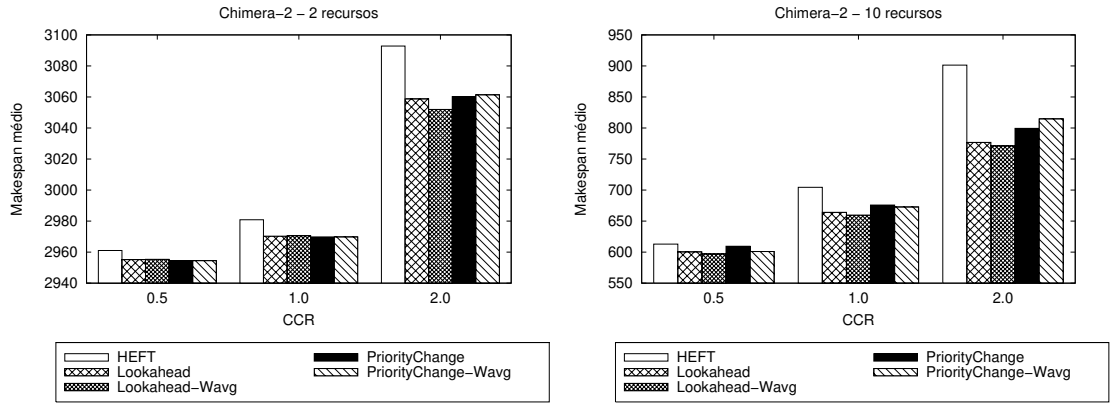


Figura 5.11: *Makespan* médio para DAGs Chimera-2 usando 2 e 10 recursos.

Discussão

Analisando os resultados para cada DAG, observamos que os algoritmos propostos parecem gerar melhores resultados quando há mais recursos e quando o CCR é maior. O exemplo mostrado na Figura 5.6 nos dá uma dica que pode explicar esse comportamento: quando o algoritmo realiza o *lookahead* para a tarefa 7, ele verifica que a comunicação entre a tarefa 7 e a tarefa 9 deve ser priorizada em detrimento do EFT da tarefa 7. Portanto, quando o CCR é mais alto e existem mais recursos para os quais as tarefas podem ser atribuídas, a informação obtida pelo *lookahead* pode prever quando transferências de dados futuras podem ser muito custosas, e assim evitá-las.

Uma segunda observação é que a alteração da lista de prioridades não parece melhorar significativamente os resultados apesar da complexidade adicional. Dessa forma, considerar o impacto das decisões de escalonamento somente nos filhos da tarefa é suficiente para tratar os problemas apresentados em [82, 106], que motivaram a idéia de relaxar parcialmente a ordem de prioridade.

Tempo de execução dos algoritmos

Um ponto importante para avaliar nos algoritmos propostos é quanto tempo eles levam para executar. Como eles consideram um espaço de busca mais amplo que o HEFT, também possuem uma maior complexidade de tempo. O tempo de execução de cada algoritmo para simulações com 10 recursos é mostrado na Figura 5.12. Como esperado, as versões com *lookahead* levaram mais tempo para executar que o HEFT original, e a alteração da lista de prioridades faz com que os algoritmos levem mais tempo que o *lookahead* simples. Outro ponto é que um maior número de arcos resulta em maior tempo de execução. Por exemplo, o DAG Chimera-1 tem uma alta densidade de arcos e seu tempo de execução é mais alto do que para o DAG AIRSN, mesmo com o AIRSN possuindo mais nós. Contudo, em termos absolutos, como mostram os tempos de execução, nenhum dos algoritmos tem um tempo de execução proibitivamente alto quando comparado ao HEFT. O tempo de execução para as versões com *lookahead* simples (sem alteração da lista de prioridades) é no máximo quatro vezes maior que o HEFT para o DAG Chimera-2, que possui uma alta densidade de arcos. Mesmo assim, em termos absolutos, isso é menos que $0,5$ segundos, o que não é proibitivo e torna os resultados atrativos em termos práticos, especialmente em relação aos benefícios esperados no *makespan*. Para outros DAGs com menor densidade de arcos, a versão com *lookahead* foi duas vezes mais lenta que o HEFT original.

Os resultados apresentados pelos algoritmos propostos, em simulações com alguns DAGs de aplicações reais, indicam que as versões com *lookahead* podem melhorar o escalonamento do HEFT, especialmente em casos onde a comunicação é alta, sem ap-

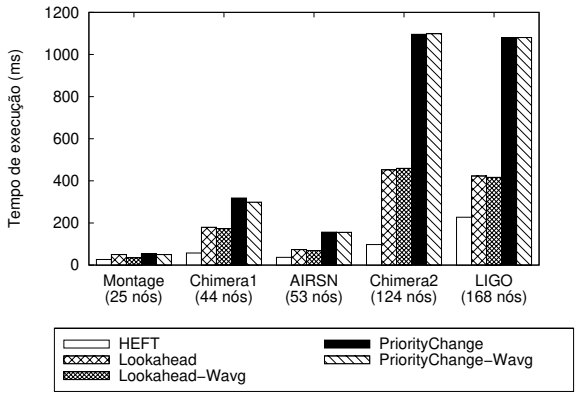


Figura 5.12: Tempos de execução dos algoritmos.

resentar tempos de execução proibitivos.

5.3 Algoritmo Bi-Critério

Para minimizar o *makespan* do *workflow*, desenvolvemos um algoritmo de escalonamento que considera a instanciação (ou criação) de novos serviços no conjunto de recursos ao escalonar os nós que compõem o *workflow de serviços*, ao invés de usar somente recursos que já existem. Dessa forma, o tempo de execução do *workflow* pode ser minimizado pela criação dos serviços necessários em recursos com alta capacidade de processamento. Mas alguns pontos a serem tratados devem ser considerados no desenvolvimento de tal algoritmo. Primeiramente, o algoritmo deve considerar quanto custa enviar e publicar o serviço em um novo recurso. Note que, em um *utility grid* [22], a criação de um novo serviço pode implicar em custos monetários. Dessa forma, a criação de muitos serviços pode levar a uma execução do *workflow* muito cara. Segundo, criar serviços de forma negligente pode levar ao desperdício de recursos computacionais, como utilização desnecessária de largura de banda ou processamento. Além disso, a publicação de novos serviços de forma deliberada pode atrasar o término de *workflows* que já estão em execução, pois os enlaces e processamento ficariam em uso constante para que tais serviços sejam transferidos e publicados nos recursos remotos. Em terceiro lugar, a não criação de serviços (ou a criação de poucos serviços) pode resultar em tempos de execução altos para os *workflows*. Assim, há um *trade-off* claro entre criar mais serviços para acelerar a execução de determinado DAG, ou criar menos serviços para não desperdiçar recursos (ou dinheiro) e para não interferir na execução de outros DAGs [18].

5.3.1 Escalonamento de Serviços

Seja $\mathcal{S}$ o conjunto de todos os serviços instanciados em todos os recursos. Consideramos que cada recurso r_i tem um conjunto de serviços que já estão instanciados $\mathcal{S}_i = \{s_{i,1}, ..., s_{i,q}\} \in \mathcal{S}$, $q \geq 0$. Além disso, cada tarefa t_a tem um conjunto de serviços candidatos $\zeta_a \subseteq \mathcal{S}, \zeta_a \neq \varnothing$, que implementam a tarefa t_a. Dessa forma, uma tarefa t_a pode ser escalonada em um recurso r_i se $\exists\ s_{i,p} \mid s_{i,p} \in \zeta_a$.

Nossa abordagem para esse problema foi de *criar serviços quando necessário*. Isso significa que o algoritmo criará novos serviços somente quando a não criação resulte em atraso na execução do *workflow*. Para chegar a isso, é indispensável que serviços que estejam no caminho crítico do DAG sejam escalonados nos melhores recursos através da criação de serviços para eles. Para serviços que não estão no caminho crítico, o algoritmo utiliza o ALAP [89] (do inglês *as late as possible*) para determinar quando um serviço deve ser criado. Em termos gerais, os passos do algoritmo são:

1. Selecionar qual tarefa do grafo (que representa um serviço) deverá ser a próxima a ser executada;

2. Determinar se um serviço deve ser criado para a tarefa selecionada;

3. Escalonar a tarefa selecionada no melhor recurso, dadas as condições para criação do serviço.

Esses três passos são repetidos até que todos os nós do DAG estejam escalonados.

No primeiro passo, a próxima tarefa a ser escalonada é a tarefa não escalonada com o maior prioridade $\mathcal{P}_i$. O segundo passo utiliza o ALAP da tarefa selecionada no primeiro passo. Intuitivamente, o ALAP de uma tarefa é o tempo máximo de início de uma tarefa tal que o caminho crítico do DAG não tenha seu tamanho aumentado. Dessa forma, através do ALAP podemos determinar quanto uma tarefa pode ser atrasada sem aumentar o tamanho do escalonamento, ou em quanto o seu *EST* pode ser aumentado. O ALAP de t_a é computado subtraindo-se $\mathcal{P}_a$ (prioridade de t_a) do tamanho do caminho crítico do DAG. Assim, uma tarefa t_{cc} no caminho crítico do grafo não pode ser atrasada, pois seu ALAP é sempre igual ao seu *EST*. Conseqüentemente, tarefas no caminho crítico serão sempre escalonadas no melhor recurso disponível, e os serviços necessários serão criados naquele recurso.

Para determinar se um serviço deve ser criado para executar a tarefa t_a do *workflow*, para cada recurso r_i que já tem o serviço de t_a disponível, t_a é inserido no escalonamento (fila) de r_i e o EST de t_a é computado. Seja $EST(t_a, r_i)$ o EST da tarefa t_a no escalonamento do recurso r_i. Se $\exists\ s_{i,q} \mid s_{i,q} \in \zeta_a$ e $EST(t_a, r_i) \leq ALAP_{t_a}$, então não é necessário criar um novo serviço para t_a, e t_a é escalonada no recurso r_i que proporcionar o menor EFT ($EST + w_{a,i}$, sendo $w_{a,i}$ seu tempo de execução naquele recurso). Caso contrário, t_a

é escalonada no recurso que tem o menor EFT mais o tempo de criar o novo serviço no recurso. O Algoritmo 10 mostra uma visão geral desses passos.

Algoritmo 10 Visão geral do algoritmo bi-critério

```
1: Computa prioridade, EST e ALAP para cada tarefa
2: while há tarefas não escalonadas do
3:    t ← tarefa não escalonada com maior prioridade
4:    melhor_recurso_t ← NULL
5:    melhor_tempo_t ← ∞
6:    R_t ← recursos com serviços em ζ_t
7:    for all r_i ∈ R_t do
8:       Computa EST(t, r_i)
9:       if (EST(t, r_i) ≤ ALAP_t) AND (EST(t, r_i) + w_{t,i} < melhor_tempo_t) then
10:         melhor_recurso_t ← r_i
11:         melhor_tempo_t ← EST(t, r_i) + w_{t,i}
12:      end if
13:   end for
14:   if melhor_recurso_t == NULL then
15:      R ← todos os recursos disponíveis
16:      for all r_i ∈ R do
17:         Computa EST(t, r_i)
18:         if r_i não tem serviços em ζ_t then
19:            custo_criar_servico_t ← custo_enviar_código_t + custo_instanciar_t
20:            EST(t, r_i) ← EST(t, r_i) + custo_criar_servico_t
21:         end if
22:         if EST(t, r_i) + w_{t,i} ≤ melhor_tempo_t then
23:            melhor_recurso_t ← r_i
24:            melhor_tempo_t ← EST(t, r_i) + w_{t,i}
25:         end if
26:      end for
27:   end if
28:   Escalona t em melhor_recurso_t
29:   Recomputa EST e ALAP para cada tarefa
30: end while
```

A primeira linha do Algoritmo 10 computa os atributos EST, Prioridade e ALAP. Após isso, uma iteração para escalonar cada tarefa é iniciada (linha 2). A próxima linha seleciona a tarefa não escalonada com o maior prioridade para ser escalonada, enquanto as linhas 4 e 5 inicializam duas variáveis usadas no algoritmo. O conjunto $\mathcal{R}_t$, que contém os recursos que possuem o serviço apto a executar a tarefa t, é criado na linha 6. Em seguida, a iteração entre as linhas 7 e 13 procura pelo melhor recurso para executar t

sem ultrapassar o ALAP de t. Se nenhum recurso for encontrado (linha 14), então o algoritmo inicia a busca pelo melhor recurso no conjunto de todos os recursos (linha 15). Se o recurso atual r_i não possuir o serviço necessário (linha 18), o algoritmo soma ao EST de t ambos os custos inerentes à criação do novo serviço e à execução da tarefa (linhas 19 e 20). Então, o algoritmo verifica se o escalonamento corrente tem o melhor *EST + tempo de execução da tarefa* de cada recurso já testado (linha 22). Se sim, ele é eleito como o melhor recurso atual (linhas 23 e 24). Antes da iteração externa do algoritmo retornar ao início, a tarefa corrente é escalonada no melhor recurso encontrado (linha 28), e os atributos são recalculados (linha 29), já que o novo escalonamento pode alterar os valores de EST e ALAP.

Note que o caminho crítico do grafo é atualizado dinamicamente a cada iteração do laço externo. Se uma tarefa t que não pertence ao caminho crítico pode somente ser escalonada em um recurso que resulte em um EST maior que o ALAP de t, então t estará no caminho crítico na próxima iteração. Ainda, a criação de novos serviços pode ser controlada por um multiplicador no ALAP na linha 9 do algoritmo, o que daria um controle sobre o *trade-off* entre a criação de serviços e o *makespan* de acordo com o ambiente considerado.

Em tempo de execução, a criação de um novo serviço para a tarefa t é realizada após o término de todos os seus predecessores. Essa política tem o objetivo de não usar recursos antes do início da tarefa t e seu *workflow*, pois isso poderia atrasar a execução de outros serviços que estão executando. Adicionalmente, a criação de serviços em tempo de escalonamento pode desperdiçar largura de banda e processamento se o recurso de destino sair da grade. Além disso, se considerarmos a situação em que a criação de um novo serviço ou a utilização de um recurso tem custos monetários, criar serviços que não serão utilizados pode levar ao aumento de custos.

5.3.2 Resultados Experimentais

Comparamos o algoritmo proposto com uma versão do algoritmo HEFT (*Heterogeneous Earliest Finish Time* [95]). A única diferença do algoritmo usado na comparação com o HEFT é que, para cada tarefa, ele utiliza apenas os recursos que possuem o serviço necessário para a tarefa t sendo escalonada, ao invés de considerar todos os recursos disponíveis.

Variamos o número de grupos de 2 a 25, cada grupo tendo de 2 a 10 recursos com capacidades geradas aleatoriamente no intervalo $(10, 100)$. Capacidades dos enlaces entre grupos foram geradas aleatoriamente no mesmo intervalo, assim como as capacidades dos enlaces entre recursos dentro do mesmo grupo. Nesta simulação, utilizamos 16 DAGs, sendo 15 gerados aleatoriamente e mais o grafo da aplicação CSTEM.

Um parâmetro importante pode influenciar o desempenho do algoritmo que não cria serviços: o número de serviços já disponíveis nos recursos. Seja δ o número esperado de serviços existentes para cada tarefa do *workflow* e seja $P_{s_t,r_i} = \frac{\delta}{|\mathcal{R}|}$ a probabilidade de um serviço s_t, que pode executar a tarefa t do DAG, existir no recurso r_i. Simulamos δ variando de 1 a 5. Uma simulação com $\delta = 3$, por exemplo, significa que $\forall\ t \in \mathcal{V}$: $E(|\zeta_t|) = 3$, ou seja, que é esperado que, para cada tarefa t do DAG, o número de recursos que podem executar t seja 3. Também simulamos uma situação hipotética onde todos os serviços existiam em todos os recursos, isto é, $\delta = |\mathcal{R}|$. Essa simulação teve como objetivo avaliar se a política de ALAP restringiria o uso dos recursos de modo a influenciar negativamente no *makespan* dos DAGs.

Outra característica que pode influenciar os resultados são as capacidades dos recursos onde os serviços já existentes estão disponíveis. Supondo que serviços são geralmente instanciados em recursos com boa capacidade, também simulamos cenários onde os serviços já existentes só poderiam estar em recursos de um conjunto $\mathcal{R}_{M\%} = \{r_i \in \mathcal{R} | p_{r_i} \geq mediana\}$, onde *mediana* é a mediana do conjunto de capacidades de processamento de todos os recursos em $\mathcal{R}$. Em outras palavras, serviços poderiam somente existir em recursos que possuíssem capacidade de processamento igual ou maior que a mediana das capacidades de todos os recursos. Nesses cenários, $P_{s_t,r_i} = \frac{\delta}{|\mathcal{R}_{M\%}|}$.

Cada algoritmo foi executado 2.000 vezes para cada quantidade de grupos. Os custos relativos à transferência e instanciação de um novo serviço foram obtidos de uma distribuição normal de acordo com o tamanho médio (250Kb) e desvio padrão (120Kb) de arquivos *.gar* encontrados em nosso laboratório e tamanhos de arquivos encontrados na literatura [79]. Esses arquivos são chamados de *grid archives*, e cada um deles possui o código fonte necessário para publicação de um serviço no Globus Toolkit 4. Os resultados mostram intervalos de confiança de 99%.

Comparamos o *makespan* médio, *speedup* médio e SLR médio dos algoritmos. Também medimos qual foi a porcentagem de serviços usados pelo algoritmo proposto que já existiam nos recursos, representados por barras nos gráficos (eixo do lado direito). Rótulos "*Existente*" são para o algoritmo que não cria novos serviços (abordagem HEFT-semelhante), rótulos "*Proposto*" são para o algoritmo proposto, e rótulos com "*Mediana*" são para simulações onde os serviços existentes poderiam somente existir nos recursos com capacidade de processamento igual ou melhor que a mediana.

Schedule Length Ratio

A Figura 5.13(a) mostra o SLR médio para $\delta = 1$. Considerando que os serviços já existentes poderiam estar em qualquer recurso, o algoritmo proposto melhora o desempenho do algoritmo que utiliza apenas serviços existentes em torno de 43%, com 2 grupos, e em torno de 53% quando há 25 grupos. Esse resultado é alcançado com o algoritmo proposto

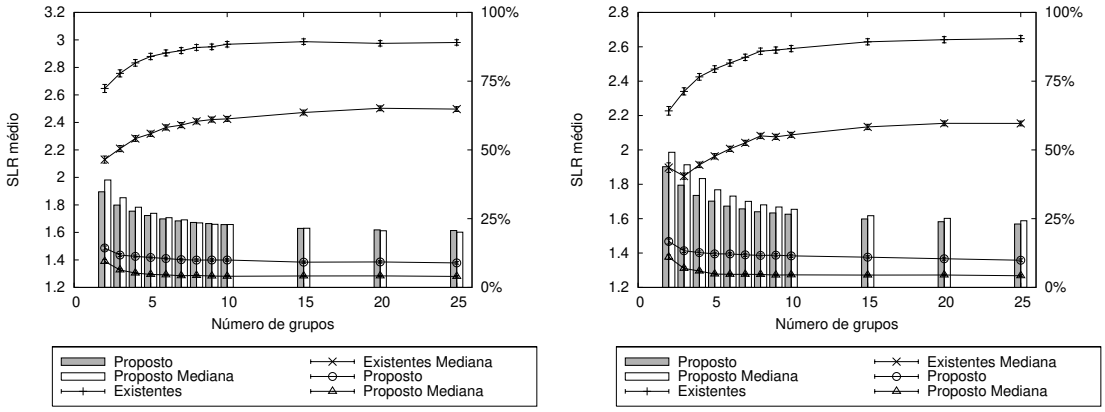


Figura 5.13: SLR médio para $\delta = 1$ e $\delta = 2$.

utilizando serviços existentes para 34% das tarefas com 2 grupos, e em torno de 21% para 10 ou mais grupos. Quando foi considerado que os serviços já existentes poderiam estar apenas nos melhores recursos, para 2 grupos o SLR foi 34% menor, enquanto que com 25 grupos essa diferença foi de 49%, respectivamente usando serviços existentes para 39% e 21% das tarefas. Então, a utilização do ALAP para decidir quando criar novos serviços melhora a qualidade do escalonamento e permite que o *workflow* use uma quantidade razoável de serviços já existentes, criando outros quando necessário.

Podemos observar o mesmo padrão para $\delta = 2$ nos resultados de SLR (Figura 5.13(b)) e *speedup* (Figura 5.15(b)), assim como para os resultados de *makespan* (Figura 5.17(b)), com um desempenho levemente melhor do algoritmo que não cria serviços em comparação ao $\delta = 1$. Isso se explica pelo fato do escalonador ter mais opções de recursos para cada tarefa, aumentando a probabilidade que um bom recurso seja escolhido. Quando os serviços existentes poderiam estar em qualquer recurso, o SLR do algoritmo proposto foi 34% menor para 2 grupos e 48% menor para 25 grupos. Essa melhoria foi alcançada utilizando serviços já existentes para 44% e 23% das tarefas, respectivamente. Quando os serviços existentes poderiam estar somente nos recursos com capacidade de processamento igual ou acima da mediana, a melhoria variou de 27% para 2 grupos, a 41% para 25 grupos, usando serviços existentes para 49% e 24% das tarefas, respectivamente.

Quando $\delta = |R|$ podemos observar que o algoritmo proposto é um pouco pior que o HEFT [2]. A diferença de SLR varia de 3% a 5% (Figura 5.14). Note que esta é uma situação hipotética, pois publicar todos os serviços em todos os recursos pode levar a uma

[2]Nesse caso as tarefas poderiam ser escalonadas em qualquer recurso, então o algoritmo usado na comparação foi o HEFT original.

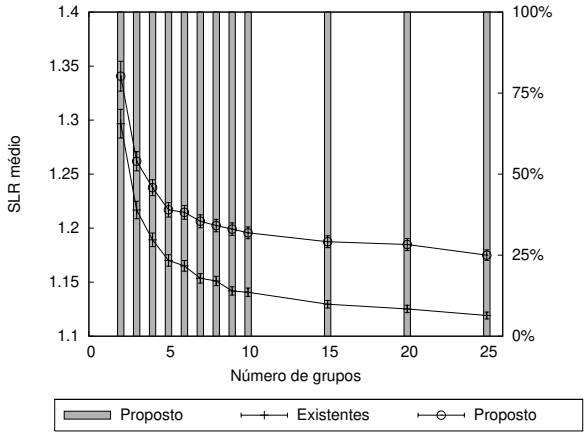


Figura 5.14: SLR médio para $\delta = |\mathcal{R}|$.

execução cara e desperdício de recursos.

Speedup

Para o *speedup* médio com $\delta = 1$ (Figura 5.15(a)), a melhoria fica entre 74%, para 2 grupos, e 113%, para 25 grupos, com serviços já existentes podendo estar em qualquer recurso. No cenário onde os serviços já existentes poderiam estar apenas nos melhores recursos, esses números são de 50% e 92%, respectivamente. O *speedup* para $\delta = 2$ (Figura 5.15(b)) foi melhorado em 48% para 2 grupos e 91% para 25 grupos com os serviços já instanciados podendo estar em qualquer recurso. Com os serviços existentes podendo estar apenas nos melhores recursos, esses números foram de 32% e 68%, respectivamente. Quando $\delta = |R|$ (Figura 5.16), o algoritmo proposto foi levemente inferior ao HEFT, e a diferença entre eles variou entre 3% e 6%.

Note que quanto maior o número de grupos, maior a diferença entre o algoritmo proposto e o HEFT-semelhante. Isso também é observado nos resultados de SLR, e pode ser explicado pelo fato de que, quanto maior o número de recursos, maior a probabilidade de bons recursos não estarem sendo usados pelo DAG quando utiliza apenas serviços já existentes. Dessa forma, instanciar novos serviços pode fazer com que esses recursos sejam utilizados, aperfeiçoando a execução do *workflow*.

Makespan

A Figura 5.17(a) mostra o *makespan* médio para $\delta = 1$ e a Figura 5.17(b) mostra o *makespan* médio para $\delta = 2$. Para $\delta = 1$, a média obtida pelo algoritmo proposto foi 44% menor com 2 grupos e 55% menor com 25 grupos em execuções com qualquer

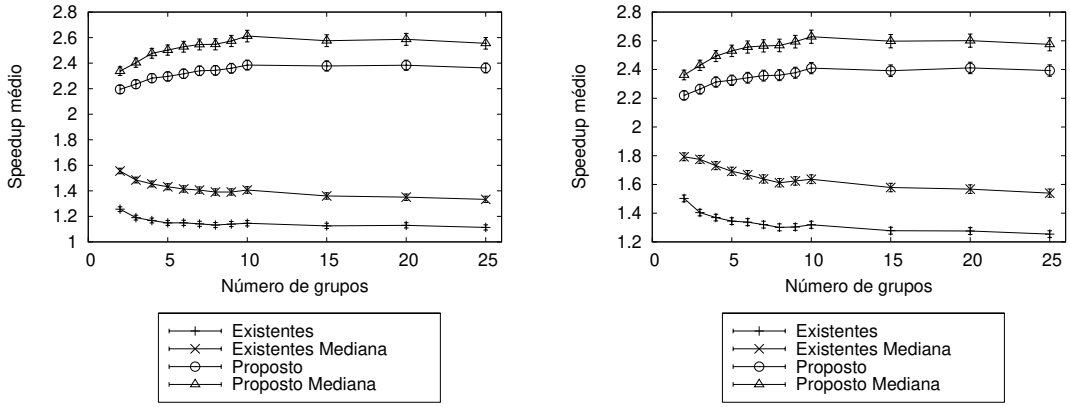


Figura 5.15: *Speedup* médio para $\delta = 1$ e $\delta = 2$.

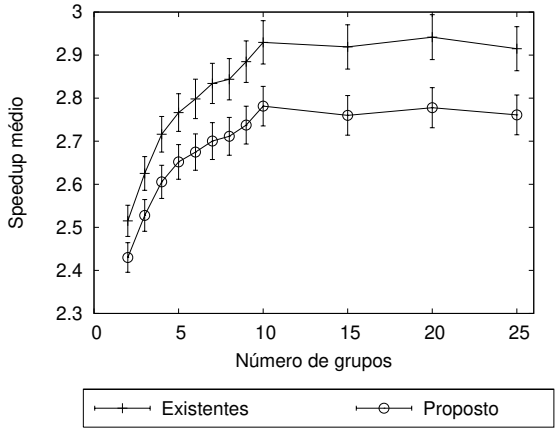


Figura 5.16: *Speedup* médio para $\delta = |R|$.

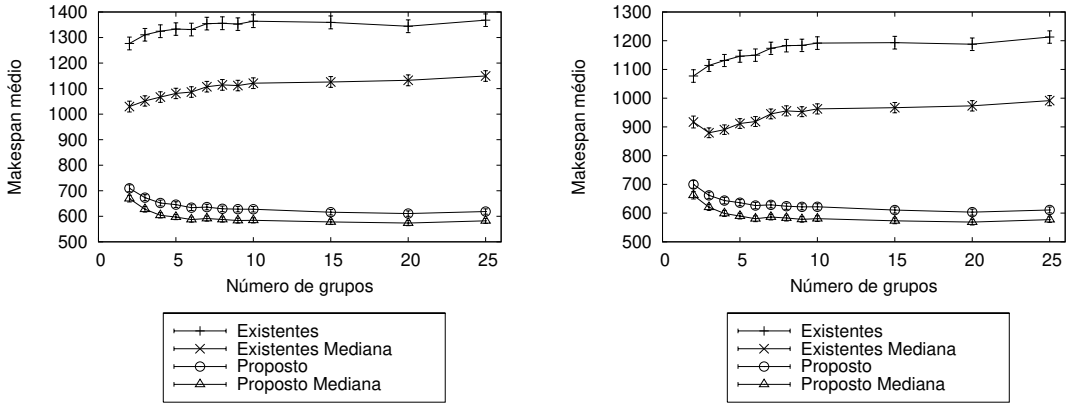


Figura 5.17: *Makespan* médio para $\delta = 1$ e $\delta = 2$.

recurso podendo hospedar os serviços já existentes. Para o cenário em que os serviços já existentes poderiam estar apenas nos recursos de maior capacidade, a melhoria foi de 35% para 2 grupos e 49% para 25 grupos. O *makespan* médio para $\delta = 2$ apresenta resultados similares. A melhoria vai de 35% a 50% quando serviços já existentes podiam estar em qualquer recurso, e de 28% a 42% quando os serviços já existentes podiam estar apenas nos recursos com capacidade igual ou superior à mediana.

Para $\delta = |R|$ (Figura 5.18) podemos observar que o algoritmo proposto tem um desempenho um pouco abaixo do HEFT, apresentando *makespans* na faixa de 3% a 6% maiores.

Discussão

Os resultados para $\delta = 1$ e $\delta = 2$ mostram que o algoritmo proposto pode melhorar significativamente o desempenho no escalonamento de *workflows* em grades baseadas em serviços. Essa melhora é alcançada através do escalonamento de novos serviços nos recursos com melhor desempenho. Isso é alcançado utilizando vários serviços que já existem através do uso do conceito de ALAP, que determina quais tarefas precisam de um novo recurso para não atrasar a execução do DAG. Quando comparado à execução tradicional de tarefas nos serviços existentes, a estratégia proposta mostra uma melhoria na execução do *workflow*, realizando uma melhor utilização dos recursos. Ainda, simulações com δ variando de 3 a 5 apresentaram os mesmos padrões, com o algoritmo proposto ainda obtendo melhores resultados nesses casos que o algoritmo HEFT-semelhante.

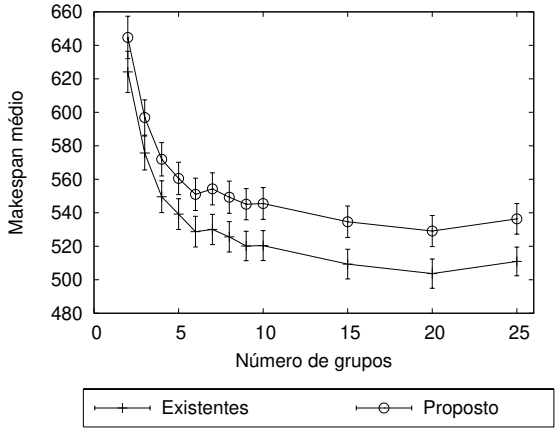


Figura 5.18: *Makespan* médio para $\delta = |R|$.

5.4 Escalonamento de Múltiplos DAGs

O ponto inicial no escalonamento de múltiplos *workflows* é decidir se os DAGs serão escalonados de forma independente ou se serão combinados em um DAGs e escalonados após isso. Portanto, os DAGs podem ser escalonados independentemente e seqüencialmente usando-se algum algoritmo de escalonamento, ou eles podem ser combinados para compor um único DAG que será escalonado, conseqüentemente realizando o escalonamento de múltiplos *workflows*. Outra abordagem é escalonar os DAGs independentemente, porém intercalando suas partes. Dessa forma, para escalonar mais de um DAG em um conjunto de recursos compartilhados podemos, em geral, usar uma das três estratégias:

- Escalonar os DAGs independentemente, um após o outro;
- Escalonar os DAGs independentemente, intercalando partes de cada DAG sendo escalonado;
- Combinar os DAGs em um único, e escalonar esse DAG resultante.

Para escalonar múltiplos DAGs, supomos que, em um dado momento, temos tarefas de N *workflows* para serem escalonadas. Note que isso não significa necessariamente que precisamos escalonar todas as tarefas de todos os DAGs no início do escalonamento. Quando um ou mais *workflows* chegam para serem escalonados, podemos considerar todas as tarefas ainda não executadas dos *workflows* que chegaram anteriormente. Dessa forma, o novo *workflow* pode ser escalonado em conjunto com os DAGs já escalonados, aproveitando-se dos espaços deixados por eles devido às dependências de dados. Quais

tarefas e quais DAGs são reescalonados quando um novo DAG chega podem ser definidos pelo *middleware*, que pode basear-se em alguma política pré-definida.

Nesta seção, descrevemos quatro algoritmos para escalonamento de múltiplos DAGs [15]:

- **Algoritmo seqüencial**: escalona um DAG após o outro nos recursos disponíveis. Um novo DAG só pode ser escalonado em um recurso após todas as tarefas já escalonadas naquele recurso. Esse algoritmo utiliza a primeira estratégia;
- **Algoritmo de busca de espaços**: também utiliza a primeira estratégia, escalonando os DAGs independentemente. Entretanto, faz uma busca por espaços entre as tarefas já escalonadas, portanto uma tarefa de um processo recém-escalonado pode ser escalonada antes de uma tarefa escalonada anteriormente, desde que a tarefa escalonada primeiro não sofra atraso;
- **Algoritmo de Intercalação**: usa a segunda estratégia. O algoritmo escalona pedaços de cada DAG em turnos, intercalando suas tarefas nas filas dos recursos disponíveis;
- **Algoritmo de agrupamento de DAGs**: utiliza a terceira estratégia. Antes de escalonar os DAGs, o algoritmo os combina em um único DAG. Esse DAG é escalonado, concretizando o escalonamento dos DAGs que o compõe.

Para implementar e avaliar essas abordagens de escalonamento para múltiplos DAGs, utilizamos uma versão modificada do algoritmo PCH. A modificação no algoritmo almeja melhorar a utilização dos espaços de transmissão de dados deixados pelas dependências de dados no escalonamento através da diminuição do tamanho dos agrupamentos gerados. Mais especificamente, a modificação é na busca em profundidade que realiza a composição dos agrupamentos de tarefas. Ao invés da busca parar quando o algoritmo encontra uma tarefa que não possui sucessores não escalonados, essa versão modificada pára quando encontra uma tarefa t_i que possui algum predecessor não escalonado, e t_i não é incluída no agrupamento. Dessa forma, os agrupamentos gerados possuem apenas tarefas com todos os predecessores já escalonados. Além disso, os agrupamentos gerados são potencialmente menores que os gerados pelo PCH original, o que aumenta a probabilidade do agrupamento ser encaixado entre outras tarefas nas filas dos recursos. Após criar cada agrupamento, na fase de seleção de recursos o PCH modificado busca o recurso que minimize o EFT do agrupamento. Portanto, o critério para escolher um recurso para um agrupamento cls_k é minimizar o EFT_{cls_k}, definido como:

$$EFT_{cls_k} = \max_{t_i \in cls_k} EFT_{t_i}$$

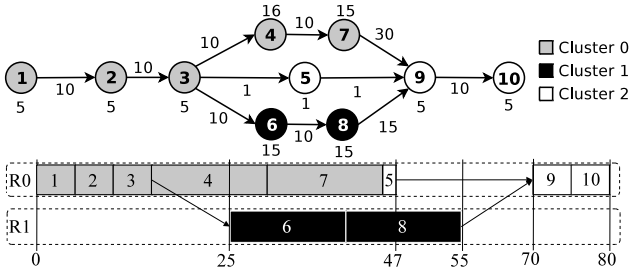


Figura 5.19: Exemplo de escalonamento utilizando o PCH modificado.

Um exemplo dos agrupamentos criados pelo PCH modificado, junto com o escalonamento resultante, é mostrado na Figura 5.19. Nesse exemplo, supomos que ambos os recursos têm capacidade de processamento igual a 1 e o enlace entre eles tem largura de banda igual a 1. Primeiramente, o agrupamento 0 é criado, onde a busca em profundidade inicia-se na tarefa 1, indo através das tarefas 2, 3, 4 e 7, e ao atingir a tarefa 9, a busca pára e essa tarefa não é adicionada ao grupo. Em seguida o agrupamento 0 é escalonado no recurso 0. O agrupamento 1, composto pelas tarefas 6 e 8, é escalonado no recurso 1, que resulta no menor EFT para a tarefa 8. Finalmente, o agrupamento 2, composto pelas tarefas 5, 9 e 10, é escalonado no recurso 0.

Os passos de agrupamento e seleção de recursos, como apresentados nesta seção, são utilizados pelos quatro algoritmos de escalonamento para múltiplos DAGs descritos a seguir.

5.4.1 Escalonamento Seqüencial

O primeiro algoritmo, e o mais direto, para escalonar múltiplos DAGs é simplesmente realizar o escalonamento de um DAG após o outro. Isso significa que, dados dois DAGs G_1 e G_2, uma tarefa $t \in G_2$ só pode ser escalonada em um recurso $r \in \mathcal{R}$ após todas as tarefas de G_1 escalonadas em r. Deste modo, o tempo inicial disponível em cada recurso após o escalonamento de um DAG $\mathcal{G}$ é o EFT da última tarefa de $\mathcal{G}$ escalonada no recurso. Essa abordagem é ilustrada no Algoritmo 11.

Esse é o mecanismo padrão utilizado quando múltiplos DAGs são escalonados (podendo utilizar diferentes heurísticas para realizar o escalonamento). Os DAGs são escalonados em ordem de chegada, um após o outro, com o escalonamento sendo iniciado pelo evento da chegada de um DAG. Portanto, o escalonador não modifica o escalonamento corrente e nem insere tarefas de novos *workflows* entre as tarefas já escalonadas.

Algoritmo 11 Escalonamento seqüencial

```
1: DAGs ⇐ workflows a serem escalonados.
2: for all G ∈ DAGs do
3:    while há tarefas não escalonadas em G do
4:       cls ⇐ cluster de G a ser escalonado
5:       r ⇐ recurso selecionado pelo PCH para cls
6:       Escalona o cluster cls em r
7:       Tempo(r) ⇐ EFT_cls
8:    end while
9: end for
```

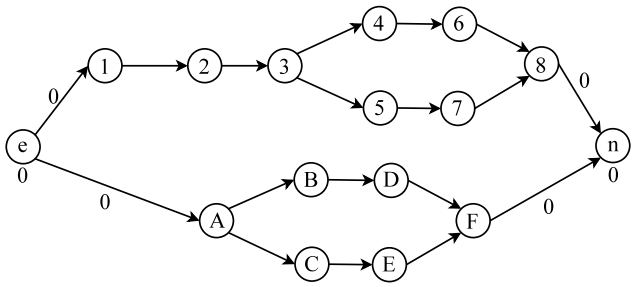


Figura 5.20: Exemplo de agrupamento de dois DAGs em um único através da adição de um nó de saída e um nó de entrada.

5.4.2 Agrupamento de DAGs

Além da estratégia de escalonar múltiplos *workflows* considerando-os como entidades separadas, outra abordagem é combinar todos os DAGs em apenas um através da criação de um nó de entrada e um nó de saída, conectando-os aos DAGs a serem escalonados. O novo nó de entrada tem custo 0, assim como seus arcos, que são conectados a cada nó de entrada dos DAGs que estão sendo escalonados. De forma similar, o novo nó de saída possui custo 0 e tem seus arcos provenientes de todos os nós de saída dos DAGs sendo escalonados. Dessa forma, um DAG único é criado, sendo escalonado em seguida nos recursos disponíveis. Sejam dois processos, $P0 = \{t_1, ..., t_8\}$ e $P1 = \{t_A, ..., t_F\}$. O DAG resultante após combinar $P0$ e $P1$ é mostrado na Figura 5.20, onde a tarefa t_e é o novo nó de entrada com custo 0, e a tarefa t_n é o novo nó de saída, também com custo 0. Os arcos adicionados e conectados a esses nós também têm custo 0, enquanto as tarefas e arcos existentes mantêm seus custos originais.

O Algoritmo 12 ilustra a abordagem de agrupamento de DAGs. A entrada do algoritmo é um grupo de *workflows* a serem escalonados ($DAGs$). O primeiro passo do algoritmo é criar um novo DAG (G_{grupo}), que será o DAG composto por todos os *workflows* no grupo $DAGs$. Para cumprir essa tarefa, o algoritmo insere as tarefas de custo nulo $t_{entrada}$ e t_{saida}

nesse novo DAG (linha 3). Após isso, no laço entre as linhas 4 e 7, cada nó de entrada de cada DAG existente é conectado ao nó recém-criado $t_{entrada}$. Da mesma forma, cada nó de saída dos DAGs existentes é conectado à tarefa t_{saida}. Após criar G_{grupo}, o algoritmo continua com o escalonamento utilizando o PCH.

Algoritmo 12 Group DAGs

1: $DAGs \Leftarrow$*workflows* a serem escalonados.
2: cria DAG G_{grupo}
3: insere as tarefas de custo nulo $t_{entrada}$ e t_{saida} em G_{grupo}.
4: **for all** $\mathcal{G} \in DAGs$ **do**
5: cria arco de custo nulo $(t_{entrada}, t_{G_0})$
6: cria arco de custo nulo (t_{saida}, t_{G_n})
7: **end for**
8: **while** há tarefas não escalonadas em G_{grupo} **do**
9: $cls \Leftarrow$ *cluster* de G_{grupo} a serem escalonadas
10: escalona o *cluster* cls no recurso selecionado pelo
11: **end while**

5.4.3 Algoritmo de Busca de Espaços

Nesta seção, apresentamos um algoritmo que escalona tarefas em cada recurso considerando os espaços entre tarefas já escalonadas [11]. Ao iniciar o escalonamento, se não há processos já escalonados nos recursos disponíveis, o escalonador não precisa fazer a busca por espaços, portanto o escalonamento continua com o PCH dinâmico com a extensão adaptativa. Se existem tarefas atualmente atribuídas a um ou mais recursos, o algoritmo faz a busca por espaços entre essas tarefas para encaixar os agrupamentos dos DAGs sendo escalonados.

Seja o escalonamento (fila) de um recurso r o conjunto $S_r = \{t_1, t_2, ..., t_k\}$. O espaço (*gap*) $g_{c,r}$ para o agrupamento de tarefas c, $c = \{t_1^c, t_2^c, ..., t_m^c\}$, no recurso r é definido como:

$$g_{c,r} = \min_{t_j \in S_r}(j)$$

tal que $(EST(t_{j+1}, r) - EFT(t_j, r)) \times margem\_seg > tamanho_{c,r}$ e $EST(t_{j+1}, r) - EST(t_1^c, r) > tamanho_{c,r}$.

Chamamos de *margem de segurança*, $margem\_seg$, um espaço livre na lacuna encontrada que não é preenchido, podendo assim absorver possíveis perdas de desempenho dos recursos. Dessa maneira, se o desempenho de um recurso é pior que o esperado, o

agrupamento inserido no espaço pode executar com pouca interferência nas tarefas previamente escalonadas naquele recurso. A margem de segurança é relativa ao tamanho do espaço encontrado. Por exemplo, se queremos uma margem de segurança de 10%, usamos $margem\_seg = 0,9$. Ainda, definimos o tamanho de um agrupamento c em S_r como a diferença entre o EFT da última tarefa do agrupamento e o EST da primeira tarefa do agrupamento em r, ou $tamanho_{c,r} = EFT(t_m^c, r) - EST(t_1^c, r)$.

Para evitar *deadlocks* uma verificação é feita quando o algoritmo de busca de espaços encontra um espaço para determinado agrupamento de tarefas. Antes de iniciar o escalonamento, o algoritmo gera, para cada tarefa do grafo, um conjunto de tarefas das quais essa tarefa depende. Esse conjunto das quais a tarefa t_i depende é composto por todas as tarefas em qualquer caminho de $t_{entrada}$ a t_i. Antes de atribuir um espaço a um agrupamento c, o algoritmo verifica se existe alguma tarefa adiante no espaço encontrado da qual t_m^c (a última tarefa do agrupamento) depende. Se existe tal tarefa, o espaço não pode ser atribuído ao agrupamento, e o algoritmo continua com a busca. A busca de espaços é mostrada no Algoritmo 13.

Algoritmo 13 busca_espaço(S_r, c)

1: $tamanho_{c,r} \Leftarrow EFT(t_m^c, r) - EST(t_1^c, r)$
2: $k \Leftarrow$ número de tarefas em S_r
3: $i \Leftarrow 1$
4: **if** $(EST(t_1, r) \times margem\_seg) > tamanho_{c,r}$ **then**
5: Computa ESTs e EFTs para $t_j^c \in c$ no espaço atual
6: $D_{t_m^c} \Leftarrow$ tarefas adiante no escalonamento das quais t_m^c depende
7: **if** $(EST(t_1, r) - EST(t_1^c, r) > tamanho_{c,r})$ e $D_{t_m^c} = \emptyset$ **then**
8: $g_{c,r} = 0$; **retorna** $g_{c,r}$
9: **end if**
10: **end if**
11: **for** $i = 1$ to $k - 1$ **do**
12: **if** $((EST(t_{i+1}, r) - EFT(t_i, r)) \times margem\_seg) > tamanho_{c,r}$ **then**
13: Computa ESTs e EFTs no espaço atual $\forall t_j^c \in c$
14: $D_{t_m^c} \Leftarrow$ tarefas adiante no escalonamento das quais t_m^c depende
15: **if** $(EST(t_{i+1}, r) - EST(t_1^c, r) > tamanho_{c,r})$ e $D_{t_m^c} = \emptyset$ **then**
16: $g_{c,r} \Leftarrow i$;
17: **retorna** $g_{c,r}$
18: **end if**
19: **end if**
20: **end for**
21: $g_{c,r} \Leftarrow k$;
22: **retorna** $g_{c,r}$ *//nenhum espaço encontrado*

O algoritmo de busca por espaços determina primeiro o tamanho do agrupamento c sendo escalonado e o número de tarefas na fila de r (linhas 1 e 2). Então, ele verifica se a primeira tarefa no recurso tem $EST > 0$ e se c é menor que essa lacuna no começo do escalonamento, considerando a margem de segurança (linha 4). Se c encaixa-se nesse espaço, o algoritmo verifica se há espaço descontando seu EST, pois não queremos interferir na execução de tarefas que já estão na fila, e verifica se há um *deadlock* (linha 6). Se o agrupamento não couber no começo do escalonamento ou se existir *deadlock*, o algoritmo inicia a iteração sobre as tarefas no escalonamento (linha 11). O algoritmo busca por lacunas entre a tarefa na posição atual, t_i, e a tarefa seguinte, t_{i+1} (linhas 12 a 18). Se não forem encontrados espaços, a última posição do escalonamento no recurso é retornada (linha 20).

O algoritmo de seleção de recursos com busca de espaços é mostrado no Algoritmo 14. Para todos os recursos, o algoritmo procura por lacunas e insere o *cluster* c na posição retornada pela busca de espaços (linhas 2 e 3). A inserção é feita após a tarefa que está na posição retornada, iniciando na posição 1. Na linha 5 o algoritmo computa o EST do sucessor de c. Finalmente, o algoritmo retorna o recurso com o menor EST para o sucessor de c (linha 7).

Algoritmo 14 seleciona_melhor_recurso(c)

1: **for all** $r \in \mathcal{R}$ **do**
2: $g_{c,r} \Leftarrow$ busca_espaço(S_r, c)
3: $escalonamento \Leftarrow$ Insere c em S_r na posição $g_{c,r}$
4: calcula_EFT(t^c_m);
5: $tempo_r \Leftarrow$ calcula_EST(sucessor(t^c_m))
6: **end for**
7: **retorna** recurso r com o menor $tempo_r$

Um exemplo de escalonamento gerado é mostrado na Figura 5.21. Dois processos, $P0 = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}\}$ e $P1 = \{t_A, t_B, t_C, t_D, t_E, t_F\}$, são escalonados com a busca de espaços considerando que os recursos $R0$ e $R1$ têm a mesma capacidade de processamento. O primeiro agrupamento de $P0$ a ser escalonado é $cls_0 = \{t_1, t_2, t_3, t_4, t_7\}$. Então $cls_1 = \{t_6, t_8\}$ é escalonado e, finalmente, $cls_2 = \{t_5, t_9, t_{10}\}$ é escalonado. Após escalonar $P0$, $P1$ é escalonado, começando com $cls_0 = \{t_A, t_B, t_D\}$ na lacuna encontrada antes de t_6. Em seguida $cls_1 = \{t_C, t_E, t_F\}$ é escalonado no espaço entre t_5 e t_9.

No exemplo, $EST(t_C, R0) = 46$ e $EFT(t_F, R0) = 65$. Também, $EFT(t_5, R0) = 46$ e $EST(t_9, R0) = 70$. As tarefas t_C, t_E e t_F podem ser escalonadas no espaço entre t_5 e t_9 considerando uma margem de segurança de no máximo 20% ou, mais precisamente, $margem\_seg > \frac{19}{24}$, pois $tamanho(\{t_C, t_E, t_F\}, R0) = 19$ e $EST(t_9, R0) - EFT(t_5, R0) = 24$.

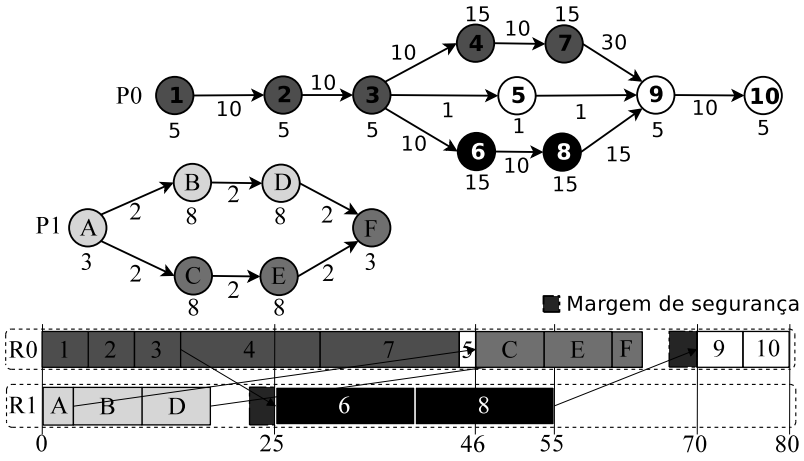


Figura 5.21: Dois DAGs e o escalonamento resultante com a busca de espaços.

5.4.4 Algoritmo de Intercalação

Nesta seção, propomos um algoritmo que intercala processos com o objetivo de minimizar o tempo de execução dos *workflows* [13]. A intercalação de processos pode ser abordada de forma a priorizar ou não processos que já estão escalonados. Essa priorização pode ser feita considerando características dos processos, tais como pesos das tarefas e custos de comunicação, tamanho do caminho crítico dos processos, quantidade de *clusters* de tarefas de cada processo, e assim por diante.

No cenário de escalonamento de múltiplos processos, à medida que processos chegam para serem escalonados, o passo de seleção de tarefas pode considerar as tarefas não executadas de processos já escalonados e dos processos que chegam. Para realizar essa seleção nosso algoritmo considera que o primeiro processo a chegar, esteja ele já escalonado ou não, será o primeiro processo que terá tarefas selecionadas para escalonamento. Em seguida, o segundo processo terá tarefas selecionadas para escalonamento. Em suma, o algoritmo seleciona um grupo de tarefas de cada processo em ordem de chegada, escalonando cada grupo de tarefas selecionado, até que acabem os processos e tarefas a serem escalonadas. Durante o escalonamento de *clusters* intercalados, o algoritmo também realiza o preenchimento de espaços, buscando o melhor uso dos recursos disponíveis. Com essa intercalação de processos, os espaços provenientes das transmissões de dados são utilizados para processamento de outros DAGs. O Algoritmo 15, que é executado no evento da chegada de um DAG $\mathcal{G}$ para escalonamento, mostra em mais detalhes este procedimento.

Na primeira linha, o Algoritmo 15 coleta os *clusters* que ainda não foram enviados para execução, enquanto na linha 2 cria o grupo *DAGs*, que contém os processos aos quais tais *clusters* pertencem. O processo que chegou para ser escalonado é adicionado

Algoritmo 15 Visão geral da intercalação

1: $CLS_{fila} \Leftarrow$ Clusters não executados de processos escalonados.
2: $DAGs \Leftarrow$ Processos que têm tarefas em CLS_{fila}.
3: $DAGs \Leftarrow \{DAGs\} \cup \{\mathcal{G}\}$.
4: Atribui prioridades a cada grafo de $DAGs$ de acordo com a política de prioridades.
5: **while** existem tarefas não escalonadas **do**
6: $G \Leftarrow$ próximo grafo com maior prioridade.
7: Escalona N *clusters* de $\mathcal{G}$ com procura de espaços, sendo N de acordo com a política de prioridades.
8: **end while**
9: Continua execução da fila de tarefas utilizando o algoritmo adaptativo dinâmico

ao grupo $DAGs$ na linha 3. Na linha 4 uma política de prioridade pré-definida atribui a cada processo um nível de prioridade que será usado para selecionar os processos e decidir quantos *clusters* de cada processo serão escalonados. A iteração entre as linhas 5 e 8 realiza o escalonamento dos processos, selecionando na linha 6 os grafos em seqüência de prioridade e na linha 7 realizando o escalonamento dos *clusters*, colocando cada *cluster* no recurso que proporciona menor EST para o sucessor de sua última tarefa.

A política de prioridades altera a ordem de escalonamento dos processos e o número de *clusters* de cada processo a serem escalonados em cada iteração do algoritmo. O valor N, utilizado na linha 7, representa o número de *clusters* do processo selecionado a serem escalonados. Esse valor é definido pela política de prioridades, com cada grafo tendo seu próprio N. Ainda, o valor de N pode ser variável para um mesmo grafo. Por exemplo, um grafo que já executou grande parte de suas tarefas pode ter sua prioridade aumentada pela política de prioridades. Ainda, a política de prioridades pode definir se um processo já escalonado pode ser reescalonado na chegada de novos processos, ou a quantidade máxima de processos que podem ser re-escalonados no evento da chegada de um novo processo.

Nos experimentos realizados neste trabalho, utilizamos prioridade FCFS (*First Come, First Served*), com $N = 1$ independente da prioridade atribuída a cada grafo. Assim, o escalonamento dos *clusters* é realizado de forma circular, começando no primeiro grafo que chegou e terminando quando não há mais *clusters* não escalonados. Note que, dessa forma, quanto maior um *cluster*, maior é o número de tarefas do grafo ao qual este *cluster* pertence que serão escalonadas. Dessa forma, a comunicação entre tais tarefas será suprimida, enquanto *clusters* curtos serão intercalados com *clusters* de outros processos, tendo seu tempo de comunicação utilizado para processamento.

A Figura 5.22 mostra um exemplo do resultado do escalonamento de três processos com o algoritmo de intercalação, onde $P_0 = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$, $P_1 = \{A, B, C, D, E, F\}$, e $P_2 = \{a, b, c, d, e, f\}$, com P_2 sendo outra instância do processo P_1. Para tornar o exem-

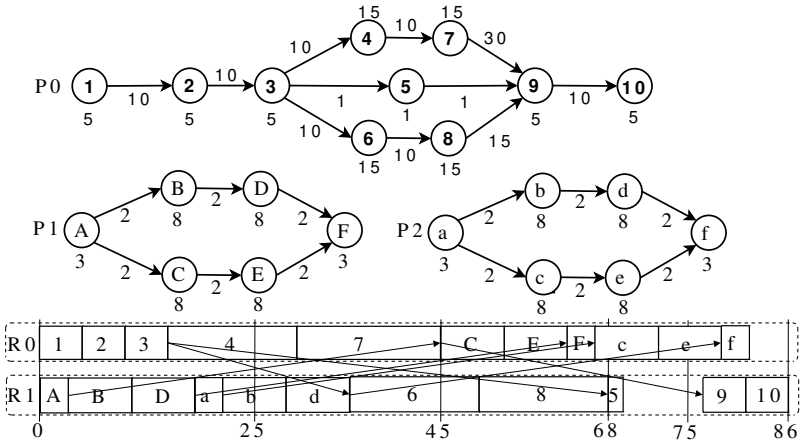


Figura 5.22: Exemplo de escalonamento de três DAGs utilizando intercalação.

plo simples, assumimos que os dois recursos possuem a mesma capacidade de processamento. O primeiro *cluster* a ser escalonado é o *cluster* $cls_{1,P0} = \{1, 2, 3, 4, 7\}$. Em seguida, os *clusters* $cls_{1,P1} = \{A, B, D\}$ e $cls_{1,P2} = \{a, b, d\}$ são escalonados. Então, retornando ao primeiro processo, o *cluster* $cls_{2,P0} = \{6, 8\}$ é escalonado. Continuando a iteração entre os processos, os *clusters* $cls_{2,P1} = \{C, E, F\}$ e $cls_{2,P2} = \{c, e, f\}$ são escalonados. Finalizando o processo, o *cluster* $cls_{3,P0} = \{5, 9, 10\}$ é escalonado. Note que há apenas um espaço entre as tarefas dos três processos, tornando baixa a taxa de ociosidade dos recursos.

Enquanto o algoritmo de preenchimento de espaço não interfere no escalonamento de processos já escalonados, a intercalação permite mecanismos de priorização de processos de acordo com políticas específicas. Por exemplo, um processo P_h que tem dependências de dados muito custosas pode ter um grupo escalonado primeiro, e outros processos com menor comunicação podem ser intercalados visando a utilização do processamento que estaria ocioso entre as transmissões de dados do processo P_h.

5.4.5 Reescalonamento de tarefas

Quando um recurso não se comporta como esperado, o algoritmo dinâmico permite mudanças no escalonamento corrente para evitar o envio de tarefas para tal recurso. Para fazer isso, o algoritmo reescalona tarefas não executadas que estão na fila do recurso. Quando há apenas um processo escalonado em um conjunto de recursos, o reescalonamento pode ser feito através da realocação das tarefas originalmente no recurso com mau desempenho. Nesse caso, por haver apenas tarefas de um processo em todos os recursos, a seleção do novo recurso demanda que sejam verificadas apenas as precedências entre tare-

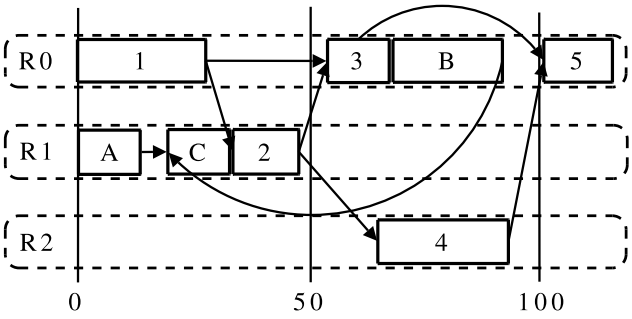


Figura 5.23: Exemplo de *deadlock* entre processos após reescalonamento.

fas de um único DAG. Quando mais de um processo compartilha o mesmo conjunto de recursos concomitantemente, o reescalonamento se torna mais complicado. Ao realocar as tarefas, há a possibilidade de ocorrência de *deadlocks* entre tarefas de processos diferentes, o que chamamos de *deadlock* interprocessos. Um exemplo dessa situação é mostrada na Figura 5.23, onde existem dois processos: $P1 = \{t_1, t_2, t_3, t_4, t_5\}$ e $P2 = \{t_A, t_B, t_C\}$. Essa situação ocorreria se, por exemplo, a tarefa t_B fosse reescalonada de um recurso com mau desempenho na lacuna entre as tarefas t_3 e t_5.

Ainda que algoritmos complexos possam ser desenvolvidos para tratar esse problema, utilizamos um algoritmo direto que o trata de maneira simples. Se o algoritmo dinâmico constatar que o desempenho em certo recurso foi aquém do esperado ao fim de um turno, o algoritmo procura por um novo recurso para cada agrupamento. Mas se um agrupamento é reescalonado no fim do escalonamento corrente de um recurso, *deadlocks* também podem ocorrer. Para evitar *deadlocks*, o algoritmo apresentado ordena todas as tarefas no escalonamento por seus ESTs quando são reescalonadas. Dessa forma, uma tarefa nunca será escalonada antes de um de seus predecessores ou após algum de seus sucessores. Quando as tarefas são ordenadas por EST, o escalonador tenta manter o agrupamento reescalonado iniciando em um tempo próximo ao EST no recurso de origem, e ao mesmo tempo tenta minimizar o tempo de término do agrupamento. Uma visão geral do reescalonamento é mostrada no Algoritmo 16.

5.4.6 Resultados Experimentais

Nesta seção, realizamos uma avaliação gradual dos quatro algoritmos apresentados, incrementando as comparações com diferentes configurações. Iniciamos com a comparação entre o algoritmo de busca de espaços e o algoritmo seqüencial, ou seja, sem a busca de espaços. Em seguida, inserimos também o algoritmo de intercalação de DAGs nos resultados, avaliando seu desempenho em relação à busca de espaços e ao algoritmo seqüencial. Completando as comparações, inserimos o algoritmo de agrupamento de DAGs nas sim-

Algoritmo 16 reescalona(S_r)

```
1: C_res ⇐ clusters de r a serem reescalonados
2: for all c ∈ C_res do
3:    for all r ∈ R do
4:       escalonamento ⇐ Insere c em S_r
5:       Ordena S_r por EST
6:       calcula_EFT(t^c_m);
7:       tempo_r ⇐ calcula_EST(sucessor(t^c_m))
8:    end for
9: end for
10: retorna recurso r com o menor tempo_r
```

ulações, realizando uma comparação global entre os quatro algoritmos.

As simulações apresentam resultados para o escalonamento de até 10 *workflows*, de P_0 a P_9, em um ambiente de grades considerando variações no número de recursos e carga externa à grade executando concomitantemente com as tarefas dos DAGs. Assim, analisamos o desempenho dos algoritmos com variações tanto no número de *workflows* quanto no número de recursos, que são variáveis e que potencialmente variam em um ambiente real de grade. Essas informações são relevantes no sentido que apresentam o comportamento dos algoritmos no que remete à dinamicidade e à escalabilidade. Os resultados são separados em *escalonamento inicial* (*makespan* logo após a finalização do escalonamento) e *execução* (*makespan* após a execução com carga externa). Adicionalmente, apresentamos uma breve simulação utilizando a heurística HEFT. O objetivo dessa simulação é verificar se há indícios de alterações sensíveis nas conclusões alcançadas se outra heurística é utilizada.

Utilizamos 16 DAGs nas simulações, sendo 15 gerados aleatoriamente mais o grafo da aplicação CSTEM. Os custos de computação das tarefas foram gerados aleatoriamente no intervalo $(500, 11000)$, enquanto os custos de comunicação foram gerados no intervalo $(500, 1100)$. Os resultados apresentam intervalos de confiança de 95%. O algoritmo de busca de espaços foi executado com $margem\_seg = 0,95$. Optamos por uma margem de segurança relativamente baixa, pois uma margem de segurança alta afeta o desempenho do algoritmo, gerando uma subutilização dos recursos e prejudicando o *makespan* final e a comparação do escalonamento inicial com outros algoritmos. Por outro lado, essa margem contribui para melhorar os resultados após a execução das tarefas. Os processos foram numerados em ordem de chegada, ou seja P_0 foi o primeiro processo recebido pelo escalonador, enquanto P_9 foi o último. Os recursos, arranjados em grupos, tiveram sua capacidade de processamento gerada no intervalo $(50, 200)$, enquanto as capacidades dos enlaces foram escolhidas aleatoriamente no intervalo $(40, 80)$ para pares de recursos dentro

do mesmo grupo, e no intervalo $(5, 40)$ para pares de recursos em grupos diferentes.

Escalonamento Inicial

Nesta seção, apresentamos resultados para o escalonamento inicial, que é o escalonamento entregue pelos algoritmos considerando os atributos computados utilizando as capacidades dos recursos e enlaces, assim como os custos de tarefas e dependências de dados recebidos pelo escalonador. O escalonamento inicial não considera a execução das tarefas dos *workflows*, mas apenas o *makespan* resultante do escalonamento dos DAGs.

Busca de Espaços Neste conjunto de resultados, avaliamos o *makespan* inicial dos processos escalonados utilizando o algoritmo de busca de espaços. A comparação utiliza dois DAGs (processo 0 e processo 1). O escalonamento com busca de espaços é comparado com o escalonamento seqüencial, onde o primeiro processo escalonado executa todas as suas tarefas em um recurso antes que o próximo processo inicie nesse mesmo recurso. Com isso, o segundo processo pode ser escalonado nos recursos livres ou em recursos já utilizados pelo primeiro processo, porém somente após as tarefas do processo 0.

A Figura 5.24 mostra o SLR e *speedup* médios para o processo 1 no escalonamento inicial (antes da execução) com e sem a busca de espaços para comunicação média e alta. Quanto menor o número de grupos, maior o SLR para ambos os algoritmos, pois há menos opções de recursos. Mas, quanto menor o número de grupos, maior o ganho da busca de espaços quando comparada ao algoritmo sem ela. Isso ocorre porque o processo 0 fica concentrado nos melhores recursos, deixando espaços em seu escalonamento. Então, quando o processo 1 é escalonado, há menos opções de recursos livres, o que faz dos espaços boas opções de escalonamento para os agrupamentos do processo 1. Por outro lado, com mais grupos, apesar do algoritmo com a busca de espaços continuar melhor, a diferença no SLR é menor que quando há menos grupos. Portanto, mesmo quando há muitas opções de recursos (número de grupos alto), a busca por espaços pode encontrar lacunas que valem a pena serem utilizadas no escalonamento do processo 1. Essa análise é também válida para os resultados de *speedup* com comunicação média, reforçando as conclusões. Note que o escalonamento inicial não interfere no processo 0, então seu *makespan* é o mesmo para ambos os algoritmos. Então, omitimos seus resultados para o escalonamento inicial.

Em um cenário de comunicação alta, os resultados de SLR mostram o mesmo comportamento que o SLR para comunicação média. As curvas são ligeiramente diferentes das curvas para comunicação média, pois as diferenças de SLR são maiores com poucos grupos quando comparadas às com comunicação média. Isso ocorre porque o escalonamento em poucos recursos resulta em muitas tarefas do mesmo processo no mesmo recurso quando se utiliza a busca de espaços, suprimindo os altos custos de comunicação. Por outro lado,

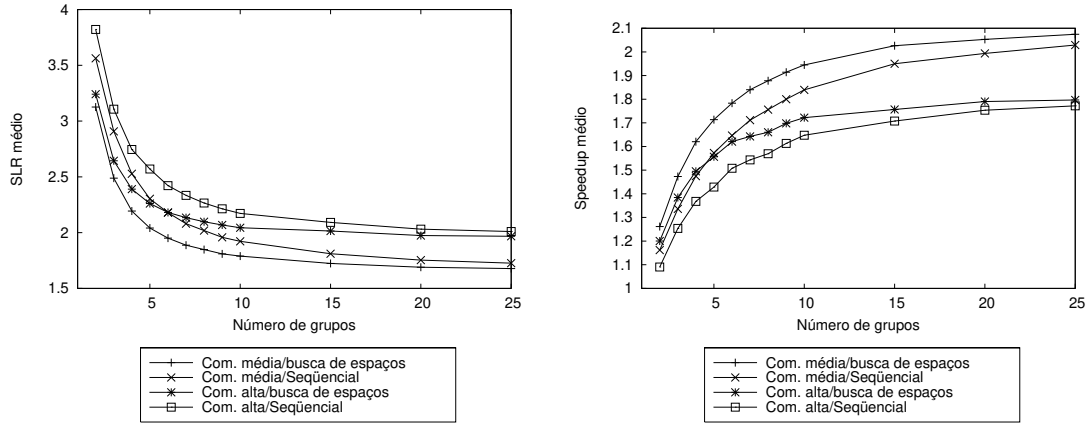


Figura 5.24: SLR e *speedup* médios para P1 no escalonamento inicial.

mais recursos permitem o espalhamento das tarefas, diminuindo o uso de espaços e aumentando os custos de comunicação, aproximando os resultados de ambos os algoritmos. Para os resultados de *speedup* com comunicação alta, podemos observar comportamento similar.

A Figura 5.25 compara as médias de *speedup* para o escalonamento inicial quando escalonamos 3 processos, chamados de $P0$, $P1$ e $P2$, com comunicação média. Naturalmente, quando existem três processos a serem escalonados, o *speedup* do processo $P2$ é menor que o do processo $P1$. Porém, note que o ganho com a busca de espaços é maior para $P2$ que para $P1$. Também observamos comportamento similar para o SLR nesse cenário.

Intercalação Comparamos o *makespan* inicial do algoritmo de intercalação com o algoritmo de escalonamento seqüencial (sem busca de espaços) e com o algoritmo de busca e preenchimento de espaços. A comparação foi feita escalonando três processos.

A Figura 5.26 mostra o SLR e *speedup* médios dos três processos para comunicação alta. Com 2 grupos, o ganho do algoritmo de busca de espaços em relação ao algoritmo seqüencial é de cerca de 16%. Quando aumentamos o número de grupos esse ganho diminui, pois quanto mais grupos, mais recursos temos para escolher. Dessa forma, recursos sem tarefas escalonadas são utilizados em detrimento da utilização de espaços encontrados no escalonamento em outros recursos. Quando utilizamos o algoritmo de intercalação, há uma melhora em relação à utilização apenas da busca de espaços. Nos resultados de *speedup*, notamos que quando há apenas 2 grupos de recursos, a intercalação de processos gera resultados equivalentes àqueles gerados pelo algoritmo seqüencial, enquanto o algoritmo com busca de espaços gera resultados melhores. Porém, para 3 grupos o algoritmo

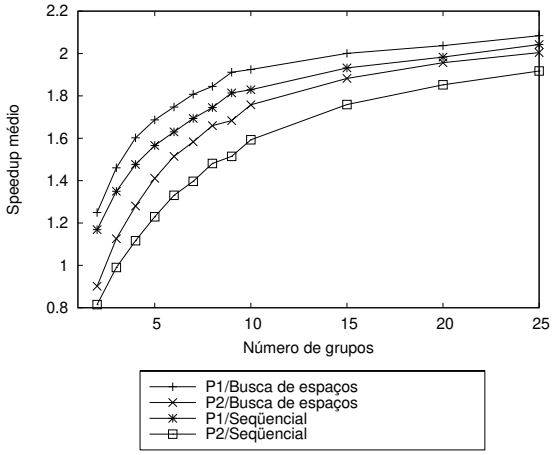


Figura 5.25: *Speedup* médio para P1 e P2 no escalonamento inicial.

de intercalação gera resultados equivalentes ao algoritmo de busca de espaços, sendo ambos melhores que o algoritmo sem busca de espaços. Isso se deve ao fato de que, tendo poucos recursos, há menos espaços a serem preenchidos, pois tarefas no mesmo recurso têm custo de comunicação igual a zero. Assim, os poucos espaços gerados são preenchidos pelo algoritmo de busca de espaços, enquanto o algoritmo de intercalação acaba gerando escalonamentos com *makespan* próximo àqueles gerados pelo algoritmo sem a busca de espaços, não havendo ganho na utilização de tempo de comunicação para processamento de tarefas de outros processos. Considerando entre 5 e 15 grupos, o algoritmo de intercalação é um pouco melhor que os outros dois algoritmos, conseguindo aproveitar mais tempo de comunicação para executar tarefas de outros processos do que o algoritmo que apenas busca espaços.

Os gráficos de SLR e *speedup* para comunicação média são mostrados na Figura 5.27, respectivamente. As curvas sem busca e com busca têm distância e comportamento semelhantes aos observados no gráfico para comunicação alta. A curva do algoritmo de intercalação neste cenário distanciou-se da curva do algoritmo com busca de espaços, pois ao termos tarefas e espaços de comunicação gerados aleatoriamente no mesmo intervalo, o algoritmo de busca de espaços encontra menos espaços onde pode encaixar tarefas quando comparado à comunicação alta. Assim, o algoritmo de intercalação consegue melhores resultados, utilizando de forma mais eficaz o tempo de comunicação para processamento de tarefas de outros processos.

No cenário de comunicação baixa, os resultados de SLR e *speedup* (Figura 5.28) mostram uma melhora do algoritmo de intercalação quando comparado ao algoritmo de busca de espaços. A mesma justificativa do cenário de comunicação média é válida: como os tempos de transmissão de dados são menores, sobram menos espaços para en-

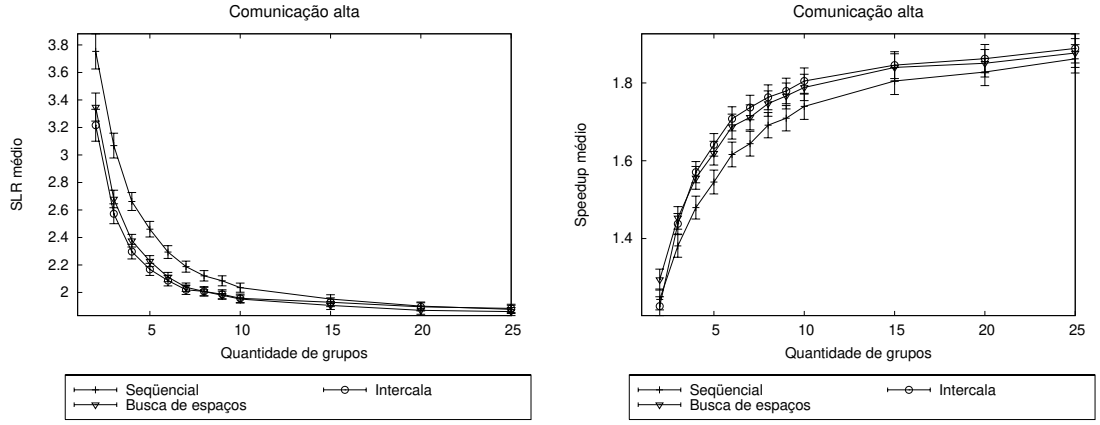


Figura 5.26: SLR e *speedup* médios para comunicação alta.

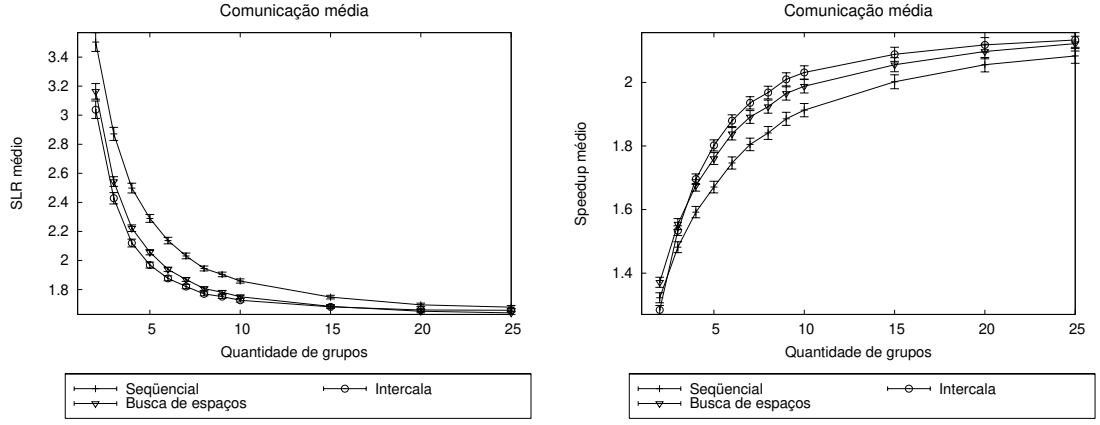


Figura 5.27: SLR e *Speedup* médios para comunicação média.

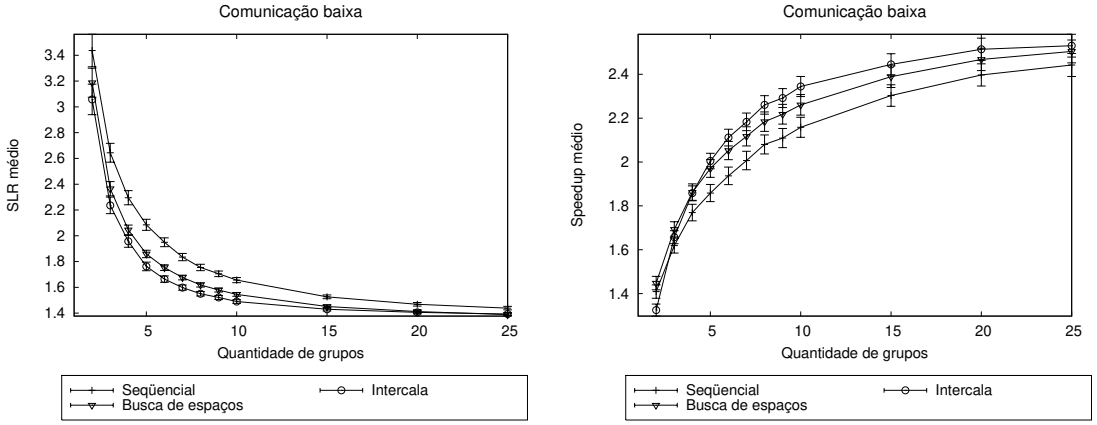


Figura 5.28: *Speedup* médio para comunicação baixa.

caixar tarefas, tornando o algoritmo de busca de espaços menos eficaz. Por outro lado, o algoritmo de intercalação continua conseguindo utilizar os tempos de comunicação para computar tarefas de outros processos, minimizando as médias de SLR e *speedup*.

Agrupamento de DAGs Neste conjunto de simulações incluímos o algoritmo de agrupamento de DAGs, apresentando, portanto, resultados do escalonamento inicial incluindo os quatro algoritmos.

A Figura 5.29 mostra o *makespan* médio, $average_{M_N}$, para os N primeiros processos, com N variando de 1 a 10 para 2 e 10 grupos de recursos.

Para 2 grupos o algoritmo de busca de espaços resultou em um *makespan* médio mais baixo no escalonamento inicial quando consideramos N de 2 a 10. O algoritmo de intercalação tem melhor desempenho que o algoritmo de agrupamento para todos os valores de N testados, além de aproximar os resultados da busca de espaços com o aumento do número de DAGs considerados. Podemos observar que os *makespans* dos algoritmos seqüencial e de busca de espaços são tão maiores quanto mais tarde os processos foram escalonados, com o algoritmo de busca de espaços tendo melhores resultados que o seqüencial. Por outro lado, os algoritmos de intercalação e agrupamento resultam em *makespans* mais balanceados, o que sugere um comportamento mais justo.

Para 10 grupos o algoritmo de intercalação resulta em *makespans* médios menores que o algoritmo de busca de espaços se consideramos 7 ou mais processos, com o algoritmo de intercalação tendo melhor desempenho também que a abordagem de agrupamento de DAGs.

A tendência continua para 25 grupos (Figura 5.30). O algoritmo de intercalação começa a apresentar melhores *makespans* quando consideramos mais de 4 processos. Pode-

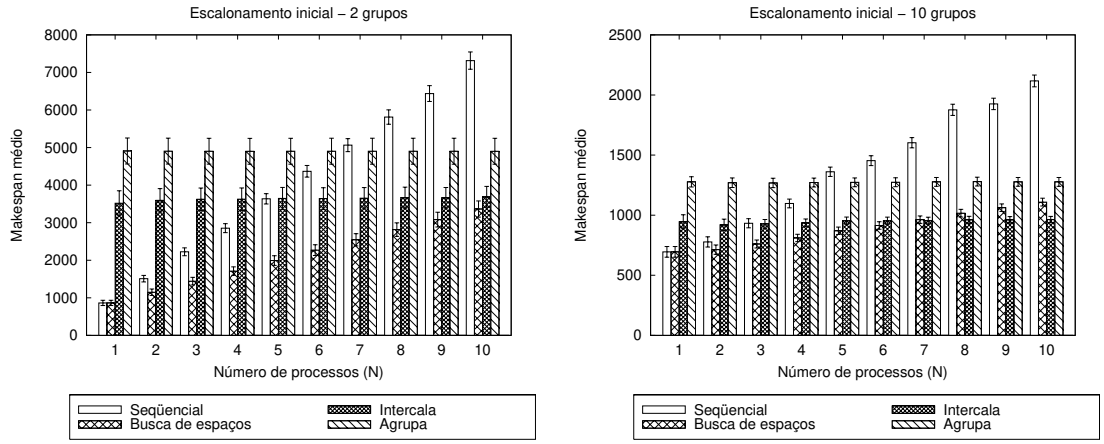


Figura 5.29: *Makespan* médio ($averageM_N$) de acordo com o número de processos no escalonamento inicial com 2 e 10 grupos de recursos.

mos observar que a diferença entre os resultados dos algoritmos é menor, pois há mais recursos para espalhar as tarefas dos processos, e os algoritmos seqüencial e de busca de espaços alcançam *makespans* mais próximos àqueles apresentados pelo algoritmo de intercalação.

A Figura 5.30 também mostra o *makespan* global médio (a média entre o *makespan* máximo entre todos os processos) para 2 a 25 grupos. Podemos observar que a intercalação e o agrupamento de DAGs resultam em escalonamentos com *makespans* globais menores que os apresentados pelos algoritmos seqüencial e de busca de espaços. As abordagens de agrupamento e intercalação resultam em *makespans* globais equivalentes.

Execução dos Processos

Busca de Espaços Após a avaliação do escalonamento inicial, analisamos agora como esse escalonamento inicial se comporta quando as tarefas são executadas em um ambiente com variação de desempenho. Dessa forma, podemos analisar como os escalonamentos iniciais de cada algoritmo se comportam em um ambiente dinâmico. Sem reescalonamento significa que todas as tarefas foram executadas nos recursos dados pelo escalonamento inicial, não importando seu desempenho. Com reescalonamento significa que as tarefas foram reescalonadas utilizando o Algoritmo 16 quando determinado pela abordagem dinâmica adaptativa. O cálculo do tamanho do turno inicial em um recurso r_i considerando múltiplos DAGs leva em conta a média sobre todas as tarefas escalonadas em r_i. Neste primeiro conjunto de resultados, a carga externa utilizada foi a mostrada no Apêndice A. Os resultados apresentam a média sobre ambos os processos com comu-

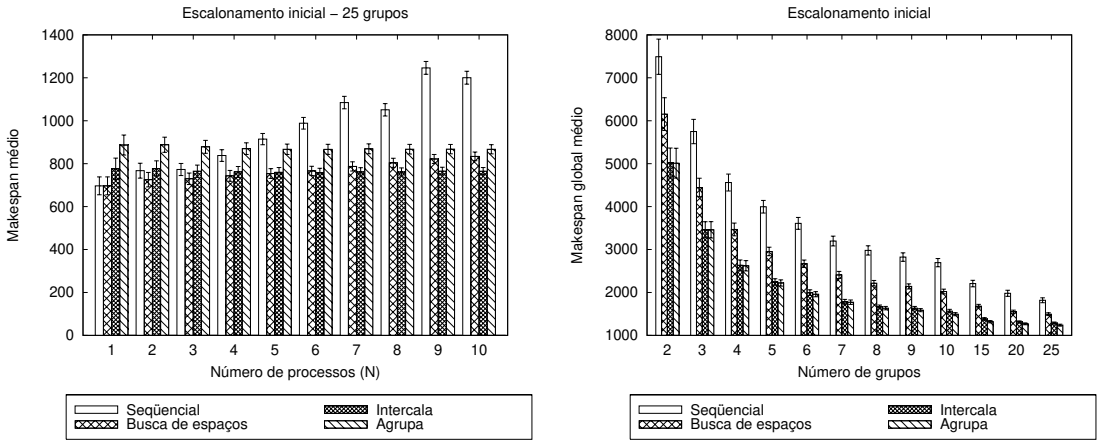


Figura 5.30: *Makespan* médio ($averageM_N$) para 25 grupos de recursos e *Makespan* global de todos os processos no escalonamento inicial.

nicação alta.

A Figura 5.31 mostra os resultados de SLR e *speedups* para a execução dos processos 0 e 1 com e sem a busca de espaços, e com e sem reescalonamento.

Para a métrica de SLR, as execuções sem reescalonamento apresentam resultados similares. Isso significa que em um ambiente com variação de desempenho a busca de espaços pode se tornar inepta se não existir uma política de reescalonamento. Isso ocorre porque todo o ganho com o uso dos espaços é perdido quando o desempenho dos recursos cai. Essa perda de desempenho funciona como uma bola de neve quando não existem espaços no escalonamento, afetando grande parte das tarefas de ambos os processos. Por outro lado, quando há lacunas no escalonamento, existe espaço para absorver os atrasos na execução das tarefas e esses atrasos não afetam os processos da mesma maneira que quando não existem espaços no escalonamento. Quando o algoritmo de reescalonamento é usado, há uma visível melhora no escalonamento.

Comparando o reescalonamento com e sem a busca de espaços podemos notar que a busca por espaços pode trazer melhorias nos resultados quando há poucos grupos. Quando há um número maior de grupos (mais que 10), os resultados de reescalonamento com e sem a busca de espaços são equivalentes. Como utilizamos o mesmo algoritmo no reescalonamento nos dois casos, sem utilizar busca por espaços durante essa etapa, os efeitos iniciais dos espaços encontrados são minimizados. Dessa forma, o algoritmo de busca de espaços, quando utilizado no escalonamento inicial, pode ser considerado menos eficiente quando há uma perda de desempenho não desprezível nos recursos computacionais utilizados com o reescalonamento proposto.

A análise anterior também é válida para os resultados de *speedup* mostrados na Figura

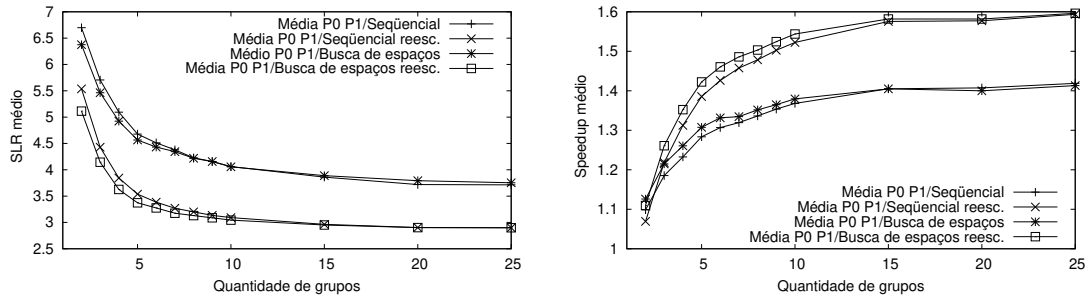


Figura 5.31: SLR e *Speedup* médios de P0 e P1 com comunicação alta após execução.

5.31, exceto para os resultados com poucos grupos. Nesse caso, devido ao fato do algoritmo não ter boas opções para fazer o reescalonamento, a ordenação de tarefas nos recursos disponíveis parece piorar o escalonamento. Entretanto, esse problema não se repete quando o número de grupos é maior que 4.

Intercalação Neste conjunto de resultados apresentamos as simulações de execução incluindo o algoritmo de intercalação. Os desempenhos dos recursos foram gerados da mesma maneira que nos resultados apresentados no conjunto anterior.

A Figura 5.32 mostra os resultados da simulação de execução com comunicação média. Podemos notar um comportamento semelhante ao encontrado nos escalonamentos iniciais: o algoritmo de intercalação consegue melhorar a média de SLR e *speedup*. As curvas do gráfico se comportam da mesma maneira que aquelas apresentadas no escalonamento inicial, sugerindo que a variação de desempenho dos recursos não afeta o ganho obtido pelo algoritmo de intercalação no escalonamento inicial.

A Figura 5.33, que mostra resultados das execuções com comunicação baixa, também segue o padrão apresentado pelas figuras que mostram resultados do escalonamento inicial, reforçando o resultado obtido para comunicação média.

Com as simulações apresentadas até aqui, comparamos os três algoritmos tanto no escalonamento inicial como na execução em ambiente compartilhado. Os resultados sugerem que o algoritmo de intercalação melhora em média os resultados obtidos com o algoritmo de preenchimento de espaços, levando a uma execução mais rápida dos múltiplos DAGs escalonados.

Agrupamento de DAGs Nas simulações de execução comparando todos os algoritmos, utilizamos uma carga externa baseada em um modelo mais geral. Como carga externa nesse conjunto de simulações utilizamos o modelo de chegada de tarefas independentes da grade que segue uma distribuição de Poisson, com cada tarefa tendo um tempo

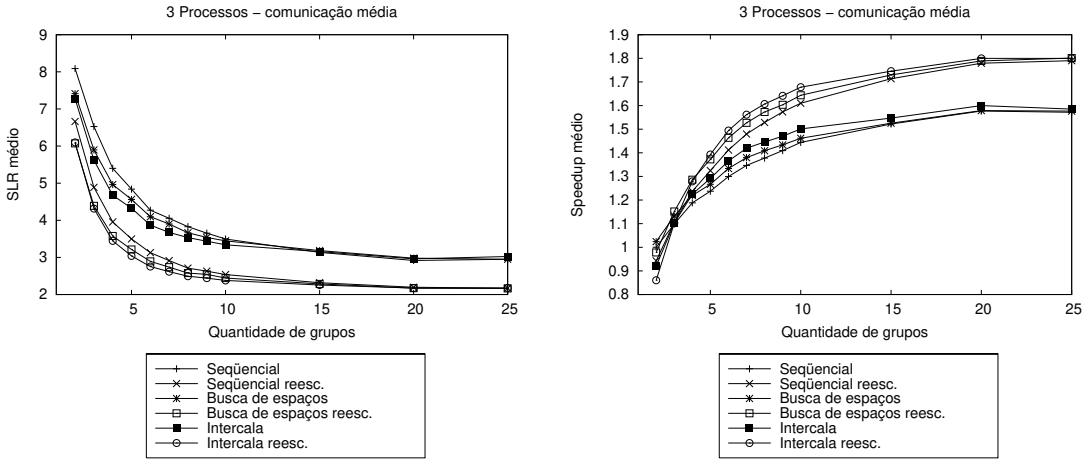


Figura 5.32: SLR e *Speedup* médios para execuções com comunicação média.

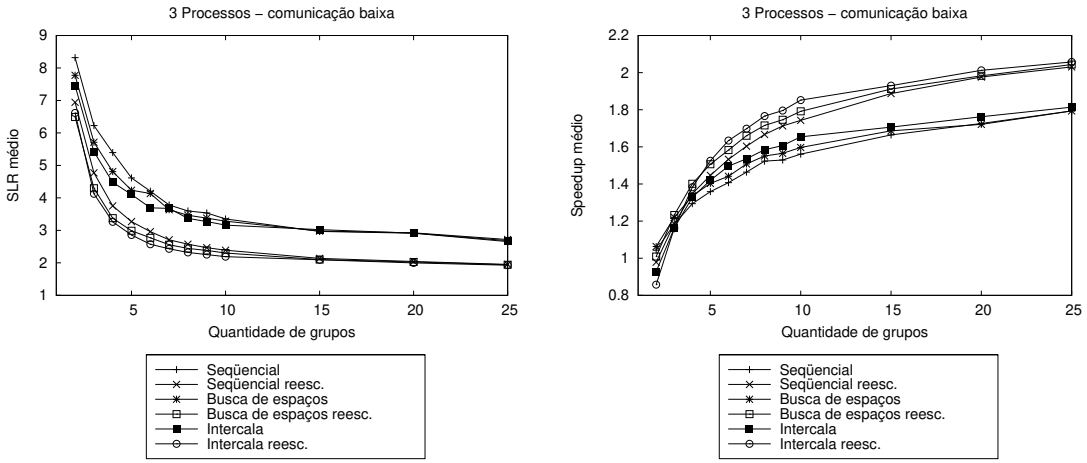


Figura 5.33: SLR e *Speedup* médios para execuções com comunicação baixa.

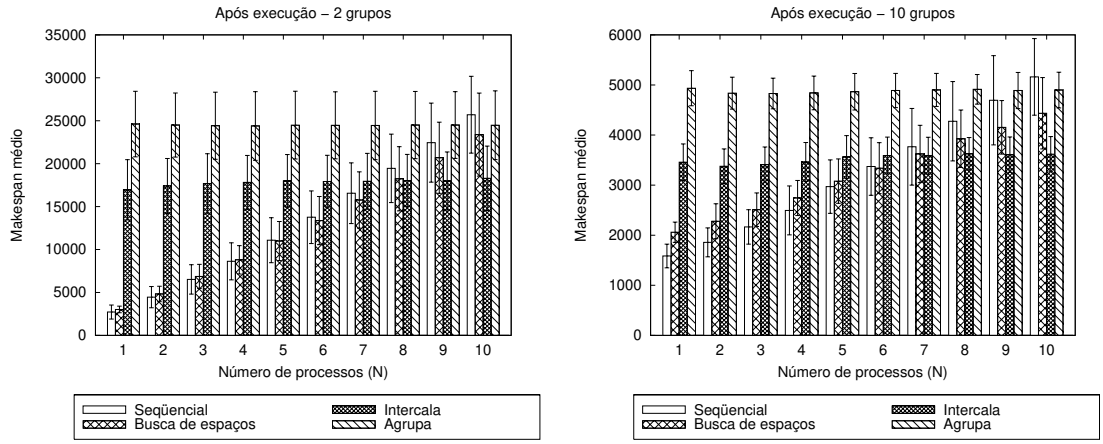


Figura 5.34: *Makespan* médio ($averageM_N$) de acordo com o número de processos após execução com 2 e 10 grupos de recursos.

de vida de $2/x$ [2], onde x é um número aleatório entre 0 e 1. Essa suposição segue o comportamento observado em aplicações reais [57].

As Figuras 5.34 e 5.35 mostram os tempos de execução finais após a simulação com carga independente da grade. Em geral, podemos observar o mesmo padrão encontrado nos resultados de escalonamento inicial.

Notamos um desempenho melhor do algoritmo seqüencial em relação aos outros quando só consideramos os processos iniciais. Justificamos esse comportamento pela pouca interferência dos processos que chegam depois daqueles processos que já estão escalonados, com as perdas de desempenho dos recursos podendo ser absorvidas pelos espaços deixados no escalonamento. Por exemplo, quando comparamos o algoritmo seqüencial com o algoritmo de busca de espaços após a execução com 2 grupos de recursos (Figura 5.34), há uma diferença nos resultados se comparados aos resultados de escalonamento inicial (Figura 5.29). Após a execução, o desempenho da busca de espaços fica próximo ao desempenho do algoritmo seqüencial. Isso se deve ao atraso das tarefas que preencheram os espaços de tais processos, que finalizam após o tempo estimado de término, atrasando outras tarefas. Esse comportamento sugere que em uma situação de carga externa alta com poucos recursos disponíveis, o algoritmo de busca de espaços pode não ser vantajoso, pois as tarefas podem não conseguir fazer uso eficiente das lacunas encontradas.

O mesmo raciocínio pode ser aplicado aos outros algoritmos: não havendo espaço livre no escalonamento, o atraso de algumas tarefas reflete em outras tarefas por não haver espaço para esse atraso ser dissipado. Entretanto, o algoritmo de intercalação continua resultando em *makespans* médios menores que os outros algoritmos quando múltiplos DAGs são executados, incluindo *makespans* médios menores que o algoritmo de busca de

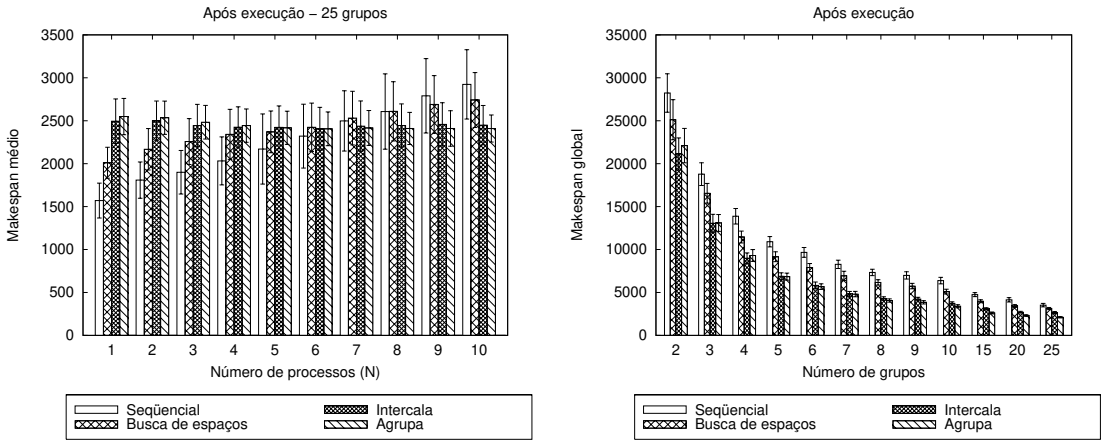


Figura 5.35: *Makespan* médio ($averageM_N$) para 25 grupos e *Makespan* global de todos os processos após execução.

espaços para 8 ou mais processos.

Quando há mais grupos de recursos (10 e 25, nas Figuras 5.34 e 5.35), o algoritmo seqüencial continua tendo um aumento no *makespan* menor que os outros algoritmos quando comparado com o escalonamento inicial. Também podemos notar que o algoritmo de intercalação não mantém seus melhores escalonamentos quando há mais de 4 DAGs. Isso pode ser explicado pelo fato da intercalação produzir escalonamentos que utilizam os melhores recursos mais intensamente, executando mais tarefas nesses recursos e deixando menos espaços livres no escalonamento que outras abordagens. Assim, os atrasos se espalham pelo escalonamento, enquanto nos outros algoritmos as tarefas são distribuídas em mais recursos e atrasos têm menos impacto nas tarefas seguintes.

A Figura 5.35 mostra também o *makespan* global após a execução dos *workflows*. Podemos observar o mesmo padrão apresentado antes da execução dos processos, sem nenhum algoritmo apresentar discrepâncias ou comportamento fora do esperado. Ainda que existam diferenças em termos de valores de *makespan* absolutos, o que é inevitável com uma carga externa compartilhando tempo de CPU, em termos relativos os resultados de execução mostram poucas variações em relação ao escalonamento inicial. Isso sugere que tanto o PCH quanto as estratégias apresentadas não são significativamente afetadas pela carga externa, sendo aptas a manter escalonamentos tão bons quanto eram antes da execução das tarefas, em termos relativos.

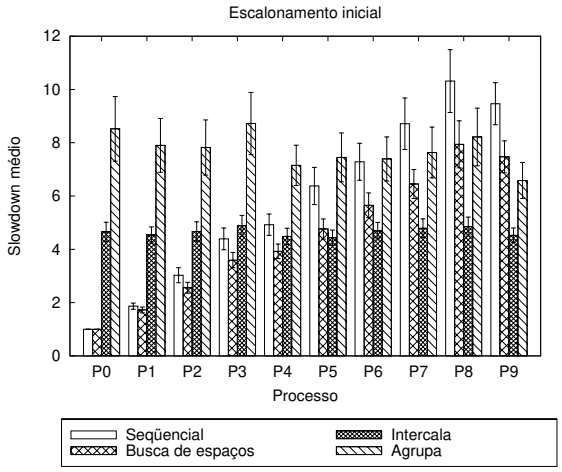


Figura 5.36: *Slowdown* de cada processo no escalonamento inicial com 2 grupos de recursos.

Justiça

Nesta seção, avaliamos como os algoritmos para escalonamento de múltiplos DAGs se comportam em relação à justiça. Utilizamos a métrica de *slowdown* para verificar quais algoritmos distribuem as tarefas nos recursos de modo que os *workflows* usem a grade compartilhando os recursos sem beneficiar algum DAG. Primeiramente apresentamos os resultados para o escalonamento inicial.

Justiça no escalonamento inicial Os resultados de *slowdown* para cada processo com 2, 10 e 25 grupos são mostrados respectivamente nas Figuras 5.36, 5.37, e 5.38.

Para 2 grupos de recursos, existem variações consideráveis no *slowdown* de processos diferentes, exceto na abordagem de intercalação, que mantém o *slowdown* de todos os processos no mesmo nível. Isso significa que o algoritmo de intercalação resulta em escalonamentos mais justos para a métrica considerada.

O mesmo padrão pode ser observado quando há 10 grupos de recursos. A intercalação de DAGs escalona todos os processos de forma que eles tenham valores de *slowdown* similares. Nesse cenário, a abordagem de grupos também resulta em *slowdowns* próximos, mas com valores mais altos que os dados pela intercalação de processos.

Assim como para 2 e 10 grupos, com 25 grupos o algoritmo de intercalação resulta em *slowdowns* uniformes para todos os processos, permanecendo com valores baixos, próximos a $1,0$. O algoritmo de agrupamento também apresenta *slowdowns* uniformes, porém novamente mais altos que os dados pela abordagem de intercalação.

As abordagens de agrupamento e intercalação apresentam *makespans* globais equiva-

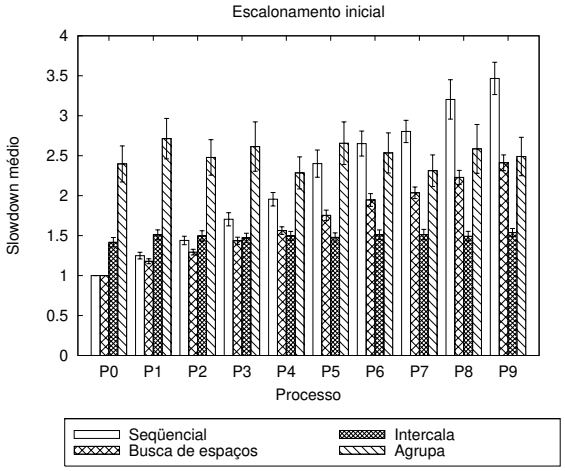


Figura 5.37: *Slowdown* de cada processo no escalonamento inicial com 10 grupos de recursos.

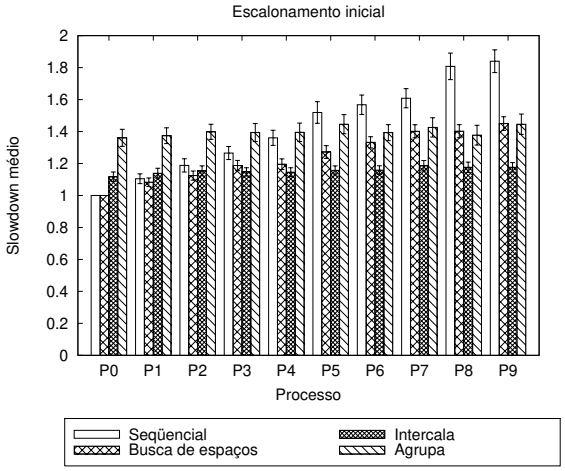


Figura 5.38: *Slowdown* de cada processo no escalonamento inicial com 25 grupos de recursos.

Tabela 5.1: Índice de justiça de Jain para *slowdowns* no escalonamento inicial.

	Número de grupos		
Algoritmo	**2**	**10**	**25**
Seqüencial	0,778268	0,881259	0,964416
Busca de espaços	0,800188	0,934998	0,986881
Intercala	0,998989	0,999576	0,999722
Agrupa	0,993800	0,997085	0,999609

lentes, porém, como mostrado nos resultados de *slowdown*, a abordagem de intercalação resulta em escalonamentos mais justos. Como o algoritmo de agrupamento trata os DAGs como se fosse apenas um, processos menores, que têm caminhos menores, tendem a ser escalonados mais tarde, pois suas tarefas recebem prioridades baixas quando os atributos do DAG agrupado são calculados. Esse comportamento acaba resultando em *slowdowns* mais altos para esses processos, pois quando são escalonados sozinhos eles têm *makespans* baixos. Isso não acontece com o algoritmo de intercalação porque tarefas de processos menores são tomadas para serem escalonadas na mesma freqüência que tarefas de processos maiores. Dessa forma, o escalonamento de DAGs menores termina antes, enquanto processos maiores continuam sendo escalonados e intercalados. Isso leva a um comportamento mais justo, tendo como conseqüência *slowdowns* similares para *workflows* pequenos e grandes. Além disso, o algoritmo de intercalação mostrou ser escalável, mantendo bons resultados com maior quantidade de grupos de recursos.

Para resumir e corroborar os resultados de *slowdown*, utilizamos outra métrica: o *Jain's Fairness Index* [59]. Dado um conjunto de valores $X = \{x_1, x_2, ..., x_n\}$, o índice de justiça de Jain é definido como:

$$J_f = \frac{(\sum_{i=1}^{n} x_i)^2}{n \sum_{i=1}^{n} x_i^2}$$

O índice de justiça varia de $\frac{1}{n}$ (injusto) a 1 (justo). A Tabela 5.1 mostra essa métrica para o escalonamento inicial dos DAGs.

Os resultados confirmam que o algoritmo de intercalação é o mas justo, seguido pela abordagem de agrupamento de DAGs. Adicionalmente, o algoritmo de intercalação é o que apresenta menor variação no índice com a variação do número de grupos de recursos.

Justiça após execução Após avaliar a justiça com o *slowdown* e o índice de justiça de Jain para o escalonamento inicial, apresentamos a mesma avaliação para os *makespans*

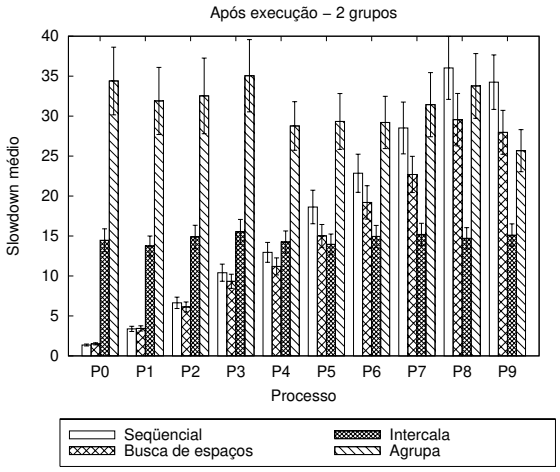


Figura 5.39: *Slowdown* de cada processo após execução com 2 grupos de recursos.

após a simulação de execução dos processos.

A métrica de *slowdown* para simulações com 2, 10 e 25 grupos é mostrada nas Figuras 5.39, 5.40, e 5.41, respectivamente. Esses resultados mostram que a abordagem de intercalação continua sendo justa após a execução das tarefas, apresentando valores de *slowdown* mais baixos que os de outras abordagens para a maioria dos processos.

A Tabela 5.2 apresenta o índice de justiça de Jain após a simulação de execução das tarefas. Observamos que a intercalação e o agrupamento continuam apresentando um resultado mais justo, com o índice ficando próximo aos apresentados antes da execução das tarefas (Tabela 5.1). Por outro lado, os resultados para os algoritmos seqüencial e de busca de espaços mostram uma piora no índice quando comparados ao escalonamento inicial. Uma explicação para esse comportamento é que esses algoritmos tendem a acumular tarefas do mesmo DAG em intervalos de tempo menores nas filas dos recursos. Com isso, tarefas do mesmo *workflow* permanecem mais próximas no escalonamento. Assim, picos de carga externa nos recursos podem potencialmente afetar mais tarefas do mesmo DAG, resultando em uma execução menos justa.

Avaliação utilizando HEFT

Para aumentar a confiança e amplitude dos resultados e das conclusões apresentadas, mostramos nesta seção algumas avaliações utilizando a heurística HEFT, em vez do PCH. A Figura 5.42 mostra os resultados de *slowdown* utilizando o HEFT como heurística para 10 grupos de recursos no escalonamento inicial, além de mostrar o *slowdown* utilizando HEFT para 10 grupos de recursos após a execução. Podemos observar o mesmo padrão apresentado pelo PCH na Figura 5.37. Esses resultados reforçam as conclusões alcançadas

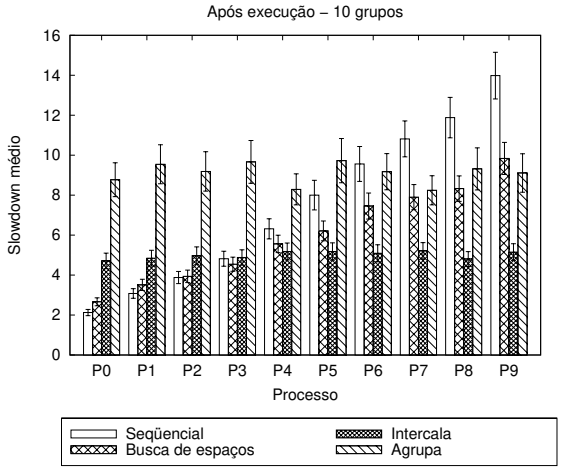


Figura 5.40: *Slowdown* de cada processo após execução com 10 grupos de recursos.

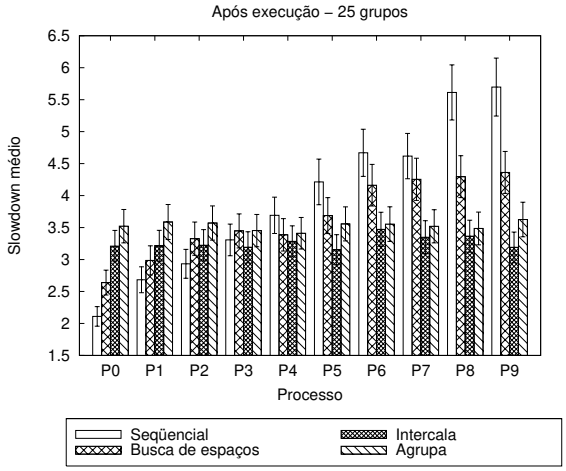


Figura 5.41: *Slowdown* de cada processo após execução com 25 grupos de recursos.

Tabela 5.2: Índice de justiça de Jain para *slowdowns* após execução.

	Número de grupos		
Algoritmo	**2**	**10**	**25**
Seqüencial	0,682643	0,790583	0,921262
Busca de espaços	0,703943	0,878235	0,976378
Intercala	0,998656	0,998800	0,999145
Agrupa	0,992068	0,997034	0,999698

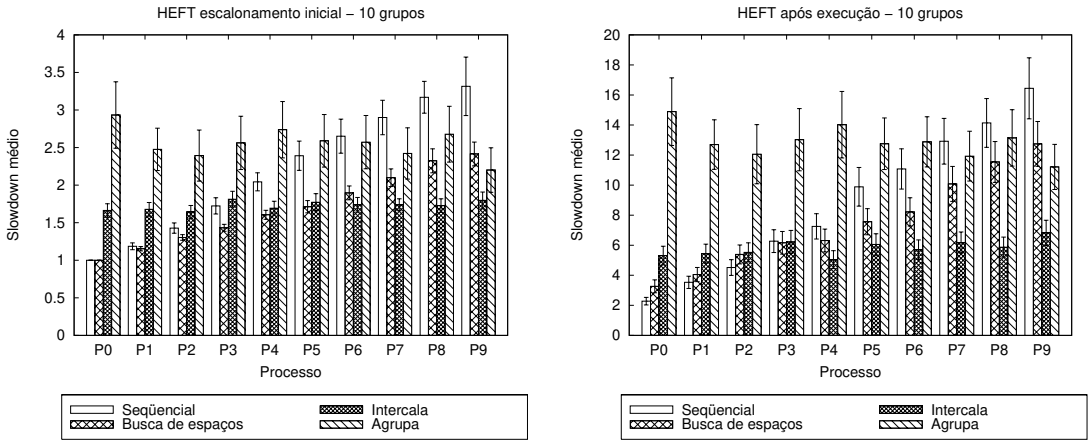


Figura 5.42: *Slowdown* de cada processo escalonado usando HEFT no escalonamento inicial com 2 grupos e após execução com 10 grupos de recursos.

quando a heurística PCH foi utilizada.

Discussão

Como contribuição geral dos resultados apresentados, podemos concluir a partir das simulações que a abordagem de intercalação é a melhor opção quando há mais de 4 processos a serem escalonados e quando o número de recursos não é baixo. Quando a quantidade de recursos disponível é baixa (2 grupos nos resultados mostrados), o algoritmo de busca de espaços resultou em *makespans* médios melhores, o que o torna uma boa opção nesse cenário. Entretanto, a busca de espaços por si só parece ser mais afetada por carga externa que o algoritmo de intercalação, com este último se tornando melhor que o primeiro após a execução quando há mais de 6 processos. A tabela 5.3 mostra um resumo dos resultados observados para o *slowdown* e *makespan* médio de cada algoritmo.

Ainda que o algoritmo seqüencial apresente resultados piores de *makespan* médio que os outros algoritmos quando consideramos a carga externa, podemos observar que este é o algoritmo menos afetado, pois suas médias são proporcionalmente mais próximas às dos outros algoritmos que no escalonamento inicial. Atribuímos esse comportamento ao fato de haver mais espaços entre as tarefas nesse tipo de escalonamento, o que pode absorver eventuais atrasos na execução das tarefas com pouca interferência no restante do escalonamento.

As quatro estratégias avaliadas (escalonamento seqüencial, intercalação, agrupamento e busca de espaços) mostraram comportamento variado quando observamos o *makespan* de diferentes quantidades de processos. Nos resultados de escalonamento inicial, podemos

Tabela 5.3: Resumo dos resultados.

Algoritmo	*Makespan* médio	Justiça
Seqüencial	Menor para os primeiros *workflows* e maior para os últimos. Menos afetado por carga externa devido aos espaços no escalonamento.	Não. Os primeiros *workflows* têm *slowdown* menor.
Busca de espaços	Menor para os primeiros *workflows* e maior para os últimos. Melhor desempenho com número baixo de recursos	Não. Os primeiros *workflows* têm *slowdown* menor.
Intercalação	Todos os *workflows* têm *makespans* similares Escalável, mantém bons escalonamentos e justiça com variação na quantidade de recursos.	Sim. Resulta em *slowdowns* similares para todos os *workflows*.
Agrupamento	Todos os *workflows* têm *makespans* similares. *Workflows* menores podem ser escalonados mais tarde, afetando o *slowdown*.	Sim. *Workflows* têm *slowdown* ligeiramente diferentes.

observar que o escalonamento seqüencial fornece vantagem absoluta ao primeiro processo escalonado, assim como a busca de espaços, porém esta última com melhor média de *makespan* porque utiliza espaços vazios para processar outras tarefas. Por outro lado, a estratégia de agrupamento e a de intercalação resultam em *makespans* piores para os primeiros processos, mas com melhorias no *makespan* dos últimos processos. Na média, isso resulta em melhor uso dos recursos, pois o *makespan* global da intercalação e do agrupamento são menores que dos algoritmos de busca de espaços e seqüencial. Adicionalmente, o agrupamento e a intercalação são capazes de produzir escalonamentos mais justos que as outras abordagens. Em termos de *slowdown*, a abordagem de intercalação mostra os melhores resultados, alcançando o *slowdown* mais baixo e homogêneo na maioria dos casos.

5.5 Trabalhos Adicionais

Nesta seção apresentamos, um resumo dos trabalhos adicionais desenvolvidos em colaboração com outros autores. Os trabalhos englobam colaborações com Carlos Roberto Senna, cujos conteúdos incluem avaliações da execução de *workflows* em grades orientadas a serviços com processadores *multicore*, a utilização de máquinas virtuais em grades computacionais com processadores *multicore*, além de uma arquitetura para suporte a execução de *workflows* de serviços em sistemas de grade/nuvem híbridos. Outro trabalho adicional, em colaboração com André Luís Vignatti e Flávio Keidi Miyazawa, foi

o desenvolvimento de algoritmos para balanceamento de carga em sistemas heterogêneos utilizando teoria dos jogos.

Avaliação da Execução de *Workflows* em Grades Orientadas à Serviços

A Internet em conjunto com seus padrões abertos permite às organizações virtuais compartilharem seus recursos. A união desses conceitos e tecnologias resultou na criação das organizações virtuais multi-institucionais [47]. Em uma organização virtual, o conceito de serviço torna-se importante tanto como base para as aplicações como para a colaboração entres os participantes. Esse ambiente naturalmente colaborativo permite aos usuários o estabelecimento de ligações entre serviços, organizando-os como *workflows* ao invés de aplicações isoladas. Essa composição de serviços traz novas dificuldades como resultado da interação entre participantes das organizações virtuais. Em relação à execução de *workflows* de serviços, existem *overheads* a serem considerados pelo *middleware*. É importante o conhecimento desses *overheads* quando comparada a execução de serviços em relação à execução de tarefas, para que seja possível aprimorar a coordenação e execução dos *workflows*. Além dos trabalhos em escalonamento de *workflows*, realizamos experimentos que avaliam tais overheads, tempos de comunicação e processamento envolvidos na execução de *workflows* de serviços em grades computacionais que gerenciam *workflows* de serviços e que são capazes de realizar instanciação dinâmica de serviços.

No nosso ambiente de testes utilizamos o Globus Toolkit (GT) [44] como infra-estrutura de grades. O GT é uma implementação da OGSA (Open Grid Services Architecture) [46], proposto pelo OGF (Open Grid Forum) [3]. No OGSA todos os recursos lógicos e físicos são modelados como serviços na grade, trazendo ao ambiente conceitos como SOA (Service Oriented Architecture) [33].

Para gerenciar a composição de serviços utilizamos o *Grid Process Orchestration* (GPO) [88]. O GPO é um *middleware* para interoperabilidade de aplicações distribuídas que requerem composição de serviços em grades. Ele permite a criação e gerência de fluxos de aplicações, tarefas e serviços na grade. A arquitetura GPO é baseada nos conceitos apresentados pela OGSA. A arquitetura GPO consiste de três componentes principais: o *GPO Run*, o $GPOMaestroFactory$ e arquivos especificados na GPO Language (GPOL). Esses componentes são mostrados na Figura 5.43.

O *GPO Run* recebe o *workflow* submetido pelo usuário e encaminha a requisição a ser processada. O *GPO Maestro* foi construído usando os conceitos de fábrica/instância fornecidos pelas infra-estruturas de grade. O *GPO Run* utiliza o serviço *GPO Maestro Factory* para criar novas instâncias que são responsáveis pela execução do *workflow* submetido. O serviço de escalonamento é responsável pelo escalonamento do *workflow*

[3] http://www.ogf.org/

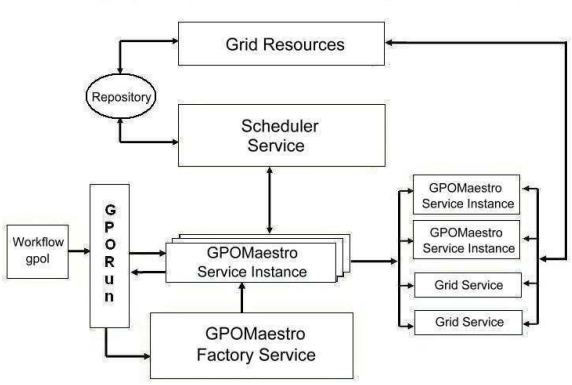


Figura 5.43: Arquitetura da infra-estrutura.

baseando-se em informações dadas pelo repositório de recursos e de serviços, que possui informações coletadas por monitores de recursos na grade.

A avaliação feita neste trabalho foi baseada em uma aplicação de filtro de mediana utilizado em processamento de imagens representadas por matrizes de pontos [68]. Esse filtro substitui o valor de um *pixel* da imagem pelo valor da mediana de sua vizinhança, de acordo com o tamanho da vizinhança a ser considerada. Dessa forma, para uma vizinhança de tamanho 1 (filtro de $3x3$), 9 elementos da matriz são ordenados e o valor da mediana desses *pixels* é tomado como novo valor para o *pixel* central. Para uma vizinhança de tamanho 2 (filtro de $5x5$), 25 elementos são ordenados e o mesmo procedimento de substituição pelo *pixel* da mediana é realizado. Nos experimentos executados utilizamos filtros de vizinhança 1, 2 e 3.

Implementamos uma variedade de cenários de filtros para execução em paralelo utilizando GPOL (*Grid Process Orchestration Language*) [88], que é uma linguagem baseada em XML para especificação *workflows* de serviços. As diferentes maneiras de executar o filtro de mediana implementadas incluem:

- Aplicação do filtro à imagem em um único arquivo de matriz utilizando apenas um recurso.
- Divisão do arquivo em blocos menores, aplicando o filtro em cada bloco em paralelo utilizando um único recurso, para então reagrupar os blocos gerando a imagem final.
- Divisão do arquivo em blocos menores, aplicando o filtro em cada bloco em paralelo utilizando mais de um recurso, para então reagrupar os blocos no recurso de origem e gerar a imagem final.

Por exemplo, o *workflow* GPOL mostrado na Figura 5.44 foi utilizado para aplicar um filtro de mediana $3x3$ no arquivo */tmp/A-2000x2000*, cujo conteúdo representa uma

imagem com resolução $2.000x2.000$. Nesse *workflow* o arquivo de imagem é dividido em cinco blocos que são processados por cinco instâncias do serviço de aplicação do filtro de mediana executadas em paralelo, e então os arquivos são combinados em um único arquivo resultado. Como diferentes escalonadores podem ser usados no GPO, a GPOL oferece a opção de deixar em aberto as definições de recursos onde cada instância será criada e onde a execução ocorrerá. No nosso exemplo o escalonador recebe o *workflow* e fica responsável por indicar os recursos para cada instância. Em seguida o GPO substitui o *host* no campo *uri* de acordo com o escalonamento. Se não há um escalonador disponível, o usuário pode simplesmente preencher o campo *uri* com o recurso onde a instância deve ser criada.

Na Figura 5.44, em (1) as variáveis de *workflow* são criadas. Essas variáveis contêm informações que são passadas para os serviços durante a execução. Em (2) as instâncias do serviço de processamento são criadas. Existem cinco instâncias, uma para cada bloco da matriz. Em (3) a operação de divisão em cinco blocos é invocada, e em (4) as cinco instâncias criadas são utilizadas para aplicar o filtro de mediana em cada bloco. Finalizando, em (5) os blocos resultantes são combinados e o arquivo final é gerado.

Aplicamos o filtro a matrizes com tamanhos de $50x50$ a $5.000x5.000$. Os filtros aplicados foram de $3x3$, $5x5$ e $7x7$. Para cada matriz e para cada tamanho de filtro realizamos as seguintes medições de tempo

- Tempo do filtro de mediana: o tempo total gasto na aplicação do filtro.
- Tempo de instanciação: tempo médio gasto para instanciar todos os serviços utilizados na aplicação do filtro.
- Tempo de transmissão: tempo médio gasto na transmissão dos dados necessários (matriz e/ou blocos da matriz) através da rede para aplicar o filtro utilizando serviços remotos.
- Tempo de execução do GPO: o tempo total do início ao final da orquestração promovida pelo GPO.
- Tempos de divisão e reagrupamento: tempo gasto na divisão dos arquivos de matriz e seu posterior reagrupamento para gerar a matriz resultante.
- Tempo cliente: o tempo que o filtro levou para executar do ponto de vista do usuário que submeteu o *workflow*.

Em seguida avaliamos como a instanciação dinâmica de serviços na grade interfere na execução do *workflow*.

```
<?xml version="1.0" encoding="UTF-8"?>
<gpo:job name="ip-A2000x2000-5p-1rcz">

  <gpo:definitions name="job_Definitions">
(1) <gpo:variables>
    <gpo:variable name="retCode" type="int" value="0"/>
    <gpo:variable name="sliceIn" type="string"
    value="5 3 /tmp/A-2000x2000"/>
    <gpo:variable name="sliceOut" type="string"/>
    <gpo:variable name="mFIn-1" type="string"
    value="3 /tmp/A-2000x2000_s1 /tmp/A-2000x2000_smf1"/>
    ...
    <gpo:variable name="mFIn-5" type="string"
    value="3 /tmp/A-2000x2000_s5 /tmp/A-2000x2000_smf5"/>
    <gpo:variable name="mergeIn" type="string"
    value="5 3 /tmp/A-2000x2000 /tmp/A-2000x2000-MF3-5p"/>
    <gpo:variable name="mergeOut" type="string"/>
    </gpo:variables>

    <gpo:gservices>
(2)  <!-- instance 1 -->
      <gpo:gsh name="IA-mf1"
          uri="host:8080/factip/FactIPService"
          type="Factory"/>
      ...
      <!-- instance 5 -->
      <gpo:gsh name="IA-mf5"
          uri="host:8080/factip/FactIPService"
          type="Factory"/>

    </gpo:gservices>
  </gpo:definitions>

<gpo:process name="job_Process">
   <!|Divide file in 5 parts -->
(3)  <gpo:invoke name="IA-mf1" method="sliceMatrixToFiles">
       <gpo:argument variable="sliceIn" type="string"/>
       <gpo:return variable="sliceOut" type="string"/>
     </gpo:invoke>

(4)  <gpo:flow>
         <!|Aply median filter in parallel way -->
       <gpo:invoke name="IA-mf1" method="MFilterMatrix">
          <gpo:argument variable="mFIn-1" type="string"/>
          <gpo:return variable="retCode" type="int"/>
       </gpo:invoke>

       <gpo:invoke name="IA-mf2" method="<MFilterMatrix">
          <gpo:argument variable="mFIn-2" type="string"/>
          <gpo:return variable="retCode" type="int"/>
       </gpo:invoke>

       <gpo:invoke name="IA-mf3" method="MFilterMatrix">
          <gpo:argument variable="mFIn-3" type="string"/>
          <gpo:return variable="retCode" type="int"/>
       </gpo:invoke>

       <gpo:invoke name="IA-mf4" method="MFilterMatrix">
          <gpo:argument variable="mFIn-4" type="string"/>
          <gpo:return variable="retCode" type="int"/>
       </gpo:invoke>

       <gpo:invoke name="IA-mf5" method="MFilterMatrix">
          <gpo:argument variable="mFIn-5" type="string"/>
          <gpo:return variable="retCode" type="int"/>
       </gpo:invoke>

    </gpo:flow>

(5) <!|Merge block files into a new file -->
    <gpo:invoke name="IA-mf1" method="mergeSliceFiles">
      <gpo:argument variable="mergeIn" type="string"/>
      <gpo:return variable="mergeOut" type="int"/>
    </gpo:invoke>

   <!-- Return value to GPO_Client -->
   <gpo:return variable="retCode" type="int"/>
 </gpo:process>
</gpo:job>
```

Figura 5.44: Exemplo de um *workflow* em GPOL.

Os resultados experimentais mostram que o uso da composição de serviços não tem um *overhead* significativo. Um estudo de caso sugere que a migração da execução tradicional de aplicação seqüencial para aplicação paralela em grades vale a pena, pois o processo de migração pode ser feito gradativamente, sem interferência nas aplicações em utilização, e promovendo a melhoria de execução com os serviços em paralelo. Ainda, os experimentos mostram que um usuário com um recurso de capacidade média pode obter execuções até 5 vezes mais rápidas no ambiente de testes utilizado ao aplicar o filtro de mediana. Outra observação a ser feita é que distribuir um conjunto de tarefas em recursos medianos pode trazer resultados tão bons quanto a utilização de recursos mais potentes quando tais tarefas têm tempos de execução similares e existe uma tarefa que depende de todas desse conjunto (no caso, a operação de reagrupamento). Adicionalmente, avaliamos que a instanciação dinâmica e a coordenação automática de referências a novos serviços podem ser executadas sem interferência do usuário, oferecendo um tratamento ao comportamento dinâmico da grade. Detalhes dessas avaliações podem ser encontradas em [84], [85] e [86].

A avaliação do desempenho dos recursos é importante no contexto de grades para que o *middleware* possa obter o desempenho dos recursos e, dessa forma, ter informações atualizadas em seus repositórios. Tais informações são utilizadas pelo escalonador em suas tomadas de decisão. Informações detalhadas sobre os tempos envolvidos na execução dos passos necessários ao uso de serviços em grades podem ser incorporadas aos algoritmos de escalonamento, auxiliando na otimização da função objetivo.

Desempenho de Máquinas Virtuais na Execução de *Workflows* de Serviços

A virtualização [90] é o processo de apresentar um grupo ou subconjunto lógico de recursos computacionais. O *software* de virtualização abstrai o *hardware* através da criação de interfaces para as máquinas virtuais (VMs), que representam recursos virtualizados, como CPUs, memória RAM, conexões de rede e periféricos. Cada máquina virtual é um ambiente de execução logicamente isolado e independente dos outros. Com isso, cada VM pode ter seu próprio sistema operacional, aplicações e serviços de rede.

Neste trabalho avaliamos como máquinas virtuais influenciam na execução de *workflows* de serviços em grades computacionais. Escolhemos uma aplicação amplamente utilizada por pesquisadores da área de redes de computadores: a simulação de cenários de rede utilizando o *Network Simulator 2* (NS2). Construímos *workflows* que executam vários cenários de redes WiMAX em paralelo e executamos tais *workflows* para avaliar o desempenho de máquinas virtuais e reais na grade.

Utilizamos o Xen como *software* de máquina virtual em nosso ambiente de testes. O Xen é um *software* de código aberto baseado na tecnologia de paravirtualização [1]. Um ambiente virtual Xen consiste de vários componentes que trabalham em conjunto para fornecer uma plataforma virtualizada. A arquitetura do Xen 3.0 inclui o *Xen Virtual*

Tabela 5.4: Especificações dos recursos computacionais da grade.

Nome	Processador	Núcleos	Clock	RAM	Disco	É VM? (host)
Cronos (C)	Intel Core 2 Quad	4	2.4 GHz	4 GB	2x320 GB	Não
Medusa (M)	Intel Xeon E5430	8	2.66 GHz	2 GB	19 GB	Não
Temis (T)	Intel Xeon E5430	8	2.66 GHz	1.3 GB	95 GB	Não
Helios (H)	Intel Xeon E5430	4	2.66 GHz	6.6 GB	390 GB	Sim (Temis)
Persefone (P)	Intel Xeon E5430	4	2.66 GHz	6.6 GB	390 GB	Sim (Temis)
Caos (O)	Intel Xeon E5430	4	2.66 GHz	6.6 GB	430 GB	Sim (Medusa)
Eros (E)	Intel Xeon E5430	4	2.66 GHz	6.6 GB	430 GB	Sim (Medusa)

Machine Manager (VMM), cuja função é abstrair a camada inferior de *hardware* físico e fornecer acesso a diferentes máquinas virtuais. O *Domain 0* é uma VM especial que permite acesso à interface de controle do VMM, através do qual outras VMs podem ser criadas, destruídas e gerenciadas. Na implementação de nosso ambiente de testes, utilizamos o Globus Toolkit 4 como ambiente de grades instalado em máquinas com sistema operacional Debian GNU/Linux conectadas por uma rede *gigabit ethernet*.

Nosso ambiente de testes é composto de sete recursos computacionais, dos quais três são máquinas físicas (Cronos, M1 e M2), e quatro máquinas virtuais. Sobre a máquina física M1 estão três recursos computacionais: Temis, Helios e Persefone. Temis é uma "máquina física Debian GNU/Linux", enquanto Helios e Persefone são VMs Xen em Temis. Em M2 utilizamos o mesmo esquema, com Medusa sendo a máquina física, e Eros e Caos sendo as máquinas virtuais Xen em Medusa. A Tabela 5.4 resume as características dos recursos computacionais que compõem o nosso ambiente de testes.

O cenário base de aplicação é simular no NS2 a comunicação entre *subscriber stations* uniformemente distribuídas e uma estação base. A simulação gera informações sobre latência, quantidade de pacotes, desvio padrão e intervalos de confiança para fluxos de *uplink* e *downlink* em diferentes modelos de transmissão de dados. Utilizando esse cenário base, construímos vários outros através da combinação de diferentes valores para os parâmetros de entrada, como diferentes sementes e quantidade de nós móveis na rede sem fio simulada. Utilizamos simulações de 20, 40 e 60 segundos, onde esse tempo representa quantos segundos de tráfego de rede serão simulados com seus detalhes gravados em arquivos de registro.

As simulações em NS2 são caracterizadas pelo uso intensivo de recursos, pois cada cenário deve ser simulado várias vezes até que os resultados convirjam de acordo com as características da rede simulada. Outra característica desse tipo de simulação é o uso intensivo de disco quando *traces* são armazenados para serem processados em seguida. Sendo assim, grades computacionais são ambientes propícios para execução dessas simulações.

Para observar o comportamento das máquinas virtuais, implementamos serviços que executam simulações do NS2 com diferentes parâmetros. Avaliamos como as VMs e seus recursos hospedeiros se comportam quando o escalonador considera-os como recursos separados, sem possuir informações extras que indiquem quais recursos são VMs. Para realizar essa avaliação, realizamos os seguintes experimentos:

- Avaliação do desempenho dos recursos individualmente para estabelecer parâmetros de desempenho.
- Execução dos mesmos cenários da avaliação anterior, porém utilizando *workflows* de serviços para avaliação de *overheads*.
- Execução dos cenários em paralelo com diversas combinações de quantidade de processos em paralelo e diferentes recursos computacionais.
- Execução dos *workflows* com e sem armazenamento de *logs* e *traces* para avaliar concorrência de disco.
- Execução de *workflows* com tantos cenários em paralelo quantos núcleos disponíveis para avaliar concorrência de CPU.

Com os experimentos executados pudemos concluir que o uso de VMs como recursos computacionais pode ser interessante quando certos cuidados são tomados pelo gerenciador de recursos, prestando atenção às concorrências introduzidas quando máquinas virtuais no mesmo recurso físico são utilizadas ao mesmo tempo. Concorrência involuntária em CPU, memória RAM e escrita/leitura de disco podem ocorrer nesse tipo de ambiente e prejudicar a execução das tarefas da grade. Os resultados indicam perdas de desempenho quando existe concorrência em situações de uso intenso de disco e quando o número de CPUs é extrapolado involuntariamente. Dessa forma, um gerenciador de recursos que tenha ciência da existência e localização de máquinas virtuais na grade é necessário para que seja possível explorar as vantagens das VMs sem prejudicar o desempenho global do sistema. Detalhes dos resultados podem ser encontrados em [87].

Assim como no trabalho apresentado na seção anterior, este também trata da avaliação do desempenho de recursos computacionais. Entretanto, este trabalho considera máquinas virtuais, que inserem novas características e *overheads* que devem ser levados em conta pelo escalonador. Novamente, a avaliação do desempenho dos recursos é importante para que o *middleware* possa obter o desempenho dos recursos e, dessa forma, ter informações atualizadas em seus repositórios, auxiliando o escalonador a otimizar sua função objetivo.

Execução de *Workflows* de Serviços em Sistemas de Grade/Nuvem Híbridos

A computação em nuvem fornece acesso a recursos computacionais sob demanda, enquanto a computação em grade permite usuários compartilharem recursos heterogêneos em diferentes domínios administrativos. Neste trabalho, discutimos características e requisitos de um sistema híbrido composto por tecnologias de grade e nuvem e propomos uma infra-estrutura que é capaz de gerenciar a execução de *workflows* de serviços nesse sistema.

Na computação em nuvem, os usuários podem alcançar de forma transparente capacidades de processamento e armazenamento virtualmente infinitas. Através da virtualização, a nuvem oferece recursos computacionais com configurações de *hardwares* diferentes, como quantidade de memória e número de núcleos no processador. Nesse paradigma, detalhes são escondidos do usuário, que não necessita de conhecimento ou experiência no controle da infra-estrutura. Nuvens são geralmente classificadas em três modelos: *software as a service* (SaaS), *platform as a service* (PaaS) e *infrastructure as a service* (IaaS). Esses três modelos permitem o uso dos padrões da computação orientada a serviços, fornecendo a possibilidade do usuário criar ligações entre serviços, organizando-os como *workflows*.

No modelo SaaS o consumidor usa uma aplicação mas não tem controle sobre o ambiente. O Google Apps [4] e o Salesforce.com [5] são exemplos desse modelo. No modelo PaaS o consumidor usa um ambiente hospedeiro para suas aplicações. O Google App Engine [6] e o Amazon Web Services [7] são exemplos de PaaS. No modelo IaaS o consumidor utiliza recursos computacionais, tais como poder de processamento e armazenamento. Neste modelo o consumidor pode controlar o ambiente, incluindo a instalação de aplicações. O *Amazon Elastic Compute Cloud* (EC2), o Globus Nimbus [8], e o Eucalyptus [72] são exemplos desse modelo.

Atualmente o modelo mais popular é o IaaS. De maneira simplificada, podemos entender esse modelo de nuvem como um conjunto de servidores acessíveis pela Internet. Esses servidores podem ser gerenciados, monitorados e mantidos dinamicamente e remotamente, sendo a virtualização um conceito fundamental para isso. Através da virtualização os usuários têm acesso a máquinas virtuais personalizadas e isoladas, tendo controle sobre o recurso virtual da grade sem interferência de outros participantes.

Nossa infra-estrutura supre a falta de suporte à execução de serviços em grades e nuvens, oferecendo instanciação automática de serviços nos recursos obtidos dinamicamente

[4] http://www.google.com/apps/
[5] http://www.salesforce.com/
[6] http://code.google.com/intl/en/appengine/
[7] http://aws.amazon.com/
[8] http://workspace.globus.org/

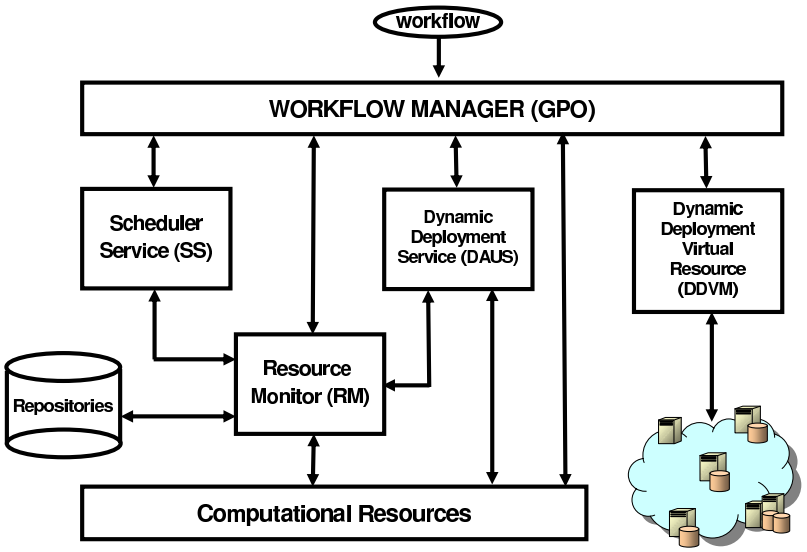


Figura 5.45: A arquitetura proposta.

da nuvem e controlando a execução do *workflow* de serviços através de um gerente de *workflows* que interage com a nuvem de maneira transparente, sem interferência do usuário. Dessa forma, este trabalho agrega o suporte a execução de *workflows* em grades e nuvens. A infra-estrutura é direcionada a sistemas que utilizam o paradigma de computação orientada a serviços. Para isso, combinamos uma grade orientada a serviços baseada no Globus Toolkit 4 (GT4) com a nuvem Nimbus, uma opção baseada na Amazon Elastic Compute Cloud (EC2) [9].

A arquitetura proposta, formada por um conjunto de serviços, é mostrada na Figura 5.45. Seus principais componentes são o gerente de *workflow*, o escalonador, o serviço de publicação de serviços, o serviço para publicação dinâmica de recursos virtuais na nuvem e repositórios utilizados por esses componentes.

Em uma execução típica na grade, o usuário submete um *workflow* ao *Grid Process Orchestrator* (GPO), que é o gerente de *workflows* do sistema. O GPO analisa o *workflow*, consultando o serviço de escalonamento (SS), que identifica os melhores recursos para cada serviço envolvido. Se um novo serviço precisa ser publicado, o GPO realiza essa tarefa através do *Deployment and Undeployment Service* (DAUS), que publica o novo serviço e retorna sua localização (*endpoint reference*) ao GPO. Usando essa referência, o GPO cria as instâncias necessárias durante a execução do *workflow*. Quando a execução finaliza, o GPO pode automaticamente eliminar as instâncias criadas e adicionadas ao DAUS ou

[9] http://aws.amazon.com/ec2/

pode deixar os serviços para uso futuro se necessário. Se a execução do *workflow* demandar mais recursos, o GPO inicia o *Dynamic Deployment Virtual Resource* (DDVM), que procura por recursos na nuvem. O DDVM faz requisições de recursos à nuvem informando sua localização ao GPO. Através do DDVM, o GPO usa o recurso da nuvem de maneira similar ao uso de recursos locais da grade. Ele publica e instancia os serviços dinamicamente, além de monitorar os recursos através dos serviços de infra-estrutura. Mais detalhes da arquitetura podem ser encontrados em [19].

A arquitetura proposta pode ser útil em cenários que aparecem atualmente na execução de aplicações em grades. Se considerarmos que a minimização do *makespan* é um objetivo importante a ser alcançado, a infra-estrutura tem sempre a opção de requisitar recursos à nuvem quando os recursos disponíveis não podem fornecer um *makespan* satisfatório. Por exemplo, isso pode ocorrer quando a grade está sobrecarregada com muitas submissões em um pico de carga. Nesse cenário, um novo *workflow* poderia ter seu *speedup* prejudicado fortemente, pois teria que esperar muitas outras tarefas finalizarem para prosseguir com a sua execução.

Outro cenário que visualizamos como possível aplicação para a infra-estrutura proposta é quando um *workflow* tem mais tarefas paralelas que a quantidade de recursos disponíveis na grade. Um exemplo de aplicação desse tipo é o Montage, mostrado anteriormente na Figura 4.2. Montage é uma aplicação de processamento de imagens que constrói mosaicos do céu para pesquisas em astronomia. O tamanho de seu DAG depende da área do céu a ser gerada. Por exemplo, para 1 grau quadrado do céu, um DAG de 232 nós é executado. Para 10 graus, um DAG com 20.652 nós é executado, tratando de um volume de dados próximo a 100 GB. O céu completo tem em torno de 400.000 graus quadrados [34]. Para tal aplicação, uma infra-estrutura elástica é desejável, onde o número de recursos pode ser adaptado de acordo com o tamanho do DAG a ser executado. No nosso sistema híbrido, a infra-estrutura pode evitar o uso da nuvem quando a grade é suficiente para a execução do *workflow*. Por outro lado, quando o DAG é muito grande, a infra-estrutura pode obter recursos de nuvens para comportar a execução.

Outro cenário é quando existem *deadlines* para a execução de *workflow*. Se o escalonador determinar que a grade por si só não é capaz de fornecer recursos na quantidade e qualidade necessárias para a execução do *workflow* dentro de um limite de tempo, ela pode requisitar à nuvem recursos para compor a infra-estrutura até que esta seja capaz de cumprir o *deadline*.

Com a necessidade de publicação dinâmica de serviços, porém utilizando os recursos da nuvem de forma cautelosa para não onerar a execução em termos financeiros, a infra-estrutura proposta é uma das possíveis aplicações do algoritmo de escalonamento bicritério proposto nesta tese.

Balanceamento de Carga em Sistemas Heterogêneos Utilizando Teoria dos Jogos

Para alcançar bom desempenho em sistemas distribuídos heterogêneos como as grades e as nuvens, a maneira de se distribuir tarefas nos recursos é um problema importante. Como são ambientes potencialmente grandes, uma gerência de recursos descentralizada é uma opção para controlar a distribuição das tarefas, já que uma entidade central pode não ser capaz de manter todas as informações necessárias atualizadas constantemente. Além disso, nas grades os recursos são oriundos de usuários e organizações distintas, podendo existir para diferentes propósitos, o que pode introduzir conflitos de interesse quando tais recursos são disponibilizados a outros usuários. Nesse contexto, um usuário pode estar somente preocupado com suas tarefas e seus recursos, e ele pode querer utilizar os melhores recursos disponíveis para executar suas tarefas e minimizar o uso de seus próprios recursos. Dessa forma, um comportamento egoísta pode ser observado, e isso pode ser visto como um jogo não-cooperativo em que cada usuário está interessado somente em executar suas tarefas rapidamente utilizando recursos alheios.

Em sua essência, o balanceamento de carga pode ser centralizado ou descentralizado, e estático ou dinâmico. Um algoritmo de balanceamento de carga centralizado [67] tem uma entidade central que concentra informações sobre os recursos do sistema, sendo também responsável pela atribuição de tarefas (carga) aos recursos. Por outro lado, um algoritmo descentralizado [81] executa em todos os recursos, sendo que cada recurso tem uma visão parcial do sistema e toma decisões locais para realizar o balanceamento de carga. Um algoritmo estático [61] usa informação não modificada obtida antes de sua execução para realizar o balanceamento, ao passo que um algoritmo dinâmico [69] pode utilizar informações em tempo de execução enquanto realiza o balanceamento de carga para tomar decisões. Abordagens híbridas (ou semi-estáticas) também existem [91]. Neste trabalho tratamos de algoritmos para balanceamento de carga dinâmico descentralizado.

Vários trabalhos tratam do balanceamento de carga em um arcabouço de teoria dos jogos, mais especificamente com jogadores egoístas. Em sistemas não-distribuídos, Even-dar e outros [42] estudaram o tempo de convergência para alcançar o equilíbrio de Nash em problemas de balanceamento de carga em *Elementary Step System* (ESS), mostrando limitantes superiores e inferiores para uma variedade de casos. Goldberg [50] considera um protocolo fracamente distribuído que simula o ESS. Nesse protocolo uma tarefa escolhe máquinas aleatoriamente e migra se a carga é mais baixa. Ele utiliza uma função potencial para mostrar limitantes superiores no número de passos para alcançar o equilíbrio de Nash. Even-dar e Mansour [43] consideram o caso em que todos os usuários escolhem migrar ao mesmo tempo. Nesse modelo as tarefas migram de máquinas sobrecarregadas para máquinas com menor carga de acordo com probabilidades calculadas considerando que elas possuem informações de carga de todas as máquinas. Berenbrink e outros [8]

propõem um protocolo distribuído que necessita de pouca informação global, em que uma tarefa precisa requisitar a informação de carga de apenas uma outra máquina, migrando se tal máquina tem carga menor. Todos esses trabalhos consideram um modelo de rede completa, onde um nó pode enviar carga a qualquer nó da rede, o que difere de nosso trabalho, onde consideramos um modelo mais geral com topologias de rede arbitrárias. Outro diferencial do nosso trabalho é que no nosso modelo os recursos, ao invés das tarefas, são os jogadores. Esse modo de representar o jogo reflete melhor um sistema distribuído real, pois o dono de uma tarefa a atribui a qualquer recurso que possa executá-la, sem se preocupar com uma possível sobrecarga do recurso. Em outras palavras, a sobrecarga de recursos é um problema a ser tratado pelos próprios recursos, e não pelo dono da tarefa.

A larga escala dos sistemas distribuídos em conjunto com o comportamento egoísta de seus usuários motivou o desenvolvimento de algoritmos de balanceamento de carga distribuídos. Tais algoritmos tentam garantir que cada recurso no sistema tem a mesma quantidade de trabalho para realizar. Para fazer isso, um algoritmo de balanceamento de carga atribui ou remove tarefas (ou parte de tarefas) aos recursos, diminuindo a quantidade de trabalho em um recurso e aumentando em outro. Após uma seqüência finita de movimentos, é esperado que uma partilha justa de trabalho exista nos recursos, com cada recurso tendo uma quantidade de trabalho compatível com sua capacidade de processamento.

Devido à falta de controle centralizado, nosso modelo supõe que cada recurso é uma entidade independente com um objetivo próprio. Ainda, se existir uma tentativa de implementação de algum tipo de regra por um controle centralizado, cada entidade independente pode simplesmente recusar-se a obedecer essas regras se elas não forem ao encontro de seus objetivos. Mesmo que entidades concordem em obedecer certas regras, a alta escalabilidade do sistema faz com que o projeto de algoritmos centralizados se torne difícil em situações práticas. Assim, definimos regras que são distribuídas e que vão de encontro aos objetivos egoístas de cada entidade.

Em nosso modelo, de um ponto de vista global, procuramos balancear a carga do sistema. Do ponto de vista do usuário, cada um quer permanecer em um estado onde estão satisfeitos com a carga atribuída a ele. Se todos os usuários estão satisfeitos, a troca de carga entre eles pára e eles podem concentrar-se no processamento de tal carga. Assim, devido à falta de controle central, mais importante que o balanceamento de carga é que os usuários estejam satisfeitos. Portanto, o conceito de solução usado neste trabalho é quando todos os usuários estão satisfeitos, o que chamamos de estado de *equilíbrio de Nash*. Então, apresentamos alguns conjuntos distintos de regras a serem usadas pelos usuários tal que cada conjunto de regras leva os usuários ao equilíbrio de Nash. Como métrica de comparação, para cada conjunto de regras que apresentamos, utilizamos o número de passos necessários para atingir o equilíbrio de Nash, apresentando simulações

para realizar essa medição.

Cada recurso é representado por um nó na rede, e um nó tem o objetivo de minimizar sua própria carga mandando parte de sua carga para seus vizinhos na rede. Assumimos que um nó está satisfeito se sua carga é menor ou (aproximadamente) igual à carga mínima de seus vizinhos. Consideramos que a carga pode ser dividida em partes de tamanho infinitesimal. Assim, quando o sistema atinge um equilíbrio aproximado, os outros nós têm suas cargas aproximadamente iguais à carga do balanceamento ótimo. Portanto, questões sobre o *preço da anarquia* [73] e medidas similares não são interessantes neste modelo. Adicionalmente, fazendo com que os nós tenham um balanceamento igual ao ótimo, o sistema permanece em equilíbrio, pois os nós não têm incentivo para enviar carga para os outros nós.

Como resultados desse trabalho, mostramos que a rede não estabiliza se regras não são utilizadas. Dessa forma, regras são criadas com as condições que sejam simples, distribuídas e sigam os objetivos dos jogadores. Discutimos como essas regras podem ser criadas e apresentamos três conjuntos delas, chamadas Agressiva, Equalitária e Passo Limitado. Para cada conjunto de regras, apresentamos provas formais de corretude. Para avaliar o desempenho de cada conjunto de regras, realizamos simulações que mostram empiricamente que o método Agressivo é o mais apropriado, enquanto o Passo Limitado se torna impraticável.

Capítulo 6

Conclusão

O problema de escalonamento de tarefas dependentes em sistemas heterogêneos, como as grades computacionais, tornou-se importante com o aumento da demanda computacional por processos de *workflows*. Esse aumento de demanda é destacado na e-Ciência, com aplicações representadas por DAGs em diversas áreas do conhecimento. Com isso, o estudo de escalonadores de tarefas dependentes representadas por DAGs levando em consideração características inerentes às grades, como a heterogeneidade e o comportamento dinâmico no desempenho e disponibilidade dos recursos, são importantes para determinar qual a melhor estratégia para extrair de forma eficiente o poder computacional disponível.

Nesta tese, apresentamos algoritmos e avaliações em quatro classes de escalonadores de tarefas dependentes, fornecendo as seguintes contribuições:

1. **Escalonamento estático de um único DAG**: esse é o problema básico de escalonamento de tarefas dependentes representadas por DAGs, e portanto é o mais freqüentemente abordado na literatura. Entre os algoritmos mais conhecidos podemos citar o *Heterogeneous Earliest Finish Time* (HEFT), que é destaque pela simplicidade e bons resultados na média. Esse algoritmo é normalmente utilizado em comparações entre algoritmos dessa classe. Nesta tese apresentamos uma extensão ao HEFT que utiliza a técnica de *lookahead* para estimar o impacto de decisões ótimas locais no escalonamento de tarefas que estão adiante no DAG. Resultados de simulações mostram que essa extensão pode trazer melhorias no *makespan* sem tornar o tempo de execução do algoritmo proibitivo.

2. **Escalonamento dinâmico de um único DAG**: em um ambiente com variações de carga externa nos recursos, algoritmos que realizam escalonamento com informações atualizadas podem trazer benefícios na execução de DAGs. Nesta tese propomos uma estratégia de escalonamento dinâmico baseada em turnos, em que o algoritmo escalona as tarefas submetendo-as em grupos para execução após uma avaliação dos

recursos. Apresentamos uma extensão adaptativa para esse algoritmo, em que o tamanho de cada turno é variável e dependente do desempenho dos recursos, do tamanho das tarefas do DAG e considera o histórico de desempenho dos recursos. A abordagem dinâmica simples, sem a extensão adaptativa, mostrou que pode melhorar o *makespan* do escalonamento quando simulamos cargas externas baseadas em medições nos recursos computacionais em uma rede de laboratório de pesquisas. Simulações também mostraram que existe uma relação entre essa melhoria e a quantidade de turnos utilizada no algoritmo. A adição da extensão adaptativa mostrou-se efetiva, pois tornou o tempo de execução dos *workflows* menor quando comparado ao tempo de execução sem a extensão, fornecendo uma melhoria no *speedup* dos processos em relação ao escalonamento dinâmico.

3. **Escalonamento bi-critério**: enquanto no problema tradicional de escalonamento estático o objetivo mais comum é a minimização do *makespan*, um problema de escalonamento bi-critério tem duas variáveis a serem otimizadas. O problema que tratamos nesta tese é o de escalonamento bi-critério com objetivo de minimizar o *makespan* e o número de serviços criados em uma grade orientada a serviços. Nesse contexto, a grade tem a habilidade de criar serviços sob demanda para que os nós do DAG sejam executados. A criação deliberada de serviços pode acarretar perda de desempenho dos recursos, com desperdício de processamento e largura de banda, além de custos monetários se considerarmos ambientes de nuvem e o conceito de *utility computing*. Com o algoritmo apresentado, que utiliza a técnica de ALAP (*as late as possible*), conseguimos obter *makespans* melhores que aqueles utilizando apenas os serviços que já existem nos recursos sem criar serviços em demasia.

4. **Escalonamento de múltiplos DAGs**: sendo uma área ainda pouco explorada, o escalonamento de múltiplos DAGs no mesmo conjunto de recursos é um problema em aberto. A sua importância está na característica compartilhada das grades e, mais atualmente, das nuvens. A criação de algoritmos que forneçam, além da melhoria do *makespan* e uso eficiente dos recursos, uma execução justa entre os processos é importante. Nesta tese, descrevemos quatro algoritmos para escalonamento de múltiplos workflows. Apresentamos avaliações desses algoritmos no que concerne *makespan* e justiça quando há até 10 DAGs compartilhando o mesmo conjunto de recursos heterogêneos. Os resultados mostram que, enquanto as abordagens de escalonamento seqüencial e de busca de espaços priorizam DAGs que são escalonados primeiro, as abordagens de agrupamento de DAGs e intercalação conseguem distribuir melhor os recursos, fornecendo execuções mais justas em ambientes compartilhados. Adicionalmente, a intercalação e o agrupamento de DAGs conseguem *makespans* globais menores que as outras estratégias, o que sugere um

melhor aproveitamento dos recursos.

O escalonamento de DAGs em sistemas compartilhados e heterogêneos é um tema que ainda pode ser abordado de diferentes maneiras, o que abre espaço para uma diversidade de trabalhos futuros. Na área de escalonamento dinâmico, sistemas de predição de desempenho mais avançados e que compartilham informações com o escalonador são de grande importância para evitar reescalonamento de tarefas por conseqüência da queda de desempenho inesperada dos recursos. No escalonamento bi-critério diferentes tipos de problemas podem ser abordados, com o algoritmo proposto sendo avaliado em diferentes situações. Por exemplo, a movimentação e criação de máquinas virtuais na computação em nuvem é uma importante fonte de *overhead*. Dessa forma, um algoritmo que minimize o tempo de execução de DAGs levando em consideração a quantidade de máquinas virtuais criadas ou movidas pode ser derivado do algoritmo proposto. Finalmente, a área de escalonamento de múltiplos DAGs tem sido pouco estudada, apesar de sua importância, o que tornam amplas as opções de continuidade do trabalho aqui proposto nesta classe de algoritmos. Um trabalho inicial de estudos do comportamento de heurísticas existentes para escalonamento de um único DAG em ambientes com múltiplos DAGs seria um passo inicial para o desenvolvimento de algoritmos específicos para esse tipo de problema. Outro trabalho futuro possível é o desenvolvimento de algoritmos de reescalonamento para múltiplos DAGs, que são algoritmos importantes em ambientes dinâmicos compartilhados e demandam cuidados adicionais, sendo intrinsecamente mais complexos que os algoritmos para apenas um DAG.

Referências Bibliográficas

[1] T. Abels, P. Dhawan, and B. Chandrasekaran. An overview of xen virtualization. Technical report, Dell Enterprise Technology Center, 2009.

[2] Y. Amir, B. Awerbuch, A. Barak, R. S. Borgstrom, and A. Keren. An opportunity cost approach for job assignment in a scalable computing cluster. *IEEE Transaction on Parallel and Distributed Systems*, 11(7):760–768, 2000.

[3] 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 *Supercomputing '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pp 1-14, Baltimore, EUA, 2002. IEEE Computer Society Press.

[4] D. A. Bader and R. Pennington. Cluster computing: applications. *The International Journal of High Performance Computing*, 15(2):181–185, 2001.

[5] R. Bajaj and D. P. Agrawal. Improving scheduling of tasks in a heterogeneous environment. *IEEE Transactions on Parallel and Distributed Systems*, 15(2):107–118, 2004.

[6] D. M. Batista, N. L. S. da Fonseca, and F. K. Miyazawa. A set of schedulers for grid networks. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pp. 209-213, Seul, Coréia, 2007. ACM Press.

[7] D. M. Batista, A. C. Drummond, and N. L. S. da Fonseca. Scheduling grid tasks under uncertain demands. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 2041–2045, Fortaleza, Ceará, Brasil, 2008. ACM.

[8] P. Berenbrink, T. Friedetzky, L. A. Goldberg, P. Goldberg, Z. Hu, and R. Martin. Distributed selfish load balancing. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 354–363, 2006.

[9] L. F. Bittencourt and E. R. M. Madeira. A dynamic approach for scheduling dependent tasks on the xavantes grid middleware. In *4th ACM International Workshop on Middleware for Grid Computing*, Melbourne, Austrália, 2006.

[10] L. F. Bittencourt and E. R. M. Madeira. On the distribution of dependent tasks over non-dedicated grids with high bandwidth links. In *III Workshop TIDIA FAPESP*, São Paulo, Brasil, 2006.

[11] L. F. Bittencourt and E. R. M. Madeira. Fulfilling task dependence gaps for workflow scheduling on grids. In *3rd IEEE International Conference on Signal-Image Technology and Internet Based Systems*, Shanghai, China, 2007.

[12] L. F. Bittencourt and E. R. M. Madeira. A performance-oriented adaptive scheduler for dependent tasks on grids. *Concurrency and Computation: Practice and Experience*, 20(9):1029–1049, 2008.

[13] L. F. Bittencourt and E. R. M. Madeira. Um algoritmo de escalonamento com intercalação de processos em grades computacionais. In *XXVI Simpósio Brasileiro de Redes de Computadores*, Rio de Janeiro, Brasil, mai. 2008.

[14] L. F. Bittencourt and E. R. M. Madeira. Dag scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm (aceito para publicação). In *The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP 2010)*, Pisa, Itália, fev. 2010.

[15] L. F. Bittencourt and E. R. M. Madeira. Towards the scheduling of multiple workflows on computational grids (aceito para publicação). *Journal of Grid Computing*, 2010.

[16] L. F. Bittencourt, E. R. M. Madeira, F. R. L. Cicerre, and L. E. Buzato. A path clustering heuristic for scheduling taks graphs onto a grid (short paper). In *3rd ACM International Workshop on Middleware for Grid Computing*, Grenoble, France. nov 2005.

[17] L. F. Bittencourt, E. R. M. Madeira, F. R. L. Cicerre, and L. E. Buzato. Uma heurística de agrupamento de caminhos para escalonamento de tarefas em grades computacionais. In *XXIV Simpósio Brasileiro de Redes de Computadores*, pp. 83-98, Curitiba, Brasil, mai. 2006.

[18] L. F. Bittencourt, C. R. Senna, and E. R. Madeira. Bicriteria service scheduling with dynamic instantiation for workflow execution on grids. In *GPC '09: Proceedings of*

the 4th International Conference on Advances in Grid and Pervasive Computing, pp. 177-188, Genebra, Suiça, 2009. Springer-Verlag.

[19] L. F. Bittencourt, C. R. Senna, and E. R. M. Madeira. Enabling execution of service workflows in grid/cloud hybrid systems (aceito para publicação). In *International Workshop on Cloud Management (Cloudman 2010)*, Osaka, Japão, Abr. 2010. IEEE Computer Society Press.

[20] J. Blythe, Y. Gil, and E. Deelman. Coordinating workflows in shared grid environments. In *ICAPS '04: Proceedings of The 14th International Conference on Automated Planning and Scheduling*, Whistler, Canadá, jun. 2004.

[21] C. Boeres, J. V. Filho, and V. E. F. Rebello. A cluster-based strategy for scheduling task on heterogeneous processors. In *16th Symposium on Computer Architecture and High Performance Computing*, pp. 214-221, Foz do Iguaçú, Brasil, 2004. IEEE.

[22] J. Broberg, S. Venugopal, and R. Buyya. Market-oriented grids and utility computing: The state-of-the-art and future directions. *Journal of Grid Computing*, 6(3):255–276, 2007.

[23] R. Buyya, D. Abramson, and J. Giddy. Nimrod/g: An architecture for a resource management and scheduling system in a global computational grid. *High-Performance Computing in the Asia-Pacific Region, International Conference on*, 1:283, 2000.

[24] E.-K. Byun and J.-S. Kim. Dynagrid: A dynamic service deployment and resource migration framework for WSRF-compliant applications. *Parallel Computing*, 33(4-5):328–338, 2007.

[25] L.-C. Canon and E. Jeannot. Scheduling strategies for the bicriteria optimization of the robustness and makespan. In *11th International Workshop on Nature Inspired Distributed Computing (NIDISC 2008)*, Miami, Florida, USA, April 2008.

[26] L.-C. Canon, E. Jeannot, R. Sakellariou, and W. Zheng. Comparative Evaluation of the Robustness of DAG Scheduling Heuristics. In *Integrated Research in Grid Computing, CoreGRID Integration Workshop*, p. 63-74, Greece, April 2008.

[27] J. Cao, S. A. Jarvis, S. Saini, and G. R. Nudd. Gridflow: Workflow management for grid computing. In *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003), Tokyo, Japan*, May 2003.

[28] C. Cérin and K.-C. Li, editors. *Advances in Grid and Pervasive Computing, Second International Conference, GPC 2007, Paris, France, May 2-4, 2007, Proceedings*, volume 4459 of *Lecture Notes in Computer Science*. Springer, 2007.

[29] H. Chen and M. Maheswaran. Distributed dynamic scheduling of composite tasks on grid computing systems. In *11th IEEE Heterogeneous Computing Workshop*, pp. 119-128, Washington, EUA, 2002. IEEE Computer Society.

[30] F. R. L. Cicerre, E. R. M. Madeira, and L. E. Buzato. A hierarchical process execution support for grid computing. *Concurrency and Computation: Practice and Experience*, 18(6):581–594, 2006.

[31] K. Cooper, A. Dasgupta, K. Kennedy, et al. New grid scheduling and rescheduling methods in the grads project. In *IPDPS Next Generation Software Program - NSFNGS - PI Workshop*, pp 199-206, Santa Fe, EUA, 2004. IEEE Computer Society.

[32] R. C. Corrêa, A. Ferreira, and P. Rebreyend. Integrating list heuristics into genetic algorithms for multiprocessor scheduling. In *SPDP '96: Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing (SPDP '96)*, pp.461, New Orleans, EUA, 1996. IEEE Computer Society.

[33] F. Curbera, R. Khalaf, N. Mukhi, S. Tai, and S. Weerawarana. The next step in web services. *Communications of ACM*, 46(10):29–34, 2003.

[34] E. Deelman. Clouds: An opportunity for scientific applications? (keynote in the 2008 Cracow Grid Workshops), 2008.

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

[36] E. Deelman, C. Kesselman, G. Mehta, L. Meshkat, L. Pearlman, K. Blackburn, P. Ehrens, A. Lazzarini, R. Williams, and S. Koranda. GriPhyN and LIGO, Building a Virtual Data Grid for Gravitational Wave Scientists. In *HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, 225, Edimburgo, Reino Unido, 2002. IEEE Computer Society.

[37] 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. *Scientific Programming*, 13(3):219–237, 2005.

[38] A. Dogan and F. Özgüner. Biobjective scheduling algorithms for execution time-reliability trade-off in heterogeneous computing systems. *Computer Journal*, 48(3):300–314, 2005.

[39] F. Dong and S. G. Akl. Scheduling algorithms for grid computing: state of the art and open problems. Technical report, Queen's University School of Computing, Kingston, Canadá, jan. 2006.

[40] B. Duran and F. Xhafa. The effects of two replacement strategies on a genetic algorithm for scheduling jobs on computational grids. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pp. 960-961, Dijon, França, abr. 2006. ACM.

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

[42] E. Even-Dar, A. Kesselman, and Y. Mansour. Convergence time to nash equilibrium in load balancing. *ACM Transactions on Algorithms*, 3(3), 2007.

[43] E. Even-Dar and Y. Mansour. Fast convergence of selfish rerouting. In *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 772–781, 2005.

[44] I. Foster. Globus toolkit version 4: Software for service-oriented systems. In *IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779*, pp. 2-13, Beijing, China, 2005.

[45] I. Foster and C. Kesselman. Computational grids. In *Vector and Parallel Processing - VECPAR 2000, 4th International Conference*, jun., 2000.

[46] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. 2002.

[47] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organization. *The International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[48] J. Frey. Condor DAGMan: Handling inter-job dependencies. http://www.cs.wisc.edu/condor/dagman/, 2002.

[49] N. Fujimoto and K. Hagihara. Near-optimal dynamic task scheduling of precedence constrained coarse-grained tasks onto a computational grid. In *2nd International*

Symposium on Parallel and Distributed Computing, pp. 80-87, Slovenia. IEEE, oct 2003.

[50] P. W. Goldberg. Bounds for the convergence rate of randomized local search in a multiplayer load-balancing game. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 131–140, 2004.

[51] A. Goldchleger, F. Kon, A. Goldman, M. Finger, and G. C. Bezerra. InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines. *Concurrency and Computation: Practice and Experience*, 16(5):449–459, March 2004.

[52] A. Gounaris, R. Sakellariou, N. W. Paton, and A. A. A. Fernandes. Resource scheduling for parallel query processing on computational grids. In *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, pp. 396–401, Washington, EUA, 2004. IEEE Computer Society.

[53] A. Gounaris, R. Sakellariou, N. W. Paton, and A. A. A. Fernandes. A novel approach to resource scheduling for parallel query processing on computational grids. *Distributed and Parallel Databases*, 19(2-3):87–106, 2006.

[54] M. Grajcar. Genetic list scheduling algorithm for scheduling and allocation on a loosely coupled heterogeneous multiprocessor system. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, Nova Iorque, EUA, 1999. ACM Press.

[55] T. Hagras and J. Janeček. An approach to compile-time task scheduling in heterogeneous computing systems. In *33rd International Conference on Parallel Processing Workshops*. IEEE Computer Society, ago. 2004.

[56] M. Hakem and F. Butelle. Dynamic critical path scheduling parallel programs onto multiprocessors. In *19th International Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2005.

[57] M. Harchol-Balter and A. B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems*, 15(3):253–285, 1997.

[58] E. S. H. Hou, N. Ansari, and H. Ren. A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed System*, 5(2):113–120, 1994.

[59] R. Jain, D. Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical Report TR-301, DEC Research, set. 1984.

[60] J.Yu, R. Buyya, and K. Ramamohanarao. Workflow scheduling algorithms for grid computing. *Studies in Computational Intelligence*, 146:173–214, 2008.

[61] C. Kim and H. Kameda. An algorithm for optimal static load balancing in distributed computer systems. *IEEE Transactions on Computers*, 41(3):381–384, 1992.

[62] Y.-K. Kwok and I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):506–521, 1996.

[63] Y.-K. Kwok and I. Ahmad. Benchmarking the task graph scheduling algorithms. In *IPPS/SPDP - 12th International Parallel Processing Symposium & 9th Symposium on Parallel and Distributed Processing*, mar. 1998.

[64] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.

[65] Y.-K. Kwok, K. Hwang, and S. Song. Selfish grids: Game-theoretic modeling and nas/psa benchmark evaluation. *IEEE Transactions on Parallel and Distributed Systems*, 18(5):621–636, 2007.

[66] B.-S. Lee, M. Tang, J. Zhang, O. Y. Soon, C. Zheng, P. Arzberger, and D. Abramson. Analysis of jobs in a multi-organizational grid test-bed. In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, Washington, EUA, mai. 2006. IEEE Computer Society.

[67] H.-C. Lin and C. S. Raghavendra. A dynamic load-balancing policy with a central job dispatcher (LBC). *IEEE Transactions on Software Engineering*, 18(2):148–158, 1992.

[68] C. A. Lindley. *Practical image processing in C: acquisition, manipulation and storage: hardware, software, images and text*. John Wiley & Sons, Inc., Nova Iorque, EUA, 1991.

[69] K. Lu, R. Subrata, and A. Y. Zomaya. Towards decentralized load balancing in a computational grid environment. In *Advances in Grid and Pervasive Computing, First International Conference, GPC 2006*, pp. 466-477, Taichung, Taiwan, 2006.

[70] J. MacLaren, R. Sakellariou, K. T. Krishnakumar, J. Garibaldi, and D. Ouelhadj. Towards service level agreement based scheduling on the grid. In *ICAPS '04: Proceedings of The 14th International Conference on Automated Planning and Scheduling*, Whistler, Canadá, 2004.

[71] M. Maheswaran and H. J. Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *In Seventh Heterogeneous Computing Workshop*, pages 57–69. IEEE Computer Society Press, 1998.

[72] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus open-source cloud-computing system. In *9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID 09)*, mai. 2009.

[73] C. Papadimitriou. Algorithms, games, and the internet. In *STOC '01: Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 749–753, 2001.

[74] C. H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal on Computing*, 19(2):322–328, 1990.

[75] C. Phillips, C. Stein, and J. Wein. Task scheduling in networks. *SIAM Journal on Discrete Mathematics*, 10(4):573–598, 1997.

[76] M. L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer Publishing Company, Incorporated, 2008.

[77] R. Prodan and T. Fahringer. Dynamic scheduling of scientific workflow applications on the grid: a case study. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, Santa Fe, New Mexico, mar. 2005. ACM Press.

[78] L. Qi, H. Jin, I. Foster, and J. Gawor. Provisioning for dynamic instantiation of community services. *IEEE Internet Computing*, 12(2):29–36, 2008.

[79] L. Qi, H. Jin, I. T. Foster, and J. Gawor. Hand: Highly available dynamic deployment infrastructure for globus toolkit 4. In *15th Euromicro IPDP*, Nápoles, Itália, fev. 2007. IEEE Computer Society.

[80] M. Rahman, S. Venugopal, and R. Buyya. A dynamic critical path algorithm for scheduling scientific workflow applications on global grids. In *E-SCIENCE '07: Proceedings of the Third IEEE International Conference on e-Science and Grid Computing (e-Science 2007)*, Washington, EUA, 2007. IEEE Computer Society.

[81] A. Riska, E. Smirni, and G. Ciardo. Analytic modeling of load balancing policies for tasks with heavy-tailed distributions. In *WOSP '00: Proceedings of the 2nd international workshop on Software and Performance*, Ottawa, Canadá, set. 2000.

[82] R. Sakellariou and H. Zhao. A hybrid heuristic for dag scheduling on heterogeneous systems. In *13th Heterogeneous Computing Workshop, International Parallel and Distributed Processing Symposium (IPDPS'04) Workshops*. IEEE Computer Society, abr. 2004.

[83] B. Schulze, G. Coulson, R. Nandkumar, and P. Henderson. Special issue: Middleware for grid computing: A possible future: Editorials. *Concurrency and Computation: Practice and Experience*, 19(14):1879–1884, 2007.

[84] C. R. Senna, L. F. Bittencourt, and E. R. M. Madeira. Execution of service workflows in grid environments. In *TRIDENTCOM - International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities*, Washington, EUA, abr. 2009. IEEE Computer Society.

[85] C. R. Senna, L. F. Bittencourt, and E. R. M. Madeira. Uma infra-estrutura para execucao dinamica de servicos em grades computacionais. In *XXVII Simpósio Brasileiro de Redes de Computadores*, Recife, Brasil, mai. 2009.

[86] C. R. Senna, L. F. Bittencourt, and E. R. M. Madeira. Execution of service workflows in grid environments (aceito para publicação). *International Journal of Communication Networks and Distributed Systems*, 2010.

[87] C. R. Senna, L. F. Bittencourt, and E. R. M. Madeira. Performance evaluation of virtual machines in a service-oriented grid testbed (aceito para publicação). In *International Conference on High Performance Computing & Simulation (HPCS 2010)*, Caen, França, Jun. 2010. IEEE Computer Society Press.

[88] C. R. Senna and E. R. M. Madeira. A middleware for instrument and service orchestration in computational grids. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGRID'07)*, Rio de Janeiro, Brasil, 2007. IEEE Computer Society.

[89] B. Simion, C. Leordeanu, F. Pop, and V. Cristea. A hybrid algorithm for scheduling workflow applications in grid environments (icpdp). In *On the Move Conferences*, volume 4808 of *LNCS*, pp. 1331-1348, Vilamoura, Portugal, nov. 2007. Springer.

[90] J. E. Smith and R. Nair. The architecture of virtual machines. *IEEE Computer*, 38(5):32–38, 2005.

[91] R. Subrata, A. Y. Zomaya, and B. Landfeldt. Game-theoretic approach for load balancing in computational grids. *IEEE Transactions on Parallel and Distributed Systems*, 19(1):66–76, 2008.

[92] X.-H. Sun and M. Wu. Grid harvest service: A system for long-term, application-level task scheduling. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, Nice, França, 2003. IEEE Computer Society.

[93] I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields. *Workflows for e-Science. Scientific Workflows for Grids*. Springer, 2007.

[94] D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., 2002.

[95] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[96] V. V. Vazirani. *Approximation Algorithms*. Springer, Mar. 2004.

[97] B. A. Vianna, A. A. Fonseca, N. T. Moura, L. T. Menezes, H. A. Mendes, J. A. Silva, C. Boeres, and V. E. F. Rebello. A tool for the design and evaluation of hybrid scheduling algorithms for computational grids. In *Proceedings of the 2nd workshop on Middleware for grid computing*, pp. 41–46, Toronto, Ontario, Canada, 2004. ACM Press.

[98] M. Wieczorek, S. Podlipnig, R. Prodan, and T. Fahringer. Bi-criteria scheduling of scientific workflows for the grid. In *8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2008)*, Lyon, França, mai. 2008. IEEE Computer Society.

[99] R. Wolski, N. T. Spring, and J. Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.

[100] N. Woo and H. Y. Yeom. k-Depth Look-Ahead Task Scheduling in Network of Heterogeneous Processors. In *International Conference on Information Networking, Wireless Communications Technologies and Network Applications (ICOIN'02)*, Cheju, Coréia, 2002. Springer-Verlag.

[101] A. S. Wu, H. Yu, S. Jin, K.-C. Lin, and G. Schiavone. An Incremental Genetic Algorithm Approach to Multiprocessor Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):824–834, 2004.

[102] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, 1994.

[103] J. Yu and R. Buyya. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Records*, 34(3):44–49, 2005.

[104] J. Yu, R. Buyya, and R. Kotagiri. *Workflow Scheduling Algorithms for Grid Computing*, volume 146 of *Studies in Computational Intelligence*, pages 173–214. Springer, 2008.

[105] L. Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. Qos-aware middleware for web services composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, 2004.

[106] H. Zhao and R. Sakellariou. An Experimental Investigation into the Rank Function of the Heterogeneous Earliest Finish Time Scheduling Algorithm. In *9th International Euro-Par Conference*, pp. 189-194, Klagenfurt, Áustria, ago. 2003.

[107] H. Zhao and R. Sakellariou. Scheduling multiple dags onto heterogeneous systems. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, Rohdes Island, Grécia, abr. 2006. IEEE.

[108] 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. *SIGMOD Records*, 34(3):37–43, 2005.

Apêndice A

Medições nos recursos do LRC

Para realizar simulações de execução de tarefas nos recursos da grade, utilizamos duas abordagens. A primeira foi medir o uso de CPU nos recursos do LRC - Laboratório de Redes de Computadores do IC/UNICAMP. Realizamos medições utilizando o comando UNIX *top*, que apresenta a quantidade de CPU utilizada pelos processos no momento de sua execução. Efetuamos duas medições independentes em diferentes datas. Cada medição teve duração de 7 dias e incluiu 15 recursos computacionais, dos quais 5 eram recursos utilizados como *desktop* e simulações eventuais, enquanto outros 10 eram recursos dedicados exclusivamente à execução de simulações de pesquisas do laboratório. A Figura A.1 mostra a porcentagem de tempo em que cada quantidade de CPU livre foi encontrada no sistema.

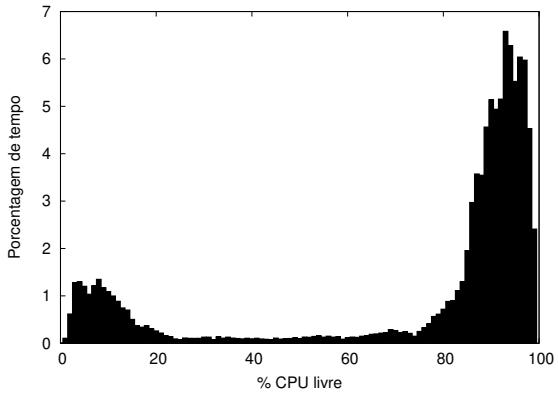


Figura A.1: Medições de CPU livre apresentadas pelos recursos do Laboratório de Redes de Computadores.

A segunda abordagem foi modelar a simulação através de uma estimativa encontrada na literatura [2], em que utilizamos uma distribuição de Poisson para chegada de proces-

sos que têm duração de $\frac{2}{x}$, com x escolhido aleatoriamente no intervalo $(0, 1]$ para cada processo. Note que esse modelo especifica quando um processo externo à grade chega e qual a sua duração, porém não especifica qual a porcentagem de CPU o processo ocupa durante sua execução. Dessa forma, consideramos que quando há um processo externo à grade sendo executado, este deixaria uma porcentagem livre de CPU escolhida aleatoriamente entre 1 e 10. Esses valores foram selecionados baseando-se no gráfico da Figura A.1, que mostra uma concentração nessa faixa de utilização de CPU.

Apêndice B

Procedimento para Gerar DAGs aleatoriamente

Apresentamos neste apêndice o método utilizado para gerar os gráficos aleatórios escalonados nos experimentos. Primeiramente são criados um nó de entrada único e um nó de saída único para, em seguida, gerar os nós internos do DAG. A abordagem para gerar um DAG aleatório com N nós é mostrada no Algoritmo 17.

Em termos gerais, o método gera um DAG em níveis, escolhendo a cada nível um número aleatório entre 1 e $\frac{N-tamanho(\mathcal{G})}{fator}$, $1 \leq fator \leq 3$. O objetivo da variável aleatória $fator$ é proporcionar a geração de grafos com profundidades diferentes. Em cada nível, os nós têm probabilidade $\frac{1}{2}$ de possuir dependência com cada nó do nível anterior. O algoritmo verifica se cada nó, exceto os nós de entrada e saída, tem pelo menos um arco que se origina nele e pelo menos um que incide nele. Ainda, o algoritmo gera arcos aleatoriamente com probabilidade $\frac{1}{10}$ entre nós com diferença de nível maior que 1.

Algoritmo 17 Gera DAG $\mathcal{G}$ aleatório com N nós

Cria nós $t_{entrada}$ e $t_{saída}$.
$nivel \Leftarrow 1$
while $tamanho(\mathcal{G}) < N$ **do**
  $fator \Leftarrow rand(1, 3)$
  $limite \Leftarrow \frac{N - tamanho(\mathcal{G})}{fator}$
  $k \Leftarrow rand(1, limite)$
  $V_{nivel} \Leftarrow k$ novos nós
  for all $n \in \mathcal{V}_{nivel}$ **do**
    for all $m \in \mathcal{V}_{nivel-1}$ **do**
      if $nivel = 1$ **then**
        Cria arco $e_{n,m}$
      else
        Cria arco $e_{m,n}$ com probabilidade $\frac{1}{2}$
      end if
    end for
    if não há arco incidindo em n **then**
      cria arco $e_{m,n}$
    end if
  end for
  $nivel \Leftarrow nivel + 1$
end while
for all $n \in \mathcal{V}_{nivel-1}$ **do**
  Cria arco $e_{n,saída}$
end for
for all $m \in \mathcal{G}$ **do**
  if não há arco com origem em m **then**
    Seleciona $n \in \mathcal{G}$ aleatoriamente tal que $nivel_n = nivel_m + 1$
  end if
  Seleciona $n \in \mathcal{G}$ aleatoriamente tal que $nivel_n > nivel_m + 1$
  Cria arco $e_{m,n}$ com probabilidade $\frac{1}{10}$
end for

Índice Remissivo