Escalonamento Dinâmico para Aplicações Autonômicas MPI em Grades Computacionais

Aline de Paula Nascimento

Tese de Doutorado

Escalonamento Dinâmico para Aplicações Autonômicas MPI em Grades Computacionais

Aline de Paula Nascimento

Tese de Doutorado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito para a obtenção do título de Doutor. Área de concentração: Processamento Paralelo e Distribuído.

Orientadores
Vinod Rebello, PhD
Cristina Boeres, PhD

Niterói, 16 de maio de 2008

Para mamãe e para Xande.

Agradecimentos

A Deus, por tornar tudo possível. Por me dar a confiança e a sabedoria necessárias para a realização deste trabalho. Por permitir que os momentos difíceis se tornassem lembranças distantes e que os momentos de felicidade fossem muito festejados. Obrigada pelo meu bem mais precioso: minha família, aonde encontro apoio, amor e alegria para caminhar com segurança nas trilhas da vida.

Ao Xande, por estar sempre ao meu lado me incentivando e me dando forças em todos os momentos. Obrigada por ser a minha melhor companhia e por me ajudar a ter disposição para cumprir esta tão longa jornada.

À minha mãe, por torcer e por rezar por mim em todos os dias da minha vida. Por sofrer com meus tropeços, mas acima de tudo por vibrar com minhas vitórias. Obrigada, principalmente, pelo seu amor incondicional.

Às minhas irmãs queridas, Gisele e Simone, que estão sempre por perto.

Aos meus sobrinhos que fazem a vida ser muito mais divertida.

Àqueles que não estão mais por aqui, mas que deixaram lembranças e sorrisos.

Aos meus orientadores, Cristina e Vinod, pela paciência, dedicação, incentivo e acima de tudo por serem mais do que professores, por serem amigos.

A todos os familiares, amigos e professores que de alguma forma contribuíram para a realização deste trabalho. Obrigada a todos que durante estes cinco anos compreenderam minhas ausências, acreditaram em meu potencial, tornaram os dias no laboratório da UFF mais alegres, ajudaram nas minhas dificuldades, dissiparam as minhas dúvidas, enfim, que colaboraram para que este sonho se tornasse realidade.

"*Esse é o dia que o Senhor fez: seja para nós dia de alegria e de felicidade.*"
(Salmos, 117, 24)

Resumo

Existem muitas aplicações da física, biologia, engenharia, entre outras áreas que requerem computação intensiva ou que trabalham com grandes conjuntos de dados para sua execução. Tais aplicações, estudadas no contexto do *e-Science*, necessitam de ambientes de execução de larga escala que forneçam grande poder computacional e grande espaço de armazenamento. Desta forma, o uso de plataformas como as Grades Computacionais vem sendo cada vez mais difundido, onde é possível oferecer um alto poder computacional a um baixo custo. Porém, as aplicações existentes necessitam ser adaptadas pelos pesquisadores para execução nas grades computacionais para que se alcance um maior desempenho. Como ambientes grades possuem natureza heterogênea, dinâmica, compartilhada e distribuída, à medida que o número de aplicações que compartilham a grade aumenta, a utilização eficiente dos seus recursos se torna cada vez mais necessária e ao mesmo tempo complexa. O desafio de aproveitar ao máximo a capacidade de processamento oferecida por uma grade é ainda maior quando os seus recursos são utilizados por pesquisadores que não possuem conhecimento, nem habilidade para gerenciar ambientes computacionais complexos. O ideal é que as dificuldades existentes sejam transparentes ao usuário. Uma abordagem promissora para o gerenciamento de grades computacionais de grande escala é criar aplicações que sejam capazes de se autogerenciar, se ajustando às mudanças que ocorram no ambiente de maneira que a sua execução seja eficaz e segura. Para isso, estas aplicações autonômicas devem ter conhecimento de informações atualizadas sobre seu próprio comportamento, as suas características internas e do ambiente de execução, sendo dotadas de funcionalidades que permitam o auto-escalonamento e a auto-recuperação de falhas. Esta tese estuda o problema de escalonamento de tarefas em grades computacionais, propondo uma estrutura de escalonamento dinâmico que permite a execução eficiente de aplicações autonômicas MPI. É considerado um sistema gerenciador de aplicações, hierárquico e distribuído que, para facilitar o desenvolvimento por parte dos pesquisadores, é automaticamente embutido em aplicações MPI. A estrutura de escalonamento é validada através da criação de heurísticas de escalonamento dinâmico, projetadas para aplicações paralelas com ou sem relações de precedência entre suas tarefas. As políticas de escalonamento apresentadas são específicas para cada tipo de aplicação, aumentando assim, o seu desempenho computacional. Foram realizadas análises em ambientes de grades reais e os resultados obtidos mostram a eficiência e a escalabilidade da abordagem proposta.

Abstract

Motivated by the growing interest in e-Science, there are now many applications in areas such as biology, physics and engineering that need large scale execution environments that provide high computational power and storage capacity. Consequently, computational grids are emerging as the platforms of choice to meet these requirements at low cost. The rapid growth in the popularity of grids means a higher number of applications competing for limited resources, so their efficient utilization will therefore become one of the key issues in the success of grids. As computational grids are in essence heterogeneous, dynamic, shared and distributed environments, managing these kinds of platforms efficiently is extremely complex but necessary. Unfortunately, few transparent grid management systems have been developed, therefore both new and existing applications must be adapted by scientists to execute specifically in these environments. Furthermore, the challenge of extracting higher performance from grid resources is more acute for non-expert users, given their lack of familiarity with the intricacies and complexities of computational grids. A promising approach to address these issues is the development of self-managing or autonomic applications. By considering up to date information about their own behavior and environmental conditions, autonomic applications are capable of adapting their execution accordingly in response to changes. This thesis studies the problem of self-scheduling in computational grids and proposes a dynamic scheduling infrastructure that permits the efficient execution of autonomic MPI applications. A hierarchical and distributed application management system (*AMS*), that is automatically embedded into existing MPI applications, transparently, is used to avoid the need for users to modify their programs for grid environments. The scheduling infrastructure is validated through the creation of novel dynamic scheduling heuristics designed for parallel applications with or without precedence constraints among their tasks. The scheduling policies presented are application specific, which increases application performance over traditional resource based management systems which only employ generic heuristics. Numerous experiments and aspects have been analysed in a real grid environment and highlight the quality, efficiency and scalability of the proposed infrastructure.

Sumário

1 Introdução **1**

1.1 Motivação 3

1.2 Contribuições 5

1.3 Organização 7

2 Taxonomia e Trabalhos Relacionados **9**

2.1 Taxonomia para Sistemas Gerenciadores de Grades 10

2.1.1 Panorama da Taxonomia 15

2.1.2 Projetos Relacionados 18

2.1.3 Classificação Geral 29

2.2 EasyGrid AMS 30

2.2.1 Arquitetura de um Processo Gerenciador 30

2.2.2 Estrutura de Gerenciamento do AMS 31

2.2.3 O Portal *EasyGrid* 32

2.2.4 Gerenciamento de Processos 32

2.2.5 Mensagens de Monitoramento 33

2.3 Resumo 34

3 Escalonamento de Tarefas **35**

3.1 Definição do Problema 35

3.2 Modelo de Escalonamento 37

3.2.1 Modelo de Aplicação 37

3.2.2 Modelo Arquitetural 37

3.2.3 Modelagem da Aplicação e da Arquitetura no EasyGrid AMS 40

3.3 Definições Importantes 41

3.4 Heurísticas de Escalonamento Consideradas 44

3.5 Resumo 47

4 Estrutura de Escalonamento Proposta **49**

4.1 Estrutura Hierárquica de Escalonamento 49

4.1.1 Escalonador Dinâmico da Máquina - HDS 51

4.1.2 Escalonador Dinâmico do Site - SDS 55
4.1.3 Escalonador Dinâmico Global - GDS 58
4.2 Resumo 63

I Aplicações *Bag-of-Tasks* **65**

5 Heurística de Escalonamento *BoT* **67**
5.1 Escalonamento Estático 67
5.2 Escalonamento Dinâmico 68
5.2.1 Considerações do Escalonamento Dinâmico 69
5.2.2 Escalonamento Dinâmico na Máquina 72
5.2.3 Escalonamento Dinâmico no Site 73
5.2.4 Escalonamento Dinâmico Global 75
5.2.5 Exemplo de Escalonamento BoT 77
5.3 Resumo 83

6 Análise Experimental **85**
6.1 Ambiente de Avaliação 85
6.2 Aplicações Avaliadas 86
6.3 Métricas de Avaliação e Parâmetros de Execução 87
6.4 Experimentos e Avaliação dos Resultados 88
6.4.1 Análise do Desvio Padrão 88
6.4.2 Benefícios da Política de Escalonamento Híbrido 92
6.4.3 Escalonamento Dinâmico Distribuído 94
6.4.4 Avaliação de Conflitos 98
6.4.5 Avaliação da Estrutura do AMS 100
6.4.6 Grade Computacional Compartilhada e Dinâmica 103
6.5 Resumo 105

II Aplicações com Relações de Precedências **107**

7 Heurística de Escalonamento *GAD* **109**
7.1 Escalonamento Estático 109
7.2 Escalonamento Dinâmico 111
7.2.1 Considerações do Escalonamento Dinâmico 111
7.2.2 Escalonamento Dinâmico na Máquina 115
7.2.3 Escalonamento Dinâmico no Site 116
7.2.4 Escalonamento Dinâmico Global 122
7.2.5 Exemplo de Escalonamento GAD 125

7.3 Resumo 133

8 Análise Experimental **135**

8.1 Ambiente de Avaliação 135
8.2 Aplicações Avaliadas 136
8.3 Métricas de Avaliação e Parâmetros de Execução 138
8.4 Experimentos e Avaliação dos Resultados 139
8.4.1 Avaliação de políticas de escalonamento local 139
8.4.2 Intrusão da heurística de escalonamento local 161
8.4.3 Avaliação da política de escalonamento local proposta 163
8.4.4 Avaliação da política de escalonamento global proposta 167
8.4.5 Grade Computacional Compartilhada e Dinâmica 172
8.5 Resumo 174

9 Conclusões e Trabalhos Futuros **175**

Capítulo 1

Introdução

Com a promessa de alto poder computacional a um baixo custo, as grades computacionais [51, 52, 53] vêm se tornando cada vez mais populares. As grades são tipicamente compostas por diversos recursos heterogêneos que podem estar distribuídos geograficamente em qualquer lugar, podendo pertencer a diversas organizações com políticas de acesso distintas e estarem interconectados por redes de capacidades variadas. Além disso, ambientes grades são compartilhados entre diferentes usuários e possuem natureza dinâmica, onde recursos podem se ligar ou desligar da rede sem aviso prévio. Todas estas características fazem com que o gerenciamento eficiente dos recursos de uma grade seja difícil.

O conceito de grades computacionais é análogo ao das redes de energia elétrica [51], que visam fornecer energia de acordo com a demanda de qualquer usuário que necessite dela, não importando a sua localização. Assim, o sucesso das grades está associado ao fato de ser possível oferecer desempenho computacional eficiente para usuários que possam se conectar a partir de qualquer ponto de acesso.

Um dos grandes desafios é projetar sistemas gerenciadores que sejam capazes de lidar eficientemente com as complexidades impostas pelas grades computacionais e permitir ao usuário um acesso simples aos recursos do sistema. Os programas gerenciadores de uma grade devem oferecer uma visão unificada do sistema, sendo responsáveis entre outras funções, por distribuir a aplicação de um usuário entre os diferentes recursos, proporcionando uma execução rápida, barata e segura. O aumento da popularidade das grades implica em um maior número de aplicações competindo pelos mesmos recursos. Logo, o ideal é que um usuário não se preocupe em saber quais dos recursos disponíveis foram alocados para a execução de sua aplicação, sendo isto feito de maneira transparente.

Atualmente, dada a grande importância destes sistemas, uma quantidade significativa de pesquisa vem sendo investida no estudo de problemas relacionados às grades computacionais. Uma das principais metas é buscar soluções para o gerenciamento das aplicações paralelas e/ou distribuídas e dos recursos oferecidos pela grade, criando *middlewares* espe-

cializados, capazes de descobrir, acessar e utilizar eficientemente os recursos disponíveis e completar com sucesso a execução de tais aplicações.

Até hoje, três filosofias de implementação de sistemas gerenciadores podem ser destacadas: Sistemas gerenciadores de recursos, *RMS (Resource Management Systems)*, Sistemas gerenciadores de usuários, *UMS (User Management Systems)* e Sistemas gerenciadores de aplicações, *AMS (Application Management Systems)*. Um *RMS* adota uma visão direcionada ao sistema (*system-centric*), sendo normalmente centralizado. Seu objetivo é maximizar a utilização dos recursos entre várias aplicações que compartilham a grade computacional. Por outro lado, um *middleware* do tipo *UMS* adota uma visão direcionada ao usuário (*user-centric*), sendo responsável por gerenciar as aplicações de apenas um usuário por vez, de acordo com os seus requisitos. Um sistema gerenciador de aplicação, *AMS*, possui propriedades similares a um *UMS* com a principal diferença de transformarem as aplicações do usuário em versões *system aware*, tornando-as cientes dos recursos necessários, podendo assim, se auto-ajustar às mudanças que ocorram na grade computacional sem a interferência do usuário. Outra vantagem é que sistemas do tipo *AMS* podem ser embutidos na aplicação ao invés de instalados nos recursos da grade, permitindo melhor portabilidade.

O *EasyGrid AMS* [22] utilizado neste trabalho é um sistema gerenciador de aplicações MPI [87], que pode ser afinado de acordo com as necessidades específicas de cada aplicação. Somente os serviços básicos de grade, como Globus [50] e a biblioteca de comunicação MPI devem estar disponíveis nos recursos da grade computacional. O *EasyGrid AMS* é automaticamente embutido na aplicação do usuário, permitindo que um grande número de aplicações MPI já existentes possam executar na grade sem a necessidade de adaptação do código do usuário.

A medida que a utilização das grades computacionais continuar a crescer, a escalabilidade e portabilidade de um *AMS* podem trazer inúmeros benefícios. Outra característica importante em um sistema gerenciador de grades é o seu grau de autonomia, isto é, a sua capacidade de autogerenciamento. Um sistema desenvolvido nos padrões da computação autônoma (*autonomic computing*) [66] é capaz de se automonitorar, ficando ciente do seu estado e do ambiente de execução onde se encontra. Sendo assim, um sistema autonômico é capaz de ajustar-se tanto às mudanças do seu comportamento quanto às variações do ambiente.

Esta tese se concentra na camada responsável pelo escalonamento dinâmico oferecido pelo *EasyGrid AMS*, a qual visa fornecer mecanismos para que as aplicações paralelas sejam capazes de se auto-otimizar, o que constitui uma das propriedades necessárias para que as aplicações se tornem autonômicas. Diferentemente de estratégias comuns de escalonamento dinâmico (que possuem comportamento *reativo*), o escalonamento aqui adotado propõe uma nova abordagem denominada *pró-ativa*. No escalonamento *pró-ativo* os processos que ainda não se encontram em execução são reescalonados sem que haja necessidade de

esperar por pedidos feitos por recursos que se tornam ociosos, minimizando a sobrecarga de escalonamento.

De uma maneira geral, no problema do escalonamento de tarefas, um dos objetivos principais é distribuir as tarefas de uma aplicação entre um conjunto de processadores com memória distribuída de modo que se minimize o tempo total de execução (*makespan*). Para encontrar soluções ótimas para tal problema podem ser necessários algoritmos com tempos exponenciais [95, 105], logo, um vasto estudo vem sendo realizado para que se produzam heurísticas que encontrem bons tempos de escalonamento em tempos computacionais aceitáveis [12, 16, 43, 77].

Muitas variações podem ser consideradas ao se tratar do problema de escalonamento de tarefas, variantes estas que podem estar relacionadas ao modelo de aplicação ou à arquitetura alvo. Logo, existem heurísticas que podem se adaptar melhor a uma classe de aplicação do que a outra [23], daí a importância de se trabalhar com um *middleware* onde se possa escolher políticas de escalonamento apropriadas. Além disso, para uma maior eficiência, é importante que o *middleware* forneça para o escalonador, informações atualizadas sobre o estado da grade computacional.

Antes de se propor uma heurística eficiente, que considere tanto a topologia de uma aplicação como as características dos recursos disponíveis na grade computacional, é preciso definir o modelo de aplicação e o modelo arquitetural. Neste trabalho, as aplicações paralelas são representadas por grafos acíclicos direcionados, e a modelagem do sistema é dada através de índices que basicamente indicam a capacidade computacional de cada processador e a velocidade dos canais de comunicação.

1.1 Motivação

As grades computacionais são ambientes que se tornaram atraentes por oferecerem grande poder computacional a um baixo custo. Várias aplicações paralelas e distribuídas que exigem ambientes computacionais de alto desempenho para que seus dados sejam processados em um tempo razoável são estudadas na área de *e-Science*. Tais aplicações são desenvolvidas em diferentes áreas da ciência como biologia, física e engenharia, podendo ter os seus códigos escritos por cientistas que não sejam necessariamente especialistas em computação. Satisfazer os requisitos de inúmeras e diferentes aplicações que executam nas grades computacionais requer o desenvolvimento de sistemas gerenciadores cada vez mais sofisticados.

Para o desenvolvimento de aplicações paralelas, a maioria dos cientistas optou pelo paradigma em que processos da aplicação se comunicam através de troca de mensagens, e a biblioteca de comunicação MPI se tornou um padrão [48]. Portanto, muitas aplicações MPI já existem como candidatas potenciais para execução em grades computacionais [73, 94]. O MPICH-G2 [88] é uma implementação que agrega serviços oferecidos pelo Globus *toolkit*

para permitir a execução das aplicações paralelas dos usuários em ambientes grades, sem a necessidade de modificação do código original. Entretanto, apesar de tal facilidade, as funcionalidades oferecidas pela biblioteca pressupõem que o ambiente de execução é estável, dedicado e homogêneo. Mesmo implementações mais recentes da biblioteca MPI que já consideram ambientes heterogêneos, como o OpenMPI [55, 58], continuam exigindo esforço do programador para que suas aplicações sejam adaptadas para execução em uma grade computacional.

Como as grades são ambientes dinâmicos, compartilhados e heterogêneos, é necessário o desenvolvimento de sistemas gerenciadores que permitam que as aplicações MPI possam se adaptar dinamicamente às mudanças da grade. Assim, uma grande motivação é fornecer às aplicações MPI já existentes uma plataforma de execução simples, eficiente e confiável, onde as aplicações sejam capazes de se autogerenciar. Apesar da adaptação automática de uma aplicação para execução na grade computacional ser uma grande vantagem, isto pode ainda não ser um atrativo decisivo para os pesquisadores. É preciso que, além disso, se ofereça um ambiente de computação seguro e eficiente.

Entretanto, devido às características das grades, em grande parte dos projetos desenvolvidos, a maioria das aplicações necessita sofrer alterações significativas em seus códigos originais para que a execução nos recursos disponíveis seja eficiente. Assim, este esforço extra exigido dos pesquisadores para conseguirem executar seus programas nas grades computacionais pode se tornar um fator inibidor do seu uso. Logo, como a simplicidade e a facilidade são chaves para o sucesso de qualquer tecnologia nova, é necessária a existência de sistemas ou serviços que permitam que as aplicações do usuário sejam capazes de executar sem que os cientistas tenham que fazer mudanças em seus códigos fontes ou até mesmo reescrevê-los. O ideal é que o grande número de aplicações MPI já desenvolvidas para *clusters* de computadores possam ser aproveitadas para execução em ambientes grades, o que seria um grande atrativo para os pesquisadores de diferentes áreas.

Além disso, como os ambientes grades são compostos por recursos que podem pertencer a diversas organizações, é importante que os sistemas gerenciadores sejam capazes de se adaptar a regras de acesso distintas impostas por cada organização. Logo, uma outra motivação é definir uma estrutura de gerenciamento, no caso deste trabalho, de escalonamento, que possa se ajustar mais facilmente às diferentes políticas existentes no ambiente de execução das grades computacionais.

Outro problema a ser considerado é como gerenciar de maneira eficiente as aplicações de cada usuário na grade. Para que as características internas, os requisitos e o paralelismo de uma aplicação possam ser explorados da melhor maneira possível, ela deve ser gerenciada individualmente. Porém, sabendo que a grade é um ambiente distribuído e compartilhado um gerenciamento que trate a existência de várias aplicações também é necessário. Contudo, um gerenciador centralizado com uma visão global dos recursos e das aplicações da grade pode não ser uma boa estratégia para ambientes de larga escala.

Assim, o desenvolvimento de uma abordagem escalável e eficiente que considere o gerenciamento interno dos processos de uma única aplicação e ao mesmo tempo o gerenciamento distribuído entre diversas aplicações é essencial.

1.2 Contribuições

As políticas de escalonamento desenvolvidas no contexto do *EasyGrid AMS* são específicas para cada tipo de aplicação e o objetivo final desta tese é propor uma estrutura de escalonamento eficiente e flexível para qualquer classe de aplicação paralela que possa ser representada por grafos acíclicos direcionados. É importante ressaltar, que o estudo e a criação de políticas de escalonamento eficientes foram necessárias para a validação da estrutura proposta.

Para otimizar a execução de uma aplicação é desenvolvido um escalonador dinâmico ciente das variações da grade e das características das aplicações. Em busca de uma maior portabilidade e melhor desempenho, as funcionalidades do escalonador são embutidas nas aplicações MPI, de forma que a própria aplicação seja capaz de se auto-escalonar (*self-scheduling*), segundo suas próprias políticas de escalonamento. Um outro ponto importante é que a existência do escalonamento dinâmico também dá suporte ao sistema de tolerância a falhas, o qual proporciona uma execução segura.

Para uma maior eficiência, o sistema de escalonamento elaborado é hierárquico, onde é possível empregar heurísticas de escalonamento diferentes para cada nível da hierarquia, de acordo com os requisitos de cada organização que oferece recursos para a grade. Assim, para cada aplicação, é criada uma hierarquia de escalonadores onde é possível definir políticas para cada recurso e para cada site que compõem a grade computacional, obedecendo suas regras de acesso. O impacto causado pela utilização de uma estrutura de escalonamento hierárquica (dentro da aplicação) e distribuída (no contexto de várias aplicações) é analisado, permitindo verificar que além de escalável, tal abordagem também se mostra eficiente.

No sistema de escalonamento proposto nesta tese, a criação de uma estrutura de escalonamento por aplicação tem o objetivo de se adequar melhor ao ambiente distribuído das grades computacionais. A tarefa de escalonamento descentralizada entre várias aplicações permite que seja alcançada uma maior escalabilidade do sistema. Para isso, a criação de aplicações *system aware* é de suma importância. Através do monitoramento do sistema, cada aplicação estará ciente, indiretamente, da existência de outras aplicações na grade, sendo possível coordenar as ações dos diversos escalonadores distribuídos por aplicação, evitando conflitos. Além disso, cada aplicação que executa pode buscar para si a máxima utilização do sistema de acordo com os recursos disponíveis e segundo as suas políticas internas, não sendo prejudicadas por políticas centralizadoras que favoreçam um determinado tipo de aplicação.

Para validação do sistema de escalonamento já implementado, um grande número de experimentos foi realizado considerando heurísticas apropriadas para duas classes distintas de aplicações, que podem ser representadas por grafos acíclicos direcionados (*GADs*). O primeiro conjunto de experimentos considera aplicações cujas tarefas são independentes e não realizam troca de mensagens entre si. Tais aplicações são vastamente executadas em ambientes distribuídos como as grades computacionais. Os resultados obtidos foram comparados com valores teóricos ótimos, pois a comparação justa entre os diversos sistemas de escalonamentos desenvolvidos para grades ainda é um grande obstáculo. Isto ocorre devido à dificuldade em criar ambientes de testes idênticos e em instalar e configurar os diversos sistemas gerenciadores, onde a escolha dos parâmetros de configuração exerce influência direta no desempenho do sistema avaliado. Daí, a importância da comparação realizada neste trabalho, que permite verificar a viabilidade da estrutura de escalonamento desenvolvida no contexto de um *AMS* e avaliar a sobrecarga produzida pela abordagem com relação a um limite inferior.

O segundo conjunto de experimentos trata aplicações com relações de precedências entre suas tarefas e com troca de mensagens entre si. Neste caso, um dos grandes desafios é mostrar que aplicações paralelas podem executar com eficácia em ambientes distribuídos de larga escala como as grades. Para isso, é importante a existência de um sistema gerenciador que considere não somente as características do sistema, mas também as características internas da aplicação. Algumas heurísticas de escalonamento dinâmico encontradas na literatura, direcionadas para aplicações com relações de precedência representadas por *GADs*, foram implementadas no *EasyGrid AMS* e comparadas com heurísticas desenvolvidas nesta tese. A partir dos experimentos realizados é possível analisar o comportamento de cada heurística considerando ambientes de testes heterogêneos e dinâmicos. Alguns dos resultados obtidos nesta pesquisa já foram publicados nos seguintes artigos:

- Managing the Execution of Large Scale MPI Applications on Computational Grids [90];
- Efficient Hierarchical Self-Scheduling for MPI Applications Executing in Computational Grids [21];
- Distributed and dynamic self-scheduling of parallel MPI Grid applications [89];
- Autonomic Application Management for Large Scale MPI Programs [91];
- On the Advantages of an Alternative MPI Execution Model for Grids [104].

Resumidamente, o objetivo desta tese de doutorado é prover aplicações MPI já existentes com funções de escalonamento de baixo custo que permitam que elas sejam auto-ajustáveis às variações da grade. A estrutura do sistema de escalonamento proposta deve ser eficiente e apropriada para um ambiente distribuído, heterogêneo, dinâmico, compartilhado e sujeito a diferentes políticas de acesso como as grades computacionais.

1.3 Organização

Esta tese está dividida da seguinte forma. No Capítulo 2 são relacionados alguns projetos na área de grades computacionais e suas principais características dentro da taxonomia proposta neste trabalho. Além disso, é feita uma breve descrição do *EasyGrid AMS*. No Capítulo 3 é descrito o problema de escalonamento, conceitos básicos e a definição dos modelos de aplicação e arquitetura aqui usados. O Capítulo 4 apresenta a estrutura de escalonamento hierárquica e distribuída desenvolvida, bem como as funções gerais de cada um dos seus escalonadores dinâmicos. A partir daí, a tese é dividida em duas partes distintas de acordo com o tipo de aplicação tratada. A Parte I é dedicada a aplicações *bag-of-tasks*. Nela, o Capítulo 5 descreve o algoritmo de escalonamento implementado e sua filosofia hierárquica e o Capítulo 6 apresenta uma análise dos resultados obtidos. A Parte II tem como foco aplicações com relações de precedências representadas por grafos acíclicos direcionados. O Capítulo 7 descreve a heurística de escalonamento proposta e a interação entre os escalonadores dos diferentes níveis da hierarquia. Os resultados desta parte são analisados no Capítulo 8. Algumas questões relativas a eficiência de um escalonador dinâmico também são discutidas nos Capítulos 5 e 7. As conclusões desta tese e futuras pesquisas são relacionadas no Capítulo 9.

Capítulo 2

Taxonomia e Trabalhos Relacionados

O objetivo deste capítulo é propor uma taxonomia para sistemas gerenciadores de grades, e descrever, dentro deste contexto, as características principais de alguns projetos relacionados com o problema de gerenciamento de aplicações e escalonamento de tarefas em grades computacionais.

Middlewares desenvolvidos para grades podem ser subdivididos em dois níveis: *middlewares básicos* e *middlewares de serviços*. Os *middlewares básicos* têm como principal finalidade permitir o acesso e a execução remota nos recursos da grade, assim como fornecer informações básicas sobre o desempenho destes recursos. Alguns exemplos clássicos são o Globus [50], que oferece serviços como autenticação, segurança, estado e disponibilidade dos recursos e transferência de arquivos, e NWS [115], que oferece serviço de monitoramento, com previsões periódicas do desempenho dos recursos e da rede. Entretanto, *middlewares básicos* não são fáceis de serem usados por usuários que não tenham experiência com grades computacionais, já que eles não fornecem ferramentas que facilitem a execução transparente das aplicações distribuídas em uma grade.

Por outro lado, *middlewares de serviços* possuem o objetivo de dar ao usuário ou a aplicação uma visão unificada da grade, facilitando o desenvolvimento de aplicações e o gerenciamento dos recursos. Sistemas gerenciadores de grades computacionais fazem parte do *middleware de serviços*, sendo responsáveis, de uma maneira geral, por descobrir e fazer acesso aos recursos da grade, mecanismos de tolerância a falhas, escalonamento, submissão de processos, segurança e confiabilidade e outros serviços que objetivam tanto a execução eficiente de uma aplicação quanto a utilização eficiente dos recursos disponíveis.

Da mesma forma que existem *middlewares* criados exclusivamente para grades computacionais, vários outros como Condor [80] e Ganglia [85] foram implementados considerando ambientes mais simples como os *clusters* de computadores. Porém, como a pesquisa desenvolvida nesta tese é direcionada a ambientes grades, a descrição dos trabalhos relacionados, realizada neste capítulo, é restrita apenas aos projetos desenvolvidos dentro do contexto das grades.

Algumas taxonomias para sistemas gerenciadores de grades computacionais já foram propostas: em [75] é apresentada uma taxonomia apenas para sistemas gerenciadores de recursos enquanto que em [117] são classificados somente sistemas gerenciadores de *workflows*, se concentrando em funções específicas que devem ser desempenhadas para o gerenciamento de uma aplicação deste tipo. Um *workflow* é definido como um conjunto de tarefas que são processadas em recursos distribuídos, em uma ordem bem definida, para atingir um objetivo específico [117].

Entretanto, apesar das taxonomias anteriores oferecerem uma descrição detalhada de várias funções do sistema e da organização virtual dos recursos da grade, elas são restritas a certos sistemas gerenciadores. Além disso, não é feita nenhuma consideração em relação aos diferentes tipos de filosofia que podem ser adotadas na implementação dos sistemas gerenciadores. Assim como introduzido no Capítulo 1, os sistemas gerenciadores de grades podem possuir as seguintes filosofias: *UMS*, Sistemas gerenciadores de usuários, *RMS*, Sistemas gerenciadores de recursos e *AMS*, Sistemas gerenciadores de aplicações. Cada filosofia possui objetivos específicos, como por exemplo, satisfazer os requisitos do usuário, maximizar o uso do sistema e executar uma aplicação segundo seus requisitos e topologia interna.

Nesta tese, é sugerida uma taxonomia que permite que sejam definidas algumas propriedades dos sistemas gerenciadores de acordo com o foco de suas filosofias de implementação. O objetivo é estabelecer as vantagens e desvantagens de cada implementação, definindo as facilidades que cada uma pode trazer com relação a sua instalação e utilização, considerando ainda, critérios de escalabilidade e eficiência.

2.1 Taxonomia para Sistemas Gerenciadores de Grades

Neste trabalho, a classificação proposta para sistemas gerenciadores de grades computacionais considera seis características principais: orientação, gerenciamento, escalonamento, instalação, autonomia e tipo de aplicação. Além do detalhamento de cada uma destas características, nesta seção é apresentado um panorama geral com as propriedades típicas dos sistemas do tipo *RMS*, *UMS* e *AMS*.

Orientação

A orientação de um sistema gerenciador indica se suas ações são direcionadas principalmente: pelos requisitos do usuário; ou se elas estão concentradas em características do sistema como maximizar a utilização dos recursos; ou se a gerência considera os requisitos e as topologias de cada aplicação. Assim, a orientação do sistema pode ser definida como *usuário*, *sistema* e *aplicação*, respectivamente.

Sistemas orientados aos usuários (*UMS*) têm o objetivo de gerenciar uma ou várias aplicações de um único usuário, de acordo com os requisitos definidos por ele próprio.

Já, sistemas do tipo *AMS* gerenciam uma só aplicação por vez, trazendo a facilidade de definir políticas de gerenciamento específicas por aplicação de acordo com seus requisitos e características internas. Por outro lado, sistemas gerenciadores do tipo *RMS* possuem uma visão global do sistema, tendo como objetivo gerenciar os recursos do sistema, maximizando a sua utilização entre todas as aplicações que compartilham a grade em um determinado momento. Tais sistemas não consideram características específicas de uma aplicação, nem os requisitos estabelecidos por um usuário.

Gerenciamento

Sistemas gerenciadores de grade oferecem serviços que objetivam uma execução eficiente, segura e transparente para o usuário. Dentre os principais serviços de gerenciamento destacam-se: descoberta e avaliação de desempenho dos recursos, tolerância a falhas, escalonamento de tarefas, submissão de processos e transferência e disponibilização de dados. Nesta taxonomia, o gerenciamento realizado por um *middleware* será dividido em apenas duas categorias: *centralizado* ou *descentralizado*.

O gerenciamento centralizado se caracteriza por possuir um componente central responsável por gerenciar várias aplicações que compartilham os recursos da grade ao mesmo tempo. Por existir um único componente gerenciador para todo o sistema, as mesmas políticas de gerenciamento são aplicadas para diferentes usuários ou aplicações em um dado instante. Além disso, a existência de um componente central para todo o sistema prejudica a escalabilidade do sistema de gerenciamento.

Por outro lado, no gerenciamento descentralizado, as decisões são distribuídas entre vários sistemas gerenciadores que podem executar simultaneamente no ambiente grade, facilitando a existência de políticas específicas para diversos tipos de aplicação ou para diferentes requisitos de usuários. Assim, os sistemas gerenciadores podem ser descentralizados por aplicação, onde cada aplicação possui seu próprio componente de gerenciamento, ou descentralizados por usuário, onde as várias aplicações de um usuário podem ser gerenciadas por um só componente. Como sistemas gerenciadores descentralizados possuem uma visão parcial do sistema, é necessário que suas ações sejam coordenadas através de um monitoramento dos recursos, para que não ocorram conflitos entre as decisões dos vários gerenciadores distribuídos na grade.

Subsistema de escalonamento

Como visto no item anterior, o escalonamento de tarefas é uma das funções desempenhadas no gerenciamento da grade. Apesar disso, ele também será destacado dentro desta taxonomia, visto que o problema de escalonamento de tarefas é um dos pontos fundamentais na operação das grades computacionais e no contexto desta tese. Dois tipos de escalonamento podem ser identificados em ambientes grades: o escalonamento de tarefas de várias aplicações que compartilham a grade (aplicações distribuídas) e o escalonamento

de tarefas de uma mesma aplicação (aplicação paralela). O objetivo é identificar os tipos de subsistemas de escalonamento existentes para as grades.

Independentemente da filosofia de gerenciamento adotada, o subsistema de escalonamento de um *middleware* especializado pode ser estruturado de três formas diferentes: centralizado, hierárquico e descentralizado, assim como apresentado em [75, 117]. A taxonomia aqui proposta identifica apenas a estrutura de escalonamento de um sistema gerenciador e não o momento em que as tarefas são escalonadas, visto que o tipo de escalonamento adotado em ambientes compartilhados e heterogêneos como as grades computacionais é tipicamente dinâmico.

Na organização centralizada, existe apenas um único escalonador que é responsável por alocar e realocar processos da aplicação em todos os recursos da grade computacional. Uma das vantagens desta abordagem é que o escalonador possui conhecimento de todo o ambiente, sendo possível gerar escalonamentos eficientes. Entretanto, apesar de facilitar o gerenciamento dos recursos da grade, a organização centralizada não é escalável e dificulta a utilização de políticas de escalonamento distintas para diferentes tipos de aplicação que possam compartilhar a grade.

A abordagem hierárquica supre algumas desvantagens do esquema anterior ao dividir a tarefa de escalonamento em múltiplos níveis. Assim, torna-se possível o emprego de políticas distintas nos diferentes níveis da hierarquia, além de oferecer uma maior escalabilidade. Nesta organização, existe um escalonador central no nível mais alto da hierarquia que distribui o trabalho entre os escalonadores situados nos níveis mais baixos. A desvantagem é que a falha do escalonador central, assim como na organização centralizada, resulta na falha de todo o sistema.

Por outro lado, escalonadores descentralizados ou distribuídos oferecem naturalmente escalabilidade, tolerância a falhas e a possibilidade de utilizar diversas políticas de escalonamento. Os escalonadores se comunicam uns com os outros, na tentativa de produzir bons escalonamentos e evitar possíveis conflitos. Entretanto, o gerenciamento dos recursos disponíveis se torna mais difícil, onde bons escalonamentos individuais não representam, necessariamente, um desempenho eficiente para todo o sistema.

A taxonomia aqui definida possui uma diferença sutil em relação à classificação apresentada em [75, 117]. Nesta tese, o escalonamento de tarefas de várias aplicações distribuídas na grade é classificado apenas como *centralizado* ou *descentralizado*, sendo tal característica definida pelo ponto de vista da aplicação. Isto é, quando existir um único escalonador responsável pelo escalonamento de tarefas de todas as aplicações que compartilham a grade, ou um grupo delas, o escalonamento é dito centralizado. Porém, quando existirem vários escalonadores distribuídos, com visões parciais do sistema, sendo responsáveis pelo escalonamento de apenas uma aplicação então o escalonamento é descentralizado. Escalonadores descentralizados não necessariamente precisam se comunicar entre si, suas ações podem ser coordenadas pelo monitoramento do ambiente computacional.

Com relação à estrutura interna do escalonador, a organização poderá ser *hierárquica*, ou não. O escalonador hierárquico se caracteriza por possuir suas funções distribuídas entre vários níveis de hierarquia. Logo, quatro categorias de escalonadores são definidas: *centralizado*, *descentralizado*, *centralizado hierárquico* e *descentralizado hierárquico*.

Instalação

A instalação do sistema é outra característica importante, no que diz respeito a sua facilidade de acesso e uso. Sistemas que precisam ser instalados em vários recursos da grade e que necessitam de muitos pré-requisitos tornam a sua distribuição e utilização mais complexa. Este problema se agrava ainda mais, ao serem considerados ambientes de larga escala. Neste taxonomia, os sistemas gerenciadores serão divididos em duas classes principais segundo o esforço para sua instalação: *embutidos* ou *instalados*.

Os sistemas que são embutidos na aplicação exigem um esforço menor para sua utilização. Tais sistemas não precisam ser instalados nos recursos da grade, já que suas funcionalidades podem ser embutidas na aplicação automaticamente ou através de modificações no seu código original. Entretanto, existem sistemas gerenciadores que necessitam da instalação de determinados serviços em vários ou todos os recursos da grade computacional. Para esta classificação, não são consideradas a instalação do Globus *toolkit*, ou das ferramentas de comunicação fornecidas pela biblioteca MPI, visto que estes são requisitos mínimos de vários *middlewares*.

Sistemas gerenciadores que são capazes de embutir suas funcionalidades automaticamente em aplicações paralelas pré-existentes, são os que oferecem aos usuários maior facilidade de utilização. Nestes casos, além das aplicações paralelas não precisarem ser reescritas, um número maior de recursos se torna potencialmente disponível, já que não existe a necessidade de instalação de serviços específicos nos vários recursos da grade.

Autonomia

Os diversos sistemas gerenciadores existentes variam também de acordo com o grau de autonomia que as suas atividades são realizadas. A idéia de computação autônoma (*autonomic computing*) [66, 74] é inspirada no funcionamento do sistema nervoso humano e o objetivo é construir sistemas que sejam capazes de se autogerenciar. O sistema deve decidir por si próprio, o que deve ser feito para que ele se mantenha estável, checando seu estado e adaptando-se automaticamente às possíveis mudanças do ambiente [107]. Sistemas autogerenciáveis se caracterizam por possuírem as seguintes propriedades [96, 107]:

- autoconfiguração (*self-configuring*): capacidade de reconfigurar as características do sistema ou da aplicação, de acordo com condições não previstas;
- auto-recuperação (*self-healing*): habilidade de detectar e recuperar-se de falhas, continuando a execução da aplicação;

- auto-otimização (*self-optimising*): capacidade de detectar um desempenho abaixo do esperado e otimizar-se para garantir uma melhora na execução, dentro da configuração já estabelecida;
- autoproteção (*self-protecting*): habilidade de detectar e proteger seus recursos de possíveis ataques, garantindo a integridade do sistema ou da aplicação.

Para que seja possível a implementação destas propriedades principais, sistemas autogerenciáveis devem possuir os seguintes atributos [107]: automonitoramento (*self-monitoring*), autoconhecimento (*self-awareness*), conhecimento do ambiente (*environment-awareness*) e auto-ajuste (*self-adjusting*). Tais características permitem que o sistema seja capaz de monitorar a si próprio, estando ciente do seu estado e do ambiente de execução, podendo assim, reagir e ajustar-se a possíveis mudanças do seu comportamento ou do próprio ambiente sem a necessidade da interferência do usuário.

Tipos de Aplicações

Grades computacionais são utilizadas para executar aplicações que necessitam de alto desempenho computacional ou que necessitam de grandes repositórios para armazenamento de dados. De uma maneira geral, as grades são conhecidas por oferecerem um ambiente propício para execução de aplicações distribuídas, entretanto aplicações paralelas também podem ser executadas com eficiência em tais ambientes.

A grande maioria dos sistemas gerenciadores existentes só apresenta suporte para aplicações cujas tarefas são independentes entre si. Este tipo de aplicação, conhecida como *bag-of-tasks* (BoT), se caracteriza por não efetuar troca de mensagens entre suas tarefas e é considerado um dos modelos ideais para arquiteturas distribuídas de larga escala como as grades [38]. Apesar de ser um tipo restrito de representação, aplicações do tipo *bag-of-tasks* são usadas em diversas situações, como por exemplo, aplicações *parameter sweep* [33], mestre-escravo [49], mineração de dados e biologia computacional. Um exemplo conhecido da execução de uma aplicação *bag-of-tasks* em uma plataforma distribuída é o projeto SETI@home [7].

Entretanto, existem aplicações paralelas cujas tarefas se comunicam entre si ou que apenas possuem relações de precedências que devem ser respeitadas durante a execução para que se alcance um objetivo específico. Um exemplo comum são as aplicações do tipo *workflow* [117] que podem ou não serem representadas por grafos acíclicos direcionados (*GADs*). Aplicações do tipo *workflow* possuem suas precedências definidas por arquivos comuns de dados que devem ser produzidos por um determinado processo para uso de outro. Existem também as aplicações paralelas cujas dependências são determinadas pelo envio de mensagens de dados [40]. Sistemas gerenciadores que tratam aplicações cujas tarefas possuam algum tipo de relação de precedência possuem um escopo de atuação maior.

Nesta taxonomia, os sistemas gerenciadores de grades também serão classificados de acordo com o tipo de aplicação suportada: aplicações com tarefas independentes entre si ou aplicações com precedências entre as tarefas.

2.1.1 Panorama da Taxonomia

Seguindo os pontos principais da taxonomia, os sistemas gerenciadores de grade podem possuir as características apresentadas na Figura 2.1. O objetivo principal desta seção é criar um panorama geral, indicando quais destas características são encontradas mais frequentemente em sistemas do tipo *AMS*, *UMS* ou *RMS*.

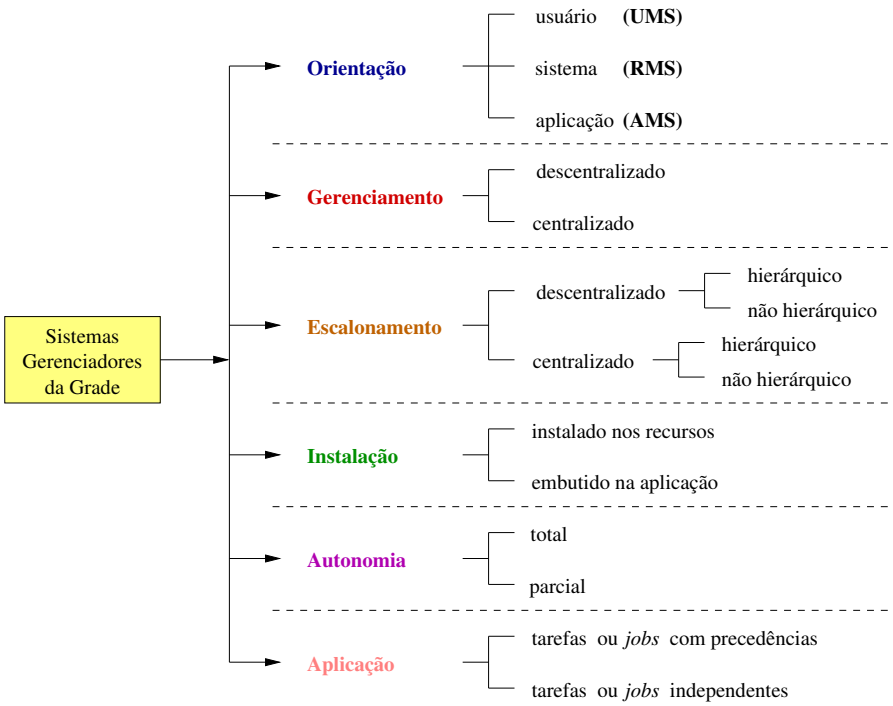

Figura 2.1: Taxonomia para sistemas gerenciadores de grades.

UMS

Um *UMS* possui uma visão direcionada ao usuário, onde a aplicação é gerenciada de acordo com suas características internas e com requisitos indicados pelo próprio usuário, como por exemplo, a execução com tempo limitado ou dentro de um determinado orçamento. Um *UMS* é coletivamente descentralizado entre vários usuários, sendo responsável por gerenciar apenas as aplicações de um usuário por vez. Neste caso, existe um *UMS* por usuário ou para cada uma de suas aplicações, que deve ser instalado em cada recurso de onde a aplicação pode ser disparada.

Para que um *UMS* satisfaça as condições estabelecidas pelo usuário da melhor maneira possível, é importante que as informações sobre as características da aplicação sejam especificadas de forma precisa. Devido à dificuldade do usuário em conhecer detalhes

das características de uma determinada aplicação, a maioria dos sistemas *UMS* foram desenvolvidos para aplicações do tipo *bag-of-tasks*. Como sistemas autogerenciáveis devem ser capazes de tomar decisões sem a influência dos usuários, *UMSs* não são caracterizados como totalmente autônomos.

É usual que o subsistema de escalonamento siga a mesma abordagem do gerenciamento. Logo, sistemas gerenciadores de usuários possuem escalonadores distribuídos por usuário. Entretanto, como descrito anteriormente na taxonomia, o tipo de escalonamento fornecido é definido pelo ponto de vista da aplicação. Logo, se o subsistema de escalonamento for responsável por escalonar várias aplicações de um mesmo usuário ele é dito centralizado. Nos sistemas do tipo *UMS*, os usuários é que devem especificar as políticas de escalonamento que serão adotadas no decorrer da execução de suas aplicações ou adotar a política padrão definida pelo sistema. Sendo assim, na tentativa de minimizar os esforços requeridos pelo usuário ao gerenciar aplicações em um sistema complexo como a grade computacional, podem ser usadas alternativas de implementação como os sistemas gerenciadores do tipo *RMS* ou *AMS*.

RMS

Os *RMSs* compõem a maioria dos sistemas gerenciadores desenvolvidos para grades computacionais, onde o objetivo é maximizar a utilização do sistema, independentemente dos requisitos e características internas das aplicações. *RMSs* são direcionados ao sistema e possuem uma visão global dos recursos da grade computacional sendo, portanto, tipicamente centralizados. Logo, várias aplicações são gerenciadas simultaneamente e o componente responsável pelo escalonamento destas aplicações também é instalado em um servidor central.

Assim como *UMSs*, os serviços necessários para o monitoramento e gerenciamento dos recursos da grade por um *RMS* devem ser previamente instalados em todos os recursos disponíveis, o que pode dificultar a sua distribuição em ambientes de larga escala como as grades computacionais. Com relação à autonomia dos serviços oferecidos, podem ser encontrados sistemas gerenciadores de recursos totalmente ou parcialmente autogerenciáveis. Nos *RMSs* existentes, não há necessidade da interferência dos usuários para que decisões de gerenciamento sejam tomadas, podendo tais funções serem desempenhadas de maneira autônoma. Como será visto no decorrer deste capítulo, na literatura foram encontrados sistemas *RMS* que trabalham com aplicações *bag-of-tasks* e também com aplicações paralelas cujas tarefas possuem relações de precedência.

AMS

Assim como os sistemas gerenciadores de usuários, *AMSs* são coletivamente descentralizados, existindo um único gerenciador por aplicação. Desta forma, as aplicações

podem ser gerenciadas segundo os seus requisitos e sua topologia interna, facilitando o uso de diferentes políticas (específicas para cada tipo de aplicação), que podem ser ajustadas durante a execução. O objetivo de um *AMS* é maximizar o desempenho da aplicação, utilizando os recursos disponíveis eficientemente.

Uma grande vantagem da implementação do tipo *AMS* em relação às outras abordagens é a capacidade de transformar a aplicação do usuário em uma versão *system aware*. Aplicações cientes do estado dos recursos da grade durante a sua execução podem se auto-ajustar de acordo com as mudanças do ambiente, buscando uma maior eficiência na sua execução. Normalmente, para que uma aplicação se torne *system aware*, as funcionalidades do sistema gerenciador são embutidas na própria aplicação de maneira automática, ou através de modificações do código fonte do usuário. Entretanto, é possível que um *AMS* seja instalado nos recursos da grade, neste caso, a aplicação se torna *system aware* requisitando serviços de gerenciamento instalados nos próprios recursos [14, 69].

Sistemas gerenciadores de aplicações oferecem maior escalabilidade, pois o esforço de gerenciar as diversas aplicações é distribuído entre vários *AMS*. Além disso, os *AMSs* automaticamente embutidos permitem que um grande número de aplicações já existentes possam ser executadas nas grades computacionais, sem a necessidade de alterar seus códigos originais. Assim como *RMSs*, os sistemas gerenciadores de aplicações também possuem a capacidade de se autogerenciar. Porém, nem todas as implementações existentes exploram esta característica, existindo sistemas totalmente ou parcialmente autônomos. Com relação ao tipo de aplicação suportada, podem ser encontrados *AMSs* que trabalham com *bag-of-tasks* e/ou com aplicações paralelas com precedência. A diferença em relação aos *RMSs* é que as características internas da aplicação são consideradas na tentativa de se obter um melhor desempenho.

A Figura 2.2 mostra a configuração típica para os sistemas gerenciadores de grades de acordo com suas características principais. Esta é uma classificação flexível, o que permite que em alguns casos, sistemas gerenciadores do tipo *AMS*, *UMS* e *RMS* não se enquadrem em todas as características relacionadas.

	Orientação	**Gerenciamento**	**Escalonamento**	**Instalação**	**Autonomia**	**Aplicação**
UMS	usuário	descentralizado por usuário	centralizado	instalado nos recursos	parcial	tarefas ou *jobs* independentes
RMS	sistema	centralizado	centralizado	instalado nos recursos	total	tarefas ou *jobs* com precedências
AMS	aplicação	descentralizado por aplicação	descentralizado por aplicação	embutido na aplicação	total	tarefas ou *jobs* com precedências

Figura 2.2: Configuração típica para sistemas gerenciadores de grades.

2.1.2 Projetos Relacionados

Muitos projetos na área de gerenciamento de grades computacionais foram desenvolvidos nos últimos anos. Esta subseção tem o objetivo de apontar alguns dos principais sistemas, indicando suas características principais dentro dos aspectos destacados na taxonomia e na estrutura de escalonamento descritas anteriormente.

GrADS

O *framework* GrADS [13, 14, 110] fornece ferramentas para minimizar o esforço de cientistas e engenheiros em executar seus programas em ambientes complexos como as grades computacionais. O gerenciador de aplicações do GrADS possui ferramentas e bibliotecas que permitem que o usuário crie aplicações que possam ser encapsuladas como programas objetos configuráveis (*COPs*), que podem ser otimizados pelo gerenciador para executar em uma coleção específica de recursos [13]. Um *COP* inclui o código da aplicação, um mapeador de tarefas e um modelo que estima o desempenho da aplicação em um conjunto de recursos.

A infra-estrutura do GrADS determina quais são os recursos disponíveis e utiliza o mapeador do *COP* e o modelo de desempenho para escalonar a aplicação em um subconjunto de recursos apropriados. Quando a aplicação é disparada, utiliza-se o monitor de contratos para verificar a viabilidade do contrato de desempenho, considerando os requisitos da aplicação e a capacidade dos recursos. Caso o contrato seja violado, o reescalonador é acionado para propor uma nova alocação da aplicação.

O reescalonamento oportunista também pode ser empregado quando é detectado o término da execução de alguma aplicação da grade. Neste caso, deve ser verificado se a realocação das tarefas de uma determinada aplicação pode trazer benefícios à sua execução. A política de escalonamento empregada no GrADS visa minimizar o tempo final de execução de uma aplicação, e para isso podem ser usadas duas abordagens no reescalonamento. Na primeira delas, *stop and restart*, a aplicação é suspensa e migrada quando recursos melhores forem encontrados, neste caso a aplicação é reiniciada no novo conjunto de recursos a partir do ponto onde foi suspensa. Na segunda abordagem, *process swapping*, processadores mais poderosos são escolhidos para substituírem outros processadores mais lentos que já façam parte da grade virtual da aplicação. Apesar de menos flexível, pois considera apenas o conjunto de recursos previamente definidos para a aplicação, esta abordagem é menos custosa [13].

O escalonador do GrADS cria um modelo dos recursos da grade computacional utilizando serviços para estimativa de desempenho oferecidos pelo Globus MDS e NWS, sendo, portanto, altamente dependente de outros *middlewares*. Na tentativa de evitar conflitos com o reescalonamento de várias aplicações que executam simultaneamente na grade, o GrADS oferece um serviço de reescalonamento central, que deve ser contactado pelos

monitores de contrato das diferentes aplicações [110]. Para cada aplicação, o escalonador do GrADS cria uma matriz que reflete o desempenho de cada tarefa em cada recurso. Tais informações podem ser usadas por heurísticas de escalonamento como *Sufferage*, *Max-Min* e *Min-Min* [13]. O escalonador oferecido pelo GrADS fornece suporte para aplicações do tipo *workflow*.

Neste trabalho, GrADS é classificado como um *AMS*, já que suas características indicam a existência de um gerenciador GrADS por aplicação *system aware*. Além disso, podem ser identificados vários serviços que visam a criação de uma aplicação autônoma como autoconfiguração, auto-recuperação (tolerância a falhas), automonitoramento, monitoramento do ambiente de execução, auto-otimização, entre outros.

AppLeS

AppLeS [15, 34] é um *middleware* específico para escalonamento de tarefas, cujo foco é oferecer escalonamento adaptativo e eficiente para as aplicações que executam na grade computacional. Para cada aplicação, são criados agentes de escalonamento que trabalham tanto com informações do sistema como da aplicação, onde o objetivo é definir uma melhor alocação de tarefas nos recursos disponíveis [35].

Cada aplicação pode possuir políticas de escalonamento específicas, pois o agente de escalonamento é embutido na própria aplicação (*system aware*) para a realização do escalonamento dinâmico. Desta forma, AppLeS é caracterizado como um *AMS*, onde o gerenciamento possui uma visão direcionada para os requisitos e características internas da aplicação. Logo, diferentes *frameworks* podem ser desenvolvidos para cada classe de aplicação, onde heurísticas de escalonamento específicas são fornecidas. O *middleware* AppLeS oferece um *framework* (*APST* [34]) para aplicações do tipo *parameter sweep*, onde o objetivo principal das políticas, como *Max-Min*, *Min-Min*, *Sufferage* e *XSufferage* [34] é minimizar o tempo de execução da aplicação.

Os agentes de escalonamento utilizam serviços oferecidos pelo NWS, para monitoramento dos recursos da grade, e do Globus, para transferência de arquivos e execução de processos. Assim, o escalonador permite que a aplicação seja auto-ajustável, se adaptando automaticamente às mudanças que venham ocorrer na grade computacional. Como cada aplicação possui seu próprio escalonador, pode-se dizer que o escalonamento é descentralizado entre as diversas aplicações que compartilham a grade. A avaliação da necessidade de se realizar um reescalonamento de tarefas ocorre periodicamente. Entretanto, não existe um valor padrão que determine o tamanho do intervalo entre cada avaliação, apenas é citado que o valor determinado influencia diretamente no desempenho do algoritmo de escalonamento [34].

Como o AppLeS é direcionado para serviços de escalonamento, ele depende que outros gerenciadores de recursos desempenhem o gerenciamento dos processos do usuário na

grade. Logo, não se pode dizer que AppLeS é um *middleware* autogerenciável. Apesar de no seu propósito principal, ele oferecer ferramentas que permitam auto-ajuste e auto-otimização da aplicação, não são disponibilizados, por exemplo, serviços de tolerância a falhas (auto-recuperação).

GridWay

O *framework* GridWay [68, 69] foi criado inicialmente para facilitar a execução das aplicações do tipo *parameter sweep* na grade computacional, ajustando o escalonamento e a execução dos processos de acordo com as mudanças do ambiente. O coração do GridWay é um agente personalizado de submissão (*submission agent*) que executa todos os estágios do escalonamento e monitora a execução correta e eficiente dos processos. O *framework* é baseado nos serviços oferecidos pelo Globus, assim como descoberta de recursos, avaliação de desempenho, etc.

A última versão disponível do escalonador do GridWay [60] também oferece suporte para submissão de processos com restrições de dependência. Esta nova funcionalidade permite a execução de aplicações do tipo *workflow*. Além disso, esta versão fornece um conjunto de políticas de escalonamento que podem ser direcionadas para dar diferentes prioridades aos processos da aplicação e aos recursos disponíveis.

O modelo de aplicação padrão requer modificações para que as aplicações se tornem *system aware*. Para isso, no GridWay devem ser especificados os requisitos iniciais da aplicação (*requirement expression*) que poderão ser ajustados durante a execução. Além disso, para criar uma prioridade entre os recursos disponíveis que atendam os requisitos especificados, deve ser indicado um *ranking* para cada recurso (*ranking expression*). Tais especificações no código fonte da aplicação permitem que ela seja auto-ajustável, podendo adaptar-se dinamicamente às mudanças da grade. As políticas de escalonamento podem ser definidas no *ranking expression* onde, por exemplo, uma aplicação *CPU-bound* pode dar maior *ranking* para os recursos que possuam CPUs mais rápidas, enquanto aplicações com computação intensiva de dados podem dar maior prioridade aos recursos que se encontram mais próximos dos dados de entrada.

Para o processo de auto-recuperação de falhas, devem ser gerados arquivos de reinicialização (*checkpointing*), de forma que seja possível recomeçar a execução de um processo a partir de um determinado ponto. Atualmente, as mudanças iniciais no código fonte da aplicação e no *requirement expression* não são feitas automaticamente, o que sugere que não existe a capacidade de autoconfiguração.

Para tratar a dinamicidade da grade computacional e a capacidade de auto-adaptação da aplicação, o *framework* GridWay emprega escalonamento e execução adaptativos. O escalonamento adaptativo aloca processos pendentes, considerando a disponibilidade dos recursos, seus estados, e os processos da aplicação já submetidos. Ele é implementado

por um gerenciador (*dispatch manager*) que recolhe informações periódicas do sistema para escalonar as tarefas da aplicação de acordo com os seus requisitos. Já, a execução adaptativa tem o objetivo de migrar processos em execução para recursos que possam oferecer melhor desempenho. Ela é disparada por eventos criados dinamicamente pela grade ou pela própria aplicação.

A política de escalonamento do GridWay é definida para cada aplicação, segundo suas características e requisitos. Logo, o gerenciamento adotado possui uma visão direcionada para a aplicação *system aware* do usuário, sendo classificado como *AMS*. Na bibliografia pesquisada, existem indicativos da existência de uma infra-estrutura GridWay por aplicação, o que caracteriza um gerenciamento descentralizado. Com relação a capacidade de se autogerenciar, vários módulos implementados no *framework* permitem auto-ajuste, auto-recuperação e auto-otimização. Entretanto, para a aplicação se tornar auto-ajustável é necessário que ocorra uma mudança no código fonte da aplicação, sendo que tal interferência não é feita automaticamente pelo sistema.

Nimrod/G

O sistema Nimrod/G [26] tem o objetivo de gerenciar a execução de aplicações do tipo *parameter sweep* nos recursos da grade segundo os requisitos do usuário, levando em consideração os custos que ele tem para disponibilizar na execução da sua aplicação. Tal tipo de gerenciamento, baseado na economia da grade computacional (*grid economy*), pode trazer alguns benefícios [25] como: ajudar a construir grades computacionais de larga escala, motivando organizações a cederem recursos ociosos para execução de processos de outros domínios; oferecer incentivos para que os usuários utilizem poucos recursos para resolverem problemas de baixa prioridade, encorajando primeiramente a solução de problemas críticos; facilitar o desenvolvimento de políticas de escalonamento voltadas aos requisitos do usuário e não somente do sistema, entre outras vantagens.

Neste trabalho, Nimrod/G é classificado como *UMS*, sendo as políticas de escalonamento baseadas nas escolhas do usuário e diferenciadas por aplicação. A integração do modelo de economia com o sistema de escalonamento influencia a maneira como os recursos são selecionados para satisfazer os requisitos do usuário, como por exemplo, ter uma aplicação executada dentro de um *deadline* ou custo pré-estabelecido. Para isso, o usuário submete um contrato ao sistema que negocia os recursos para que seus processos sejam executados.

Nimrod/G fornece ao usuário uma interface para controle e supervisão dos seus experimentos, de onde o usuário pode variar parâmetros relacionados ao tempo e custo, que influenciam diretamente no escalonamento produzido. Existe um componente central responsável por criar e disparar processos do usuário nos recursos escolhidos. Além disso, tal componente grava o estado dos processos, permitindo que em caso de falhas o processo

seja reexecutado (auto-recuperação).

Nimrod/G possui uma arquitetura hierárquica onde existe um escalonador central no topo da hierarquia, interagindo com escalonadores locais para criar um escalonamento que minimize o custo da computação de acordo com os requisitos dos usuários. Para obter informações do sistema em relação a disponibilidade e poder computacional dos recursos, Nimrod/G utiliza funções fornecidas por *middlewares* básicos como Globus e NWS.

Por ser classificado como um *middleware* do tipo *UMS*, onde os donos das aplicações e os donos dos recursos são responsáveis pelas políticas de escalonamento adotadas, não é possível dizer que Nimrod/G é um sistema autogerenciável. Algumas funções como a capacidade de se auto-recuperar de falhas podem ser encontradas, porém, a maioria das decisões do gerenciador depende da interferência do usuário.

Gridbus

Outro sistema do tipo *UMS*, também dirigido pelos requisitos da economia da grade computacional, é o Gridbus [29, 111]. Porém, diferentemente do Nimrod-G, ele é direcionado para a execução de aplicações que possuam computação intensa de dados (*data grids* [37]). Desta forma, o Gridbus também possui funções para acesso de dados remotos e para a otimização da transferência de dados [111]. Assim, além do gerenciamento e escalonamento das aplicações serem baseados em requisitos definidos pelo usuário, como *deadline* e orçamento, a proximidade dos dados requeridos pela aplicação também é considerada. Para a definição das aplicações do tipo *parameter sweep*, o Gridbus estende a linguagem para modelagem de parâmetros (baseada em XML) [112] usada pelo Nimrod-G, suportando também parâmetros dinâmicos, isto é, que tenham seus valores determinados durante a execução.

O Gridbus *toolkit* possui um conjunto de ferramentas para suportar e gerenciar a execução de aplicações *e-Science* e *e-Business* de um determinado usuário. Dentre estas ferramentas se encontram: *G-monitor*, que faz a autenticação do usuário e permite o monitoramento dos processos em execução; *Grid Service Broker*, que toma as decisões de escalonamento nos recursos da grade; *Grid Market Directory - GMD*, onde os fornecedores de serviços publicam os serviços oferecidos pelos recursos juntamente com seu custo; *GridBank*, que mantém as contas dos fornecedores de serviços e dos usuários; e *Libra* que é um escalonador responsável por mapear os processos de um usuário entre os recursos de um *cluster*.

O escalonador do Gridbus recebe como entrada o conjunto de nós que oferecem serviços e o conjunto de processos do usuário. A estratégia de escalonamento combina os requisitos dos processos com os serviços oferecidos pelos recursos. Se os processos necessitam de dados remotos, o escalonador interage com o serviço de monitoramento para obter informações sobre a capacidade dos canais de comunicação entre os repositórios de

dados e os recursos computacionais. Estas informações oferecidas por serviços do Globus e NWS, em conjunto com as informações do *GMD*, são usadas para que o escalonador possa criar alocações que minimizem a quantidade de dados transferida e atenda aos requisitos de qualidade definidos pelo usuário. Como cada usuário requer uma instância diferente do *Broker* [112], o sistema de escalonamento pode ser caracterizado como centralizado entre as diversas aplicações de um usuário e hierárquico já que além do escalonador central da grade, cada *cluster* pode possuir o seu próprio escalonador [59].

O serviço de gerenciamento oferecido é descentralizado por usuário, e o Gridbus também disponibiliza um módulo que permite o gerenciamento de aplicações do tipo *workflow* [117]. Por ser um sistema orientado ao usuário, o Gridbus não pode ser classificado como totalmente autonômico, apesar de oferecer algumas funções como auto-recuperação de falhas e auto-otimização.

OurGrid/MyGrid

O OurGrid *toolkit* [8] fornece um conjunto de ferramentas para a execução de aplicações do tipo *bag-of-tasks* nas grades computacionais. Seu objetivo é oferecer para o usuário abstrações para lidar com a grade, escalonamento, acesso a recursos e segurança. Três partes distintas compõem o OurGrid: OurGrid *Community*, MyGrid e SWAN. O OurGrid *Community* pode ser visto como um *RMS* responsável por obter recursos que serão usados por instâncias do MyGrid. Já, SWAN é o componente que garante que o acesso aos recursos será feito de maneira segura, protegendo-os de possíveis ataques.

O MyGrid [38, 41] pode ser visto como um alocador de usuários, devendo, portanto, ser instalado na máquina de cada usuário da grade computacional. O objetivo do MyGrid é oferecer um ambiente simples, completo e seguro. A simplicidade deve ser caracterizada pelo mínimo de esforço que um usuário deve exercer para executar suas aplicações. Um sistema completo deve oferecer serviços que cubram todo o ciclo de uso de um sistema computacional, do desenvolvimento à execução, passando através da instalação, atualização e manipulação de arquivos. Finalmente, MyGrid é abrangente no sentido que todas as máquinas que o usuário possui acesso podem ser usadas pelo *middleware*.

O usuário é responsável por fornecer ao sistema uma descrição dos seus processos e um ponto de entrada para o OurGrid *Community*. Desta forma, o usuário pode especificar os requisitos dos processos que são usados pelo MyGrid para o escalonamento das tarefas nos devidos recursos. O OurGrid *Community* é composto por uma série de pontos (*peers*), que representam sites da grade computacional. Uma instância do MyGrid interage com o OurGrid local, requisitando recursos. Caso o número de recursos requeridos, seja maior que o disponibilizado pelo site, o OurGrid local pode entrar em contato com outros pontos do OurGrid *Community*. O MyGrid atua como um sistema gerenciador de usuários, possuindo uma implementação voltada às suas necessidades.

O MyGrid é instalado na máquina base (*home machine*) de cada usuário, de onde é coordenada a execução das suas aplicações. Tipicamente, uma *home machine* contém os dados de entrada da aplicação e coleta as informações produzidas na execução. As máquinas da grade onde são executadas as tarefas da aplicação não são customizadas pelo usuário. O ideal é que usuários não precisem instalar softwares ou tratar tais máquinas individualmente, elas devem ser manipuladas através das abstrações criadas no MyGrid. Entretanto, as máquinas da grade devem possuir um mínimo de serviços instalados para que possam ser usadas pelo usuário [38]. Uma interface chamada *GridMachine* encapsula o conjunto mínimo de serviços que devem estar disponíveis em uma máquina para que ela possa ser adicionada à grade do usuário. Tais serviços são: execução remota, cancelamento de uma tarefa em execução, transferência de arquivos da máquina base para máquinas da grade e vice-versa e checagem de conexão (*ping*).

Dado que este sistema só trata aplicações do tipo *bag-of-tasks*, o escalonador do MyGrid não utiliza informações sobre as tarefas da aplicação ou a capacidade dos recursos da grade. É usada a política *workqueue* [43] que utiliza apenas as informações necessárias para escalonar a aplicação, como por exemplo, o nome das tarefas a serem executadas e os recursos disponíveis. Para melhorar o desempenho do algoritmo devido à sua falta de informação relativa ao sistema, foi desenvolvido o *workqueue* com replicação, que também é usado como um mecanismo de tolerância a falhas. O escalonador do MyGrid fica localizado na *home machine*, sendo descentralizado entre os diferentes usuários da grade e centralizado no que diz respeito às aplicações que são disparadas da mesma *home machine*.

Com relação à autonomia dos serviços oferecidos podemos dizer que o MyGrid é parcialmente autogerenciável, visto que o usuário tem que tomar certas decisões como a especificação dos requisitos da sua aplicação, sugerindo não ser autoconfigurável. Entretanto, funções como auto-recuperação, auto-otimização e autoproteção são bem definidas.

Legion

Legion [36, 61, 62] foi desenvolvido como um sistema operacional para grades computacionais, com o objetivo de ser um *middleware* completo, tratando todos os desafios impostos pelo ambiente. É um sistema orientado a objetos, que visa atender os requisitos tanto dos usuários como dos administradores do sistema, agindo como um mediador que procura fornecer uma alocação de recursos que satisfaça as duas partes. Isto é alcançado através de uma abordagem modular do escalonamento.

A arquitetura do Legion segue um modelo de camadas, sendo os diferentes componentes da grade representados por objetos. A camada de serviços oferece heurísticas de escalonamento básicas que podem ser customizadas pelos usuários. Além disso, os usuários também podem criar suas próprias heurísticas de acordo com os seus objetivos e características internas da aplicação. A filosofia do escalonamento implementado é baseada na

negociação de serviços entre agentes autônomos, que atuam na aplicação (consumidores) e nos recursos do sistema (produtores). O escalonador é responsável pela coordenação da aplicação, decidindo o mapeamento das tarefas nos recursos de acordo com o estado atual do sistema e com as políticas internas de cada recurso.

Para obter informações do sistema é mantido um repositório (*Collection*) que armazena o estado dos recursos. Uma vez que o escalonador gera um mapeamento, ele o passa para o *Enactor* que é responsável por negociar e validar o escalonamento proposto. Durante a execução, baseado no progresso da aplicação e na carga computacional do sistema, pode ocorrer a recomputação do escalonamento. Como cada aplicação pode possuir sua própria política de escalonamento, pode-se identificar a existência de um escalonamento distribuído entre as aplicações.

O projeto do Legion considera aspectos chaves como: autonomia dos sites, suporte para heterogeneidade, escalabilidade, segurança e tolerância a falhas. Várias características autonômicas podem ser identificadas como: auto-otimização, autoproteção e auto-recuperação. Porém, por ser classificado como um sistema gerenciador de usuários, *UMS*, onde algumas decisões dependem da interferência do mesmo, como por exemplo, a capacidade de reconfiguração, Legion não é considerado um sistema totalmente autogerenciável.

Condor-G

O sistema gerenciador Condor-G [54] é um *RMS* que integra ferramentas disponibilizadas pelo Globus *toolkit* e pelo sistema Condor [80], permitindo que usuários tenham acesso a recursos de domínios diferentes como se eles pertencessem ao seu domínio pessoal. Assim, os usuários possuem uma visão unificada da grade computacional. O usuário define os processos que serão executados e o Condor-G trata da descoberta e aquisição de recursos, inicialização, monitoramento, gerenciamento da execução, detecção e tratamento de falhas, e notificação de término.

O Condor-G explora vários serviços fornecidos pelo Globus como: transferência de arquivos, autenticação de usuários, submissão remota e disseminação de informações sobre o estado dos recursos da grade. Através do sistema Condor, são agregados serviços que permitem a execução de processos do usuário de forma transparente em recursos que estejam ociosos. Embora o sistema Condor seja conhecido por esta característica, ele também pode ser configurado para compartilhar recursos [75]. O objetivo do Condor é maximizar a utilização dos recursos, permitindo um melhor aproveitamento das máquinas que estariam subutilizadas. Condor também oferece uma ferramenta para escalonamento que é capaz de gerenciar aplicações representadas por grafos acíclicos direcionados, DAGMan (*Directed Acyclic Graph Manager*) [45].

O *RMS* Condor-G possui um serviço para gerenciamento da computação (agente Condor-G) que permite que a grade seja tratada como um recurso local. Através de

uma interface, o usuário é capaz de submeter e cancelar processos, obter informações do status dos processos ou da história da sua execução e também ser informado de possíveis problemas ocorridos durante a execução. No caso de falhas, o agente Condor-G é capaz de ressubmeter processos.

Uma das funções do agente Condor-G é determinar onde serão executados os processos dos usuários, isto é, qual tipo de política de escalonamento será adotada. O Condor-G possui um escalonador central, que segue os moldes do sistema Condor, onde o objetivo é maximizar a utilização dos recursos disponíveis. Não existem políticas de escalonamento que se diferenciem por tipo de aplicação. O mecanismo *Class-Ad* (*Condor Classified Advertisement*) é usado para combinar pedidos de recursos dos usuários com as ofertas de recursos do sistema. As informações do estado da grade computacional são recolhidas pelo Globus MDS, e os recursos disponíveis podem receber prioridades de acordo com as preferências do usuário, como por exemplo, custos de alocação, tempo de início ou tempo de fim da computação

Os serviços oferecidos pelo Condor-G podem ser classificados como autonômicos, como por exemplo, o suporte para tolerância a falhas (auto-recuperação), para segurança do sistema (autoproteção) e para o escalonamento de tarefas (auto-otimização). Neste último caso, se o objetivo for a utilização de recursos ociosos, o monitoramento da grade permitirá que a execução se adapte às mudanças que possam ocorrer no ambiente computacional, para isso, são fornecidas bibliotecas para *checkpointing* e migração de processos [112]. Logo, dentro das funcionalidades propostas, o *RMS* Condor-G pode ser classificado como autogerenciável.

GridFlow

GridFlow [30] pode ser caracterizado como um sistema gerenciador de recursos (*RMS)*, cujo foco principal é oferecer serviços para o gerenciamento e escalonamento de aplicações do tipo *workflow* em uma grade computacional. O gerenciamento de recursos é hierárquico sendo dividido em três níveis: recurso, que trata de um recurso particular da grade; local, que consiste em vários recursos que pertençam a uma determinada organização; e global, que inclui todos os recursos da grade computacional que possam pertencer a diferentes organizações. O *framework* do GridFlow fornece ao usuário um portal com interface gráfica que facilita a composição das aplicações *workflow*.

O sistema para gerenciamento e escalonamento de *workflows* é projetado em níveis seguindo o mesmo padrão do gerenciamento de recursos. Desta forma, o gerenciamento ocorre nos níveis da tarefa, do *sub-workflow* e do *workflow*, existindo gerenciadores de recurso, gerenciadores locais e um gerenciador global.

O gerenciador global recebe pedidos do portal e simula a execução da aplicação para definir um escalonamento eficiente. O escalonamento produzido pode ser submetido à

aprovação do usuário ou levado diretamente para execução. Devido à natureza dinâmica da grade, podem ocorrer atrasos na execução da aplicação que não tenham sido previstos pelo escalonamento inicial produzido, neste caso, o restante da aplicação pode ser submetida a um novo escalonamento. Além disso, o gerenciador global fornece uma interface para o monitoramento da execução do *workflow*.

Um *workflow* pode ser alocado em diferentes grades locais, onde o gerenciador local é o responsável pelo escalonamento dos *sub-workflows*. Diferentemente dos gerenciadores globais, os escalonadores locais devem lidar com possíveis conflitos entre *sub-workflows*. Nestes casos, é necessário se estabelecer prioridades entre as tarefas ou usar uma política do tipo *First Come First Served*, logo, é difícil fornecer uma previsão precisa do tempo de início, execução e fim de um *workflow*. Finalmente, no nível da tarefa, os gerenciadores de recurso são responsáveis pelo escalonamento e execução das tarefas paralelas objetivando a máxima utilização do recurso.

O *framework* GridFlow utiliza uma variedade de ferramentas sofisticadas [31, 93] que devem ser instaladas para que seja possível fazer a análise e estimativa do desempenho de um recurso ou de uma aplicação. O objetivo principal da política de escalonamento é minimizar o tempo de execução da aplicação. Entretanto, o gerenciamento das aplicações é feito de forma centralizada, não existindo políticas de escalonamento específicas para diferentes tipos de aplicações.

Com relação à autonomia dos serviços oferecidos, pouco pode ser concluído. O monitoramento dos recursos do sistema permite que o sistema se auto-ajuste caso seja necessário optar por novos escalonamentos. Entretanto, a capacidade de se auto-reconfigurar, ou de se recuperar de falhas não é explicada com clareza nos artigos pesquisados. Logo, pode-se dizer, que apesar do avanço em se trabalhar com uma classe de aplicações mais complexa do tipo *workflow*, ainda pouco foi feito em relação ao desenvolvimento de um sistema totalmente autônomo.

Cactus

O *framework* Cactus [4, 5] é um ambiente *open source* para solução de problemas de aplicações da física e da engenharia. O seu formato modular torna possível que cientistas de diferentes instituições coordenem suas pesquisas, usando o Cactus como ferramenta colaborativa e unificadora para execução de aplicações paralelas.

Cactus possui um núcleo central (*flesh*) que é responsável por conectar os módulos da aplicação (*thorns*). Os *thorns* podem ser implementações de aplicações científicas dos usuários ou implementações de ferramentas padrão da grade que ofereçam distribuição de dados, tolerância a falhas, paralelismo, comunicação, entre outras. Dependendo da necessidade, *thorns* com conhecimento da infra-estrutura da grade podem ser incluídos no código dos usuários sem a necessidade de mudanças nos *thorns* das aplicações [6].

O usuário possui acesso simples e transparente aos recursos da grade computacional através de um portal. O objetivo é esconder atrás de um único ponto, as complexidades inerentes à arquitetura e aos *softwares* paralelos e distribuídos. O portal fica hospedado na *Web*, oferecendo uma interface acessível e universal a uma determinada plataforma de computação científica. Desta forma, cria-se um ambiente de laboratório tradicional, onde vários cientistas que compartilham os mesmos interesses e aplicações podem colaborar entre si, com simulações e coleta de dados para análise.

Para melhor lidar com o balanceamento de carga nos recursos disponíveis, Cactus fornece a habilidade de decompor a aplicação em subproblemas de tamanhos diferentes que possam se ajustar ao poder computacional local. Para descoberta de recursos são explorados serviços do Globus *toolkit*, e o processo de seleção é baseado no mesmo algoritmo utilizado pelo alocador de recursos do Condor [80]. Para que seja possível a migração de processos entre os recursos, mecanismos de *checkpointing* também são implementados e um conjunto de *thorns* é definido para que as aplicações possam se adaptar às possíveis mudanças da grade computacional, inclusive no caso de falhas.

Por oferecer um *framework* que gerencia a grade entre as diversas simulações de um problema de forma centralizada, Cactus pode ser classificado como um sistema gerenciador de recursos. Além disso, ele pode ser visto como um *framework* autogerenciável, fornecendo um conjunto de ferramentas que permitem auto-otimização, autoconfiguração, autoproteção e auto-recuperação.

EasyGrid

A fim de apresentar a classificação geral dos sistemas gerenciadores segundo a taxonomia proposta nesta tese, uma descrição inicial do *EasyGrid AMS*, utilizado neste trabalho, é feita a seguir.

O *EasyGrid AMS* é um sistema gerenciador de aplicações que é automaticamente embutido em aplicações MPI já existentes. Tais aplicações podem ser do tipo *bag-of-tasks* ou possuírem relações de precedência entre suas tarefas. O sistema de escalonamento e gerenciamento é descentralizado por aplicação, buscando desta forma oferecer maior eficiência e escalabilidade em ambientes como as grades computacionais. Além disso, cada aplicação possuirá uma hierarquia de processos gerenciadores e de agentes escalonadores, possibilitando que sejam definidas políticas distintas tanto entre as aplicações como também para diferentes níveis de uma hierarquia.

Os escalonadores distribuídos entre as diversas aplicações coordenam suas ações através do monitoramento do ambiente durante a execução, evitando possíveis conflitos. O objetivo principal do subsistema de escalonamento é minimizar o tempo de execução final da aplicação MPI. As informações sobre o estado do sistema e da aplicação são recolhidas por funcionalidades oferecidas pelo próprio *EasyGrid AMS*, tornando-o independente de

outros *middlewares* básicos. Para sua execução em uma grade computacional, é necessária apenas a instalação do Globus e da biblioteca de comunicação MPI.

O *EasyGrid AMS* provê a aplicação do usuário com capacidade de automonitoramento, autoconhecimento e conhecimento do ambiente. Além disso, a camada responsável pelo escalonamento dinâmico permite que a aplicação se auto-ajuste, otimizando a sua execução e se reconfigurando quando necessário. A habilidade de auto-recuperação é fornecida pelo sistema de tolerância a falhas. A Seção 2.2 descreve com maior detalhes a estrutura e as funcionalidades do *EasyGrid AMS*.

2.1.3 Classificação Geral

Um resumo da classificação dos sistemas gerenciadores de grade em relação à taxonomia apresentada na Figura 2.1, pode ser visualizada na Tabela 2.1. É importante ressaltar que o tipo identificado para o sistema gerenciador é aquele que mais se aproxima, pois em alguns casos, algumas características do sistema ainda podem estar em desenvolvimento ou simplesmente não foram identificadas no material bibliográfico pesquisado.

Cada característica tratada nesta classificação (orientação, gerenciamento, subsistema de escalonamento, instalação, autonomia e tipo de aplicação), assim como as particularidades de cada sistema gerenciador foram descritas nas subseções anteriores. Os detalhes de cada projeto relacionado bem como suas características básicas foram discutidas dentro dos principais pontos da taxonomia proposta, e a Tabela 2.1 mostra um resumo do que foi apresentado neste capítulo.

É importante ressaltar que sistemas gerenciadores que tratam apenas tarefas independentes não possuem infra-estrutura para tratar relações de precedência. Porém, os sistemas que lidam com tarefas com relações de precedência também têm a habilidade de tratar aplicações com tarefas independentes, visto que este é um tipo de aplicação que possui características internas mais simples. Sistemas totalmente autonômicos também constituem uma vantagem para o usuário, já que por serem autogerenciáveis não necessitam que os usuários tenham que tomar decisões para otimizar desempenho durante a execução de suas aplicações.

O que se pode observar é que o *EasyGrid AMS* possui características que juntas oferecem maior escalabilidade, flexibilidade e autonomia ao gerenciamento da aplicação nos recursos disponíveis. A escalabilidade é alcançada através do gerenciamento e escalonamento descentralizados entre as várias aplicações que compartilham a grade computacional. Além disso, para cada aplicação, o esforço de escalonar tarefas é dividido em níveis, tornando mais simples e flexíveis as políticas de escalonamento. Desta forma, cada aplicação pode possuir políticas que se adequem mais às suas características e requisitos.

Outra vantagem do *EasyGrid AMS* é a sua simples instalação, todas as funcionalidades oferecidas são embutidas automaticamente na aplicação do usuário sem que sejam

feitas alterações no código original da aplicação. O objetivo é criar aplicações *system aware* e autogerenciáveis, que possam se ajustar de maneira eficiente às mudanças do ambiente computacional.

Tabela 2.1: Classificação dos Sistemas Gerenciadores de Grade

Sistemas	Orientação	Gerenciamento	Instalação	Autonomia	Escalonamento	Aplicação
EasyGrid	aplicação	descentralizado por aplicação	embutido na aplicação	total	descentralizado hierárquico	tarefas com precedências
GrADS	aplicação	descentralizado por aplicação	instalado nos recursos	total	centralizado hierárquico	tarefas com precedências
AppLeS	aplicação	descentralizado por aplicação	embutido na aplicação	parcial	descentralizado	tarefas independentes
GridWay	aplicação	descentralizado por usuário	instalado nos recursos	parcial	descentralizado	*jobs* com precedências
Nimrod/G	usuário	descentralizado por usuário	instalado nos recursos	parcial	centralizado hierárquico	tarefas independentes
Gridbus	usuário	descentralizado por usuário	instalado nos recursos	parcial	centralizado hierárquico	*jobs* com precedências
MyGrid	usuário	descentralizado por usuário	instalado nos recursos	parcial	centralizado	tarefas independentes
Legion	usuário	descentralizado por usuário	instalado nos recursos	parcial	descentralizado	tarefas independentes
Condor-G	sistema	centralizado	instalado nos recursos	total	centralizado	*jobs* com precedências
GridFlow	sistema	centralizado	instalado nos recursos	parcial	centralizado hierárquico	*jobs* com precedências
Cactus	sistema	centralizado	instalado nos recursos	total	centralizado	tarefas com precedências

2.2 EasyGrid AMS

Nesta tese, cada aplicação de um usuário é gerenciada pelo *EasyGrid AMS* e dentro deste contexto serão analisadas políticas de escalonamento específicas para cada tipo de aplicação. Nesta seção é feita uma descrição do *framework EasyGrid* proposto em [22].

2.2.1 Arquitetura de um Processo Gerenciador

O *framework EasyGrid* é um sistema gerenciador de aplicações MPI, que trabalha com criação dinâmica de processos usando a biblioteca LAM/MPI [78]. O *EasyGrid* adota

uma hierarquia de processos gerenciadores, cujos projetos são baseados em uma *subsumption architecture* [24] apresentada na Figura 2.3.

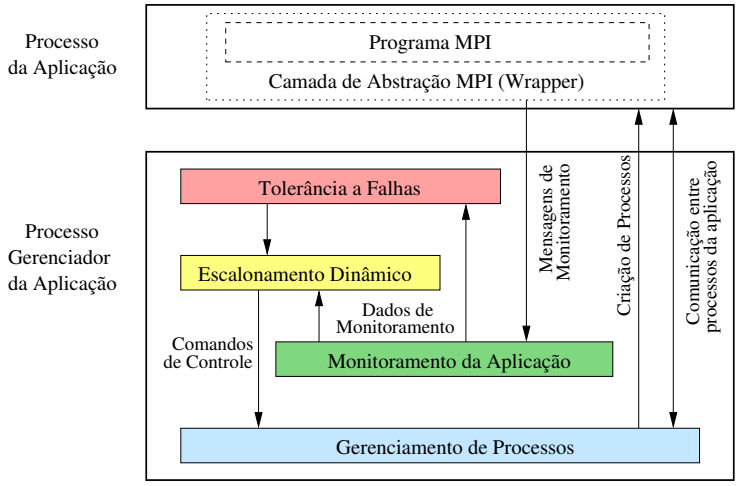

Figura 2.3: Estrutura em camadas do *EasyGrid AMS*.

Cada camada de um processo gerenciador oferece um conjunto de funcionalidades que são essenciais para a execução eficiente de uma aplicação MPI na grade computacional. Nesta arquitetura, as camadas mais altas podem modificar o comportamento dos serviços oferecidos pelas camadas mais baixas e assim se adaptarem melhor às mudanças da grade.

A camada mais baixa, que realiza o gerenciamento de processos, é responsável pela criação dinâmica dos processos MPI da aplicação do usuário e pelo redirecionamento de mensagens entre eles. A camada de monitoramento fornece informações necessárias para o reescalonamento e para a detecção de falhas de processos da aplicação que são usadas pelas camadas de escalonamento dinâmico e de tolerância a falhas.

2.2.2 Estrutura de Gerenciamento do AMS

Os processos gerenciadores oferecem serviços diferentes de acordo com a sua posição na hierarquia do *EasyGrid*. Eles são embutidos automaticamente na aplicação do usuário, permitindo que ela seja capaz de se autogerenciar (*autonomic application*). A Figura 2.4 mostra um exemplo de grade computacional com três sites, onde é possível visualizar os três níveis da hierarquia de gerenciadores que é composta pelas seguintes entidades:

- GM (*Global Manager*): o gerenciador global fica no topo da hierarquia supervisionando os sites da grade onde a aplicação do usuário pode executar;
- SM (*Site Manager*): em cada um dos sites existe um gerenciador que é responsável por alocar os processos da aplicação no seu site;

- HM (*Host Manager*): o gerenciador da máquina existente em cada um dos recursos disponíveis na grade é responsável pelo escalonamento, criação e execução dos processos do usuário no seu respectivo recurso.

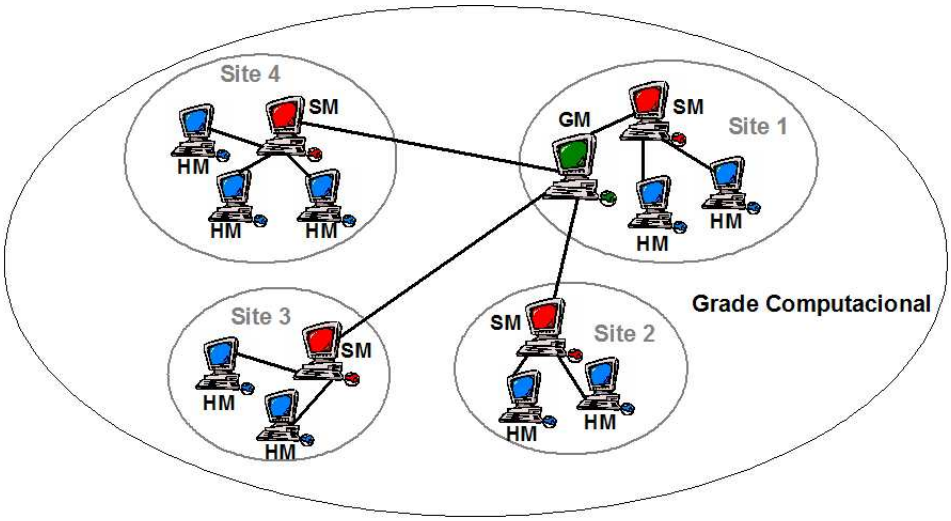

Figura 2.4: Hierarquia de processos gerenciadores do *EasyGrid AMS*.

2.2.3 O Portal *EasyGrid*

O Portal de escalonamento [18] do *framework EasyGrid* trabalha como um *RMS* simplificado. Ele deve escolher a política de escalonamento e de tolerância a falhas apropriada para a aplicação, determinar uma alocação de processos inicial, compilar a aplicação, gerenciar as credenciais de acesso a grade do usuário, criar o ambiente MPI, incluindo a transferência de arquivos necessários e garantir uma execução segura do gerenciador global, isto é, prover um mecanismo de tolerância a falhas ao GM.

2.2.4 Gerenciamento de Processos

Com exceção do GM, todos os processos gerenciadores e da aplicação do usuário são criados dinamicamente. O GM é criado através do Portal *EasyGrid* no momento em que a aplicação do usuário é disparada na grade computacional. A partir daí, o GM cria um SM por site que é responsável por criar um HM em cada recurso do seu respectivo site. Após a criação dos processos gerenciadores, os processos da aplicação do usuário também são criados dinamicamente pelos HMs de acordo com a política de escalonamento local definida.

Todos os processos são criados com comunicadores distintos, o que reforça a estrutura hierárquica do *EasyGrid AMS*, visto que a comunicação só pode acontecer entre processos pais e filhos. A opção por utilizar comunicadores distintos entre pares de processos facilita

a elaboração dos mecanismos de tolerância a falhas e ainda, permite que processos filhos sejam criados dinamicamente em momentos diferentes, sem exigir sincronização para que todos possuam um só comunicador [113]. Assim, os processos gerenciadores, GM, SMs e HMs devem ser responsáveis por rotear as mensagens trocadas entre os processos da aplicação do usuário.

Os processos do usuário e os processos gerenciadores podem executar no mesmo processador competindo pela CPU. Entretanto, a intrusão dos processos gerenciadores do *EasyGrid AMS* é mínima já que eles trabalham como *daemons* orientados a eventos, consumindo ciclos de CPU apenas para processar mensagens.

2.2.5 Mensagens de Monitoramento

As mensagens de monitoramento utilizadas pelo *EasyGrid AMS* ajudam a manter informações atualizadas tanto sobre a aplicação em execução como sobre o desempenho dos recursos da grade. Estas mensagens são disparadas sempre que uma tarefa da aplicação do usuário termina a sua execução em um determinado recurso e seu término deve ser avisado aos gerenciadores do site e global. Juntamente com esta mensagem indicativa de *fim de execução* seguem informações como porcentagem de utilização da CPU, tempo estimado para finalizar a execução da aplicação no recurso, quantas tarefas estão executando no momento, etc.

É possível que seja ativado um monitoramento ainda mais detalhado do estado da aplicação do usuário assim que ela for disparada na grade computacional. Desta forma, é informado o momento de criação de cada processo (*init*), quando são realizadas as trocas de mensagens (*send* e *receive*) e quando cada processo termina sua execução (*finalize*). Entretanto, tal abordagem é mais intrusiva, pois um número maior de mensagens circulará no sistema devido a cada mensagem extra de monitoramento. Logo, apenas a mensagem que indica o *fim da execução* de um processo é obrigatoriamente monitorada, dando suporte, por exemplo, à camada de escalonamento dinâmico.

Pode ocorrer que as tarefas da aplicação demorem muito tempo para terminar sua execução e para que as informações disponibilizadas pelas mensagens de monitoramento não fiquem desatualizadas, mensagens de *heartbeat* são usadas (subseção 4.1.1). As mensagens de *heartbeat* também podem ser vistas como mensagens de monitoramento que ajudam a manter informações precisas sobre o estado do sistema e da aplicação. Considerações em relação à frequência destas mensagens são feitas no Capítulo 4.

As informações empacotadas em uma mensagem de monitoramento são usadas pelas camadas de tolerância a falhas, cujas políticas estão em implementação [44, 90], e de escalonamento dinâmico, que será explicada detalhadamente nesta tese.

2.3 Resumo

Este capítulo apresentou uma taxonomia para sistemas gerenciadores de grade, identificando as características principais que podem ser encontradas nos sistemas do tipo *UMS*, *RMS* e *AMS*. O objetivo é classificar os tipos de sistemas gerenciadores de grade, de forma que seja possível identificar as vantagens e desvantagens de se optar por uma determinada filosofia de implementação.

Além disso, foram descritas de forma sucinta a arquitetura e as principais funções e características do *EasyGrid AMS*. A estrutura de processos gerenciadores criada e suas funcionalidades são embutidas automaticamente na aplicação do usuário, permitindo que ela se torne uma aplicação *system aware*. A arquitetura apresentada é utilizada para elaboração do sistema de escalonamento, que fornece à aplicação do usuário a capacidade de se auto-otimizar em relação às mudanças do ambiente grade. No próximo capítulo será introduzido o problema de escalonamento e suas características principais.

Capítulo 3

Escalonamento de Tarefas

O foco principal das heurísticas de escalonamento, implementadas e propostas neste trabalho, é alocar as tarefas da aplicação paralela nos processadores disponíveis da arquitetura alvo de forma que o tempo total de execução da aplicação (*makespan*) seja minimizado e que as relações de precedência entre as tarefas sejam obedecidas. Este problema é dito NP-Completo [56, 67, 95, 105, 109] e devido à natureza heterogênea, dinâmica e compartilhada das grades computacionais, o escalonamento de tarefas de uma aplicação paralela se torna ainda mais difícil de ser tratado.

Apesar de não ser estudado nesta tese, o problema de escalonamento de tarefas pode ter outros objetivos, como minimizar o *throughput* do sistema [65], minimizar os gastos do usuário (economia da grade) [27], ou ainda ser direcionado a aplicações específicas de tempo real cujas tarefas possuam *deadlines* que devam ser respeitados [28, 84, 99].

Além da criação de uma estrutura de escalonamento eficiente, um dos objetivos desta tese de doutorado é a criação de heurísticas de escalonamento dinâmico que possam ser embutidas em diferentes tipos de aplicações paralelas MPI representadas por *GADs*. As heurísticas desenvolvidas devem tratar as características da aplicação e estarem cientes do ambiente de execução para que seja possível otimizar o desempenho de uma aplicação, em ambientes de grades abertos, minimizando o seu tempo final de execução.

3.1 Definição do Problema

Algoritmos de escalonamento podem ser caracterizados como estáticos [11, 12, 20, 47, 72, 76, 77, 101, 108] ou dinâmicos [9, 16, 39, 43, 82, 99, 118]. Quando o conjunto de tarefas a ser escalonado e algumas de suas características são conhecidos antes da execução da aplicação, então, pode se propor heurísticas que abordem tanto o escalonamento estático quanto o dinâmico. Ou seja, o escalonador pode mapear e ordenar as tarefas nos processadores antes da execução da aplicação (escalonamento estático) ou enquanto a aplicação executa (escalonamento dinâmico). Porém, se as tarefas e as possíveis precedências

entre elas não são conhecidas, ou somente são conhecidas durante a execução da aplicação, a heurística de escalonamento usada deve ser dinâmica.

Em ambientes de grades, recursos podem ser adicionados ou retirados do sistema em qualquer momento. Além disso, como os recursos podem estar executando aplicações de vários usuários diferentes fica difícil prever o poder computacional total que pode estar disponível para uma aplicação antes da sua execução. Desta forma, projetar algoritmos de escalonamento estáticos que sejam eficientes para tais ambientes é um grande desafio, já que previsões feitas em tempo de compilação podem não mais valer no momento de execução da aplicação. Porém, como as heurísticas estáticas são executadas previamente sem competir com a execução da aplicação, técnicas de escalonamento mais complexas podem ser empregadas na tentativa de se conseguir um melhor tempo de execução paralelo (*makespan*).

Por outro lado, algoritmos de escalonamento dinâmico utilizam informações que são disponibilizadas durante a execução da aplicação, podendo tomar melhores decisões de escalonamento e assim adaptar a execução da aplicação do usuário às mudanças que possam ocorrer no desempenho ou na configuração de uma grade computacional. Entretanto, apesar das decisões serem baseadas em informações atualizadas, elas devem ser tomadas de forma rápida evitando a criação de um algoritmo de escalonamento complexo que atrase a execução da aplicação paralela. Uma outra vantagem da utilização do escalonamento dinâmico é dar suporte para a implementação de técnicas de tolerância a falhas [44], já que tarefas podem ser realocadas entre processadores durante a execução da aplicação, sempre que for detectada uma falha.

Apesar de uma grade computacional ter como uma de suas principais características o comportamento dinâmico, o escalonamento estático pode ser proveitoso no que se diz respeito à análise mais detalhada da topologia de uma aplicação paralela como um todo, principalmente no caso de aplicações que possuam relações de precedência entre suas tarefas. Esta habilidade, fornecida pelo escalonador estático juntamente com o acesso do escalonador dinâmico a informações precisas, pode fazer com que as decisões de escalonamento sejam tomadas de forma rápida e eficiente. Logo, é possível a utilização de uma abordagem híbrida [19, 46, 83] que busca juntar as vantagens do escalonamento estático e dinâmico.

Na implementação proposta neste trabalho, o *EasyGrid AMS* pretende utilizar a metodologia de escalonamento híbrida [21, 89] onde o objetivo principal é minimizar o *makespan* da aplicação paralela. Em tempo de compilação é possível uma melhor análise da aplicação e dos recursos, enquanto que durante a execução, a aplicação ciente das mudanças da grade pode se ajustar considerando informações mais precisas que são disponibilizadas ao longo da sua execução.

3.2 Modelo de Escalonamento

Visto que o problema de escalonamento de tarefas possui muitas maneiras de ser tratado, ao se propor uma heurística que alcance os objetivos estabelecidos de forma eficiente, devem ser especificados a classe de aplicações tratada e o tipo de arquitetura alvo usado. A partir daí, é possível construir heurísticas com características mais adequadas ao *modelo de escalonamento* adotado.

3.2.1 Modelo de Aplicação

Este trabalho supõe que uma aplicação paralela é representada por um grafo de tarefas acíclico direcionado (GAD) denotado por $G = (V, E, \varepsilon, \omega)$, onde V é o conjunto de nós do grafo e E o conjunto de arcos. Cada nó $v \in V$ representa uma tarefa do grafo com peso $\varepsilon(v)$ que indica a quantidade de operações realizadas pela tarefa v. Enquanto que um arco $(u, v) \in E$ representa a relação de precedência entre as tarefas u e v, ou seja, a tarefa u deve completar sua execução antes que a tarefa v comece. Associado a este arco pode existir um peso, $\omega(u, v)$, que informa a quantidade de dados enviada de u para v. Neste caso, é importante ressaltar que uma tarefa é uma unidade de computação indivisível. O conjunto das tarefas predecessoras e sucessoras imediatas de uma tarefa v é dado por $pred(v)$ e $succ(v)$, respectivamente, definidos como:

$$pred(v) = \{u \mid (u, v) \in E\}$$

$$succ(v) = \{z \mid (v, z) \in E\}$$

3.2.2 Modelo Arquitetural

O modelo arquitetural especifica as características da arquitetura alvo considerada. O conjunto de recursos computacionais que modela os processadores e a rede de interconexão devem ser definidos de forma que uma heurística apropriada possa ser desenvolvida.

Neste trabalho, considera-se que a grade computacional é formada por um conjunto limitado de processadores heterogêneos que possuem sua própria memória local e se comunicam através de trocas de mensagens, estando conectados segundo alguma topologia. Tal arquitetura pode ser representada por um grafo $G_r = (R, E_r)$, onde $R = \{r_1, r_2, ..., r_m\}$ é o conjunto de m nós que representam os recursos computacionais (processadores) e $E_r = \{(r_i, r_j)/r_i, r_j \in R\}$ o conjunto de arestas que representam os canais de comunicação entre eles.

Modelo de Comunicação

Outro ponto importante a ser analisado dentro do modelo arquitetural é o modelo de comunicação que se trabalha. Vários modelos podem ser considerados, porém o mais

popular nesta área de pesquisa é o modelo de latência, onde o único parâmetro de comunicação considerado é o atraso (latência) que um dado pode sofrer na rede de interconexão quando ele precisa ser transmitido entre dois processadores.

Um modelo mais completo é o modelo LogP [42], onde além da latência de comunicação são considerados custos para envio e recebimento de mensagens (sobrecargas de envio e recebimento). Porém, devido às dificuldades encontradas ao se trabalhar com este modelo [17, 20, 72], já que mais parâmetros de comunicação são considerados, a maioria das heurísticas já propostas preferem a simplicidade do modelo de latência.

Nesta tese será considerado como parâmetro de comunicação apenas a latência do canal de comunicação. Entretanto, como na arquitetura do *EasyGrid AMS* os processos da aplicação do usuário não se comunicam diretamente, mas através dos processos gerenciadores (GM, SM e HM), a sobrecarga para que uma mensagem seja recebida por um processo gerenciador será modelada juntamente com o tempo que esta mensagem gasta para ser enviada em um canal de comunicação.

No *EasyGrid AMS* podem existir três tipos distintos de padrão de comunicação entre as tarefas da aplicação do usuário. A comunicação pode acontecer entre tarefas que se encontram no mesmo recurso (Figura 3.1), entre tarefas que se encontram em recursos distintos do mesmo site (Figura 3.2), ou entre tarefas que se encontram em recursos de sites diferentes (Figura 3.3). Estes três esquemas de comunicação podem ser visualizados a seguir, onde as setas indicam o caminho percorrido por uma mensagem e as tarefas v_0, v_1, v_2 e v_3 pertencem à aplicação do usuário.

Na Figura 3.1, as tarefas v_1 e v_2 estão alocadas em um mesmo recurso r_4, onde o HM responsável pelo gerenciamento deste recurso intermedia a comunicação entre as tarefas.

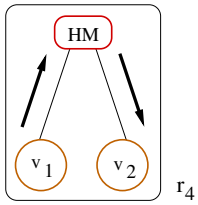

Figura 3.1: Esquema de comunicação entre tarefas alocadas em um mesmo recurso.

Neste exemplo, como todos os agentes da comunicação estão em um mesmo recurso, o custo da comunicação a ser considerado é o tempo que o HM demora para consumir a mensagem vinda de v_1 e redirecioná-la para v_2. Este tempo está diretamente relacionado com a quantidade de processos que executam juntamente com o HM. Ou seja, tão logo o HM ganhe a CPU, de acordo com a política de escalonamento do sistema operacional, a mensagem será consumida.

Para análise da comunicação entre tarefas pertencentes a recursos distintos do mesmo site, suponha que a tarefa v_0 esteja alocada no recurso r_3 e v_2 em r_4, como é possível

visualizar na Figura 3.2. Quando v_0 envia uma mensagem para v_2, tanto os processos gerenciadores da máquina como o processo gerenciador do site são envolvidos na comunicação. Logo, o HM de r_3 recebe a mensagem enviada por v_0 e a repassa para o SM. O SM é responsável por redirecionar esta mensagem para o HM que gerencia o recurso r_4, onde se encontra a tarefa v_2. Neste caso, além do tempo que cada processo gerenciador gasta para consumir a mensagem, também deve ser considerada a latência de comunicação entre os diferentes recursos.

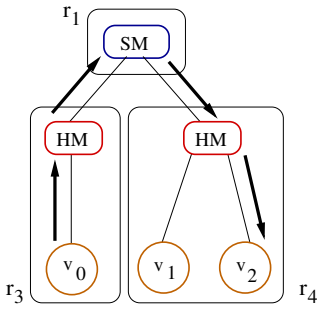

Figura 3.2: Esquema de comunicação entre tarefas alocadas em recursos distintos do mesmo site.

Assim como no exemplo anterior, a Figura 3.3 mostra o caso onde são considerados a latência de comunicação e o tempo que cada processo gerenciador gasta para consumir uma mensagem. Neste caso, como a comunicação é feita entre v_0 alocada em r_3 e v_3 alocada em r_5, que pertencem a diferentes sites, o gerenciador global também participa do redirecionamento da mensagem.

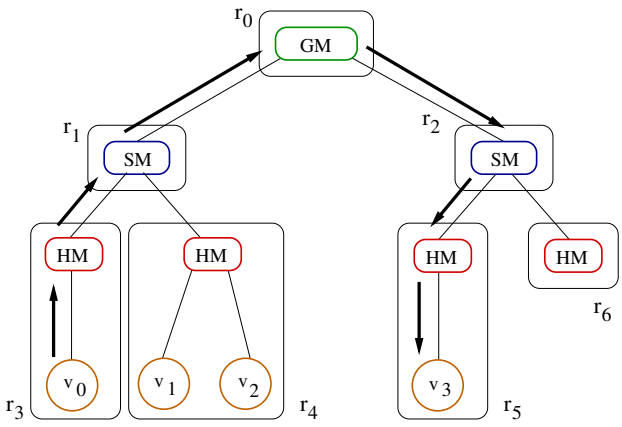

Figura 3.3: Esquema de comunicação entre tarefas alocadas em recursos de sites diferentes.

É importante ressaltar que no *EasyGrid AMS* as tarefas são criadas (colocadas para execução) segundo políticas de escalonamento internas associadas a cada máquina. Logo, em determinados casos, uma tarefa destino v_i quando criada, pode não precisar esperar por mensagens vindas de suas predecessoras. Isto ocorre, sempre que no momento da criação de v_i as mensagens enviadas pelas suas tarefas predecessoras já estejam armazenadas no *buffer* do HM correspondente.

3.2.3 Modelagem da Aplicação e da Arquitetura no EasyGrid AMS

No *EasyGrid AMS* um programa modelador [86] pode ser usado para capturar as características da arquitetura disponível, como a capacidade de processamento dos recursos e a latência associada aos canais de comunicação. Estes valores são coletados em um dado momento antes da execução da aplicação do usuário.

Nesta tese, o valor fornecido pelo modelador para indicar o poder computacional estimado de cada processador da grade é chamado de índice de retardo computacional (*computational slowdown index*, *csi*). O *csi* é um valor normalizado disponibilizado através do Portal do *EasyGrid AMS*, e está relacionado com a velocidade de processamento da CPU considerando todas os processadores da arquitetura quando dedicados à aplicação do usuário. O *csi* indica quanto tempo é necessário para executar uma operação e o recurso $r_j \in R$ mais rápido possui o menor *csi*.

Atualmente, para simplificar o trabalho do usuário é utilizado o seguinte esquema para modelagem de aplicações no *EasyGrid AMS*. Uma máquina r_j cujo csi_j é conhecido deve ser escolhida para executar a aplicação do usuário. Assim, é gerado um arquivo que armazena os tempos de execução das tarefas bem como o comportamento que revela as relações de precedência entre elas. Desta forma, é possível gerar o grafo acíclico direcionado desta aplicação. Os pesos das tarefas são então modelados dividindo o tempo gasto na execução das tarefas pelo csi_j da máquina r_j.

As heurísticas de escalonamento implementadas nesta tese utilizam o *csi* para que seja possível estimar o tempo de execução de uma tarefa da aplicação em cada $r_j \in R$. Como a capacidade de processamento de um recurso pode variar durante a execução da aplicação do usuário, já que a grade computacional é compartilhada e dinâmica, o poder computacional de um recurso é atualizado através de mensagens de monitoramento do *EasyGrid AMS*. Esta nova estimativa é feita pelo escalonador dinâmico conforme será explicado na Subseção 4.1.1.

Da mesma forma, para cada canal de comunicação entre dois processadores é coletado um índice de latência de comunicação (*communication delay index*, *cdi*), que também deve ser atualizado durante a execução da aplicação. Neste trabalho, a latência de comunicação entre dois recursos r_i e r_j é modelada através da mensagem de monitoramento enviada de r_i para r_j. Como os relógios de cada recurso são sincronizados no início da execução do programa, a latência é definida da seguinte forma:

$$cdi(r_i, r_j) = tempo\_recebimento(r_j) - tempo\_envio(r_i) \tag{3.1}$$

Com o objetivo de simplificar o modelo de comunicação adotado para a implementação das heurísticas de escalonamento dentro da estrutura do *EasyGrid AMS*, o valor calculado para $cdi(r_i, r_j)$ considera uma mensagem de tamanho fixo. Esta mensagem de monitoramento possui, atualmente, um tamanho aproximado de 90 *bytes*.

3.3 Definições Importantes

Nesta seção são apresentadas algumas formalizações das características associadas tanto à aplicação quanto à arquitetura utilizadas por várias heurísticas de escalonamento encontradas na literatura e pela estratégia proposta neste trabalho.

Definição 1: *Tempo de execução*

Cada tarefa da aplicação possui um tempo estimado em cada um dos recursos disponíveis na arquitetura. É possível que tal representação seja feita através de uma matriz de tempo onde cada posição (i, j) indica o tempo de execução de uma tarefa v_i em um recurso r_j [3]. Tais matrizes podem ser classificadas como consistentes ou inconsistentes. As matrizes inconsistentes permitem que um mesmo recurso seja rápido na execução de uma tarefa e lento em relação a uma outra. Isto faz sentido, pois nem sempre as tarefas possuem os mesmos requisitos disponíveis em um recurso, como memória ou velocidade de processamento. Entretanto, matrizes consistentes facilitam a estimativa de tempo das tarefas da aplicação, podendo ser substituídas pelo índice de retardo computacional (csi) que indica a capacidade de processamento de um recurso. Logo, a maioria das heurísticas utilizam um modelo uniforme, onde o tempo de execução estimado de uma tarefa v em um recurso r_j, antes da execução da aplicação é dado por:

$$et(v, r_j) = \varepsilon(v) \times csi(r_j) \tag{3.2}$$

Definição 2: *Tempo de Comunicação*

No *EasyGrid AMS*, o tempo gasto na comunicação entre dois processos é modelado de acordo com as situações apresentadas na subseção 3.2.2, onde o custo de comunicação, $comm(u, v)$, deve considerar a latência de comunicação (cdi) de todos os canais por onde passa a mensagem, desde a origem u até o destino v. Entretanto, na maioria dos modelos de escalonamento propostos é possível a comunicação direta entre duas tarefas u e v da aplicação do usuário. Logo, nas heurísticas desenvolvidas para o modelo de comunicação direta entre as tarefas [20, 47, 83, 108], o tempo para que um dado enviado de uma tarefa u alocada em r_j chegue em v alocado em r_k é dado por:

$$comm(u, v) = \begin{cases} 0, & r_j = r_k \\ \omega(u, v) \times cdi(r_j, r_k), & r_j \neq r_k \end{cases} \tag{3.3}$$

Definição 3: *Tempo de Início e Tempo de fim*

O tempo de início de execução de uma tarefa v em um processador r_j é denotado por $st(v, r_j)$. O tempo que v finaliza sua execução em r_j é representado por $ft(v, r_j)$. Sabendo que $ready(r_j)$ é o tempo em que o processador r_j está pronto para executar uma tarefa,

$st(v, r_j)$ e $ft(v, r_j)$ são definidos por:

$$st(v, r_j) = \max_{u \in pred(v)} \{ready(r_j), ft(u, r_x) + comm(u, v)\} \tag{3.4}$$

$$ft(v, r_j) = st(v, r_j) + et(v, r_j) \tag{3.5}$$

Definição 4: *Nível Escalonado*

O nível escalonado de uma tarefa v, $blevel(v)$, é o caminho mais longo desde v até um vértice de saída do grafo (vértice sem sucessores), considerando não somente a topologia do grafo, mas também o escalonamento definido pelo escalonador estático. No *EasyGrid AMS*, quando o *blevel* é calculado antes do início da execução da aplicação, a latência de comunicação considerada é a prevista pelo modelador [86]. Sendo r_j o recurso onde v foi alocada pelo escalonamento estático, o $blevel(v)$ é calculado da seguinte forma:

$$blevel(v) = \begin{cases} \varepsilon(v) \times csi(r_j), & succ(v) = \emptyset \\ \varepsilon(v) \times csi(r_j) + \max_{w \in succ(v)} \{comm(v, w) + blevel(w)\}, & succ(v) \neq \emptyset \end{cases} \tag{3.6}$$

Para o cálculo do nível escalonado ($blevel$), deve ser considerado que se v_i e v_{i+1} são executadas consecutivamente em um mesmo recurso r_j, então deve ser definido que v_{i+1} é sucessora de v_i, mesmo que não exista o arco $(v_i, v_{i+1}) \in E$.

Definição 5: *Caminho Crítico Escalonado*

O caminho crítico de um grafo é o caminho mais longo que se pode percorrer dentro de um grafo. Neste caso, para calcular o caminho crítico escalonado, a alocação inicial feita pelo escalonamento estático deve ser considerada, sempre que são referenciadas as tarefas sucessoras de v. Ao se considerar uma tarefa v para reescalonamento, as suas sucessoras ainda não foram tratadas pelo escalonador dinâmico. Como no cálculo do *blevel* escalonado, o *EasyGrid AMS* também usa a latência de comunicação prevista pelo modelador, quando o $cpath(v)$ é calculado antes do início da execução da aplicação. O caminho crítico de uma tarefa v alocada em r_j é dado por [83]:

$$cpath(v, r_j) = ft(v, r_j) + \max_{w \in succ(v)} \{comm(v, w) + blevel(w)\} \tag{3.7}$$

Definição 6: *Nível Topológico*

Diferentemente do nível escalonado ($blevel$), o nível topológico de uma tarefa não é definido baseado em informações do escalonamento, mas somente na topologia do grafo. O nível topológico de uma tarefa v é dado por:

$$level(v) = \begin{cases} 0, & pred(v) = \emptyset \\ \max_{u \in pred(v)} \{level(u) + 1\}, & pred(v) \neq \emptyset \end{cases} \tag{3.8}$$

Definição 7: *Bloco de Tarefas*

Com o objetivo de minimizar os custos de uma heurística de escalonamento dinâmico, muitos algoritmos não consideram todas as tarefas da aplicação para reescalonamento em um determinado momento da execução. Nestes casos, os algoritmos propostos analisam apenas um bloco de tarefas do grafo [83]. A definição de um bloco está diretamente ligada ao nível topológico das tarefas. O menor bloco de tarefas B_l que pode ser definido é o que considera apenas as tarefas de um único nível l do grafo. Neste caso, as tarefas do bloco são independentes entre si.

Porém, na tentativa de melhorar o desempenho do algoritmo dinâmico proporcionando uma visão maior do grafo, um bloco de tarefas pode ser composto por tarefas que pertençam a mais de um nível topológico consecutivo do grafo. Na heurística de escalonamento proposta nesta tese, sendo $level(v)$ o nível topológico de uma tarefa v, o bloco de tarefas B_l será definido por:

$$B_l = \{v \in V / level(v) \leq l\} \tag{3.9}$$

O objetivo de considerar um bloco que contenha todas as tarefas menores do que um determinado nível l, é criar oportunidade para que tarefas já reescalonadas possam participar de vários eventos de escalonamento enquanto não tiverem sido executadas. Esta estratégia foi adotada para que a heurística se adapte melhor a um ambiente dinâmico como as grades computacionais.

Definição 8: *Granularidade*

A granularidade de uma tarefa é baseada em uma relação local entre comunicação e computação, enquanto a granularidade do grafo é uma quantidade global que caracteriza o grafo [57]. A granularidade de uma tarefa v é dada por:

$$g(v) = \frac{\min\{\varepsilon(u)\}}{\max\{\omega(u,v)\}},\ \forall u \in pred(v) \tag{3.10}$$

A granularidade de um grafo G é:

$$g(G) = \min\{g(v)\} \tag{3.11}$$

Se $g(G) > 1$ diz-se que o grafo tem granularidade grossa, caso contrário, possui granularidade fina [57]. O conceito de granularidade é importante, pois a relação entre comunicação e computação estabelece até que ponto é vantajoso paralelizar ou não uma aplicação. Aplicações de granularidade grossa possuem mais computação em relação à comunicação, enquanto aplicações de granularidade fina o tempo gasto com comunicação é maior que o tempo gasto com a computação das tarefas da aplicação.

Nesta tese, são estudadas aplicações de granularidade grossa e do tipo *CPU-bound*. Aplicações de granularidade grossa são amplamente executadas em grades computacionais

conseguindo maior desempenho em suas execuções. Apesar das redes de interconexão estarem se tornando cada vez mais rápidas, o custo de comunicação entre as tarefas de uma aplicação ainda prejudica a execução em ambientes largamente distribuídos como as grades.

O conceito de granularidade referenciado em [57] é definido para uma aplicação em arquiteturas homogêneas. Entretanto, apesar deste trabalho tratar de ambientes heterogêneas este conceito ainda pode ser aplicado, visto que a comunicação medida entre as tarefas é realizada em uma ordem de grandeza de milissegundos, enquanto a menor tarefa executa em no mínimo 1 segundo, na máquina mais rápida da grade.

3.4 Heurísticas de Escalonamento Consideradas

A grande maioria dos sistemas gerenciadores de grades adotam políticas de escalonamento dinâmicas projetadas especificamente para aplicações do tipo *bag-of-tasks* [26, 29, 36, 68]. Tipicamente as estratégias de escalonamento seguem a abordagem clássica de *list-scheduling* [2], onde a idéia básica é definir prioridades para as tarefas da aplicação, e assim usá-las para estabelecer um critério para associar as tarefas aos recursos disponíveis. Algumas heurísticas conhecidas são: *workqueue*, *workqueue* com replicação [43], *sufferage*, *Max-Min* e *Min-Min* [33]. Nestes casos, o objetivo principal é minimizar o *makespan* total da aplicação. Por outro lado, como visto no Capítulo 2, também existem sistemas gerenciadores que adotam heurísticas de escalonamento baseadas na economia da grade [27], onde a estratégia considera o quanto o usuário deseja pagar para obter sua aplicação executada em um determinado intervalo de tempo.

Atualmente, mais pesquisas vem sendo desenvolvidas de maneira que os sistemas gerenciadores de grades sejam capazes de gerenciar de maneira eficiente aplicações cujas tarefas possuam algum tipo de relação de precedência. A maioria destes trabalhos considera aplicações do tipo *workflow* que podem ou não ser representadas por grafos acíclicos direcionados [64, 81, 98]. Entretanto, muitos sistemas gerenciadores, apesar de suportarem a execução de tais aplicações, não oferecem uma política de escalonamento adequada que tratem as dependências existentes entre as tarefas. Nestes casos, a heurística de escalonamento considera apenas as tarefas prontas em um determinado momento, isto é, as tarefas cujas precedências já foram satisfeitas. Tal abordagem, referenciada como *task-based* [16] é simplista, adotando uma alocação gulosa que considera apenas informações da tarefa pronta não considerando suas tarefas subsequentes.

Por outro lado, uma abordagem que procura uma alocação de tarefas eficiente considerando toda a topologia do grafo que representa o *workflow* é chamada *workflow-based* [16]. Tal estratégia produz escalonamentos melhores, porém dependendo do tamanho do grafo considerado pode não ser escalável. Logo, na tentativa de minimizar este problema, a cada

ativação do escalonador dinâmico apenas as tarefas pertencentes a um determinado bloco B_l do grafo [19, 83] são consideradas. A definição do tamanho deste bloco de tarefas é decisiva na eficiência do algoritmo. Quanto maior o tamanho do bloco mais informações da topologia do grafo são disponibilizadas para o escalonador podendo levar a decisões mais eficientes. Porém, quanto maior o número de tarefas tratado mais custosa pode ser a heurística, tornando-a não escalável.

Sabendo que a execução dos escalonadores dinâmicos concorrem com a execução da aplicação do usuário, as heurísticas dinâmicas são projetadas de maneira que suas decisões sejam tomadas rapidamente. Por isso, devido a sua menor complexidade, os algoritmos do tipo *list-scheduling* são vastamente empregados.

Apesar de não serem empregadas na maioria dos sistemas gerenciadores existentes, muitas estratégias de escalonamento dinâmico projetadas especificamente para aplicações representadas por *GADs* podem ser encontradas na literatura [19, 32, 39, 83]. Nesta tese, além da estrutura de escalonamento proposta, objetiva-se também a criação de heurísticas de escalonamento dinâmico eficientes para aplicações paralelas que executam em uma grade computacional. Tais aplicações podem ser do tipo *bag-of-tasks* ou possuir relações de precedência entre suas tarefas.

A seguir, são descritas algumas heurísticas de escalonamento híbridas do tipo *list-scheduling*, que foram implementadas no contexto do *EasyGrid AMS* para comparação com as heurísticas produzidas nesta tese. Estas estratégias foram escolhidas por terem sido projetadas especificamente para aplicações representadas por *GADs*, por não considerarem replicação de tarefas, e por possuírem bom desempenho computacional. São elas: *Minimum Partial Completion Time Static Priority* (PS), *Minimum Completion Time Static Priority* (CS) e *Minimum Completion Time Dynamic Priority* (CD), propostas em [83]. Por se tratarem de heurísticas híbridas, PS, CS e CD são compostas por duas fases de escalonamento: *fase estática* e *fase dinâmica*. Logo, ao iniciar a execução da aplicação um escalonamento estático já foi definido e a fase dinâmica se concentra em reescalonar tarefas de acordo com as flutuações do sistema.

Nestas heurísticas, a fase dinâmica do algoritmo utiliza prioridades previamente calculadas pelo escalonador estático. Nas três variações (PS, CS e CD), o escalonador dinâmico considera para reescalonamento apenas as tarefas pertencentes a um bloco B_l do grafo. Como descrito em [83], as tarefas de um bloco são reescalonadas uma única vez e diferentemente da Equação 3.9, B_l pode conter apenas as tarefas que pertençam ao nível l do grafo, ou também pode compreender um número maior de tarefas considerando os próximos L níveis. O valor de L influencia diretamente na eficiência e complexidade do algoritmo. O bloco B_l pode ter suas tarefas reescalonadas a partir do momento que a primeira tarefa pertencente a B_{l-1} for disparada para execução. As diferenças entre as estratégias PS, CS e CD são relativas às prioridades adotadas durante o processo de escalonamento dinâmico.

PS: A heurística PS utiliza o nível escalonado (*blevel*, Equação 3.6) para dar prioridades às tarefas de B_l que são candidatas ao reescalonamento. Assim, a tarefa $v \in B_l$ com maior *blevel* é selecionada e alocada no recurso r onde $ft(v,r)$ é minimizado.

CS: No algoritmo CS, as tarefas $v \in B_l$ também são ordenadas de acordo com o seu *blevel*. A tarefa v com maior *blevel* deve ser escalonada no recurso r que minimiza o caminho crítico de v, $cpath(v,r)$ (definido na Equação 3.7).

CD: A política de escolha de tarefas de CD é baseada no caminho crítico da tarefa definido pelo escalonamento estático. Desta forma, a tarefa $v \in B_l$ com maior $cpath(v,r)$ é selecionada, onde r é o recurso onde v foi previamente escalonada. A escolha do recurso onde v deverá ser reescalonada segue o mesmo critério do algoritmo CS.

Para melhor adaptação aos ambientes dinâmicos como as grades computacionais, a implementação feita da políticas PS, CS e CD no *EasyGrid AMS*, determina que um bloco B_l conterá apenas as tarefas do nível l do grafo. Isto permite que um menor número de tarefas seja considerado durante um evento de escalonamento, o que proporciona que um maior número de eventos ocorra durante a execução da aplicação. Tal abordagem torna a heurística mais sensível às mudanças do ambiente.

No estudo realizado em [83], as heurísticas PS, CS e CD mostraram melhor desempenho quando comparadas com a heurística *Generational Scheduling* (GS) [32]. GS é uma estratégia dinâmica, gulosa e de baixa complexidade, que reduz o problema de escalonamento de tarefas com relações de precedência, considerando apenas o conjunto das tarefas prontas para execução [46]. Tais tarefas se caracterizam por não possuírem relações de precedência entre si, e sempre que uma nova tarefa termina sua execução mais tarefas prontas podem ser adicionadas a este conjunto. A ordem em que as tarefas são escolhidas para serem reescalonadas segue a política *first come first served*, onde a primeira tarefa v que se torna pronta para execução é escalonada no recurso r que minimiza $ft(v,r)$. A abordagem utilizada pelo algoritmo GS é simples e não considera prioridades de tarefas relacionadas com a topologia do grafo, como nas estratégias anteriores.

O estudo de heurísticas de escalonamento é essencial para ressaltar a importância de políticas específicas para aplicações que possuam relações de precedência entre suas tarefas. Nestes casos, o conhecimento da topologia do grafo pode levar a decisões de escalonamento mais eficientes. Com o objetivo de simplificar, nesta tese, as aplicações do tipo *bag-of-tasks* serão referenciadas como aplicações BoT, enquanto aplicações com relações de precedência serão chamadas de aplicações GAD.

3.5 Resumo

Este capítulo descreveu as propriedades principais do problema de escalonamento, mostrando as vantagens e desvantagens que podem ser obtidas ao se usar uma abordagem de escalonamento estática ou dinâmica. Além disso, foram definidas as características da arquitetura e o modelo de aplicação usados nesta tese. Algumas particularidades do modelo de comunicação do *EasyGrid AMS* também foram apresentadas, pois como visto na Seção 2.2 do Capítulo 2, não existe comunicação direta entre os processos da aplicação do usuário. O modelo de comunicação reforça a estrutura hierárquica onde só existe troca de mensagens entre processos pais e filhos. Algumas técnicas de heurísticas de escalonamento e definições importantes também foram brevemente descritas. O próximo capítulo apresenta a estrutura de escalonamento proposta neste trabalho, a qual é baseada na hierarquia dos processos gerenciadores.

Capítulo 4

Estrutura de Escalonamento Proposta

Este capítulo descreve a estrutura de escalonamento hierárquica proposta dentro do *EasyGrid AMS*. O objetivo é criar uma estrutura que possa proporcionar uma execução eficiente das aplicações paralelas e distribuídas em ambientes dinâmicos e heterogêneos como as grades computacionais. Além disso, também é importante que a estrutura proposta permita a utilização de diferentes heurísticas de escalonamento que possam se adequar aos requisitos do usuário sem interferir nas regras de acesso dos diversos recursos que são disponibilizados em uma grade.

4.1 Estrutura Hierárquica de Escalonamento

A camada responsável pelo escalonamento dinâmico apresenta uma hierarquia de escalonadores dinâmicos que estão associados com os processos gerenciadores (GM, SM e HM) distribuídos nos três níveis da hierarquia do *AMS*. A Figura 4.1 apresenta um exemplo de grade computacional com três sites, onde é possível identificar os seguintes agentes de escalonamento:

- GDS (*global dynamic scheduler*): localizado no topo da hierarquia, o escalonador dinâmico global é responsável pelo reescalonamento de tarefas entre sites. Ele está associado ao GM.
- SDS (*site dynamic scheduler*): subordinado ao GDS e associado a cada SM, o escalonador dinâmico do site é responsável pelo reescalonamento de tarefas somente entre os recursos do seu site.
- HDS (*host dynamic scheduler*): no nível mais baixo da hierarquia de escalonadores se encontra o escalonador dinâmico da máquina. Associado a cada HM, ele é responsável

por definir as políticas de autorização da aplicação na máquina alvo e disparar os processos do usuário.

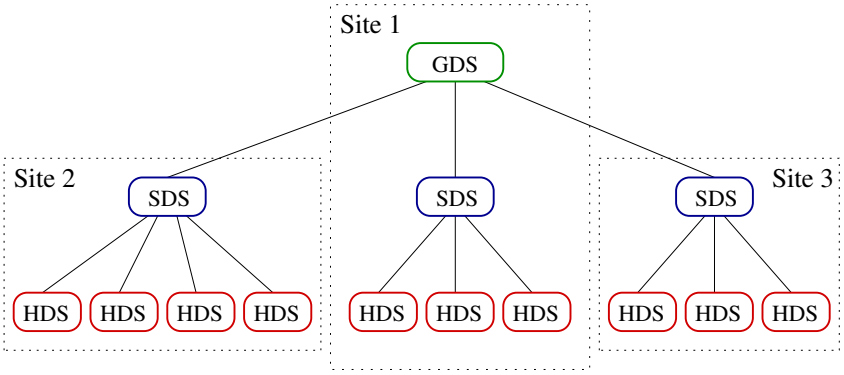

Figura 4.1: Exemplo da hierarquia de escalonadores dinâmicos do *EasyGrid AMS*.

Os escalonadores dinâmicos possuem funções específicas de acordo com o seu nível na hierarquia. Cada aplicação possui sua própria hierarquia de escalonadores, sendo possível que políticas de escalonamento distintas sejam empregadas em cada escalonador. As políticas usadas devem ser definidas considerando as políticas de acesso estabelecidas por diferentes instituições e também de acordo com as características da aplicação, como por exemplo: lidar com diferentes prioridades associadas às aplicações de usuários locais e usuários remotos, disparar mais de um processo da aplicação por máquina, disparar processos apenas quando uma máquina estiver ociosa, entre outras.

A estrutura hierárquica e distribuída proposta possui duas características essenciais para execução em grades computacionais: *flexibilidade* e *escalabilidade*. Visto que diferentes políticas de escalonamento podem ser definidas por aplicação ou até mesmo em diferentes níveis da hierarquia, o sistema se torna mais flexível podendo usar heurísticas que se adequem melhor aos diferentes requisitos da aplicação ou do sistema. Além disso, como não existe um escalonador central para todo o sistema, mas escalonadores distribuídos por aplicação, é possível alcançar maior escalabilidade em ambientes de larga escala como as grades.

A validação da estrutura proposta é feita através de experimentos. Para isso, foram estudadas diversas estratégias de escalonamento para que fosse possível desenvolver heurísticas dinâmicas eficientes e de baixa sobrecarga. As funções de cada escalonador da hierarquia serão detalhadas a seguir, independentemente das heurísticas de escalonamento adotadas. De uma maneira geral, pode se dizer que coletivamente os escalonadores dinâmicos estimam o tempo de execução restante da aplicação em cada recurso da grade e consequentemente, o *makespan* total da aplicação. Através de políticas específicas de escalonamento é possível verificar se o *makespan* pode ser minimizado, caso onde um *evento de escalonamento* deve ser disparado para a realocação das tarefas entre os diversos recursos disponíveis.

Para que um escalonador dinâmico seja eficiente, é essencial que ele baseie suas decisões em informações que estejam sempre atualizadas, já que a grade é um ambiente de execução dinâmico sujeito a mudanças frequentes. Existem várias formas para que um sistema gerenciador obtenha informações atualizadas sobre o estado dos recursos da grade computacional. Isto pode ser feito através de *middlewares* básicos como o Globus *toolkit* ou o NWS [114], através de funções do sistema operacional, ou através de mecanismos que sejam implementados no próprio sistema gerenciador. Neste trabalho, o escalonador dinâmico obtém informações do sistema através da camada de monitoramento do *EasyGrid AMS*. As funções implementadas não usam informações fornecidas por *middlewares* básicos na tentativa de tornar o sistema mais independente.

Como será discutido na Subseção 4.1.1, a camada de monitoramento oferece informações aos escalonadores dinâmicos sempre que um processo da aplicação termina sua execução ou através de mensagens de *heartbeat*. As mensagens de *heartbeat* podem ser usadas para diminuir o intervalo entre o recebimento das informações atualizadas.

4.1.1 Escalonador Dinâmico da Máquina - HDS

O escalonador dinâmico da máquina é responsável por determinar a ordem dos processos e consequentemente, o instante que um processo da aplicação deve ser criado na sua máquina. As políticas usadas para se determinar a ordem de execução das tarefas podem ser definidas pelo escalonador estático ou seguir o modelo *dataflow*, onde as tarefas podem ser executadas à medida que ficam prontas, isto é, quando todas as suas dependências forem satisfeitas e os respectivos dados estiverem disponíveis para o processamento.

O instante de criação de uma tarefa também pode seguir várias políticas diferentes, a serem selecionadas através do Portal. O *EasyGrid AMS* permite que vários processos da aplicação do usuário sejam criados ao mesmo tempo e, portanto, executem concorrentemente. Existem benefícios em se executar processos concorrentemente em um mesmo processador e o número ótimo depende da duração e das características de E/S do processo [90].

Outra política que pode ser interessante é disparar processos da aplicação apenas em recursos que estejam ociosos. Esta política pode ser usada dependendo das regras de acesso aos recursos da grade e também na tentativa de beneficiar a execução de processos locais ou com maior prioridade. Porém, determinar se um recurso está ocioso ou dedicado para apenas uma aplicação é difícil e algumas funções do sistema operacional são usadas para que a decisão tomada seja mais precisa. Os detalhes desta política (HDS/ociosa) serão apresentados no final desta subseção.

Independentemente da política de escalonamento local adotada durante a execução da aplicação, sempre que um processo u do usuário (tarefa) termina, duas informações são coletadas pela camada de monitoramento do *EasyGrid AMS* e disponibilizadas ao HDS:

1. tempo de CPU do processo u ($cpu\_time(u)$)

2. tempo parede do processo u ($wall\_clock\_time(u)$)

O tempo de CPU de um processo u é o tempo que este efetivamente usou o processador ao qual está alocado, enquanto o tempo parede inclui todo o tempo de espera e execução desde que u foi disparado até o seu término. Após o recebimento destes dois valores, o HDS pode estimar o desempenho oferecido pelo recurso, chamado de poder computacional, e o tempo restante de execução considerando as tarefas alocadas no recurso.

Logo, seja $R_k = \{r_1, r_2, ..., r_x\}$ o conjunto de x recursos disponíveis em um site s_k, u o último processo que finalizou a execução em $r_j \in R_k$ e nt_j o número máximo de tarefas que são disparadas concorrentemente em r_j. O fator de utilização da CPU de r_j (*percentage CPU utilization*, pcu_j) é definido como:

$$pcu_j = \frac{cpu\_time(u)}{wall\_clock\_time(u)} \times nt_j \tag{4.1}$$

O poder computacional relativo (*relative computational power*, cp_j) oferecido pelo recurso é então calculado pelo HDS_j, considerando o índice de retardo computacional do recurso (csi_j) fornecido pelo Portal *EasyGrid*, e o fator de utilização da CPU (pcu_j), já calculado, ou seja:

$$cp_j = \frac{pcu_j}{csi_j} \tag{4.2}$$

Após o cálculo do poder computacional, o HDS_j estima o tempo restante de execução (*estimated remaining time*, ert) das tarefas da aplicação alocadas, mas ainda não criadas, em r_j. Neste caso, são consideradas apenas as tarefas prontas, onde $V_j \subset V$ representa o conjunto destas tarefas:

$$ert_j = (\sum_{v_i \in V_j} \varepsilon(v_i)) \times \frac{1}{cp_j} \tag{4.3}$$

Além dos valores cp_j e ert_j calculados, o HDS_j associado ao recurso r_j também verifica o tempo em que r_j estará disponível para a execução de uma nova tarefa pronta. A estimativa deste tempo, denotado por $ready_j$, baseia-se no relógio atual de r_j. Porém, uma normalização de $ready_j$ é feita para que se considere que todos os recursos tenham começado a executar a aplicação do usuário no tempo 0 (zero).

Finalmente, os valores cp_j, ert_j e $ready_j$ são disponibilizados para o respectivo SDS, para que possam ser usados de acordo com a política de escalonamento definida para o site. Estes valores são adicionados à mensagem de monitoramento que é enviada pelo HM para informar ao SM que o processo u finalizou sua execução em r_j. Portanto, o HDS não cria nenhuma mensagem adicional, as informações calculadas são enviadas para o gerenciador do site, através de mensagens de monitoramento já existentes no *EasyGrid AMS*.

Além das mensagens de monitoramento já existentes, mensagens de *heartbeat* podem ser usadas para informar o cp_j, ert_j e $ready_j$ de um recurso r_j ao SM correspondente. O

heartbeat é disparado quando o tempo de envio da última mensagem de monitoramento enviada entre o HM e o SM e o tempo atual ultrapassar um limite determinado de tempo, $I_{monitor}$. Isto pode acontecer em duas ocasiões: (1) quando não existem processos executando em r_j, ou (2) quando o processo que está executando em r_j é muito demorado. Nestes casos, o cálculo do fator de utilização da CPU (pcu_j) se torna mais complicado, pois não existe um último processo u (ou pelo menos recente) do qual possa ser recolhido o tempo de CPU e o tempo parede. Logo, o HDS_j se baseia em funções do sistema operacional para realizar este cálculo. Note que, o intervalo $I_{monitor}$ entre envios de *heartbeat* também pode ser definido pelo usuário ou pelo próprio *AMS*.

Na implementação atual, quando uma mensagem de *heartbeat* for enviada, o HDS_j deve recolher o número médio de processos que executou no último minuto para estimar o fator de utilização da CPU. Tal valor é lido do arquivo de sistema /proc/loadavg. Como o valor oferecido pelo arquivo /proc/loadavg representa uma média, pode ser que ele não indique precisamente o estado atual do recurso quando ocorrer uma variação de carga em um curto espaço de tempo [114]. Logo, sabendo que este arquivo é atualizado de 5 em 5 segundos, duas leituras consecutivas com um intervalo mínimo de 5 segundos são realizadas e o valor pcu_j é estimado considerando a diferença entre os valores lidos. Devido a este fato, atualmente o valor mínimo para $I_{monitor}$ é estabelecido em 5 segundos.

Através de experimentos foram analisadas algumas funções oferecidas pelo sistema operacional para avaliação do fator de utilização da CPU. Porém, nenhuma medição se mostrou tão eficiente quanto a verificação do cpu_time e $wall\_clock\_time$ de um processo que tenha acabado de finalizar a sua execução.

O Algoritmo 1 resume as ações executadas por um HDS, enquanto a execução da aplicação não for finalizada.

Algoritmo 1 : HDS_j

```
1   dispara tarefas segundo a política local;
2   se (recebe mensagem fim(u) do monitor) então
3       calcula pcu_j baseado em cpu_time(u) e wall_clock_time(u);
4       calcula cp_j e ert_j;
5       verifica ready_j;
6       adiciona cp_j, ert_j e ready_j a mensagem recebida e repassa para SM;
    fim se
7   se (expirou I_monitor) então
8       calcula pcu_j baseado em /proc/loadavg;
9       calcula cp_j e ert_j;
10      verifica ready_j;
11      adiciona cp_j, ert_j e ready_j a mensagem de heartbeat e envia para SM;
    fim se
12  se (recebe pedido tarefas_ask de SDS) então
13      selecionar_tarefas_HDS(tarefas_ask);
    fim se
```

Além das funções já descritas, o HDS desempenha funções importantes no escopo dos escalonamentos do site e global, realizados respectivamente pelo SDS e GDS. Nestas situações, o HDS será ativado sempre que um pedido para ceder tarefas lhe for feito pelo SDS (linha 12), caso onde ele deve selecionar quais tarefas serão cedidas, se possível. Tal procedimento, selecionar_tarefas_HDS, será detalhado de acordo com a política de escalonamento adotada nas Subseções 5.2.2 e 7.2.2.

Política HDS/ociosa

Neste trabalho, a política HDS/ociosa dispara processos em um determinado recurso apenas quando este oferecer no mínimo 80% de sua CPU para a aplicação do usuário. Neste caso, considera-se que o recurso está ocioso, ou seja, estará praticamente dedicado à aplicação do usuário. A porcentagem pode ser ajustada pelo *middleware*, entretanto este valor foi escolhido experimentalmente, na tentativa de desconsiderar o tempo que a CPU gasta executando processos do sistema, interfaces, editores de texto, protetores de tela, ou outras aplicações que consomem pouca CPU. Na implementação corrente, existem duas maneiras de verificar se um recurso está *ocioso*, conforme descrito a seguir.

A primeira técnica é utilizada quando não existem tarefas da aplicação executando no recurso, logo, a pesquisa por ociosidade é feita utilizando o arquivo /proc/stat do sistema operacional. Este arquivo indica, em sua primeira linha, o número de *jiffies* gasto pela CPU nos modos usuário, nice, sistema e ocioso. Um *jiffy* é equivalente a um centésimo de segundo na maioria das CPUs e, portanto, a cada 10ms um *jiffy* é obrigatoriamente gasto em algum dos modos. Para verificar se a CPU está no mínimo 80% ociosa, devem ser realizadas no mínimo duas leituras consecutivas dos valores do arquivo /proc/stat. De acordo com o intervalo de tempo medido entre as leituras, basta verificar se no mínimo 80% dos *jiffies* foram gastos em modo ocioso. Em caso positivo, o recurso é dito ocioso e então, liberado para executar aplicações do usuário.

No segundo caso, quando já existem processos da aplicação do usuário executando, o número de *jiffies* em modo ocioso praticamente não se altera, dependendo das características de E/S dos processos. Logo, para verificar se um recurso r_j está sendo dedicado para a aplicação, verifica-se o fator de utilização da CPU (pcu_j) referente ao último processo executado. Se este valor for acima de 80% o recurso r_j continua liberado para a aplicação do usuário, caso contrário o recurso fica bloqueado para a criação de novos processos da respectiva aplicação. Visto que, uma tarefa em execução não é interrompida pelo HDS até o seu término, no caso em que o recurso voltar a ser compartilhado por outras aplicações, a tarefa continuará sua execução até o final e, então, novas tarefas não serão disparadas no recurso até que este seja considerado ocioso novamente.

4.1.2 Escalonador Dinâmico do Site - SDS

O escalonador dinâmico do site, SDS, é responsável por reescalonar tarefas somente entre as máquinas do seu site, com o objetivo de minimizar o *makespan*. Considerando $S = \{s_1, s_2, ..., s_q\}$ o conjunto de q sites disponíveis na grade computacional do usuário, cada site s_k pode possuir diferentes políticas de escalonamento e executá-las em momentos distintos. Como definido anteriormente, o conjunto de recursos pertencentes ao site s_k é denotado por R_k.

A política de escalonamento do site pode seguir critérios distintos até mesmo para as tarefas de uma mesma aplicação, como por exemplo, o tratamento diferenciado entre as tarefas já prontas para execução e as tarefas cujas pendências ainda não tenham sido satisfeitas. Assim, aplicações paralelas cujas tarefas possuam relações de precedências entre si (aplicações GAD), possuem em determinados momentos dois conjuntos distintos de tarefas: *prontas* e *pendentes*. Em aplicações do tipo *bag-of-tasks* (aplicações BoT), as tarefas são independentes entre si, estando sempre prontas para executar. Logo, suas tarefas podem ser tratadas através de alguma política de balanceamento onde critérios como afinidade e prioridade possam ser usados [33].

Neste trabalho, o escalonador dinâmico do site trata separadamente as tarefas *prontas* e *pendentes*, disparando *eventos de escalonamento* específicos sempre que for detectada a necessidade de reescalonamento de cada grupo. Desta forma, é possível que *eventos de escalonamento* de tarefas prontas e pendentes ocorram com diferentes frequências, sendo prioritário, por exemplo, o reescalonamento das tarefas que já se encontram prontas para execução.

Quando o SDS_k não está executando um evento de escalonamento, ele calcula a média do tempo de execução de tarefas prontas do site s_k ($\overline{ert}_k$) e a soma do poder computacional considerando todos os recursos de s_k (tcp_k). Estes valores são incluídos nas mensagens de monitoramento enviadas entre o SM e GM e são usados para o escalonamento dinâmico global. Assim, com os valores ert_j e cp_j enviados por cada $r_j \in R_k$, o cálculo de $\overline{ert}_k$ e tcp_k é especificado abaixo:

$$\overline{ert}_k = \frac{\sum_{r_j \in R_k} ert_j}{|R_k|} \tag{4.4}$$

$$tcp_k = \sum_{r_j \in R_k} cp_j \tag{4.5}$$

A estrutura de escalonamento proposta permite que o SDS dispare eventos de escalonamento locais. Para isso, é necessário que o SDS empregue políticas de escalonamento para verificar se existe a necessidade que um reescalonamento local seja realizado. Os intervalos entre cada uma destas verificações ou validações devem ser estabelecidos segundo critérios que podem considerar os requisitos da aplicação, sua topologia ou até mesmo o

tipo de heurística de escalonamento adotada. O intervalo mínimo entre cada validação do escalonamento de tarefas *prontas* será denotado por I_{min} (Algoritmo 2, linha 5). Já, no escalonamento de tarefas *pendentes*, o intervalo entre cada verificação está ligado com a topologia do grafo (Algoritmo 2, linha 11), onde a informação utilizada é relacionada ao nível topológico l das tarefas (Definição 6, Seção 3.3) que estão em execução em s_k.

A definição de um valor ou critério para I_{min} é importante, já que intervalos pequenos acarretam em uma maior intrusão no sistema, enquanto que intervalos grandes fazem com que o escalonador demore a reagir às possíveis mudanças da grade computacional. Da mesma forma, determinar o tamanho do bloco de tarefas pendentes sujeitas ao reescalonamento é essencial para um bom desempenho do algoritmo. Considerações em relação ao I_{min} e ao tamanho do bloco de tarefas ideal serão feitas nos Capítulos 5 e 7, de acordo com o tipo da aplicação do usuário. As ações do escalonador dinâmico do site s_k podem ser visualizadas no Algoritmo 2, apresentado a seguir. Um evento de escalonamento local só é ativado ($schedAtivo = 1$, Algoritmo 2, linhas 8 e 14) quando o SDS detecta que existem tarefas para serem reescalonadas no site.

Algoritmo 2 : SDS_k

```
1   se (recebe mensagem do monitor ou de heartbeat de algum HDS) então
2       atualiza tcp_k e ert_k;
3       adiciona tcp_k e ert_k a mensagem recebida e repassa para o GM;
    fim se
4   se (schedAtivo = 0) então
5       se (I_min ultrapassado) então
6           (aloc, R_ask, R_sub, tarefas_ask) ← escalonar_prontas_local();
7           se (tarefas_ask ≠ ∅) então
8               schedAtivo ← 1;
9               contPedido ← |R_ask|;  contResposta ← 0;
10              envia tarefas_ask_j, ∀r_j ∈ R_ask;
            fim se
        fim se
11      se (Nivel l ultrapassado) então
12          (aloc, R_ask, R_sub, tarefas_ask) ← escalonar_pendentes_local();
13          se (tarefas_ask ≠ ∅) então
14              schedAtivo ← 1;
15              contPedido ← |R_ask|;  contResposta ← 0;
16              envia tarefas_ask_j, ∀r_j ∈ R_ask;
            fim se
        fim se
    fim se
17  se (schedAtivo = 1) então
18      verificar_recebimento_tarefas_SDS(aloc, R_sub);
    fim se
```

Quando não existir nenhum escalonamento local já em ação no site s_k, ou seja, $schedAtivo = 0$ (Algoritmo 2, linha 4), o SDS_k pode verificar se existe a necessidade de

realizar um novo reescalonamento local. Como explicado anteriormente, o intervalo entre cada verificação de tarefas prontas respeitará o valor I_{min} (Algoritmo 2, linha 5). No caso de tarefas pendentes a verificação será feita sempre que a primeira tarefa de um nível topológico l terminar a sua execução (Algoritmo 2, linha 11).

Antes de disparar um evento de escalonamento, o SDS_k deve empregar um critério de avaliação que permita identificar se o *makespan* de s_k pode ser minimizado (Algoritmo 2, linhas 6 e 12). Em caso positivo, o SDS_k deve ser capaz de definir qual a melhor alocação de tarefas nos recursos do seu site, denotada por $aloc$. Desta forma, o SDS_k também identifica o conjunto dos recursos que devem ceder tarefas, representado por $R_{ask} \subset R_k$, o conjunto de recursos que devem receber tarefas ($R_{sub} \subset R_k$) e o conjunto de tarefas que serão pedidas no reescalonamento, denotado por $tarefas_{ask}$. O escalonamento local é então ativado, $schedAtivo = 1$ (Algoritmo 2, linhas 8 e 14).

Tendo sido verificada a necessidade de um evento de escalonamento, o SDS_k envia o pedido de tarefas, $tarefas_{ask_j}$, para cada $r_j \in R_{ask}$ (Algoritmo 2, linhas 10 e 16). Este pedido pode identificar as tarefas específicas que devem ser cedidas por r_j ou apenas especificar uma quantidade de tarefas estabelecida pela política de escalonamento do SDS. Normalmente, só existe a necessidade de se identificar quais tarefas devem ser cedidas por r_j quando as aplicações tratadas possuem relações de precedência. Nestes casos, a escolha de uma tarefa específica pode significar a diminuição do *makespan* da aplicação.

Após o envio de mensagens solicitando tarefas, o SDS_k deve verificar se os pedidos foram aceitos ou não. O algoritmo para verificar o recebimento de tarefas que é disparado na linha 18 do Algoritmo 2, pode ser visualizado a seguir.

Algoritmo 3 : verificar_recebimento_tarefas_SDS*(aloc, recursos)*

```
1  se (recebe ack e pacote_HDS de r_j ∈ R_ask) então
2      guardar pacote_HDS;
3      contResposta++;
   fim se
4  se (recebe nack de R_ask) então
5      contResposta++;
   fim se
6  se (contResposta = contPedido) então
7      distribuir_tarefas(pacote_HDS, recursos, aloc);
8      schedAtivo ← 0; /* encerra evento de escalonamento local */
   fim se
```

Cada escalonador dinâmico da máquina alocado em $r_j \in R_{ask}$, ao receber a mensagem com o pedido $tarefas_{ask_j}$ enviado pelo SDS_k, irá analisar se é possível atendê-lo, selecionando as tarefas para o reescalonamento (Algoritmo 1, linha 13). Se o pedido de tarefas não puder ser atendido pelo HDS_j, um *nack* é enviado como resposta ao SDS_k (Algoritmo 3, linha 4). Caso contrário, as tarefas são selecionadas pelo HDS_j de acordo

com $tarefas_{ask_j}$, e então enviadas juntamente com um ack para o SDS_k (Algoritmo 3, linha 1). Assim que o SDS_k recebe um pacote com as tarefas cedidas por algum $r_j \in R_{ask}$, ele deve guardá-lo até que todos HDSs alocados em R_{ask} tenham respondido ao pedido de escalonamento (Algoritmo 3, linha 2).

Para o controle do número de mensagens que foram disparadas solicitando tarefas e do número de respostas que já foram recebidas, são usadas as variáveis $contPedido$ e $contResposta$, inicializadas no Algoritmo 2, linhas 9 e 15. Assim, apenas quando a quantidade de respostas recebidas for igual a quantidade de pedidos disparados, (Algoritmo 3, linha 6), o SDS_k poderá redistribuir os pacotes de tarefas recebidos entre os recursos de s_k de acordo com a alocação de tarefas definida (Algoritmo 3, linha 7). Após a redistribuição o evento de escalonamento local pode ser encerrado.

É importante ressaltar que um SDS_k também desempenha funções no escalonamento dinâmico global em duas situações: quando recebe tarefas do GDS para serem redistribuídas no seu site (Algoritmo 4, linha 1), ou quando é selecionado para ceder tarefas ao GDS para que estas possam ser reescalonadas em outros sites (Algoritmo 4, linha 3).

Algoritmo 4 : SDS_k_escalonamento_global()

```
1    se (recebe pacote_GDS) então
2        distribuir_tarefas(pacote_GDS, R_k, aloc);
     fim se
3    se (recebe pedido tarefas_ask do GDS) então
4        selecionar_tarefas_SDS(tarefas_ask);
     fim se
```

4.1.3 Escalonador Dinâmico Global - GDS

O escalonador dinâmico global, GDS, é capaz de reescalonar tarefas entre os diferentes sites da grade. O GDS possuirá comportamento distinto em relação ao tipo de aplicação tratada: BoT ou GAD. Para aplicações BoT o escalonador global tem o poder apenas de ativar eventos de escalonamento, enquanto que para aplicações GAD ele também pode trabalhar como mediador entre pedidos de reescalonamento feitos pelos sites.

Nas aplicações BoT, apesar de não possuir conhecimento do poder computacional, nem da alocação de tarefas entre os recursos da grade, o GDS trabalha com as médias $\overline{ert}_k$ e tcp_k enviadas pelos sites através das mensagens de monitoramento. Considerando que neste caso todas as tarefas da aplicação estão prontas para executar, é razoável que o escalonamento global baseie suas decisões nas médias oferecidas com o objetivo de manter o sistema inteiro balanceado. Assim, o GDS dispara eventos de escalonamento (balanceamento) assumindo que cada site s_k já está internamente balanceado.

No caso de aplicações GAD, devido às relações de precedências existentes, a utilização de médias não se aplica ao cálculo de um escalonamento eficiente. Como apenas o SDS_k de cada site s_k conhece o poder computacional dos seus recursos e a alocação atualizada de tarefas do seu site, somente estes podem especificar com maior precisão o escalonamento atual de s_k e o seu *makespan*. Assim, sempre que um SDS_k realizar um escalonamento local e verificar que o *makespan* de s_k pode ser minimizado, um pedido de tarefas pertencentes a outro site pode ser feito ao GDS. Neste caso, o GDS desempenha um papel de mediador.

Ainda para aplicações GAD, o GDS mantém informações atualizadas periodicamente pelos sites, como o *makespan* parcial de cada site s_k e os valores estimados do tempo de execução final (ft) das tarefas que ainda não executaram. Dado que o GDS possui valores ft periodicamente atualizados, sempre que um SDS_k precisar calcular o seu *makespan*, ele pode pedir ao GDS os tempos de fim das tarefas u que não pertençam ao seu site, mas que sejam predecessoras imediatas de suas tarefas, sem que este pedido tenha que ser transmitido até o site em que a tarefa u está alocada.

Logo, no reescalonamento de tarefas de aplicações GAD, o GDS pode agir como um *mediador* de pedidos de tarefas entre os sites, avaliando se estes pedidos podem ser atendidos sem um prejuízo do *makespan* da aplicação, como também um *ativador* de eventos de escalonamento, caso este onde apenas as tarefas *prontas* são tratadas. Os passos executados pelo escalonador dinâmico global podem ser visualizadas no Algoritmo 5.

Algoritmo 5 : GDS

```
1   se (schedAtivo = 0) então
        /* ativador de escalonamento global: */
2       se (I_min ultrapassado) então
3           (aloc, S_ask, S_sub, tarefas_ask) ← escalonar_prontas_global();
4           se (tarefas_ask ≠ ∅) então
5               schedAtivo ← 1;
6               contPedido ← 0;  contResposta ← |S_ask|;
7               envia tarefas_ask_k, ∀s_k ∈ S_ask;
            fim se
        fim se
        /* mediador de escalonamento global: */
8       se (recebe pedido tarefas_ask de SDS) então
9           selecionar_tarefas_GDS(tarefas_ask);
10          se (tarefas_ask ≠ ∅) então
11              schedAtivo ← 1;
12              envia tarefas_ask_k, ∀s_k ∈ S_ask;
            fim se
        fim se
    fim se
13  se (schedAtivo = 1) então
14      verificar_recebimento_tarefas_GDS(aloc, S_sub);
    fim se
```

Assim como no escalonamento realizado pelo site, o GDS só pode participar de um evento de escalonamento global, caso nenhum outro evento já esteja ativado (Algoritmo 5, linha 1). Da mesma forma como já explicado anteriormente no escopo do escalonamento do site, o intervalo I_{min} deve ser respeitado no caso do rescalonamento de tarefas *prontas*. Neste caso, o GDS trabalha como ativador de escalonamento global (Algoritmo 5, linhas 2 a 7), logo, baseado nos valores $\overline{ert}_k$ e tcp_k recebido de cada site $s_k \in S$, o GDS pode decidir disparar um evento de escalonamento.

No Algoritmo 5, linha 3, o GDS determina qual a alocação de tarefas prontas ideal entre os sites da grade, denotada por $aloc$, e identifica o conjunto de sites S_{ask} e S_{sub} que devem ceder e receber tarefas, respectivamente. Caso seja verificada a necessidade de um evento de escalonamento, o GDS envia para cada site $s_k \in S_{ask}$ um pedido com as tarefas que devem ser cedidas, $tarefas_{ask_k}$ (Algoritmo 5, linha 7). Note que, como o evento de escalonamento global só é ativado diretamente pelo GDS para tarefas prontas, os pedidos de tarefas enviados aos sites não precisam identificar tarefas específicas, mas apenas uma quantidade de trabalho que deve ser cedida.

Quando o GDS trabalha como mediador do escalonamento entre sites (Algoritmo 5, linhas 8 a 12), ele será ativado ao receber um pedido específico de tarefas de algum dos sites s_k da grade (Algoritmo 5, linha 8). Neste caso, segundo a política de escalonamento adotada, o GDS identificará quais as tarefas que podem ser cedidas (Algoritmo 5, linha 9), de forma que o *makespan* não seja aumentado. Caso seja possível o reescalonamento global, o GDS enviará o pedido com as tarefas selecionadas para os sites correspondentes. As tarefas pedidas para cada site s_k são identificadas em $tarefas_{ask_k}$ (Algoritmo 5, linha 12).

Após o envio de mensagens solicitando tarefas, o GDS verificará se os seus pedidos foram respondidos (Algoritmo 5, linha 14). O algoritmo para verificar o recebimento de tarefas é análogo ao algoritmo verificar_recebimento_tarefas_SDS explicado no escalonamento do site e pode ser visualizado a seguir.

Algoritmo 6 : verificar_recebimento_tarefas_GDS*(aloc, recursos)*

```
1  se (recebe ack e pacote_SDS de S_ask) então
2      guardar pacote_SDS;
3      contResposta++;
   fim se
4  se (recebe nack de S_ask) então
5      contResposta++;
   fim se
6  se (contResposta = contPedido) então
7      distribuir_tarefas(pacote_SDS, recursos, aloc);
8      schedAtivo ← 0; /* encerra evento de escalonamento global */
   fim se
```

O esquema de comunicação entre os escalonadores dinâmicos pode ser visualizado nas Figuras 4.2 e 4.3, seguindo os algoritmos especificados neste capítulo. Cada passo da comunicação é representado por setas numeradas de (1) a (6). Como será explicado nos Capítulos 5 e 7, os conjuntos S_{ask}, R_{ask}, S_{sub} e R_{sub}, que indicam os recursos que devem ceder e receber tarefas, são determinados de acordo com a política de escalonamento dinâmico estabelecida. A Figura 4.2 apresenta a primeira parte do protocolo de comunicação, onde o GDS dispara um evento de escalonamento global requisitando tarefas para serem reescalonadas. A Figura 4.3 mostra a segunda parte da comunicação que só é realizada quando o GDS recebe a confirmação do seu pedido de reescalonamento. Caso não sejam cedidas tarefas, o GDS encerra o evento de escalonamento global. Cada passo da comunicação é explicado a seguir.

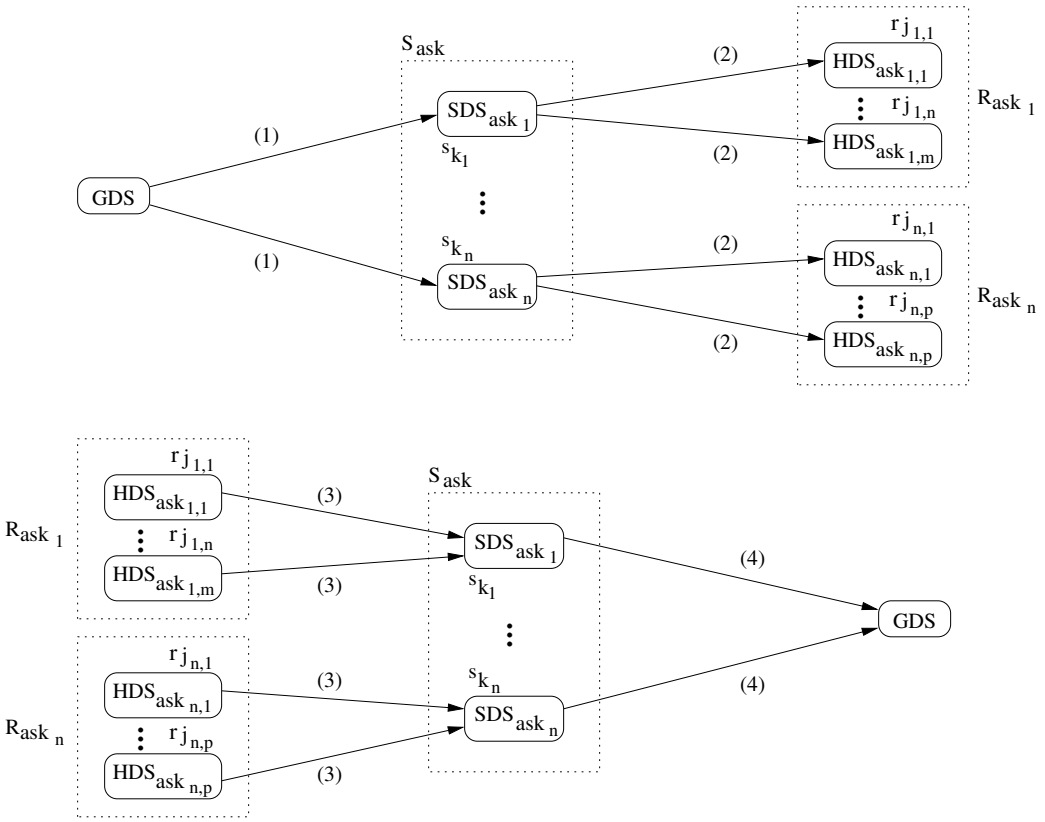

Figura 4.2: Esquema de comunicação entre os escalonadores dinâmicos do *EasyGrid AMS* ao ser disparado um evento de escalonamento global.

(1) O GDS envia um pedido de tarefas para cada SDS associado aos sites $s_k \in S_{ask}$ (Algoritmo 5, linha 7).

(2) Cada SDS_{ask}, após receber o pedido do escalonador global (Algoritmo 4), envia uma mensagem correspondente para cada HDS_{ask} do seu site, cujo recurso $r_j \in R_{ask}$ tenha sido selecionado para ceder tarefas.

(3) Cada HDS_{ask} que recebe um pedido de tarefas (Algoritmo 1, linha 12), deve enviar uma resposta para o SDS_{ask} correspondente. Tal resposta pode ser a confirmação do pedido juntamente com as tarefas cedidas, ou simplesmente a negação do pedido de reescalonamento.

(4) Após aguardar pelas respostas provenientes dos respectivos HDS_{ask}, cada SDS_{ask} envia uma mensagem para o GDS confirmando ou não o seu pedido de tarefas.

(5) Tendo guardado as tarefas recebidas de cada SDS_{ask}, o GDS de acordo com a política de escalonamento global, envia mensagens com as tarefas que devem ser reescalonadas em cada site $s_k \in S_{sub}$ (Algoritmo 6).

(6) Da mesma forma, seguindo a política de escalonamento local, cada SDS_{sub} envia mensagens com as tarefas cedidas para os recursos r_j do seu site, tal que $r_j \in R_{sub}$ (Algoritmo 4).

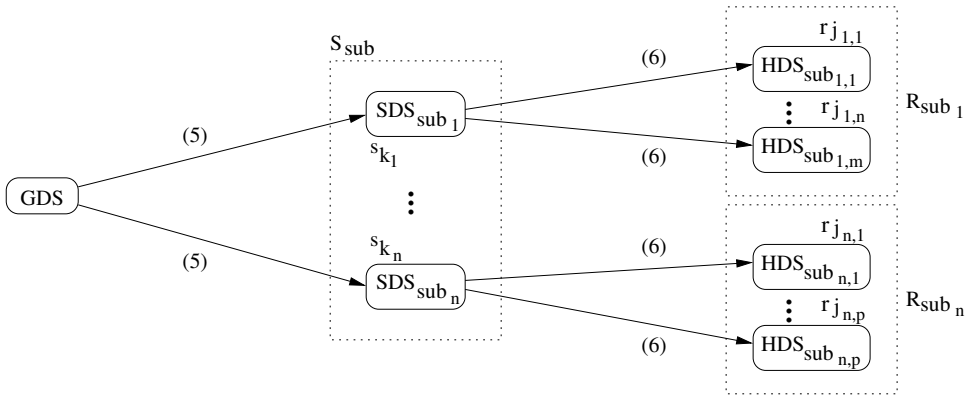

Figura 4.3: Esquema de comunicação entre os escalonadores dinâmicos do *EasyGrid AMS* quando o GDS recebe um pacote de tarefas para serem reescalonadas.

O esquema de comunicação e todos os algoritmos apresentados neste capítulo mostram a interação entre os escalonadores dinâmicos da hierarquia do *EasyGrid AMS*, independentemente do tipo de aplicação tratada. Nos Capítulos 5 e 7, para um melhor entendimento das heurísticas propostas, as seguintes funções serão detalhadas de acordo com a política de escalonamento adotada.

- selecionar_tarefas_HDS (Algoritmo 1, linha 13), realizada pelo HDS;
- escalonar_prontas_local (Algoritmo 2, linha 6), escalonar_pendentes_local (Algoritmo 2, linha 12), distribuir_tarefas (Algoritmo 3, linha 7 e Algoritmo 4, linha 2) e selecionar_tarefas_SDS (Algoritmo 4, linha 4), realizadas pelo SDS;
- escalonar_prontas_global (Algoritmo 5, linha 3), selecionar_tarefas_GDS (Algoritmo 5, linha 9) e distribuir_tarefas (Algoritmo 6, linha 7), realizadas pelo GDS

4.2 Resumo

Este capítulo apresentou a estrutura de escalonamento dinâmico desenvolvida no contexto do *EasyGrid AMS*. É proposta uma estrutura de escalonamento hierárquico, onde políticas distintas podem ser aplicadas nos diferentes níveis hierárquicos. Tal característica visa oferecer uma melhor adaptação à execução em grade computacionais, podendo adequar as heurísticas escolhidas às políticas de acesso de diferentes organizações. Além disso, como cada aplicação possui sua própria hierarquia de escalonadores, o esforço de escalonar tarefas é dividido entre as diversas aplicações que compartilham os recursos da grade, tornando o sistema mais escalável.

Nos próximos capítulos são apresentadas as políticas de escalonamento criadas e os experimentos realizados para validação da estrutura proposta. Algumas dificuldades em relação à implementação das principais funções de um escalonador dinâmico foram analisadas, assim como as justificativas nas soluções escolhidas. A pesquisa é dividida em duas partes de acordo com o tipo de aplicação utilizada: aplicações *bag-of-tasks* (Parte I) e aplicações com precedências (Parte II).

Parte I

Aplicações *Bag-of-Tasks*

Capítulo 5

Heurística de Escalonamento *BoT*

Nesta primeira parte da tese, o conjunto de aplicações considerado é do tipo *bag-of-tasks* (BoT), logo, a heurística de escalonamento implementada pode ser vista como uma subclasse do problema de escalonamento de tarefas: balanceamento de carga. Esta classe de aplicações é caracterizada por possuir tarefas independentes que podem ser executadas sem ordem pré-definida. A independência entre as tarefas traz algumas facilidades, como a possibilidade de ressubmeter tarefas em qualquer ordem, em qualquer máquina, no caso de falhas. Além disso, o atraso na execução de uma tarefa não influencia a execução de outras tarefas alocadas em outras máquinas. Apesar de não existirem restrições de precedência no processo de escalonamento, o problema de encontrar um escalonamento ótimo no menor tempo possível continua de difícil solução [67, 70].

Em [97] foi criado um ambiente onde é possível escolher entre várias estratégias de balanceamento de carga para aplicações BoT. Entretanto, diferentemente da abordagem de escalonamento *pró-ativa* utilizada nesta tese, a maioria dos trabalhos existentes oferece técnicas de escalonamento ou de balanceamento *reativas*, isto é, que esperam por solicitações do sistema ou da aplicação para serem ativadas. Uma das vantagens da utilização de um escalonamento *pró-ativo* é a diminuição da sobrecarga de escalonamento [104].

Neste capítulo serão descritas as fases de escalonamento estático e dinâmico implementadas no *EasyGrid AMS* para aplicações BoT. Entretanto, somente a heurística dinâmica, que é alvo deste trabalho, será explicada em detalhes.

5.1 Escalonamento Estático

Muitas heurísticas que tratam o problema de escalonamento estático de tarefas em ambientes heterogêneos já foram propostas [12]. O *framework EasyGrid* disponibiliza através do seu Portal de escalonamento um conjunto destas heurísticas que podem ser utilizadas pelo usuário na tentativa de encontrar um escalonamento eficiente para sua aplicação [18].

Neste trabalho, para aplicações BoT, a heurística de escalonamento ordena as tarefas decrescentemente de acordo com o seu peso $(\varepsilon(v), v \in V)$ e as distribui entre os recursos

(máquinas) disponíveis considerando uma política gulosa, onde a tarefa de maior custo é dada para a máquina que pode receber a maior carga de trabalho, conforme pode ser visto no Algoritmo 7. Este algoritmo possui os seguintes parâmetros: o conjunto de tarefas a serem distribuídas, o conjunto de *recursos* e a carga de trabalho que cada recurso deseja receber (*wload*), que é proporcional ao seu desempenho computacional estimado.

É importante ressaltar que, nos Algoritmos 3, 4 e 6 apresentados no Capítulo 4, a função distribuir_tarefas() possui como parâmetros o pacote de tarefas que será distribuído, o conjunto de recursos que deve receber tarefas e qual deve ser a alocação das tarefas cedidas nos recursos destinos correspondentes (*aloc*). Porém, para aplicações BoT não é necessário que o reescalonamento dinâmico realizado pelo SDS defina uma alocação rígida para as tarefas nos recursos. Logo, neste capítulo, a variável *aloc* usada para aplicações GAD, será substituída pela carga de trabalho destinada a cada recurso. Quando a função distribuir_tarefas() é chamada pelo SDS, o parâmetro *recursos* indica uma lista de máquinas que podem receber tarefas. Quando esta mesma função é executada pelo GDS, *recursos* identifica um conjunto de sites da grade.

Nesta política, os recursos também são ordenados decrescentemente, porém de acordo com a carga de trabalho que cada um deles pode receber (linha 3). A cada passo, a primeira tarefa da lista é dada para o primeiro recurso (linha 5), e em seguida a lista de recursos é reordenada considerando a tarefa recebida. O objetivo de alocar primeiramente as tarefas mais longas é tentar evitar que na fase final da distribuição existam tarefas maiores do que a carga de trabalho que um recurso ainda pode receber. Esta situação pode acontecer pois as tarefas são indivisíveis. Além disso, tal distribuição de tarefas também favorece a política de escalonamento dinâmico implementada, como será visto a seguir.

Algoritmo 7 : distribuir_tarefas*(tarefas, recursos, wload)*

```
1  listaTarefas ← tarefas em ordem decrescente de peso (ε);
2  enquanto (listaTarefas ≠ ∅) faça
3      listaRecursos ← recursos em ordem decrescente de wload;
4      r ← first(listaRecursos);   v ← first(listaTarefas);
5      aloca v em r, adicionando v ao pacote_r;
6      wload(r) ← wload(r) − ε(v);
7      listaTarefas ← listaTarefas −{v};
   fim enquanto
8  para todo r_k ∈ recursos faça
9      envia pacote_{r_k} para r_k;
   fim para
```

5.2 Escalonamento Dinâmico

Quando a execução da aplicação do usuário é iniciada, as suas tarefas já estão previamente alocadas nos processadores disponíveis segundo a política de escalonamento estático.

Porém, o reescalonamento de tarefas pode ser necessário devido à dificuldade de se obter estimativas precisas em relação ao desempenho dos recursos e do ambiente de execução em tempo de compilação. Isto ocorre devido a natureza dinâmica e compartilhada das grades computacionais, onde é importante que as estimativas sejam sempre atualizadas no decorrer da execução da aplicação.

O escalonador dinâmico proposto baseia suas decisões em informações fornecidas pela camada de monitoramento do *AMS*. Na verdade, o que é feito é um balanceamento das cargas de trabalho associadas às filas de espera dos processadores, não importando especificamente qual tarefa será cedida, mas sim o seu tempo de execução. Na versão atual, o *AMS* foi projetado para reescalonar apenas tarefas que ainda não começaram a sua execução, já que migrar tarefas em execução pode ser um processo caro [110].

Antes da descrição das funções desempenhadas pelos diferentes escalonadores da hierarquia, algumas considerações em relação ao escalonamento dinâmico em ambientes grades serão analisadas. Como a abordagem escolhida neste trabalho é hierárquica, o processo de escalonamento e a responsabilidade em resolver as dificuldades apontadas, a seguir, podem ser divididas entre os escalonadores de diferentes níveis.

5.2.1 Considerações do Escalonamento Dinâmico

No processo de escalonamento várias etapas podem ser destacadas, e em cada uma delas diferentes políticas podem ser adotadas. É importante lembrar, que no escalonamento dinâmico não se deve gastar muito tempo nas tomadas de decisão para que o gerenciador da aplicação não sobrecarregue a execução da aplicação do usuário. Portanto, heurísticas mais simples devem ser levadas em consideração ao se implementar as funções principais de um escalonador dinâmico, que são relacionadas abaixo:

(a) Frequência de verificação de desbalanceamento;

(b) Seleção das máquinas participantes do reescalonamento;

(c) Seleção das tarefas a serem redistribuídas;

(d) Redistribuição das tarefas selecionadas;

(a) Frequência de verificação de desbalanceamento

Considerando que a grade computacional é um ambiente dinâmico, é necessário verificar de tempos em tempos se o sistema está balanceado, isto é, se a distribuição de tarefas entre os recursos da grade é satisfatória. Logo, uma questão importante é determinar com qual frequência esta verificação deve ser realizada. Intervalos muito pequenos entre cálculos de desbalanceamento podem aumentar a sobrecarga no sistema quando a

grade computacional possuir um comportamento muito inconstante, já que mais eventos de escalonamento poderão ser disparados.

Uma vantagem de intervalos de tempo pequenos entre cálculos de desbalanceamento é tornar aplicação *system aware* mais sensível às mudanças da grade. Entretanto, se uma aplicação possui um tempo de execução longo talvez não seja interessante que ela se preocupe em fazer escalonamentos dinâmicos frequentes, visto que, até o final da sua execução, a grade já pode estar com um comportamento completamente diferente. Estas considerações podem ser feitas para aplicações BoT onde a execução de tarefas em um recurso mais atrasado não prejudica a execução de tarefas em outros recursos.

Sabendo que a aplicação *system aware* deve ser uma aplicação inteligente e autonômica, uma boa abordagem seria a diminuição deste intervalo automaticamente, à medida que a aplicação se aproxima do seu final. Neste caso, o escalonador dinâmico vai se tornando mais sensível às mudanças que possam ocorrer na grade, justamente no momento em que tais mudanças acarretam maior impacto na execução da aplicação.

Na implementação do algoritmo de escalonamento para aplicações BoT, por questão de simplicidade, foi estabelecido um intervalo mínimo fixo, I_{min}, entre cada cálculo de desbalanceamento realizado, porém, outras técnicas devem ser avaliadas futuramente. O grau de desbalanceamento de uma aplicação é calculado separadamente para cada site (pelos respectivos SDSs) e para o sistema como um todo (pelo GDS).

(a.1) Índice de desbalanceamento

Um outro fator determinante para que um evento de escalonamento seja disparado é o *índice de desbalanceamento* permitido entre os recursos da grade. O índice de desbalanceamento é um valor que representa a má distribuição de trabalho entre os recursos disponíveis. Antes da execução da aplicação, o usuário pode definir a porcentagem tolerável para o desbalanceamento entre os recursos ou entre os sites da grade.

Uma boa característica verificada na política implementada nesta tese é que os índices de desbalanceamento se tornam mais sensíveis a medida que a aplicação se aproxima do final da sua execução. Tal propriedade pode ser verificada, pois quando a carga de trabalho diminui em cada recurso r_j, pequenas diferenças entre os seus tempos estimados de fim (ert_j) se tornam mais significantes em proporção a carga de trabalho restante. Isto é importante para que a atuação do escalonador aumente quando ele se torna mais necessário.

(b) Seleção de máquinas

Ao detectar que o sistema está desbalanceado, um outro problema é escolher quais máquinas irão participar do reescalonamento. No caso da heurística de escalonamento que trata de aplicações BoT, a solução implementada pode ser vista como um algoritmo de balanceamento de carga ou *balanceamento de tempo*. Neste caso, para alcançar o objetivo onde todas as máquinas terminem sua execução aproximadamente ao mesmo tempo, uma

heurística deve passar tarefas de máquinas sobrecarregadas para máquinas subcarregadas até que se alcance um equilíbrio.

Em cada evento de escalonamento pode-se optar por repassar as tarefas excedentes: (1) de todas as máquinas sobrecarregadas para todas as subcarregadas; (2) da máquina mais sobrecarregada para todas as subcarregadas; (3) de todas as máquinas sobrecarregadas para a mais subcarregada; ou simplesmente (4) da máquina mais sobrecarregada para a mais subcarregada.

A política (1) é mais eficiente no que se diz respeito a quantidade de eventos de escalonamento que serão necessários para se balancear os recursos da grade. Neste caso, como todas as máquinas são tratadas ao mesmo tempo, no final de apenas um evento de escalonamento todos os recursos estarão balanceados. Por outro lado, tal política é mais custosa pelo fato de considerar um maior número de máquinas e de tarefas da aplicação em um único processo de reescalonamento. No caso de aplicações BoT, como não existe dependência entre as tarefas, um escalonamento gradativo também pode ser adotado com eficiência.

A política (2), adotada nesta implementação, tem o objetivo de ceder carga de trabalho para as máquinas subcarregadas moderadamente, minimizando o custo do algoritmo e adaptando a execução da aplicação às mudanças ocorridas na grade de forma gradual. Como uma grade computacional pode mudar de comportamento e configuração a qualquer momento, não é necessário que o algoritmo de escalonamento dinâmico para aplicações BoT empregue um grande esforço em apenas um único evento de escalonamento. Logo, na implementação atual, em cada evento de escalonamento, o escalonador dinâmico retira carga extra apenas do recurso ou do site com maior tempo estimado restante, passando tarefas para os recursos mais ociosos (subcarregados) gradativamente.

O problema da política (2) é que tratando apenas uma máquina sobrecarregada por vez, pode ser que o sistema demore a ficar balanceado, porém uma solução simples é diminuir os intervalos mínimos entre eventos de escalonamento. A estratégia usada nas políticas (3) e (4) é ceder tarefas para somente uma máquina subcarregada por vez. Esta abordagem não é eficiente, pois a máquina subcarregada pode não ter capacidade para receber as tarefas cedidas e ao final do evento de escalonamento muito pouco terá sido feito em relação ao balanceamento do sistema.

(c) Seleção de tarefas

Após a escolha das máquinas, as tarefas que serão cedidas para a redistribuição devem ser selecionadas. Na versão atual da política de escalonamento implementada, apenas tarefas que ainda não foram criadas é que podem participar do processo de reescalonamento. Para a implementação de uma política de realocação de tarefas em execução, é necessário que sejam criados mecanismos de *checkpointing*, para que a tarefa continue a sua execução a partir do momento em que foi interrompida. No entanto, além da versão atual do *EasyGrid*

AMS não oferecer *checkpointing*, os custos relacionados à interrupção e ao reinício da execução do processo em um outro recurso também deverão ser avaliados.

Considerações em relação ao tempo de execução das tarefas que serão realocadas ainda devem ser feitas. Atualmente, o escalonador dinâmico opta por executar primeiramente as tarefas mais pesadas, sendo as tarefas da lista de prontos de um recurso classificadas em ordem decrescente de peso. Assim, ao buscar tarefas para o reescalonamento, o escalonador prioriza as tarefas que se encontram no final da lista (tarefas menores).

Esta seleção de tarefas é feita em função da heurística de redistribuição implementada, onde um único recurso sobrecarregado cede tarefas para vários recursos subcarregados. O ideal é que o conjunto de tarefas cedido para o escalonamento seja composto por várias tarefas pequenas, facilitando sua distribuição entre diferentes recursos. Logo, a escolha de tarefas menores beneficia a redistribuição proposta, já que as tarefas são unidades de computação indivisíveis.

(d) Redistribuição de tarefas

Quando um escalonador dinâmico recebe um pacote de tarefas para serem distribuídas, uma política de redistribuição eficiente deve ser escolhida. O problema não é simples, pois as tarefas são indivisíveis, podem possuir pesos diferentes e cada máquina candidata a receber novas tarefas possui capacidade de processamento diferente. Na tentativa de simplificar este processo de redistribuição, os escalonadores dinâmicos utilizam uma heurística gulosa onde a maior tarefa é dada para a máquina mais subcarregada e assim sucessivamente (Algoritmo 7).

No processo de escalonamento também deve ser tratado o redirecionamento de mensagens. As tarefas de uma aplicação podem possuir mensagens que devem ser recebidas para que elas possam iniciar a sua computação. No caso da aplicação *bag-of-tasks* aqui tratada, cada tarefa deve receber do processo mestre pelo menos uma mensagem com os dados necessários para a sua execução.

As mensagens são guardadas em um *buffer* nos HMs onde as tarefas estão alocadas e só serão encaminhadas quando a tarefa for disparada para execução. Logo, quando uma tarefa é reescalonada para uma máquina diferente, as mensagens relativas à ela devem ser redirecionadas. Na política implementada, sempre que um pacote de tarefas é escolhido e enviado para reescalonamento, um pacote de mensagens relativo às tarefas cedidas é enviado em seguida. Após a realocação das tarefas, o trabalho de redirecionar novas mensagens é realizado pelos gerenciadores locais e global.

5.2.2 Escalonamento Dinâmico na Máquina

Na política para aplicações BoT, o escalonador dinâmico da máquina pode executar tarefas assim que elas ficam prontas, utilizando o modelo *dataflow*. Não é necessário que

a ordem definida pelo escalonador estático seja obedecida, já que as tarefas são independentes. As tarefas são inseridas na fila de prontos do HM em ordem decrescente de peso.

Além de definir a ordem em que as tarefas serão executadas em um recurso r_j, o HDS_j participa do reescalonamento de tarefas, sempre que este receber um pedido de tarefas do SDS do seu site (subseção 4.1.1, Algoritmo 1, linha 13). Como será explicado na seção seguinte, o HDS_j recebe um pedido que indica a *porcentagem* do peso de tarefas pertencentes a r_j que deve ser cedido, e não quais tarefas devem ser cedidas. Tais passos, executados por um HDS_j no contexto do escalonamento do site, podem ser visualizados no Algoritmo 8.

Algoritmo 8 : selecionar_tarefas_HDS$(perc_{ask})$

```
1   LP ← tarefas da lista de prontos em ordem decrescente de ε(v);
2   pacote_HDS ← ∅
3   pesoPedido ← (Σ_{v∈LP} ε(v)) × perc_ask;
4   minPeso ← min{ε(v)}/v ∈ LP;
5   enquanto (pesoPedido > minPeso) faça
6       pesoPedido ← pesoPedido - minPeso;
7       remove v de LP;
8       pacote_HDS ← pacote_HDS + {v};
9       minPeso ← min{ε(v)}/v ∈ LP;
    fim enquanto
10  se (pacote_HDS ≠ ∅) então
11      envia ack e pacote_HDS para o SDS;
    senão
12      envia nack ao SDS;
    fim se
```

5.2.3 Escalonamento Dinâmico no Site

As funções desempenhadas pelo SDS na estrutura hierárquica de escalonamento foram detalhadas no Algoritmo 2, subseção 4.1.2. Esta seção se concentra na descrição das políticas de escalonamento implementadas, onde o objetivo é que a carga de trabalho entre os recursos do site esteja distribuída de forma balanceada, considerando para isso o poder computacional dos seus recursos e o tamanho das tarefas da aplicação.

Para verificar se um *evento de escalonamento* de tarefas prontas deve ser disparado em um site s_k, o SDS_k deve verificar se o índice de desbalanceamento do site (sii_k) é maior que um limite θ, pré-definido. Para isso, tendo recebido o ert_j e o cp_j de cada recurso $r_j \in R_k$, o SDS_k calcula o *tempo de execução alvo* de s_k ($sert_k^*$, *site estimated remaining time*), que representa dentro dos valores recebidos qual seria o menor tempo de execução para s_k naquele momento, considerando apenas as tarefas prontas.

$$sert_k^* = \frac{\sum_{r_j \in R_k} ert_j \times cp_j}{\sum_{r_j \in R_k} cp_j} \tag{5.1}$$

Após o cálculo do *tempo de execução alvo* do site s_k, o SDS_k verifica qual o recurso mais atrasado, ou seja, com maior ert_j e verifica o quanto este recurso está distante do *tempo alvo* calculado. Desta forma, calcula-se o índice de desbalanceamento do site (sii_k), que na verdade representa o grau da má distribuição dos processos prontos entre os recursos de s_k, ou seja:

$$sii_k = \frac{\max_{r_j \in R_k}\{ert_j\}}{sert_k^*} \tag{5.2}$$

Várias políticas podem ser usadas para a redistribuição das tarefas, com o objetivo de que todos os recursos aproximem o seu tempo restante de execução do *tempo alvo* calculado para o site. Ou seja, recursos sobrecarregados devem ceder tarefas para os subcarregados. Um recurso r_j é considerado subcarregado se $ert_j < sert_k^*$, e sobrecarregado se $ert_j > sert_k^*$.

Nesta implementação, em cada evento de escalonamento apenas um recurso sobrecarregado, aquele com maior ert_j (denotado por r_{max}), cede sua carga de trabalho excedente, que é distribuída entre todos os recursos subcarregados, que são representados pelo conjunto $R_{sub} \subset R_k$. Logo, para aplicações BoT, o conjunto R_{ask} definido na linha 6 do Algoritmo 2 contém como único elemento o recurso r_{max}.

Quando um evento de escalonamento é disparado pelo SDS_k, uma mensagem é enviada a r_{max}, pedindo uma porcentagem do seu tempo de processamento restante ($perc_{ask}$). A porcentagem representa o quanto r_{max} está acima do *tempo de execução alvo* do site, onde:

$$perc_{ask} = 1 - \frac{1}{sii_k} \tag{5.3}$$

Usar a porcentagem ao invés de um valor fixo é mais flexível, visto que enquanto são feitos os cálculos para o escalonamento dinâmico, a quantidade de processos esperando por execução em r_{max} pode se alterar, já que os processos do usuário continuam sendo executados. Isto garante que nunca será pedida uma carga de trabalho superior ao que o recurso pode ceder, pois a porcentagem pedida será relativa ao número de tarefas alocadas no recurso no instante do pedido. Isto evita que um recurso fique oscilando entre os estados de sobrecarregado e subcarregado, repetidamente. O algoritmo escalonar_prontas_local(), disparado na linha 6 do Algoritmo 2 (subseção 4.1.2), pode ser visualizado abaixo:

Algoritmo 9 : escalonar_prontas_local()

```
1   R_ask ← ∅;
2   calcula sert*_k e sii_k;
3   se (sii_k > θ) então
4       calcula perc_ask e determina r_max e R_sub;
5       R_ask ← R_ask + r_max;
6       tarefas_ask ← perc_ask;
    fim se
7   retorna R_ask, R_sub e tarefas_ask;
```

O escalonador dinâmico da máquina, HDS_{max} alocado em r_{max}, ao receber a mensagem com a porcentagem pedida pelo SDS_k (Algoritmo 1, linha 13, subseção 4.1.1) irá analisar se é possível atendê-lo, selecionando tarefas para o reescalonamento (Algoritmo 8). De uma maneira geral, não é permitido ceder uma carga de trabalho superior a porcentagem pedida, logo, caso o peso da menor tarefa alocada em r_{max} exceda à $perc_{ask}$, uma negação do pedido ($nack$) é enviada como resposta ao SDS_k. Caso contrário, as tarefas de menor peso são selecionadas pelo HDS_{max} de acordo com $perc_{ask}$, e então enviadas juntamente com uma confirmação (ack) para o SDS_k.

Como apresentado no Algoritmo 3, assim que o SDS_k recebe um pacote com as tarefas cedidas por r_{max} ($pacote_{HDS}$), ele deve redistribuí-las entre os recursos subcarregados do seu site s_k, de acordo com a quantidade total de tarefas recebidas. Cada $r_j \in R_{sub}$ deverá receber uma carga de trabalho ($wload_{r_j}$) equivalente a uma porcentagem do peso total cedido (W_{HDS}), segundo as equações abaixo:

$$W_{HDS} = \sum_{v \in pacote_{HDS}} \varepsilon(v) \tag{5.4}$$

$$perc_{r_j} = \frac{(sert_k^* - ert_j) \times cp_j}{\sum_{r_i \in R_{sub}} ((sert_k^* - ert_i) \times cp_i)} \tag{5.5}$$

$$wload_{r_j} = perc_{r_j} \times W_{HDS} \tag{5.6}$$

Após o cálculo da carga de trabalho que cada recurso subcarregado deve receber, o mecanismo de redistribuição de tarefas executado pelo SDS é ativado. A política de distribuição é a mesma apresentada no Algoritmo 7, onde os parâmetros de entrada são: distribuir_tarefas($pacote_{HDS}$, R_{sub}, $wload_r$).

5.2.4 Escalonamento Dinâmico Global

O escalonador dinâmico global, GDS, tem o objetivo de manter o sistema balanceado considerando todos os sites da grade computacional disponível para a execução da aplicação do usuário. Para isso, com a média do tempo de execução restante ($\overline{ert_k}$) e a soma do poder computacional (tcp_k) recebido de cada site, o GDS calcula o índice de desbalanceamento do sistema (sii) e o *tempo de execução alvo* para toda as tarefas prontas da aplicação, denotado por $gert^*$ (*global estimated remaining time*).

$$gert^* = \frac{\sum_{s_k \in S} \overline{ert}_k \times tcp_k}{\sum_{s_k \in S} tcp_k} \tag{5.7}$$

$$sii = \frac{\max_{s_k \in S}\{\overline{ert}_k\}}{gert^*} \tag{5.8}$$

É importante ressaltar que ao se calcular o $gert^*$ da aplicação, o GDS se baseia apenas nas médias de tempo recebidas de cada site. Logo, o *tempo alvo* é uma estimativa aproximada do que seria o tempo de execução ideal naquele momento. O GDS considera também, que cada site s_k está internamente balanceado, segundo o limite de tolerância θ, explicado na Subseção 5.2.3.

Análogo ao processo de reescalonamento executado pelo SDS, o GDS irá disparar um evento de escalonamento se o índice de desbalanceamento do sistema for maior que um limite pré-determinado (denotado por ϕ), que não necessariamente é o mesmo limite definido para escalonamento dinâmico do site. Tal mecanismo pode ser verificado no Algoritmo 10 que é disparado na linha 3 do Algoritmo 5 (subseção 4.1.3).

Algoritmo 10 : escalonar_prontas_global()

```
1   S_ask ← ∅;
2   calcula gert* e sii;
3   se (sii > φ) então
4       calcula perc_ask e determina s_max e S_sub;
5       S_ask ← S_ask + s_max;
6       tarefas_ask ← perc_ask;
    fim se
7   retorna S_ask, S_sub e tarefas_ask;
```

No evento de escalonamento global, o GDS envia uma mensagem ao site mais sobrecarregado, s_{max} , pedindo uma porcentagem ($perc_{ask}$) da sua carga de processamento (Algoritmo 5, linha 7, subseção 4.1.3), que representa o quanto s_{max} está acima do $gert^*$. Assim como no evento de escalonamento local, a política de redistribuição adotada assume em cada evento de escalonamento que s_{max} é o único elemento do conjunto S_{ask}, e que sua carga de trabalho extra deve ser distribuída entre os sites subcarregados da grade, S_{sub}. O SDS situado em s_{max} recebe a porcentagem, $perc_{ask}$ (Algoritmo 4, linha 3, subseção 4.1.2) e a repassa para cada um dos HDSs do seu site. A porcentagem pedida é a mesma para todos os recursos, considerando que o site está balanceado. Os passos executados pelo SDS ao receber um pedido de tarefas do GDS são apresentados no Algoritmo 11.

Algoritmo 11 : selecionar_tarefas_SDS*($perc_{ask}$)*

```
1   repassa perc_ask para todos HDS_j do seu site
2   pacote_SDS ← ∅
3   para cada (ack recebido de HDS_j) faça
4       pacote_SDS ← pacote_SDS + pacote_HDS;
    fim para
5   se (pacote_SDS ≠ ∅) então
6       envia ack e o pacote_SDS para o GDS;
    senão
7       envia nack ao GDS;
    fim se
```

Assim como no escalonamento dinâmico local, cada HDS ao receber o pedido verifica se é possível ceder tarefas para o reescalonamento. Se possível uma mensagem é enviada com as tarefas cedidas, e caso contrário uma mensagem de negação é enviada. O SDS deve esperar até que todos os recursos respondam, e só depois enviar uma única mensagem com todas as tarefas cedidas empacotadas para o GDS (Algoritmo 11, linha 6).

Após receber o pacote com as tarefas ($pacote_{SDS}$), o GDS deve distribuí-las entre os sites subcarregados de acordo com os valores de $\overline{ert}_k$ e tcp_k (Algoritmo 6, linha 7, subseção 4.1.3). A distribuição é feita da mesma maneira usada no escalonamento dinâmico do site, quando o SDS utiliza a política descrita no Algoritmo 7, que possuirá os seguintes parâmetros: distribuir_tarefas($pacote_{SDS}$, S_{sub}, $wload_s$). Logo, seja S_{sub} o conjunto de sites subcarregados, cada $s_k \in S_{sub}$ deverá receber uma carga de trabalho ($wload_{s_k}$) calculada de acordo com as seguintes equações:

$$W_{SDS} = \sum_{v \in pacote_{SDS}} \varepsilon(v) \tag{5.9}$$

$$perc_{s_k} = \frac{(gert^* - \overline{ert}_k) \times tcp_k}{\sum_{s_i \in S_{Sub}} ((gert^* - \overline{ert}_i) \times tcp_i)} \tag{5.10}$$

$$wload_{s_k} = perc_{s_k} \times W_{SDS} \tag{5.11}$$

É importante ressaltar que os valores calculados para $wload_{r_j}$ e $wload_{s_k}$ indicam a carga ideal de trabalho que os recursos e sites subcarregados devem receber, respectivamente. Entretanto, como as tarefas da aplicação do usuário são indivisíveis pode acontecer que a distribuição de tarefas não satisfaça exatamente o valor calculado. Nestes casos, alguns recursos podem receber uma carga superior a desejada, enquanto outros recursos podem receber um pouco menos. Este problema se agrava com tarefas de pesos maiores. Assim, o peso efetivamente recebido por um site s_k é denotado por W_{s_k}.

Finalmente, no último passo do escalonamento global, cada SDS localizado nos sites subcarregados deve distribuir as tarefas recebidas entre todos os seus recursos (Algoritmo 4, linha 2, subseção 4.1.2). O peso total recebido pelo SDS do site s_k (W_{s_k}) é dividido em fatias proporcionais à soma do poder computacional do site s_k. Seguindo a política de redistribuição do Algoritmo 7, onde os parâmetros de entrada serão ($pacote_{GDS}, R_k, wslice_r$), cada recurso r_j em um site $s_k \in S_{sub}$, receberá o seguinte peso de tarefas:

$$wslice_{r_j} = \frac{W_{s_k}}{tcp_k} \times cp_j \tag{5.12}$$

5.2.5 Exemplo de Escalonamento BoT

Nesta subseção, será apresentado um exemplo de escalonamento global e local considerando a grade computacional apresentada na Figura 5.1. No exemplo, as tarefas das

aplicação de um usuário já estão alocadas nos 13 recursos heterogêneos disponíveis, divididos em 3 sites diferentes. Para cada recurso r_j está representado o seu poder computacional (cp_j), o tempo estimado, em segundos, para fim de execução das suas tarefas (ert_j) e o número de tarefas que aguardam na fila de execução (TQ_j). Para cada site é apresentado a soma do seu poder computacional (tcp_k) e a média do seu tempo de execução restante ($\overline{ert}_k$).

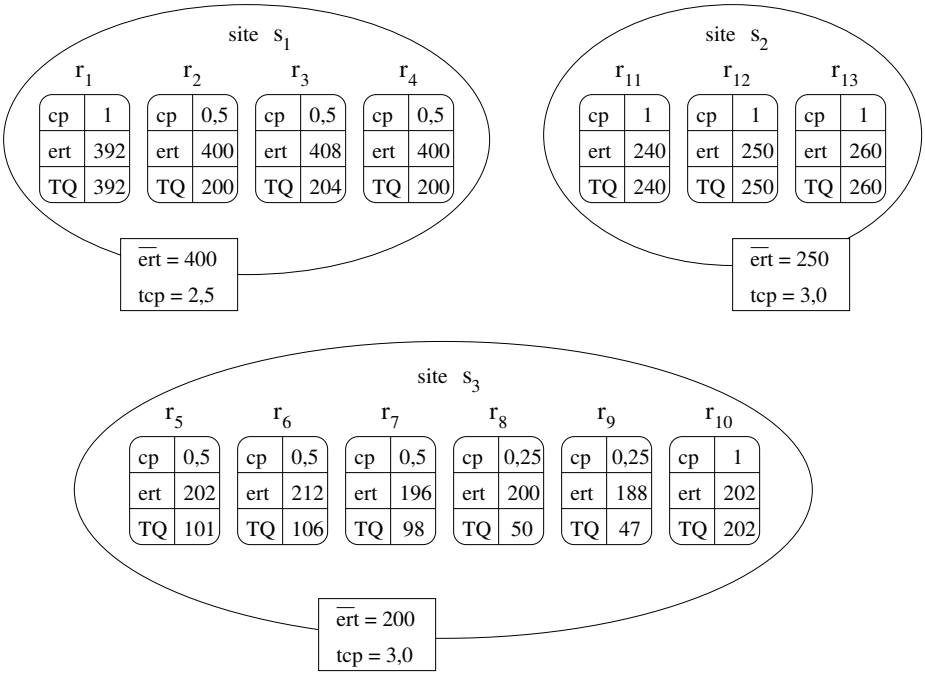

Figura 5.1: Estado dos recursos da grade antes do evento de escalonamento global.

A aplicação usada como exemplo é do tipo *bag-of-tasks*, e todas as suas tarefas possuem o mesmo peso, sendo o tempo de execução de cada tarefa na máquina com maior poder computacional igual a um segundo. Logo, neste exemplo, TQ representa também a soma dos pesos das tarefas que aguardam na fila de execução de um recurso.

Escalonamento Global

Seguindo o Algoritmo 5, o primeiro passo do GDS é verificar se o sistema está desbalanceado, calculando $gert^*$ e sii.

$$gert^* = \frac{(400 \times 2,5) + (250 \times 3,0) + (200 \times 3,0)}{2,5 + 3,0 + 3,0} = 276,47$$

$$sii = \frac{400}{276,47} = 1,45$$

Considerando que o índice de desbalanceamento global tolerável estabelecido para esta aplicação é 10% ($\phi = 1,1$), verifica-se que um evento de escalonamento global deve ser

disparado, pois $sii = 1,45 > \phi$. Neste caso, o GDS deve pedir uma porcentagem de tarefas ao site s_1, que é o site que apresenta maior $\overline{ert}$. A porcentagem pedida é então calculada e enviada a s_1 (Algoritmo 5, linha 7).

$$perc_{ask} = 1 - \frac{1}{1,45} = 0,31$$

Assim que o SDS situado em s_1 receber o pedido de tarefas ($perc_{ask}$), este é repassado para cada um de seus recursos, cujos respectivos HDSs passam a avaliar a possibilidade de ceder tarefas (Algoritmo 8). Logo, cada HDS irá verificar o peso a ser cedido de acordo com $perc_{ask}$ e com o peso total de tarefas (TQ) que existem em sua filas, onde $pesoPedido = TQ \times perc_{ask}$, segundo o Algoritmo 8, linha 3. Assim, o peso pedido para cada HDS é calculado da seguinte forma:

$$\begin{aligned}
\mathsf{HDS}_1 &= 392 \times 0,31 = 121,5 \rightarrow \text{equivale a 121 tarefas} \\
\mathsf{HDS}_2 &= 200 \times 0,31 = 62,0 \rightarrow \text{equivale a 62 tarefas} \\
\mathsf{HDS}_3 &= 204 \times 0,31 = 63,2 \rightarrow \text{equivale a 63 tarefas} \\
\mathsf{HDS}_4 &= 200 \times 0,31 = 62,0 \rightarrow \text{equivale a 62 tarefas}
\end{aligned}$$

Como neste exemplo as tarefas possuem um segundo de duração, basta converter o peso calculado para o número de tarefas que serão cedidas. Cada HDS seleciona as últimas tarefas da sua fila de espera (que em aplicações com pesos distintos representam as tarefas de menor peso) e as envia em um pacote para o SDS responsável pelo pedido. O SDS deve aguardar pela resposta de todos os HDS e só então, enviar uma resposta para o GDS (Algoritmo 11, linha 6). A Figura 5.2 mostra o número de tarefas cedidas para o GDS.

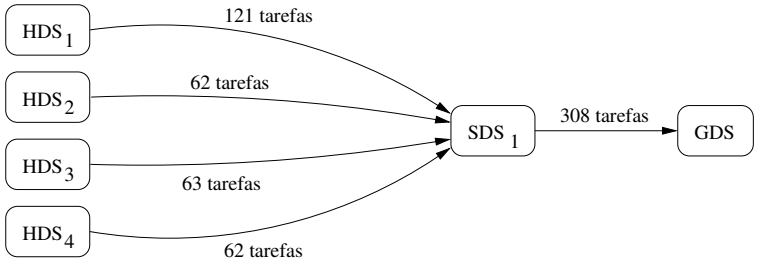

Figura 5.2: Tarefas que são cedidas por s_1 após o pedido do escalonador global.

Quando o GDS recebe as tarefas cedidas por s_1, ele deve distribuí-las entre os sites subcarregados, s_2 e s_3 (Algoritmo 6, linha 7). Para isso, é calculado $perc_{s_k}$ e $wload_{s_k}$, segundo as Equações 5.10 e 5.11, para saber quanto trabalho cada s_k pode receber.

$$perc_{s_2} = \frac{(276,47 - 250) \times 3,0}{((276,47 - 250) \times 3,0) + ((276,47 - 200) \times 3,0)} = \frac{79,41}{308,82} = 0,26$$

$$perc_{s_3} = \frac{(276,47 - 200) \times 3,0}{((276,47 - 250) \times 3,0) + ((276,47 - 200) \times 3,0)} = \frac{229,41}{308,82} = 0,74$$

$$wload_{s_2} = 308 \times 0,26 = 80,08 \rightarrow W_{s_2} = 80$$

$$wload_{s_3} = 308 \times 0,74 = 227,92 \rightarrow W_{s_3} = 228$$

Após o cálculo de quanto cada site subcarregado deve receber do pacote de tarefas cedidas no escalonamento, o GDS distribui as tarefas entre s_2 e s_3 (Algoritmo 6, linha 7) e encerra sua participação no escalonamento global. Quando cada SDS responsável pelos sites s_2 e s_3 recebe o pacote de tarefas, ele as distribui de acordo com a fatia de trabalho ($wslice_{r_j}$) que cada recurso pode receber (Algoritmo 4, linha 2). A fatia de trabalho para cada recurso é calculada a seguir, onde:

$$wslice_{r_j} = \frac{W_{s_k}}{tcp_k} \times cp_j$$

No site 2:

$$wslice_{11} = 80/3,0 \times 1,0 = 26,7$$
$$wslice_{12} = 80/3,0 \times 1,0 = 26,7$$
$$wslice_{13} = 80/3,0 \times 1,0 = 26,7$$

No site 3:

$$wslice_5 = 228/3,0 \times 0,5 = 38$$
$$wslice_6 = 228/3,0 \times 0,5 = 38$$
$$wslice_7 = 228/3,0 \times 0,5 = 38$$
$$wslice_8 = 228/3,0 \times 0,25 = 19$$
$$wslice_9 = 228/3,0 \times 0,25 = 19$$
$$wslice_{10} = 228/3,0 \times 1,0 = 76$$

A distribuição de tarefas entre os recursos subcarregados é realizada como apresentada na Figura 5.3.

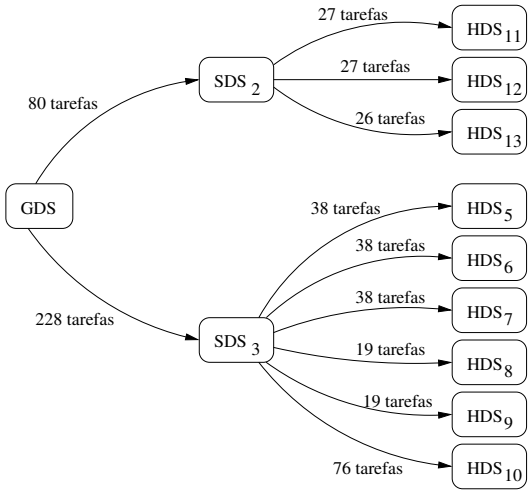

Figura 5.3: Tarefas que são redistribuídas entre os recursos de s_2 e s_3.

A distribuição das tarefas entre os recursos da grade após o evento de escalonamento global pode ser visualizada na Figura 5.4, onde $gert^* = 276,31$ e $sii = 1,001$, indicando assim, que o sistema está balanceado.

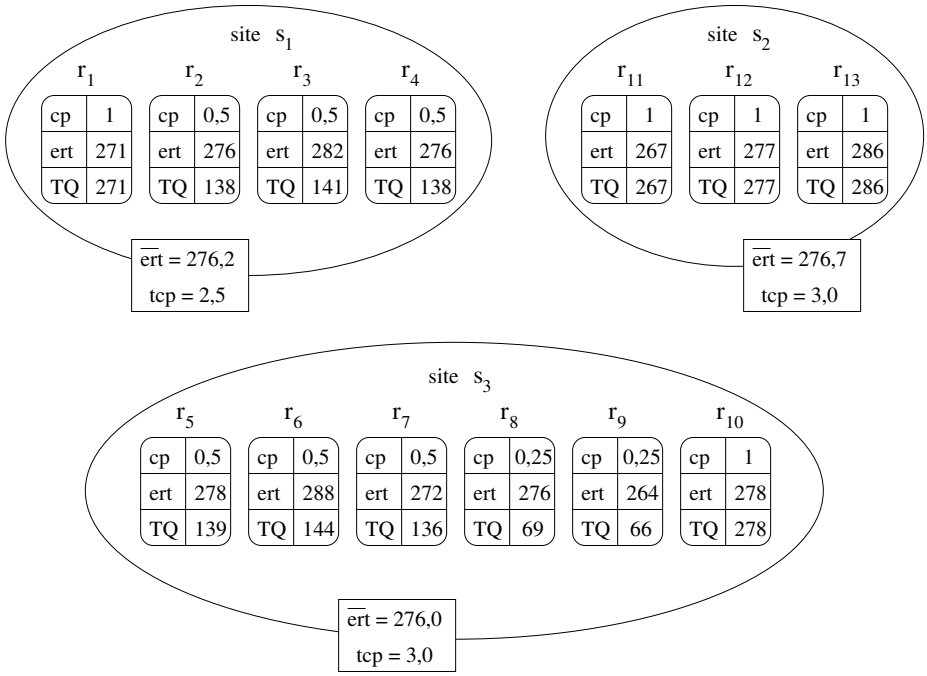

Figura 5.4: Distribuição final após escalonamento global.

Escalonamento Local

Tendo terminado o evento de escalonamento global e expirado o limite I_{min} entre intervalos de escalonamento, cada SDS_k (responsável pelo escalonamento de tarefas entre os recursos do site s_k) pode verificar se existe a necessidade de um evento de escalonamento local. Considerando a alocação de tarefas apresentada na Figura 5.4, e que o índice de desbalanceamento local tolerável estabelecido é de apenas 2% ($\theta = 1,02$), cada SDS_k decide disparar um evento de escalonamento, após o cálculo do *tempo de execução alvo* ($sert_k^*$) e do índice de desbalanceamento local (sii_k) de acordo com as Equações 5.1 e 5.2.

Site 1

$$sert_1^* = \frac{(271 \times 1,0) + (276 \times 0,5) + (282 \times 0,5) + (276 \times 0,5)}{1,0 + 0,5 + 0,5 + 0,5}$$

$$sert_1^* = 275,20 \qquad sii_1 = \frac{282}{275,20} = 1,025 \qquad perc_{ask} = 1 - \frac{1}{1,025} = 0,024$$

O SDS_1 identifica r_3 como o recurso que define o tempo de execução do site e calcula a porcentagem de tarefas que r_3 deve ceder para os recursos subcarregados, $R_{sub} = \{r_1\}$

(Algoritmo 9, linhas 2 a 4). Segundo a Equação 5.3, o SDS_1 pede ao HDS_3 $2,4\%$ da carga de trabalho associada ao seu HM, que neste exemplo equivale a 3 tarefas.

Seguindo o mesmo algoritmo, após o cálculo de $sert_2^*$ e sii_2, o SDS_2 determina que r_{max} deve ceder 3,2% de sua carga de trabalho para os recursos subcarregados, logo r_{13} cede 9 tarefas para r_{11}. Da mesma forma, o SDS_3, determina que r_6 deve ceder $3,6\%$ de sua carga de trabalho para os recursos subcarregados, $R_{sub} = \{r_7, r_8, r_9\}$.

Site 2

$$sert_2^* = \frac{(267 \times 1,0) + (277 \times 1,0) + (286 \times 1,0)}{1,0 + 1,0 + 1,0}$$

$$sert_2^* = 276,66 \qquad sii_2 = \frac{286}{276,66} = 1,033 \qquad perc_{ask} = 1 - \frac{1}{1,033} = 0,032$$

Site 3

$$sert_3^* = \frac{(278 \times 0,5) + (288 \times 0,5) + (272 \times 0,5) + (276 \times 0,25) + (264 \times 0,25) + (278 \times 1,0)}{0,5 + 0,5 + 0,5 + 0,25 + 0,25 + 1,0}$$

$$sert_3^* = 277,33 \qquad sii_3 = \frac{288}{277,33} = 1,038 \qquad perc_{ask} = 1 - \frac{1}{1,038} = 0,036$$

No site s_3 as tarefas cedidas no escalonamento devem ser divididas entre 3 recursos subcarregados. A distribuição destas tarefas é feita de acordo com o Algoritmo 7, usando os valores retornados pelas Equações 5.4, 5.5 e 5.6. Logo, sendo $W_{HDS} = 5$, s_7, s_8 e s_9 devem receber 2, 0 e 3 tarefas, respectivamente.

$$perc_{s_7} = \frac{(277,33 - 272) \times 0,5}{((277,33 - 272) \times 0,5) + (277,33 - 276) \times 0,25) + (277,33 - 264) \times 0,25)}$$

$$perc_{s_7} = 0,42 \qquad wload_{s_7} = 5 \times 0,42 = 2,1$$

$$perc_{s_8} = \frac{(277,33 - 276) \times 0,25}{((277,33 - 272) \times 0,5) + (277,33 - 276) \times 0,25) + (277,33 - 264) \times 0,25)}$$

$$perc_{s_8} = 0,05 \qquad wload_{s_8} = 5 \times 0,05 = 0,25$$

$$perc_{s_9} = \frac{(277,33 - 264) \times 0,25}{((277,33 - 272) \times 0,5) + (277,33 - 276) \times 0,25) + (277,33 - 264) \times 0,25)}$$

$$perc_{s_9} = 0,53 \qquad wload_{s_9} = 5 \times 0,53 = 2,65$$

O estado da aplicação na grade computacional após o reescalonamento local de tarefas é apresentado na Figura 5.5. O índice de desbalanceamento de cada site é $sii_1 = 1,003$, $sii_2 = 1,001$ e $sii_3 = 1,002$, estando dentro do limite de tolerância estabelecido para este exemplo ($\theta = 1,02$).

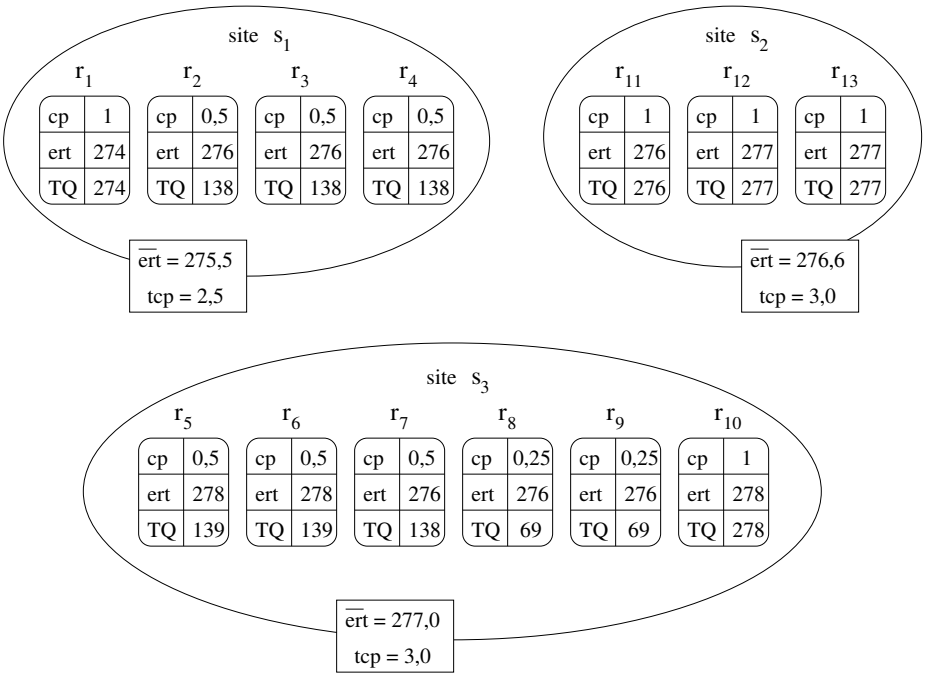

Figura 5.5: Distribuição final após escalonamento local.

5.3 Resumo

Este capítulo apresentou a abordagem de escalonamento dinâmico desenvolvida no contexto do *EasyGrid AMS*. Algumas dificuldades em relação à implementação das principais funções de um escalonador dinâmico foram analisadas, assim como as justificativas nas soluções escolhidas nesta implementação. Foi proposta uma estrutura de escalonamento hierárquico, onde políticas distintas podem ser aplicadas nos diferentes níveis hierárquicos. Tal característica visa oferecer uma melhor adaptação à execução em grade computacionais, podendo adequar as heurísticas escolhidas às políticas de acesso de diferentes organizações. Além disso, como cada aplicação possui sua própria hierarquia de escalonadores, o esforço de escalonar tarefas é dividido entre as diversas aplicações que compartilham os recursos da grade, tornando o sistema mais escalável. No próximo capítulo são apresentados os experimentos realizados para validação da estrutura proposta.

Capítulo 6

Análise Experimental

Neste capítulo é feita uma análise dos resultados obtidos nos diversos experimentos realizados, onde o objetivo é validar a política de escalonamento dinâmico proposta no contexto do *EasyGrid AMS*. Antes da descrição dos experimentos, serão apresentados o ambiente de execução usado, as características das aplicações executadas e as métricas de avaliação adotadas.

6.1 Ambiente de Avaliação

Os experimentos foram executados no Grid Sinergia, um ambiente de grade computacional real, que atualmente opera com recursos geograficamente distribuídos em 5 instituições do sudeste do Brasil. O Grid Sinergia é utilizado principalmente para a avaliação de *middlewares* desenvolvidos para ambientes grades, sendo operado como uma grade computacional aberta, sem um *RMS* global. Assim, usuários podem executar seus processos em qualquer recurso disponível, independentemente da capacidade computacional oferecida e das aplicações que já estejam em execução.

Para avaliação do escalonamento dinâmico é importante que custos reais, como a sobrecarga causada pelo próprio algoritmo e pelo sistema gerenciador de grades, sejam considerados. Enquanto é comum que a maioria das heurísticas de escalonamento sejam avaliadas através de simulações [35, 43, 82, 99], este trabalho apresenta resultados obtidos em ambientes de execução reais.

Entretanto, para evitar a influência de outras aplicações nos resultados medidos, os testes são realizados em um ambiente semi-controlado, isto é, os recursos da grade computacional são dedicados apenas às aplicações que estão sendo avaliadas. Para isso, são utilizados somente os recursos com acesso exclusivo, oferecidos pelo *Smart Grid Computing Lab* (SGCLab), localizado no Instituto de Computação da Universidade Federal Fluminense. Apesar disso, não é possível afirmar que a grade é totalmente controlada, visto que os canais de comunicação são compartilhados, podendo gerar alguma sobrecarga extra no sistema.

Para simulação de heterogeneidade ou compartilhamento dos recursos, aplicações extras do tipo CPU-bound podem ser disparadas nos recursos desejados.

A configuração do SGCLab, disponível para os testes das aplicações BoT, é composta por 32 processadores Pentium IV 2.6 GHz com 512Mb RAM, rodando Linux Fedora Core 2, Globus Toolkit 2.4 e LAM/MPI 7.0.6. As máquinas estão divididas em 4 sites, interconectados por switches Fast Ethernet e Gigabit Ethernet, assim como apresentado na Figura 6.1, onde os sites 1 e 3 possuem 8 máquinas cada, o site 2 possui 13 máquinas e o site 4 possui 3 máquinas.

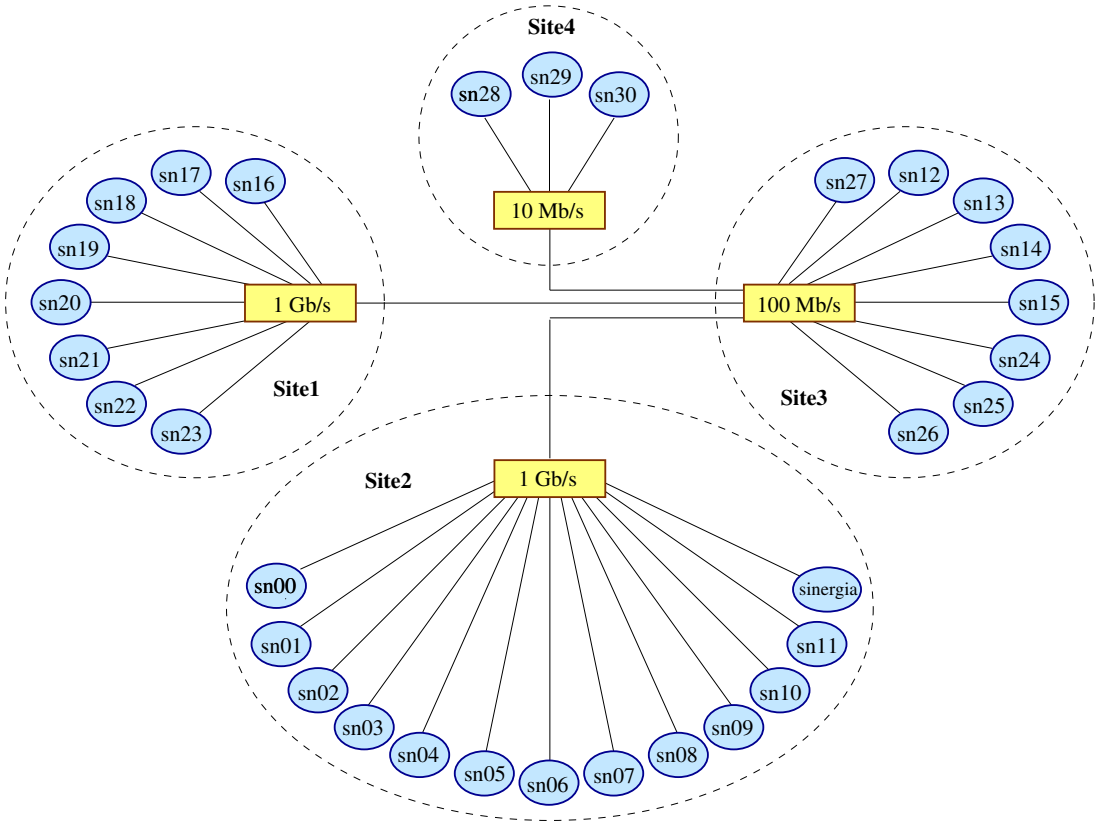

Figura 6.1: Configuração da grade computacional do SGCLab nos experimentos realizados para aplicações BoT.

6.2 Aplicações Avaliadas

Embora o objetivo final deste trabalho seja o estudo de políticas de escalonamento dinâmico que lidem com diferentes classes de aplicações paralelas representadas por *GADs*, a política descrita no Capítulo 5 trata a classe de aplicações do tipo *bag-of-tasks*. Apesar de não possuírem relações de precedência, aplicações *bag-of-tasks* também podem ser representadas por um grafo acíclico direcionado do tipo *fork-join*. Neste grafo, as tarefas recebem da tarefa origem os dados de entrada necessários para execução, e retornam os resultados para a tarefa destino. Nesta tese duas aplicações são utilizadas para a execução destes testes: PSwp e Termions.

Aplicações do tipo *parameter sweep* (PSwp) avaliam um determinado problema de acordo com os seus parâmetros de entrada que são modificados a cada execução. Estas aplicações são comumente executadas em ambientes como as grades computacionais. Um exemplo conhecido são as aplicações *parameter sweep* para comparação e avaliação de heurísticas de escalonamento, onde cada tarefa da aplicação PSwp é formada pelo algoritmo de escalonamento e seus respectivos parâmetros. Para automatização destes testes um portal de escalonamento [102] foi desenvolvido, onde é possível selecionar as heurísticas de escalonamento que serão avaliadas e seus parâmetros de entrada. Neste exemplo, o tempo de execução de cada tarefa da aplicação PSwp dependerá principalmente da complexidade da heurística escolhida, da quantidade de tarefas do GAD de entrada e da arquitetura disponível.

Para uma melhor avaliação da heurística de escalonamento dinâmico desenvolvida, os testes realizados utilizam uma versão sintética da aplicação PSwp. Desta forma, é possível controlar exatamente a duração das tarefas da aplicação, tornando possível investigar o impacto de diferentes pesos de tarefas no desempenho da heurística híbrida utilizada nesta versão do *EasyGrid AMS*.

A outra aplicação usada, Termions, é uma implementação de um problema real que computa a dispersão termal macroscópica em meios porosos [106]. Dado um meio poroso, composto de elementos fluidos e sólidos, a dispersão termal é calculada pelo movimento de partículas hipotéticas (*termions*) através do meio poroso. Para este cálculo, são usados valores como a posição inicial dos termions, sua energia e as propriedades dos meios sólido e fluido. Logo, o custo computacional para se calcular o caminho percorrido por um termion varia e não pode ser determinado *a priori*. Cada tarefa da aplicação determinará o caminho percorrido por um único termion.

Em todos os testes realizados, com exceção dos apresentados na Subseção 6.4.5, os recursos onde são alocados o GM (gerenciador global) e os SMs (gerenciadores do site) não executam tarefas da aplicação do usuário. Estas tarefas são distribuídas e executadas nos demais recursos da grade computacional, de forma que possa ser feita uma melhor análise dos tempos de execução obtidos, sem a interferência das atividades do GM e dos SMs. Nas duas aplicações, Termions e PSwp, a tarefa origem e destino é a mesma, sendo a única executada pelo gerenciador global da aplicação.

6.3 Métricas de Avaliação e Parâmetros de Execução

Para cada execução de uma aplicação na grade computacional, é calculado o desvio padrão entre os tempos de término de execução da aplicação em cada recurso $r_j \in R$ disponível. O uso do desvio padrão tem como objetivo indicar se a carga de trabalho foi bem distribuída entre os recursos da grade. Quanto menor a variação entre os tempos de término dos recursos (ft_j), para uma determinada aplicação, menor será o desvio padrão

e, portanto, melhor terá sido a distribuição de tarefas. O desvio padrão (*stdev*) é calculado da seguinte forma:

$$avg_{ft} = \frac{\sum_{r_j \in R} ft_j}{|R|} \tag{6.1}$$

$$stdev = \sqrt{\frac{\sum_{r_j \in R} (ft_j - avg_{ft})^2}{|R|}} \tag{6.2}$$

No caso da execução de aplicações em ambientes controlados, o *makespan* obtido também pode ser usado para avaliar a eficiência da heurística proposta. Quanto menor a diferença entre o tempo de execução obtido e o *makespan ótimo*, melhor é o desempenho da heurística. Entretanto, em ambientes compartilhados e dinâmicos, os valores obtidos para o *makespan* não indicam se o escalonador teve um bom desempenho ou não. Nestes casos, é difícil avaliar o poder computacional obtido pela aplicação durante a sua execução, já que o conjunto de recursos existentes além de ser compartilhado com outros usuários, pode não estar integralmente disponível durante toda a execução da aplicação.

Os parâmetros de execução usados pelos escalonadores dinâmicos nos testes realizados nas próximas seções podem ser vistos na Tabela 6.1. Conforme apresentado no Capítulo 4, nt_j, denota o número de processos da aplicação que executam concorrentemente; I_{min}, indica o intervalo mínimo entre cálculos de desbalanceamento; $I_{monitor}$, indica o intervalo máximo entre mensagens de monitoramento; θ e ϕ representam, respectivamente, os índices de desbalanceamento local e global. A escolha destes valores foi discutida no Capítulo 5, porém mais testes podem ser realizados para avaliação destes parâmetros. Em todos os testes realizados neste capítulo, os valores apresentados representam a média de no mínimo três execuções. O tempo de execução de uma tarefa em uma máquina ociosa, com $csi = 1$, será denotado por t_{exe}.

Tabela 6.1: Parâmetros de execução

nt_j	I_{min}	$I_{monitor}$	θ	ϕ
2	1s	5s	10%	10%

6.4 Experimentos e Avaliação dos Resultados

6.4.1 Análise do Desvio Padrão

Nesta subseção, os testes realizados têm o objetivo de analisar a relação do desvio padrão com o tempo de execução total de uma aplicação, com o peso das tarefas, e com o poder computacional de um recurso. Desta maneira, deseja-se identificar as características da aplicação ou da arquitetura que mais exercem influência na variação do desvio padrão.

Para tais testes, foi utilizada uma aplicação PSwp, que é disparada no site 2 da grade computacional, descrita anteriormente. A Figura 6.2 mostra a configuração virtual do site, onde dois recursos (sinergia e sn00) são dedicados apenas à execução do GM e do SM, enquanto que os demais com um HM cada, executam as tarefas da aplicação PSwp. Para simular a heterogeneidade entre os recursos do site, sn02, sn04 e sn08 executam cargas extras de processamento, disponibilizando para a aplicação do usuário 2/5, 1/2 e 2/3 do seu poder computacional, respectivamente.

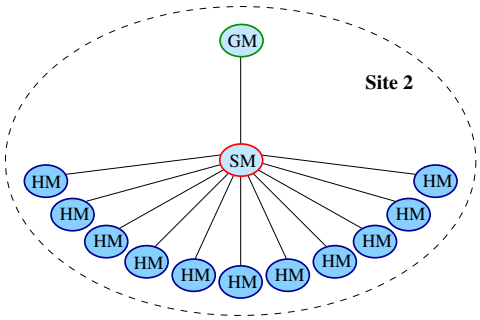

Figura 6.2: Configuração virtual do site 2.

Inicialmente, as tarefas da aplicação são distribuídas de maneira uniforme entre os HMs do site, visto que nestes experimentos, a carga extra de processamento imposta a alguns recursos, só é descoberta pela aplicação durante a sua execução. Assim, o esforço de balancear o sistema é deixado para o escalonador dinâmico, de forma que seja possível verificar sua eficiência.

Desvio padrão em relação ao tempo de execução da aplicação

Na análise da variação do desvio padrão em relação ao tempo de execução de uma aplicação são executadas várias instâncias de PSwp, variando a sua quantidade total de tarefas. Neste caso, as tarefas possuem sempre o mesmo tempo de execução, equivalente a 5 segundos em uma máquina ociosa. Além disso, as instâncias de PSwp foram elaboradas de forma que fosse possível uma *distribuição perfeita* de tarefas entre os recursos do site, considerando o poder computacional disponível.

Para cada instância do teste, o tempo total de execução teórico da aplicação é estabelecido em 5, 10, 20, 30, 40, 60 e 80 minutos, e a distribuição perfeita se caracteriza por alcançar um desvio padrão nulo, onde todos os recursos terminam a execução das tarefas ao mesmo tempo. Logo, para a aplicação com duração de 5 minutos, na distribuição perfeita, cada recurso dedicado deve executar 60 tarefas, enquanto que sn02, sn04 e sn08, devem executar, respectivamente, 24, 30 e 40 tarefas, de acordo com o poder computacional oferecido. Para o cálculo desta distribuição, não foram considerados os custos introduzidos pelo *middleware*, apenas o peso das tarefas e o poder computacional dos recursos.

A Tabela 6.2 mostra a média do desvio padrão obtida para cada uma das instâncias usadas. É importante ressaltar que nas aplicações com duração de 5, 10 e 20 minutos, o escalonador dinâmico consegue a distribuição perfeita de tarefas, apesar do desvio padrão não ter sido nulo. Isto acontece, pois os recursos sobrecarregados (sn02, sn04 e sn08) possuem um maior número de processos executando concorrentemente, gastando mais tempo com trocas de contexto entre os seus processos. Isto produz um tempo de execução total um pouco maior do que nos recursos menos sobrecarregados do site.

Tabela 6.2: Desvio Padrão em relação ao tempo de execução da aplicação PSwp

t_{exe}	*makespan teórico*	*stdev*
5s	5 min	0,54
	10 min	1,16
	20 min	2,96
	30 min	3,53
	40 min	3,37
	60 min	3,99
	80 min	2,26

A distribuição teórica perfeita não leva em consideração o tempo gasto com trocas de contexto, logo, apesar do desvio padrão teórico ser zero, isto não é obtido na prática. Além disso, a medida que o número de tarefas da aplicação aumenta, a tendência das máquinas mais sobrecarregadas é gastar ainda mais tempo com trocas de contexto e, portanto, aumentar o desvio padrão. Isto pode ser observado na Tabela 6.2, onde existe um aumento gradual do desvio padrão para as primeiras instâncias da aplicação PSwp.

Por outro lado, à medida que o desbalanceamento do site aumenta e a distribuição teórica perfeita não é a melhor opção encontrada, o escalonador dinâmico consegue optar por outras distribuições de tarefas que minimizem o *makespan final* e o desvio padrão. Isto acontece nas instâncias de 30, 40, 60 e 80 minutos, onde o comportamento crescente do desvio padrão não é mais verificado. O que se percebe, é que os valores do desvio padrão não são diretamente ligados com a quantidade de tarefas (ou tempo de execução) de uma aplicação, mas sim, com a variação entre os poderes computacionais dos recursos.

O gráfico apresentado na Figura 6.3, mostra uma comparação entre o desvio padrão que seria obtido caso o escalonador dinâmico sempre optasse pela distribuição teórica perfeita das tarefas de PSwp, e o desvio padrão real obtido nos testes, que foi sempre menor do que o peso de uma única tarefa. A projeção do desvio padrão é feita a partir da execução da aplicação com duração de 5 minutos, onde são considerados os *makespans* de cada recurso e a distribuição perfeita obtida. A partir daí, para cada instância de PSwp, é mantida, na mesma proporção, a distribuição perfeita de tarefas, atualizando os *makespans* dos recursos, segundo o número de tarefas que cada um deve executar.

A aplicação dos Termions também foi executada, aumentando gradativamente o número de tarefas da aplicação. A mesma conclusão obtida nos testes com PSwp pode ser verificada, isto é, não existe relação entre o aumento do desvio padrão e o aumento do número de tarefas da aplicação. Entretanto, como os Termions não possuem tarefas com peso exatamente conhecido (aproximadamente entre 4 e 5 segundos), é difícil identificar a distribuição ótima e, portanto, fazer análises mais profundas.

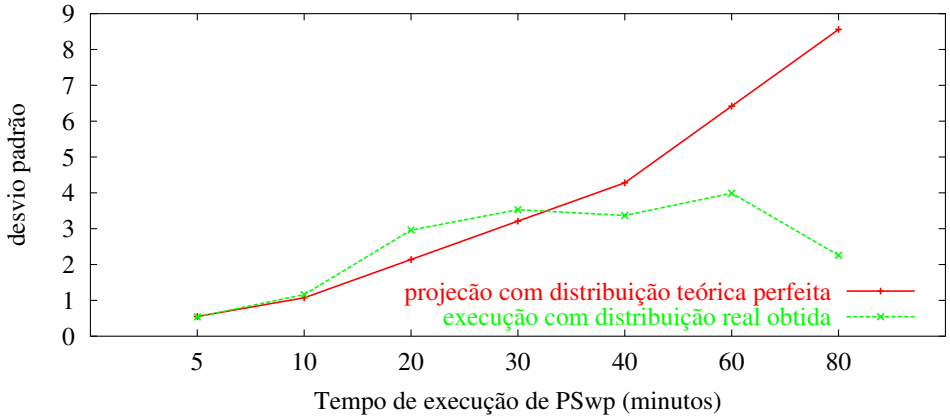

Figura 6.3: Comparação entre o desvio padrão real obtido na execução de PSwp e a sua projeção considerando a distribuição teórica perfeita das tarefas.

Desvio padrão em relação ao tempo de execução das tarefas da aplicação

Para verificar se o peso (tempo de execução) das tarefas da aplicação influencia nos valores obtidos para o desvio padrão, são executadas três instâncias de PSwp, onde são variados os pesos das tarefas. Neste caso a primeira instância de PSwp possui apenas tarefas com duração de 1 segundo, a segunda instância trabalha com tarefas de 5 segundos, e a terceira instância com tarefas de 10 segundos.

Diferentemente do experimento anterior, onde os tempos de execução estimados das instâncias são variáveis, nestes testes os *makespans* teóricos de PSwp são constantes e fixados em 5 minutos. Como cada instância possui tarefas com pesos diferentes, para que cada uma delas gaste o mesmo tempo de execução, o número de tarefas da aplicação também varia. As médias dos valores obtidos para o desvio padrão são apresentadas na Tabela 6.3.

Tabela 6.3: Desvio Padrão em relação ao tempo de execução das tarefas da aplicação PSwp

makespan teórico	t_{exe}	*stdev*
5 min	1s	0,30
	5s	0,54
	10s	4,97

Os testes realizados mostram que à medida que o peso da tarefa aumenta, maior pode ser o valor do desvio padrão. Isto pode ocorrer, pois quanto maior uma tarefa mais difícil pode se tornar o balanceamento do sistema, visto que tarefas são unidades de computação indivisíveis. Além disso, como a política de escalonamento não reescalona tarefas que já estão em execução, maior também pode ser a diferença entre os tempos de término dos recursos quando existirem alguns recursos mais lentos do que outros na grade computacional. Na implementação atual, a partir do momento que uma tarefa é disparada em um recurso, mesmo que este seja ou se torne lento, ela não pode ser interrompida.

Outro ponto verificado é com relação ao número de processos que executam concorrentemente. Apesar de alguns benefícios encontrados [90], um grande número de processos disparados ao mesmo tempo prejudica a ação do escalonador dinâmico, que teoricamente possuirá um conjunto menor de tarefas que podem ser reescalonadas, pois atualmente, não é possível migrar tarefas em execução.

6.4.2 Benefícios da Política de Escalonamento Híbrido

Para validar a utilização do escalonamento híbrido, uma aplicação PSwp foi executada nos sites 1, 2 e 3 da grade computacional, usando diferentes políticas de escalonamento. A configuração virtual dos sites é apresentada na Figura 6.4 onde apenas os recursos que possuem HMs executam tarefas da aplicação. Para caracterizar a heterogeneidade do sistema, cargas de trabalho extra foram disparadas em alguns recursos da grade.

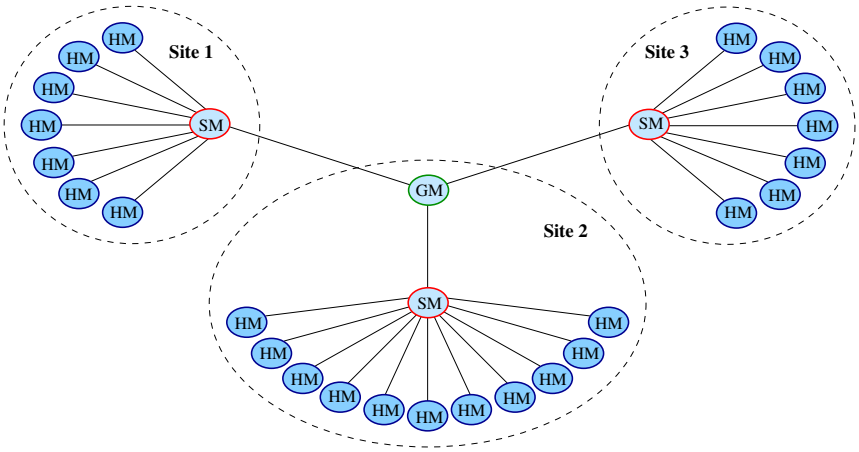

Figura 6.4: Configuração virtual dos sites 1, 2 e 3.

Uma versão heterogênea da aplicação PSwp é usada, onde as tarefas possuem custos de computação diferentes. Neste caso, metade das tarefas possui tempo de execução equivalente a 1 segundo cada, enquanto 1/4 possui duração de 5 segundos cada, e o restante, 10 segundos cada. A aplicação foi executada com 1.000, 5.000 e 10.000 tarefas e as políticas de escalonamento usadas foram as seguintes:

1 escalonamento *round-robin* usado pelo MPI, sem escalonamento dinâmico;

2 escalonamento dinâmico considerando a alocação estática inicial definida em (1);

3 escalonamento estático do tipo *list-scheduling* empregado pelo Portal *EasyGrid*, sem escalonamento dinâmico;

4 escalonamento dinâmico considerando a alocação estática inicial definida em (3).

Enquanto que nas políticas (1) e (2) o *AMS* não possui nenhum conhecimento prévio sobre as características da aplicação, nas políticas (3) e (4) os escalonadores possuem informações dos processos da aplicação como, por exemplo, as estimativas dos seus tempos de execução. Os *makespans* obtidos em cada política são apresentados na Tabela 6.4, juntamente com o desvio padrão global medido em cada execução.

Tabela 6.4: Comparação entre políticas de escalonamento diferentes no *EasyGrid AMS*

N		(1)	(2)	(3)	(4)
1000	*mspan*	392,81	211,86	191,92	191,26
	stdev	68,65	8,37	0,44	0,26
5000	*mspan*	1926,15	969,07	960,51	954,48
	stdev	367,01	7,30	1,96	0,75
10000	*mspan*	3522,50	1926,84	1911,79	1908,97
	stdev	669,53	6,98	2,22	0,99

Como pode ser observado, embora a política (1) apresente os piores resultados, os valores obtidos em (2) mostram como o escalonador dinâmico consegue melhorar uma alocação inicial ruim. Entretanto, apesar da significante melhoria, os *makespans* obtidos pela política (2) não são tão bons quanto os apresentados em (4). Isto ocorre, pois, o escalonador dinâmico em (2) não possui informações sobre o tempo de execução das tarefas da aplicação. Os tempos de execução obtidos pela política (3) indicam que escalonamentos estáticos eficientes podem produzir bons resultados quando a grade computacional é estável, característica que não é comum nestes tipos de ambientes como o Grid Sinergia.

Outra conclusão importante, é que a alocação estática inicial pode influenciar diretamente no desempenho do escalonador dinâmico. Em (2), devido a má alocação inicial, o escalonador dinâmico foi ativado várias vezes, na média 25 escalonamentos globais foram feitos, enquanto que na política (4), isto ocorreu apenas duas vezes.

Os melhores resultados são produzidos pela política de escalonamento híbrido apresentada em (4), aonde mesmo a boa alocação inicial fornecida pelo escalonamento estático é melhorada. Isto se deve ao fato, de pequenas variações no poder computacional dos recursos não poderem ser percebidas pelo escalonador estático em tempo de compilação, mesmo se tratando de um ambiente dedicado. Logo, mesmo para uma aplicação do tipo

bag-of-tasks, uma abordagem híbrida pode trazer vantagens. Assim, os benefícios podem ser ainda maiores para aplicações com restrições de precedência representadas por *GADs*. Nestes casos, como existe dependência entre as tarefas, as características da aplicação se tornam ainda mais importantes nas decisões tomadas por um escalonador.

6.4.3 Escalonamento Dinâmico Distribuído

Estes experimentos foram realizados para analisar o comportamento e a eficiência do escalonamento dinâmico distribuído e descentralizado utilizado pelo *EasyGrid AMS*. Neste caso, mais de uma aplicação é executada simultaneamente para que se possa verificar como o escalonador gerencia as tarefas da aplicação do usuário entre os recursos disponíveis.

Duas aplicações PSwp com 1.000 tarefas de pesos iguais são executadas ao mesmo tempo, onde para cada instância do teste varia-se o tempo de execução associado as tarefas em 1, 5 e 10 segundos. A aplicação PSwp1 executa nos sites 1 e 2 da grade computacional, enquanto PSwp2 utiliza os sites 2 e 3. Logo, os recursos do site 2 são compartilhados entre as duas aplicações. A Figura 6.5 mostra a grade virtual de cada uma das aplicações, onde apenas os recursos com HMs executam processos da respectiva aplicação.

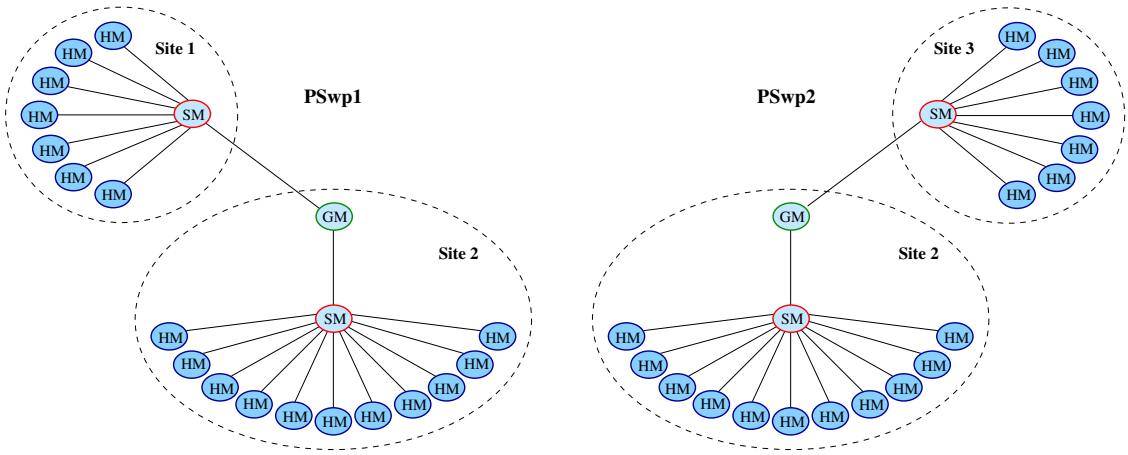

Figura 6.5: Grade virtual das aplicações PSwp1 e PSwp2.

Inicialmente, o escalonador estático distribui as tarefas de cada uma das aplicações PSwp1 e PSwp2 entre os HMs disponíveis. A distribuição é feita de maneira uniforme, pois, o escalonador de cada aplicação não possui conhecimento sobre a futura existência da outra aplicação na grade computacional. Os testes são realizados variando também, a política de HDS utilizada, com o objetivo de analisar se cada escalonador consegue satisfazer de maneira eficiente os requisitos da aplicação de acordo com a política adotada.

Aplicações com a mesma política de HDS

Nestes testes, PSwp1 e PSwp2 foram executadas de acordo com a mesma política de HDS, onde duas tarefas de uma aplicação são executadas concorrentemente em um mesmo

processador, sempre que este se encontrar disponível. Como as aplicações são idênticas e suas tarefas são distribuídas em um mesmo número de recursos com o mesmo poder computacional, espera-se que elas obtenham o mesmo *makespan*.

Os experimentos foram divididos em duas partes, onde na primeira parte as aplicações PSwp são executadas isoladamente (execução isolada) para que se determine o *makespan* sem interferências externas, e na segunda parte PSwp1 e PSwp2 são executadas simultaneamente (execução simultânea). Os valores obtidos para cada tipo de execução, *isolada* ou *simultânea*, podem ser vistos na Tabela 6.5.

Os tempos de execução medidos são indicados na coluna *mspan* e seus valores teóricos mínimos (limite inferior do *makespan*), que excluem todos os custos associados com a *gridificação* da aplicação, com o *EasyGrid AMS* e com o sistema operacional são apresentados na coluna *opt*. Para a execução simultânea, além destes valores, é calculado o tempo de execução esperado (*exp*) que é baseado no *makespan* conseguido na execução individual, porém considerando que cada aplicação não possui uso exclusivo dos recursos do site 2 ($exp = mspan_{isolada} \times opt_{simultanea}/opt_{isolada}$).

Para o cálculo do valor *opt* considera-se que cada recurso pode receber um número x de tarefas de acordo com o seu poder computacional. A distribuição é feita de forma que o *makespan* total da aplicação seja minimizado, e o valor *opt* será dado pelo recurso que oferecer maior tempo de execução. Durante a distribuição, é importante considerar que cada tarefa é uma unidade de computação indivisível.

Tabela 6.5: *Makespans* de PSwp1 e PSwp2 com mesma política de HDS

		isolada		simultânea		
	t_{exe}	*mspan*	*opt*	*mspan*	*opt*	*exp*
PSwp1	1s	57,20	56,0	83,28	80,0	81,71
	5s	282,64	280,0	409,42	400,0	403,77
	10s	564,17	560,0	816,04	800,0	805,96
PSwp2	1s	57,35	56,0	83,22	80,0	81,93
	5s	283,48	280,0	408,73	400,0	404,97
	10s	565,75	560,0	815,06	800,0	808,21

Na execução isolada a comparação entre *mspan* e *opt* mostra que a sobrecarga introduzida pelo *EasyGrid AMS* é bem pequena, não ultrapassando 2,5%, valor que ainda diminui à medida que o peso das tarefas aumenta. Com relação a execução simultânea, os resultados obtidos mostram que as aplicações PSwp1 e PSwp2 praticamente obtém o mesmo *makespan*, o que é esperado dada a similaridade entre as grades virtuais de cada aplicação e das políticas de escalonamento usadas. Neste caso, os *makespans* obtidos foram no máximo 1,9% piores que os valores esperados (*exp*) e 4,1% piores que os valores ótimos

(*opt*). É importante ressaltar que *opt* é um valor impossível de ser alcançado, se os custos associados a execução de uma aplicação em uma grade computacional forem devidamente contabilizados.

Os resultados obtidos indicam que o escalonador distribuído é capaz de gerenciar as aplicações eficientemente em um ambiente dinâmico. Ainda para a verificação da eficiência do escalonador dinâmico, a Tabela 6.6 mostra o desvio padrão medido quando PSwp1 e PSwp2 executam juntas. Como esperado, o desvio padrão cresce à medida que o peso das tarefas aumenta, devido ao fato de cada tarefa ser indivisível. Os valores na coluna *global* são relacionados ao desempenho da política de escalonamento do GDS, enquanto os outros valores se referem a política de escalonamento desempenhada pelos SDSs, dentro de cada site.

Tabela 6.6: Desvio padrão na execução simultânea de PSwp1 e PSwp2, com políticas de HDS idênticas

	t_{exe}	site 1	site2	site3	global
PSwp1	1s	0,03	0,55	-	0,88
	5s	1,74	1,83	-	1,85
	10s	3,45	2,77	-	3,58
PSwp2	1s	-	0,72	0,07	0,63
	5s	-	1,55	0,23	1,83
	10s	-	2,81	0,46	3,61

Na política de escalonamento atual, processos em execução não são considerados para o reescalonamento e nenhum mecanismo de replicação de tarefas é usado. Logo, considerando que nestes experimentos o HDS permite a execução de duas tarefas de uma aplicação concorrentemente em um mesmo recurso, o desvio padrão de até duas vezes o tempo de execução das tarefas pode ser esperado.

Aplicações com diferentes políticas de HDS

Nestes testes, os mesmos passos do experimento anterior são seguidos, apenas com uma única diferença: PSwp1 adota uma política de HDS onde os seus processos só podem ser disparados em um recurso, se este for considerado ocioso (política HDS/ociosa, Subseção 4.1.1). A aplicação PSwp2, não sofre alterações, continuando com a mesma política de escalonamento do teste anterior.

Os resultados obtidos são apresentados na Tabela 6.7, que mostra que na execução isolada das aplicações os *makespans* obtidos são praticamente iguais, e sem diferenças significativas em relação ao teste anterior. Entretanto, na execução simultânea, quando as aplicações compartilham os recursos do site 2, os tempos de execução obtidos por PSwp2

sofrem uma pequena alteração em relação a execução isolada. Esta interferência, de menos de 0,8%, é devido aos HMs da aplicação PSwp1, cujos HDSs verificam de tempos em tempos se o recurso compartilhado já se encontra ocioso.

Tabela 6.7: *Makespans* de PSwp1 e PSwp2 com políticas de HDS diferentes

		isolada		simultânea		
	t_{exe}	*mspan*	*opt*	*mspan*	*opt*	*exp*
PSwp1	1s	57,67	56,0	92,94	90,0	92,68
	5s	283,34	280,0	456,68	450,0	455,36
	10s	565,71	560,0	911,29	900,0	909,17
PSwp2	1s	57,36	56,0	57,81	56,0	57,36
	5s	283,22	280,0	284,73	280,0	283,22
	10s	565,43	560,0	568,20	560,0	565,43

Ao utilizar a política HDS/ociosa, a aplicação PSwp1 só utiliza os recursos do site 2 quando PSwp2 termina a sua execução. Desta forma, os *makespans* alcançados por PSwp1 na execução simultânea, são maiores. Porém, estes tempos não dobram, pois a política de GDS permite que as tarefas inicialmente alocadas no site 2 sejam reescalonadas para o site 1, que é dedicado exclusivamente a aplicação PSwp1. Quando PSwp2 termina, o GDS pode novamente reescalonar as tarefas de PSwp1, balanceando a carga de trabalho restante entre os sites 1 e 2.

O desvio padrão obtido com a execução simultânea das aplicações pode ser visualizado na Tabela 6.8. Os resultados dos testes, de uma maneira geral, indicam mais uma vez, que a abordagem de escalonamento distribuído entre as aplicações consegue atingir bons resultados, distribuindo os recursos disponíveis de maneira eficiente entre as aplicações que compartilham a grade.

Tabela 6.8: Desvio padrão na execução simultânea de PSwp1 e PSwp2, com políticas de HDS diferentes

	t_{exe}	site 1	site2	site3	global
PSwp1	1s	0,74	0,91	-	0,85
	5s	1,80	1,96	-	2,03
	10s	4,52	6,99	-	6,84
PSwp2	1s	-	0,30	0,06	0,75
	5s	-	2,07	0,19	1,72
	10s	-	4,19	0,38	3,52

6.4.4 Avaliação de Conflitos

Os testes a seguir têm o objetivo de certificar que não existem conflitos nas decisões tomadas pelos escalonadores dinâmicos, descentralizados e distribuídos. Isto é importante para evitar a instabilidade do sistema, onde o escalonador dinâmico opte por situações que logo em seguida precisem ser desfeitas. Assim, cada escalonador dinâmico, durante a sua execução, deve perceber a existência de outras aplicações que possam estar compartilhando os mesmos recursos da grade computacional.

A mesma grade virtual e as mesmas instâncias de testes da subseção anterior são usadas, porém considerando sempre que PSwp1 e PSwp2 adotam políticas de HDS idênticas. Entretanto, para mostrar a eficiência do escalonador dinâmico com relação a possíveis conflitos, as aplicações PSwp1 e PSwp2 são executadas com uma alocação de tarefas inicial diferente dos testes anteriores. Nesta situação, quando o escalonador estático executa, ele distribui as tarefas de PSwp1 apenas no site 1 e, as de PSwp2, no site 3. O objetivo desta alocação inicial é fazer com que os escalonadores dinâmicos só percebam a existência do site 2 quando as aplicações forem disparadas, o que deve ser feito ao mesmo tempo.

Desta forma, os escalonadores dinâmicos globais responsáveis pelas aplicações PSwp1 e PSwp2 descobrem o site 2 ao mesmo tempo, e tentam realocar parte de suas tarefas nos recursos inicialmente ociosos. Um novo recurso p_j é descoberto, quando um HM associado a ele envia uma mensagem de *heartbeat* informando ao SM que p_j está disponível. O atraso para se verificar a existência de um novo recurso é pequeno, já que a primeira mensagem de *heartbeat* é enviada assim que o HM de cada p_j inicia sua execução. Logo, os *makespans* obtidos quando as duas aplicações PSwp estão executando isoladamente são próximos dos resultados apresentados na Tabela 6.5.

A Tabela 6.9 mostra os resultados obtidos neste experimento, onde os *makespans* obtidos na execução simultânea não ultrapassam 4,3% dos valores ótimos (*opt*) e 1,6% dos valores esperados (*exp*). Os tempos de execução obtidos por processador, também mostram que a carga de trabalho de cada aplicação foi uniformemente distribuída entre os recursos disponíveis pelos seus respectivos escalonadores dinâmicos.

Tabela 6.9: *Makespans* de PSwp1 e PSwp2 (descobrindo recursos ociosos)

		isolada		simultânea		
	t_{exe}	*mspan*	*opt*	*mspan*	*opt*	*exp*
PSwp1	1	58,39	56,0	83,43	80,0	83,41
	5s	283,48	280,0	410,79	400,0	404,97
	10s	565,04	560,0	816,19	800,0	807,20
PSwp2	1s	58,58	56,0	83,41	80,0	83,68
	5s	283,12	280,0	411,06	400,0	404,45
	10s	564,47	560,0	817,71	800,0	806,38

Apesar dos bons resultados, é importante que seja mostrado como cada escalonador dinâmico reage ao perceber a existência dos novos recursos, e depois como eles percebem que estes recursos, na verdade, estão sendo compartilhados com uma outra aplicação. A Figura 6.6 apresenta a fila de espera de PSwp1 em dois recursos da grade: um dedicado a PSwp1 e o outro, inicialmente ocioso, compartilhado com PSwp2. Para melhor visualização da ação do escalonador dinâmico, a caixa da direita do gráfico mostra uma ampliação dos 15 segundos iniciais da execução. O monitoramento da fila foi feito durante a execução simultânea de PSwp1 e PSwp2 com tarefas de 10 segundos.

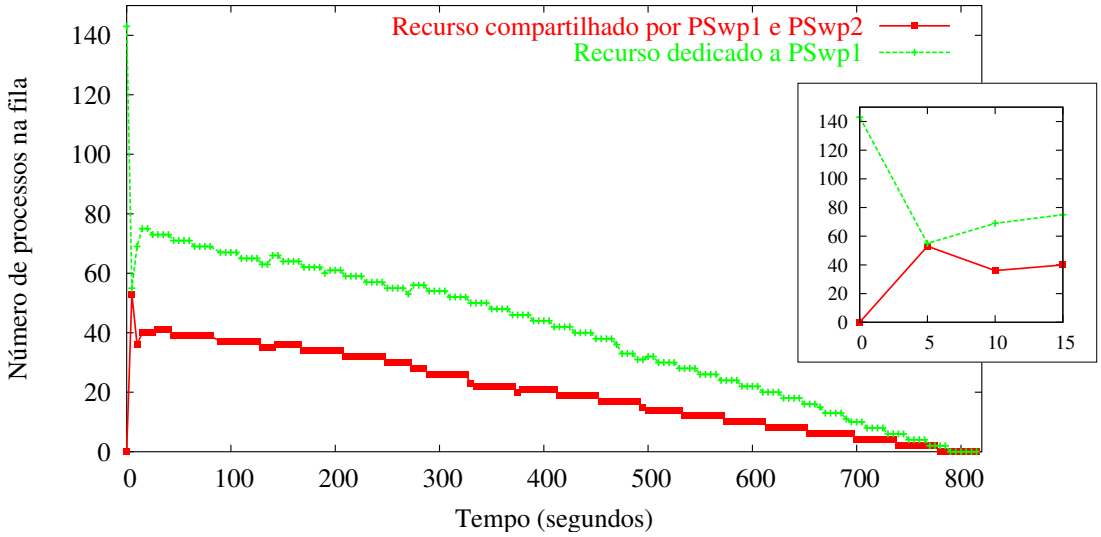

Figura 6.6: Fila de espera da aplicação PSwp1.

No começo da execução das aplicações, cada recurso dedicado (site 1) possui 143 tarefas em sua fila de espera, e assim que recursos ociosos são descobertos através do envio de um *heartbeat*, um evento de escalonamento global é disparado. O GDS de PSwp1, distribui as tarefas excedentes do site 1, uniformemente, entre os recursos disponíveis do site 2. Entretanto, 5 segundos mais tarde, quando uma segunda mensagem de *heartbeat* é enviada indicando o poder computacional disponível para a aplicação, o GDS percebe que um de seus sites (site 2), na realidade, está sendo compartilhado. Em seguida, um segundo evento de escalonamento global é disparado, e as tarefas da aplicação PSwp1 são redistribuídas considerando a presença de PSwp2.

A Figura 6.7 apresenta um outro gráfico que mostra que o escalonador dinâmico não afeta os processos em execução, visto que apenas as tarefas da fila de espera são realocadas. Para a elaboração deste gráfico, foi utilizada a função vmstat 1 do sistema operacional, que indica quantos processos estão em execução em um determinado instante. Assim, como nos outros experimentos, o HDS permite a execução de dois processos concorrentemente, logo, nos recursos dedicados a uma única aplicação, apenas dois processos devem ser detectados,

enquanto que nos recursos compartilhados, deve constar a existência de quatro processos, dois de PSwp1 e dois de PSwp2.

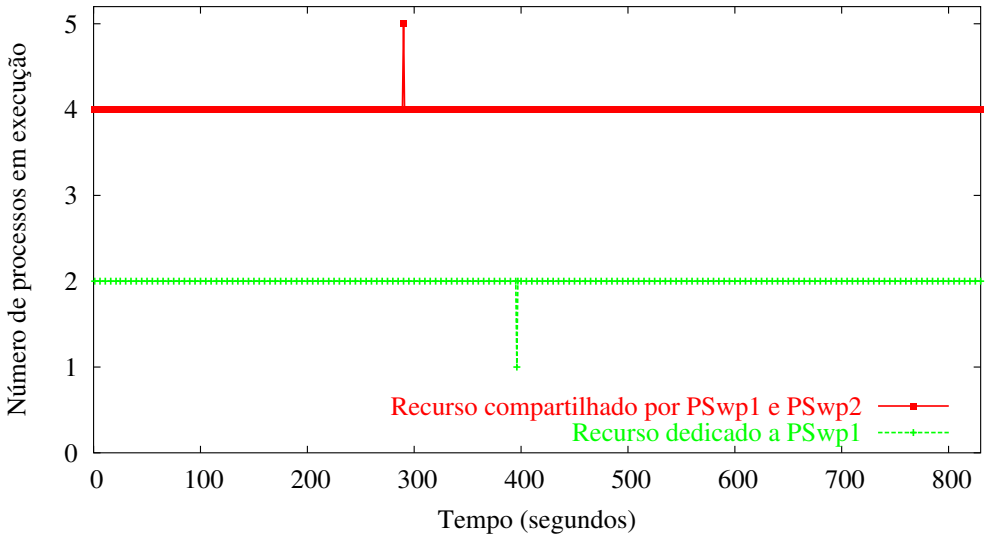

Figura 6.7: Processos de PSwp1 executando concorrentemente.

Como pode ser observado, quase todas as medições realizadas retornaram os valores esperados, exceto em dois casos. Quanto ao recurso compartilhado, vmstat retorna o valor 5, pois provavelmente, foi detectada a presença de um dos HMs, neste instante. Isto também é importante, para mostrar que a sobrecarga introduzida pelos processos gerenciadores é bem baixa, pois eles gastam um tempo de processamento irrisório, sendo difíceis de serem detectados. O outro valor diferente é verificado no recurso dedicado, no exato instante onde um processo da aplicação havia acabado a sua execução e um outro processo ainda estava sendo criado pelo *AMS*, por isso apenas 1 processo foi detectado.

6.4.5 Avaliação da Estrutura do AMS

Em uma primeira versão do *EasyGrid AMS*, aqui chamada de vr_1, os gerenciadores globais (GM) e do site (SM) podiam assumir funções de um gerenciador da máquina (HM). Uma das funções destes gerenciadores seria liberar para execução as tarefas alocadas em sua fila de espera, segundo alguma política de escalonamento local. Entretanto, com o objetivo de tornar o código do *AMS* mais modular e sucinto, optou-se por retirar esta característica em uma segunda versão.

A nova versão, denotada por vr_2, também traz uma facilidade significante na implementação do processo de reescalonamento de tarefas, visto que a política de escalonamento dinâmico só precisará tratar tarefas que são gerenciadas diretamente por HMs. Como apresentado no Capítulo 4, o escalonamento dinâmico é ativado por mensagens pelo GDS ou SDSs, sempre que forem solicitadas tarefas para determinados recursos. Tal abordagem

não poderia ser implementada no caso onde SDSs ou GDS tivessem que pedir tarefas para seus próprios gerenciadores, pois como explicado, eles fazem parte de um mesmo processo. Assim, a implementação das ações dos escalonadores deveria ser adaptada e o grau de modularidade do programa diminuiria.

Por outro lado, nesta nova versão sempre que uma aplicação for disparada, HMs deverão ser criados em todas as máquinas disponíveis, inclusive nas máquinas onde estiverem alocados SMs ou o GM. Nestes casos, tais máquinas possuirão dois processos gerenciadores competindo pela CPU, juntamente com os processos do usuário.

Esta subseção visa avaliar o impacto desta nova abordagem, analisando os *makespans* obtidos por uma aplicação PSwp considerando as duas versões implementadas. Os resultados obtidos foram satisfatórios, já que um percentual muito pequeno de sobrecarga é adicionado ao sistema. Tais experimentos mostram mais uma vez que processos gerenciadores causam pouca intrusão, gastando tempo de processamento irrisório.

Na primeira bateria de testes foi usada uma configuração virtual da grade computacional bem desfavorável, onde apenas 3 recursos estavam disponíveis, caso onde os gerenciadores foram criados como mostrado na Figura 6.8.

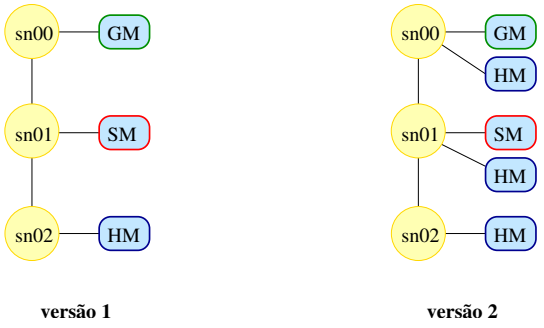

Figura 6.8: Visão geral da criação dos gerenciadores na grade com 3 recursos disponíveis.

Para esta configuração, variou-se o tempo de execução (em segundos) e a quantidade de tarefas de uma aplicação PSwp. Os *makespans* obtidos para (vr_1) e (vr_2) são apresentados na Tabela 6.10. A maior diferença percentual é verificada para PSwp com poucas e pequenas tarefas. Entretanto, para os demais casos a nova abordagem mostra uma piora de menos de 1% nos *makespans* medidos. Na verdade, durante as execuções, verificou-se que o recurso que mais se atrasa é o que possui um SM e um HM. Tal atraso não é devido apenas a execução de dois gerenciadores, mas principalmente, ao maior número de mensagens que o SM tem que redirecionar, já que na versão 2, três HMs estão subordinados a ele.

É importante ressaltar que os resultados apresentados na Tabela 6.10 foram obtidos sem a execução do escalonamento dinâmico. O objetivo era não interferir nos resultados com os possíveis benefícios trazidos pela realocação de tarefas.

Tabela 6.10: *Makespans* de PSwp para vr_1 e vr_2, variando o tempo de execução (t_{exe}) e a quantidade de tarefas

N	0,1s		0,5s		1s		10s	
	vr_1	vr_2	vr_1	vr_2	vr_1	vr_2	vr_1	vr_2
100	3,76	3,83	17,42	17,58	34,52	34,76	342,14	344,09
500	18,32	18,53	85,35	86,25	169,30	170,91	1680,20	1695,80
1000	36,34	36,85	170,50	171,78	338,62	340,72	3360,11	3380,96
10000	367,25	370,49	1705,38	1719,93	3379,89	3410,14	33576,73	33868,67

Para esta análise, uma outra bateria de testes foi realizada, onde as versões vr_1 e vr_2 do *AMS* executam com o escalonamento estático e com o escalonamento híbrido. Nesta segunda bateria de testes, varia-se o número de recursos ($|P|$) disponíveis na grade computacional. Dispara-se um HM em cada máquina adicionada e apenas um GM e um SM continuam sendo criados. As tarefas de cada instância da aplicação PSwp têm duração de 1 segundo e cada recurso executa um número fixo de 100 tarefas. Logo, cada teste realizado corresponde a execução de uma instância de PSwp com $100 \times |P|$ tarefas, onde o *makespan* teórico ótimo é igual a 100 segundos.

Os resultados obtidos são apresentados na Tabela 6.11 e na Figura 6.9. A piora do tempo de execução, quando a quantidade de tarefas aumenta, é um comportamento esperado para todas as versões, visto que apenas um SM passa a lidar com um número bem maior de mensagens relativas a cada tarefa da aplicação.

Tabela 6.11: *Makespans* de PSwp nas versões vr_1 e vr_2 do *AMS*, variando o número de recursos

número de recursos	Estático		Híbrido	
	vr_1	vr_2	vr_1	vr_2
3	101,75	102,77	101,76	102,68
4	101,94	102,98	101,96	102,42
8	103,00	104,04	102,99	102,48
12	104,06	105,10	104,19	102,63
16	104,88	106,11	104,93	102,88
24	107,09	108,56	107,26	102,97
32	110,70	111,68	109,96	105,91

Com exceção das configurações com 3 e 4 recursos, a versão vr_2, com escalonamento dinâmico ativado, apresenta os melhores *makespans*. Isto é possível, pois o escalonamento dinâmico redireciona tarefas do recurso mais sobrecarregado, que é onde se encontra o SM, para os outros recursos disponíveis. Quanto maior o número de recursos mais tarefas podem ser redirecionadas, propiciando um melhor balanceamento.

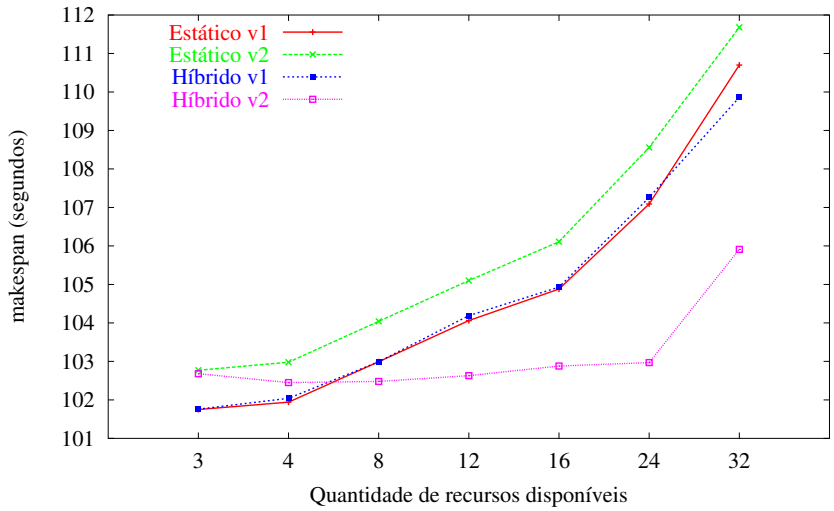

Figura 6.9: Gráfico relativo aos *makespans* da Tabela 6.11.

Os resultados obtidos foram satisfatórios, pois além da sobrecarga introduzida por vr_2 em relação a vr_1 ser muito pequena, ela diminui à medida que um maior número de máquinas é usado. Além disso, a ativação do escalonamento dinâmico consegue minimizar a sobrecarga de trabalho criada nos gerenciadores que precisam lidar com um grande número de mensagens. Logo, devido a simplicidade do código, o que permite gerar executáveis menores e tornar a manutenção do *AMS* mais fácil, vr_2 é a versão utilizada atualmente.

6.4.6 Grade Computacional Compartilhada e Dinâmica

Para verificar a robustez e eficiência do escalonamento híbrido, considerando a estrutura de escalonamento descentralizado e hierárquico proposta, alguns experimentos também foram realizados em uma grade geograficamente distribuída. Para isso, à configuração descrita anteriormente apresentada na Figura 6.1, foi adicionado mais um site, localizado na PUC-Rio. Este site é composto por 24 processadores Pentium II 400 MHz com 256Mb RAM, rodando Linux Red Hat, Globus Toolkit 2.4 e LAM/MPI 7.0.6. A conexão com a grade do SGCLab é feita através da rede GIGA, uma rede ótica experimental da RNP (Rede Nacional de Ensino e Pesquisa), com capacidade de 1 Gigabit.

As aplicações PSwp e Termions foram executadas várias vezes na grade computacional, compartilhada com outros usuários, configurando um ambiente dinâmico e não controlado. Neste tese, a execução foi realizada para aplicações PSwp com até 50.000 tarefas, variando o tempo de execução das tarefas em 1, 5 e 10 segundos, considerando a máquina mais rápida da grade, e para a aplicação Termions com até 20.000 tarefas. Estas execuções mostram a robustez da estrutura de escalonamento proposta, e novos trabalhos vêm sendo realizados para que um número ainda maior de tarefas possa ser executado sem perda de desempenho.

Como o ambiente é compartilhado e dinâmico não foram feitas análises em relação aos *makespans* obtidos. O desempenho do escalonador dinâmico foi verificado através do cálculo do desvio padrão para o tempo de término de cada máquina que executa tarefas de uma aplicação. A Tabela 6.12 mostra a média dos valores obtidos na execução de PSwp com 50.000 tarefas, variando o peso das tarefas a cada execução. Na aplicação dos Termions como não é possível variar o peso da tarefa, variou-se o número de tarefas da aplicação, sendo os resultados apresentados na Tabela 6.13.

Tabela 6.12: Média do desvio padrão na execução de 50.000 tarefas de PSwp na grade computacional

	t_{exe}	site 1	site2	site3	sitePUC	global
PSwp	1s	0,39	0,75	0,38	2,25	2,94
	5s	2,79	3,52	3,05	7,98	7,61
	10s	5,40	6,60	5,90	11,45	15,29

Tabela 6.13: Média do desvio padrão na execução dos Termions na grade computacional

	N	site 1	site2	site3	sitePUC	global
Termions	5000	1,50	1,15	1,33	1,39	5,01
	10000	1,45	1,39	2,62	4,44	5,49
	20000	4,02	3,95	2,76	4,30	5,62

Os valores calculados para o desvio padrão dos sites do SGCLab, a cada execução, são na média sempre menores que o t_{exe} de uma única tarefa, resultados que mostram a eficiência da política de escalonamento local proposta. Como explicado na Subseção 6.4.1, o desvio padrão está ligado com o poder computacional oferecido por um recurso. Como os recursos do site da PUC possuem maior *csi* (*computational slowdown index*), verifica-se que o desvio padrão neste site é maior do que o encontrado nos sites do SGCLab. Considerando as máquinas ociosas do SGCLab, o *csi* medido para as aplicações Pswp e Termions é 1. Já, no site da PUC, o valor de *csi* para a aplicação Pswp é 7, enquanto que para os Termions é igual a 3. Isto significa, por exemplo, que uma tarefa da aplicação Pswp demora 7 vezes mais para executar em uma máquina ociosa na PUC do que no SGCLab.

O desvio padrão encontrado para o escalonamento global, além de ser prejudicado pela grande diferença existente entre o poder computacional dos recursos, é também influenciado pela estratégia de escalonamento do GDS, que na tentativa de ser menos complexa, trabalha somente com valores médios informados pelos sites. Ao trabalhar com médias, o GDS pode não identificar a existência de alguns recursos de um site s_k que estejam mais carregados do que outros. É importante destacar que isto não necessariamente indica que

s_k está desbalanceado. Por exemplo, suponha um site s_k com 10 recursos que ofereçam exatamente o mesmo poder computacional (100%). Dado que um recurso r_{max} possui 20 tarefas, com tempo estimado restante igual a 200 segundos e os demais recursos possuem 19 tarefas cada, com tempo estimado restante igual a 190 segundos, não existe maneira do SDS_k propor um melhor balanceamento em s_k. Através das Equações 5.7 e 5.8, o GDS deve analisar se existe a necessidade de um evento de escalonamento global, calculando o índice de desbalanceamento global (sii).

$$gert^* = \frac{\sum_{s_k \in S} \overline{ert}_k \times tcp_k}{\sum_{s_k \in S} tcp_k} = 191 \qquad sii = \frac{\max_{s_k \in S}\{\overline{ert}_k\}}{gert^*} = \frac{200}{191} = 1,047$$

Como nestes testes, o índice de desbalanceamento global tolerável (ϕ) foi definido em 10%, não será disparado um evento de escalonamento global. Entretanto, talvez a tarefa v excedente em r_{max} pudesse ser cedida para outro site para a redução do tempo de execução final. Porém, ao se trabalhar com médias, a estratégia global não identifica que existe uma tarefa v sobrando em um dos recursos do site s_k. Logo, caso v esteja no site s_k mais lento, o desvio padrão global será maior devido ao baixo poder computacional oferecido pelos recursos de s_k. Para que o GDS identifique a tarefa v e verifique se esta pode pertencer a um outro site, mecanismos extras seriam adicionados a heurística de escalonamento, causando um aumento da sua complexidade. Este exemplo representa o pior caso que o GDS pode encontrar, que é agravado à medida que o tempo de execução de v for maior.

6.5 Resumo

Este capítulo apresenta uma análise dos resultados obtidos, onde o objetivo foi verificar a eficiência da heurística de escalonamento dinâmico implementada dentro da estrutura hierárquica e descentralizada proposta. Através de experimentos foi mostrado que o escalonador dinâmico proporciona uma boa distribuição de tarefas entre as máquinas disponíveis. Isto pode ser verificado, pois o baixo desvio padrão calculado nos testes indica que as máquinas da grade terminam a execução das tarefas de uma aplicação aproximadamente ao mesmo tempo. Além disso, os *makespans* obtidos mostram a baixa intrusão do processo de escalonamento dinâmico.

Para a validação da estrutura proposta, os testes foram realizados em um ambiente controlado, onde os resultados obtidos foram comparados com *makespans* teóricos mínimos. Desta forma, foi possível constatar a eficiência da estrutura proposta. Além disso, experimentos também foram realizados em um ambiente grade não controlado (compartilhado e dinâmico), onde foi possível verificar o bom desempenho da heurística desenvolvida.

Parte II

Aplicações com Relações de Precedências

Capítulo 7

Heurística de Escalonamento *GAD*

A segunda parte deste trabalho de tese trata o conjunto de aplicações que possuem relações de precedência entre suas tarefas. Como definido no Capítulo 3, tais aplicações são representadas por grafos acíclicos direcionados (*GADs*) e o objetivo da heurística de escalonamento proposta é minimizar o *makespan* da aplicação em uma grade computacional, obedecendo estas relações de precedência.

Heurísticas que lidam somente com aplicações BoT tendem a ser mais simples do que aquelas que tratam aplicações GAD, onde informações sobre a topologia do grafo, as características das tarefas e o volume de dados transmitidos entre elas são de suma importância para que se alcance um escalonamento possível e satisfatório. Nestes casos, a ordem de uma tarefa pode influenciar o momento de execução de todas as suas tarefas sucessoras e interferir diretamente no *makespan* final da aplicação. Já, aplicações BoT são constituídas em geral por tarefas independentes entre si, dando maior flexibilidade para a heurística de escalonamento, visto que o grafo de tarefas possui uma topologia simples, onde não existe uma ordem de execução pré-estabelecida.

A seguir, serão apresentadas as heurísticas de escalonamento dinâmico para aplicações GAD propostas neste trabalho e implementadas no *EasyGrid AMS*. Porém, inicialmente será feita uma breve descrição do algoritmo de escalonamento estático usado.

7.1 Escalonamento Estático

Como visto no Capítulo 3, o modelo de comunicação do *EasyGrid AMS* possui características peculiares, logo, é importante o desenvolvimento de heurísticas de escalonamento estáticas e dinâmicas que considerem estas particularidades. Quanto mais fiel for a modelagem da arquitetura e da aplicação, maior será a eficiência do algoritmo de escalonamento. Entretanto, como o objetivo deste tese de doutorado é o estudo de escalonamento dinâmico, não foram desenvolvidas heurísticas específicas para o escalonamento estático de aplicações GAD no *EasyGrid AMS*.

Neste trabalho, foi escolhido o algoritmo de escalonamento estático *HEFT (Heterogeneous Earliest Finish Time)*, apresentado em [108]. Esta heurística foi utilizada por ser considerada uma das melhores estratégias do tipo *list-scheduling* para escalonamento estático de tarefas em ambientes heterogêneos [18]. Além disso, o *HEFT* também não trata a replicação de tarefas, funcionalidade que ainda não é suportada pelo *EasyGrid AMS*. Assim como na grande maioria das heurísticas de escalonamento, o algoritmo *HEFT* adota o modelo de latência, onde o único parâmetro de comunicação considerado é o atraso (cdi) que um dado pode sofrer em um canal de comunicação quando é transmitido entre dois processadores.

Na heurística *HEFT*, uma tarefa v é escalonada no recurso r que minimiza seu tempo final de execução ($ft(v,r)$). A tarefa selecionada para escalonamento é aquela que possui o menor nível ($rank$). O $rank$ de uma tarefa v é o caminho mais longo desde v até um vértice de saída do grafo, considerando a topologia do grafo e as médias dos tempos de execução da tarefa v em todas as máquinas do ambiente oferecido ($\overline{et(v)}$), e também a média da latência dos canais de comunicação ($\overline{cdi}$). O cálculo do $rank$ de uma tarefa v é apresentado na Equação 7.1.

$$rank(v) = \begin{cases} \overline{et(v)}, & succ(v) = \emptyset \\ \overline{et(v)} + \max_{w \in succ(v)}\{\omega(v,w) \times \overline{cdi} + rank(w)\}, & succ(v) \neq \emptyset \end{cases} \quad (7.1)$$

É importante observar que a definição apresentada na Equação 3.6 do Capítulo 3, apesar de ser análoga ao $rank$, se refere ao nível escalonado ($blevel$), que considera além da topologia do grafo, um escalonamento estático previamente realizado. O Algoritmo 12 apresenta, em linhas gerais, a estratégia *HEFT*.

Algoritmo 12 : HEFT

```
1   listaTarefas ← tarefas em ordem decrescente de rank;
2   enquanto (listaTarefas ≠ ∅) faça
3       v ← first(listaTarefas);
4       minFT ← ∞;
5       para i = 1 até |R| faça
6           calcula ft(v, r_i) considerando espaços ociosos em r_i;
7           if (ft(v, r_i) < minFT) então
8               minFT ← ft(v, r_i);
9               r_min ← r_i;
            fim se
        fim para
10      aloca v em r_min;
11      listaTarefas ← listaTarefas - {v};
    fim enquanto
```

Após o cálculo do $rank$ de cada tarefa do grafo, é criada uma lista de tarefas que é ordenada decrescentemente pelo $rank$ calculado (linha 1). A primeira tarefa v da lista é

selecionada (linha 3) e então é buscado o recurso $r_{min} \in R$, que minimiza o tempo final de execução de v (linhas 5 a 9), considerando os espaços ociosos que possam existir entre as tarefas já escalonadas nos processadores. A tarefa v é então escalonada em r_{min} (linha 10) e retirada da lista de tarefas. O algoritmo executa até que todas as tarefas tenham sido escalonadas, respeitando suas relações de precedência, ou seja, uma tarefa só pode começar a sua execução em um tempo posterior ao recebimento de todas as mensagens vindas de suas tarefas predecessoras. Um dos diferenciais do algoritmo *HEFT*, ao escalonar uma tarefa v em um recurso r, é a tentativa de se ocupar espaços ociosos que possam existir entre as tarefas já escalonadas em r.

7.2 Escalonamento Dinâmico

Assim como nas heurísticas de escalonamento dinâmico apresentadas para aplicações BoT, no início da execução de uma aplicação GAD, as tarefas da aplicação do usuário já estão associadas aos processadores disponíveis segundo a política de escalonamento estático. Durante a execução, o escalonador dinâmico recolhe informações atualizadas sobre o estado da aplicação e da grade computacional, podendo ser ativado o reescalonamento de tarefas sempre que for verificado que o *makespan* da aplicação pode ser minimizado. Alguns aspectos em relação ao escalonamento dinâmico para aplicações GAD serão discutidos antes da descrição das respectivas heurísticas propostas e avaliadas nesta tese.

7.2.1 Considerações do Escalonamento Dinâmico

Devido às restrições adicionais causadas pelas relações de precedência existentes entre as tarefas de uma aplicação GAD, as heurísticas de escalonamento podem se tornar mais custosas do que aquelas desenvolvidas para aplicações BoT. Logo, além das considerações já feitas para o problema de escalonamento de aplicações BoT, outros pontos importantes devem ser considerados, como por exemplo, a ordenação e replicação de tarefas de um *GAD*. Como o objetivo é criar, dentro da estrutura do *EasyGrid AMS*, um escalonador dinâmico de baixa intrusão que minimize o tempo final de execução da aplicação, cada um dos tópicos abaixo deve ser analisado e implementações eficientes devem ser propostas.

(a) Frequência de verificação de escalonamento;

(b) Seleção das tarefas a serem reescalonadas;

(c) Seleção das máquinas participantes do reescalonamento;

(d) Ordenação das tarefas selecionadas;

(e) Replicação de tarefas;

(a) Frequência de verificação de escalonamento

Uma variedade de heurísticas de escalonamento dinâmico desenvolvidas para aplicações GAD relacionam a verificação de necessidade de escalonamento diretamente com a topologia do grafo [32, 46, 83]. Nestes casos, eventos de escalonamento são disparados baseados no nível topológico (*level*) das tarefas (Definição 6, Seção 3.3, Capítulo 3), onde, por exemplo, um bloco de tarefas B_l é considerado para reescalonamento quando a primeira tarefa do bloco B_{l-1} terminar ou iniciar a sua execução. Como explicado no Capítulo 3, o menor bloco B_l é composto por tarefas que possuam $level(v) = l$. Desta forma, a quantidade de eventos de escalonamento máxima a ser disparada seria equivalente a profundidade do grafo.

Em ambientes dinâmicos como as grades computacionais, tal estratégia pode não trazer bons resultados quando o número de tarefas de um bloco é muito maior do que o número de máquinas disponíveis. Nestes casos, um grande número de tarefas paralelas pertencentes a um bloco B_l é considerado para reescalonamento apenas uma única vez. No entanto, muitas variações podem ocorrer no ambiente computacional, entre os eventos de escalonamento disparados para o bloco B_l e B_{l+1}, o que representa um problema quando o intervalo entre cada verificação de escalonamento for muito grande.

Para aplicações GAD, assim como definido no Capítulo 4, o escalonador dinâmico deve se preocupar com dois tipos de tarefas distintas: *prontas* e *pendentes*. As tarefas prontas já tiveram suas precedências satisfeitas, estando na lista associada ao seus respectivos HMs prontas para executar. Já, tarefas pendentes ainda esperam por mensagens de suas predecessoras para que tenham suas pendências satisfeitas e, assim, possam migrar para o estado de tarefa pronta.

Nesta tese, a verificação de escalonamento de tarefas *pendentes* está relacionada ao nível topológico de uma tarefa da aplicação. Por outro lado, a necessidade de escalonamento de tarefas *prontas* é analisada da mesma forma definida para aplicações BoT. O objetivo é que as tarefas prontas não permaneçam um longo tempo em um estado de desbalanceamento, prejudicando a execução de suas tarefas sucessoras. Assim, como discutido anteriormente, o intervalo mínimo fixo, I_{min}, é usado entre cada verificação de reescalonamento de tarefas prontas. O valor de I_{min} influencia diretamente no desempenho do algoritmo de escalonamento dinâmico, logo, pesquisas futuras podem ser feitas para que este valor seja ajustado de acordo com as características da aplicação e do ambiente de execução.

(b) Seleção de tarefas

Algoritmos de escalonamento dinâmico projetados para aplicações GAD podem considerar todas as tarefas do grafo ou apenas uma parte delas. Normalmente, na tentativa de diminuir a complexidade do algoritmo de escalonamento, apenas uma parte do grafo é

avaliada a cada verificação de necessidade de reescalonamento [46, 83]. Como explicado no Capítulo 3, um bloco de tarefas B_l é definido e tratado a cada evento de reescalonamento dinâmico.

Não necessariamente todas as tarefas do bloco B_l precisam ser reescalonadas, logo, deve ser definido um critério para a escolha destas tarefas. Um boa estratégia é selecionar as tarefas através do caminho-crítico do grafo (Definição 5, Seção 3.3, Capítulo 3). Neste caso em especial, o caminho-crítico é relativo apenas ao bloco B_l de tarefas que está sendo analisado. Para diminuir o caminho-crítico $cpath(v, r_j)$ tenta-se reescalonar a tarefa v em um outro processador que minimize o *makespan* do bloco B_l, considerando para isso, a alocação de tarefas previamente definida pelo escalonador estático.

Outro critério usado é simplesmente optar por reescalonar todas as tarefas associadas ao bloco B_l. Nestes casos, as tarefas são ordenadas segundo alguma prioridade e reescalonadas, uma a uma, nos processadores que minimizem seus tempos de execução. Este tipo de abordagem não considera o escalonamento definido estaticamente, e um grande número de tarefas podem ser realocadas sem a real necessidade. Isto ocorre, pois o escalonamento dinâmico tende a basear suas decisões em informações locais sem ter um total conhecimento da topologia do grafo como acontece com as políticas estáticas. Exemplos de heurísticas que adotam esta metodologia são: CD, CS, PS e GS, discutidas no Capítulo 3, Seção 3.4.

Nesta pesquisa, o critério para seleção de tarefas usado é diferente quando se considera o reescalonamento de tarefas *prontas* e tarefas *pendentes*. Quando um bloco B_l é constituído por tarefas pendentes, a seleção de tarefas para reescalonamento é feita através da definição do caminho-crítico. As tarefas selecionadas, chamadas de *tarefas críticas*, são aquelas que determinam, a cada passo do reescalonamento, o caminho-crítico de B_l.

As tarefas *prontas* são selecionadas de maneira um pouco distinta, onde em um primeiro passo calcula-se o *tempo de execução alvo* das tarefas *prontas* que pertencem ao bloco B_l. Este *tempo alvo* é calculado pelo SDS e GDS da mesma forma como foi definido na política de escalonamento para aplicações BoT. Em seguida, as tarefas *prontas* de B_l selecionadas para reescalonamento, são aquelas que possuírem tempo de fim de execução maior do que o *tempo de execução alvo* calculado.

(c) Seleção de máquinas

Da mesma forma como explicado na heurística para aplicações BoT, os escalonadores dinâmicos de cada site e o escalonador global, ao dispararem um evento de escalonamento, podem definir dois conjuntos distintos de máquinas: recursos subcarregados e recursos sobrecarregados. Na heurística de escalonamento BoT todas as máquinas subcarregadas e apenas a máquina mais sobrecarregada participam do escalonamento, assim como discutido no Capítulo 5, Subseção 5.2.1.

Para aplicações GAD, após a avaliação de todos os recursos, os escalonadores dinâmicos irão selecionar a cada *evento de escalonamento* todas as máquinas do site (no caso de um reescalonamento local) ou todos os sites da grade (no caso de um reescalonamento global), e não apenas um subconjunto de recursos como implementado na heurística BoT. Ao considerar que as tarefas possuem dependências entre si é importante que um escalonamento mínimo seja realmente encontrado a cada passo. Logo, as heurísticas de escalonamento implementadas para aplicações GAD consideram todas as máquinas na tentativa de encontrar o *makespan* mínimo. Já, as heurísticas para aplicações BoT podem ter um escopo de atuação mais restrito, pois a alocação da aplicação pode ir se ajustando a cada evento de escalonamento, sem que a execução de tarefas futuras sejam prejudicadas, já que não existe dependência entre as tarefas.

(d) Ordenação de tarefas

A ordem em que as tarefas são executadas influencia o *makespan* da aplicação, principalmente no caso de aplicações com relações de precedência. O problema de ordenação de tarefas é NP-difícil [116] e heurísticas são estudadas na tentativa de se encontrar soluções satisfatórias. No problema de escalonamento de tarefas, várias abordagens podem ser usadas para ordenação de tarefas de acordo com prioridades estabelecidas pelo escalonador, como por exemplo, o *blevel* de uma tarefa.

Nesta tese, as tarefas v pertencentes à aplicação GAD são ordenadas em um recurso r_j de acordo com o seu tempo de início $st(v, r_j)$. O tempo de início de uma tarefa v é calculado segundo a Definição 3, Seção 3.3 do Capítulo 3. Apesar do *blevel* ser uma prioridade muito utilizada para seleção de tarefas o que em muitos casos reflete na própria ordenação produzida, a ordenação por *blevel* nem sempre permite que se aproveite os tempos de ociosidade de uma máquina. A ordenação pelo tempo de início de uma tarefa tem o objetivo de escalonar uma tarefa o mais cedo possível, na tentativa de minimizar o *makespan* da aplicação. Logo, nesta implementação o *blevel* de uma tarefa é usado apenas para determinar a ordem das tarefas selecionadas para escalonamento, e não para a ordenação de suas tarefas em um recurso.

(e) Replicação de tarefas

O mecanismo de replicação de tarefas pode ser muito vantajoso quando os custos de comunicação entre as tarefas for grande. Nestes casos, a utilização da replicação pode contribuir significativamente para a redução do *makespan* de uma aplicação GAD em comparação com heurísticas que não usam este mecanismo [71]. Por outro lado, cada vez que é feita a cópia de uma tarefa, mais tarefas passam a competir pelos mesmos recursos, logo o uso da replicação de tarefas indiscriminadamente exige mais memória e maior poder de processamento.

No caso de grades computacionais, os custos de comunicação entre as máquinas de um mesmo site são pequenos quando comparados com comunicação entre máquinas de diferentes sites. Isto acontece, pois geralmente, os canais de comunicação de um *cluster* são de alta capacidade. Logo, uma abordagem interessante e promissora seria fazer cópias de tarefas somente entre sites distintos, sempre que a heurística de escalonamento detectasse esta necessidade. A replicação de tarefas ainda não foi implementada no *EasyGrid AMS*, mas é um dos tópicos que devem ser investigados futuramente.

7.2.2 Escalonamento Dinâmico na Máquina

Na política para aplicações GAD, o escalonador dinâmico da máquina deve executar as tarefas na ordem definida pelo escalonador dinâmico do site, ou se ainda não houve um evento de escalonamento dinâmico, a ordem definida pelo escalonador estático deve ser obedecida. Assim como para aplicações BoT, um HDS_j também poderia alterar a ordem de execução das tarefas alocadas em seu recurso r_j, entretanto como o HDS_j possui uma visão restrita (apenas dos processos pertencentes a lista de prontos e pendentes do seu HM_j), opta-se por seguir a ordenação definida pelo SDS.

A participação de um HDS_j no processo de reescalonamento dinâmico é simples. Como será explicado, na política de escalonamento dinâmico adotada para aplicações GAD, o SDS_k especifica exatamente quais as tarefas ele precisa reescalonar e não apenas uma porcentagem de peso, como acontece na política para aplicações BoT. Logo, ao receber um pedido de tarefas do SDS_k do seu site s_k (Subseção 4.1.1, Algoritmo 1, linha 13), o HDS_j seleciona as tarefas pedidas e as cede para o SDS_k (Algoritmo 13, linha 6). Caso as tarefas pedidas já tenham sido disparadas para execução, o HDS_j envia uma mensagem negando o pedido de tarefas (Algoritmo 13, linha 7).

Note que quando um evento de escalonamento trata tarefas *pendentes*, são selecionadas tarefas apenas da lista de pendentes de um recurso, caso contrário, apenas a lista de tarefas prontas é considerada. As ações realizadas pelo HDS no escalonamento do site, são apresentadas no Algoritmo 13.

Algoritmo 13 : selecionar_tarefas_HDS$(tarefas_{ask})$

```
1  pacote_HDS ← ∅
2  para cada v ∈ tarefas_ask faça
3      remove v da lista do HM;
4      pacote_HDS ← pacote_HDS + {v};
   fim para
5  se (pacote_HDS ≠ ∅) então
6      envia ack e pacote_HDS para o SDS;
   senão
7      envia nack ao SDS;
   fim se
```

7.2.3 Escalonamento Dinâmico no Site

Para aplicações GAD, o escalonador dinâmico do site desempenha dois tipos de políticas diferentes: (1) para tarefas prontas e (2) para tarefas pendentes. O reescalonamento de tarefas prontas pertencentes a um site s_k possui características similares à heurística proposta para aplicações BoT, já no reescalonamento de tarefas pendentes uma nova abordagem é usada. As funções desempenhadas pelo SDS_k na estrutura hierárquica de escalonamento foram detalhadas no Algoritmo 2, Subseção 4.1.2. A seguir serão descritas as políticas desenvolvidas para este tipo de aplicações com relações de precedência.

É importante ressaltar que para o escalonamento dinâmico local, o SDS_k responsável pelo site s_k, manterá uma estrutura de escalonamento auxiliar que indicará o estado atual da alocação de tarefas em todos os recursos do seu site. Ou seja, para cada recurso r_j pertencente a um site s_k, o SDS_k criará uma lista ordenada com as tarefas pertencentes a r_j, onde para cada tarefa será calculado o seu tempo estimado de início e final de execução. Esta estrutura auxiliar será denotada por $aloc_l$.

A estrutura $aloc_l$ é avaliada sempre que for verificada a necessidade de reescalonamento de tarefas. Como já observado anteriormente, apenas uma parte do grafo (B_l, Definição 7, Capítulo 3) é considerada na criação da estrutura de escalonamento $aloc_l$. Tal procedimento é adotado para que o algoritmo de reescalonamento não se torne muito custoso. Os passos para criação de $aloc_l$ podem ser visualizados no Algoritmo 14.

Algoritmo 14 : construir_aloc$_l$()

```
1   nivel ← level(u), onde u foi a última tarefa a terminar em s_k
2   para todo r_j ∈ s_k faça
3       para todo v ∈ r_j faça
4           se (level(v) ≤ nivel + 1) então
5               insere v em aloc_l(r_j) obedecendo sua ordem de execução;
            fim se
        fim para
    fim para
```

Logo, a estrutura de escalonamento, $aloc_l$, conterá tarefas v ainda não executadas e com nível topológico $level(v) \leq l$ (Algoritmo 14, linha 4), onde a última tarefa u, que finalizou a execução em s_k, possui $level(u) = l-1$. Na estrutura $aloc_l$ estarão representadas todas as tarefas de B_l de acordo com o último evento de escalonamento dinâmico realizado, ou de acordo com o escalonamento estático.

Como pode ser verificado, o procedimento para determinar B_l é diferente de outras heurísticas existentes na literatura [46, 83], onde as tarefas são reescalonadas apenas uma vez. Em ambientes dinâmicos como as grades esta restrição na implementação pode resultar em um desempenho ineficiente do algoritmo.

Escalonamento Local de Tarefas Prontas

Assim como na heurística apresentada para aplicações BoT, cada SDS_k é responsável por verificar se um *evento de escalonamento* de tarefas prontas deve ser disparado no seu site s_k, calculando o *tempo de execução alvo* ($sert_k^*$) e o índice de desbalanceamento do sistema (sii_k), como apresentados respectivamente, nas Equações 5.1 e 5.2 do Capítulo 5, que também podem ser visualizadas abaixo. Caso o índice de desbalanceamento do site seja maior que um limite de tolerância θ, é disparado um *evento de escalonamento*.

$$sert_k^* = \frac{\sum_{r_j \in R_k} ert_j \times cp_j}{\sum_{r_j \in R_k} cp_j} \qquad sii_k = \frac{\max_{r_j \in R_k}\{ert_j\}}{sert_k^*}$$

Como visto anteriormente, várias políticas podem ser usadas para a escolha das máquinas que participarão do reescalonamento de tarefas. Como aplicações GAD possuem relações de precedência entre as suas tarefas, é importante que a estratégia usada trate de maneira rápida e eficiente o possível *desbalanceamento* existente entre os recursos, pois o atraso na execução de uma tarefa v influencia diretamente a execução das suas tarefas sucessoras. Assim, na política implementada para esta classe de aplicações, um subconjunto de tarefas prontas dos recursos sobrecarregados será reescalonado nos recursos subcarregados. Um recurso r_j é subcarregado se $ert_j < sert_k^*$, e sobrecarregado se $ert_j > sert_k^*$.

Ao identificar a necessidade de reescalonamento dinâmico em um site s_k, o SDS_k correspondente envia um pedido de tarefas para cada recurso r_{ask} pertencente ao conjunto de recursos sobrecarregados R_{ask}. Sendo T, o tempo em que foi disparado o evento de escalonamento, cada r_{ask} deve ceder todas as suas tarefas prontas v cujo $ft(v, r_{ask}) > sert_k^* + T$. Ao receber as tarefas pedidas, o SDS_k deve reescaloná-las no conjunto de recursos subcarregados R_{sub}, segundo a heurística local. O algoritmo escalonar_prontas_local, disparado na linha 6 do Algoritmo 2 (Subseção 4.1.2) pelo escalonador dinâmico do site é apresentado a seguir:

Algoritmo 15 : escalonar_prontas_local()

```
1  calcula sert*_k e sii_k;
2  se (sii_k > θ) então
3      cria estrutura de escalonamento do site aloc_l;
4      determina R_ask e R_sub;
5      para todo r_ask ∈ R_ask faça
6          para todo v ∈ r_ask faça
7              se (ft(v, r_ask) > sert*_k + T) então
8                  tarefas_ask ← tarefas_ask + {v};
               fim se
           fim para
       fim para
   fim se
9  retorna aloc_l, R_ask, R_sub e tarefas_ask;
```

Quando o SDS_k dispara o evento de escalonamento de tarefas prontas, ele determina os conjuntos de recursos R_{ask} e R_{sub} (Algoritmo 15, linha 4) e também o conjunto de tarefas cedidas $tarefas_{ask}$ (Algoritmo 15, linhas 6 a 8). As tarefas pertencentes ao conjunto $tarefas_{ask}$ são retiradas da estrutura auxiliar $aloc_l$ para serem reescalonadas em um próximo passo. O Algoritmo 15 retorna as seguintes informações atualizadas $aloc_l$, R_{ask}, R_{sub} e $tarefas_{ask}$.

Cada HDS_{ask}, associado ao recurso r_{ask}, ao receber o pedido de tarefas do SDS_k deve buscar as tarefas pedidas na lista de prontos associada ao seu HM. Estas tarefas são selecionadas de acordo com Algoritmo 13, explicado no âmbito de atuação do escalonador da máquina. Caso seja possível ceder tarefas, o HDS_{ask} envia uma confirmação (ack) juntamente com o pacote de tarefas cedidas. Uma tarefa não será cedida apenas se já estiver em execução. Logo, apenas se todas as tarefas pedidas já tiverem sido disparadas para execução, é que será enviado para o SDS_k uma negação ($nack$) do pedido de reescalonamento. Como as tarefas pedidas são as tarefas mais atrasadas de um recurso, ou seja, aquelas que possuam $ft(v, r_{ask}) > sert_k^* + T$, é difícil que um pedido de reescalonamento local seja negado.

Somente após receber as respostas de cada recurso $r_{ask} \in R_{ask}$, o SDS_k pode reescalonar as tarefas entre os recursos R_{sub} do site s_k, como mostrado nas linhas 6 e 7 do Algoritmo 3. Para aplicações GAD, o algoritmo distribuir_tarefas que trata tarefas prontas, é apresentado a seguir, possuindo os seguintes parâmetros: $pacote_{HDS}$, que contém todas as tarefas cedidas no processo de rescalonamento local; R_{sub} que compreende o conjunto de recursos que receberá estas tarefas; e $aloc_l$ que é a estrutura de escalonamento auxiliar determinada na linha 3 do Algoritmo 15.

Algoritmo 16 : distribuir_tarefas*($pacote_{HDS}$, R_{sub}, $aloc_l$)*

```
1   minFT ← ∞;
2   listaTarefas ← pacote_HDS em ordem decrescente de blevel;
3   enquanto (listaTarefas ≠ ∅) faça
4       para todo r_j ∈ R_sub faça
5           calcular st(v, r_j) e ft(v, r_j) considerando aloc_l;
6           se (ft(v, r_j) < minFT) então
7               (r_min, minFT) ← (r_j, ft(v, r_j));
            fim se
        fim para
8       aloca v em r_min considerando st(v, r_min);
9       insere v em pacote_{r_j};
10      remove v de listaTarefas;
    fim enquanto
11  para todo r_j ∈ R_sub faça
12      envia pacote_{r_j} para r_j;
    fim para
```

Antes de iniciar o reescalonamento, o SDS_k deve ordenar as tarefas cedidas decrescentemente pelo seu *blevel* (Algoritmo 16, linha 2). A partir daí, a lista de tarefas é percorrida, onde para cada tarefa v deve ser encontrado o recurso $r_{min} \in R_{sub}$ que minimize o tempo final de execução de v (Algoritmo 16, linhas 4 a 7). Finalmente, v é escalonada em r_{min}, no tempo inicial $st(v, r_{min})$ calculado pelo SDS_k (Algoritmo 16, linha 8).

Escalonamento Local de Tarefas Pendentes

O reescalonamento de tarefas pendentes está diretamente ligado com o nível topológico das tarefas do grafo, ocorrendo com menor frequência que o rescalonamento de tarefas prontas. No algoritmo escalonar_pendentes_local, o SDS_k também tem o objetivo de identificar os recursos aos quais serão pedidas tarefas (R_{ask}), os recursos que receberão tarefas (R_{sub}), o conjunto de tarefas pedidas ($tarefas_{ask}$) e a alocação destas tarefas dentro do site ($aloc_l$). As ações do escalonador dinâmico do site no processo de reescalonamento de tarefas pendentes são apresentadas no Algoritmo 17.

No reescalonamento de tarefas pendentes, o *makespan* de s_k é calculado considerando todas as tarefas v pertencentes a estrutura $aloc_l$, para isso é utilizado o escalonamento atual das tarefas v no site s_k. Logo, este valor, denotado por $mspan_k^{aloc}$, representa uma estimativa do *makespan* do site considerando todas as tarefas v, tal que $level(v) \leq l$.

O objetivo do algoritmo é determinar a cada passo, o recurso r_{max} que determina o *makespan* do site s_k, e assim tentar minimizar o seu tempo de execução de acordo com duas estratégias distintas: (a) buscar tarefas predecessoras críticas que pertençam a outros recursos (busca_tarefas()) ou (b) ceder tarefas para recursos menos carregados (cede_tarefas()). Os dois procedimentos serão explicados a seguir, através do exemplo mostrado nas Figuras 7.1 e 7.2, onde uma aplicação paralela é representada por uma árvore binária invertida com 15 tarefas de peso uniforme.

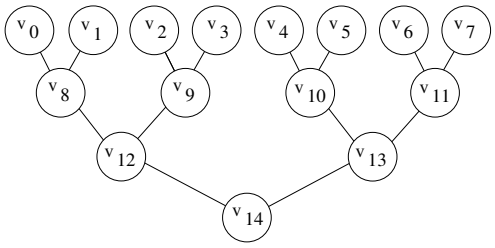

Figura 7.1: Aplicação representada por uma árvore binária invertida com 15 tarefas.

Supondo que a estrutura de escalonamento $aloc_l$ é formada pelas tarefas v_8, v_{12} alocadas em r_0 e v_9 alocada em r_1, como apresentado na Figura 7.2, o tempo de execução do site nos dois casos apresentados, será definido pelo recurso r_0, identificado como r_{max}. A explicação das estratégias busca_tarefas() e cede_tarefas() é realizada a seguir.

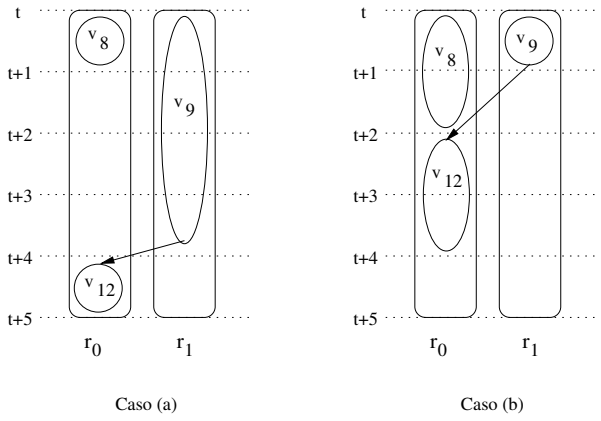

Figura 7.2: Exemplo de reescalonamento de tarefas pendentes: (a) r_0 deve receber tarefas de r_1, de acordo com a estratégia busca_tarefas(). (b) r_0 deve ceder tarefas para r_1, segundo cede_tarefas().

Caso (a)

No primeiro escalonamento, apresentado na Figura 7.2(a), suponha que o recurso r_0 oferece 100% do seu poder computacional, enquanto r_1 passa a oferecer apenas 25%. Tal mudança no ambiente acarreta em um atraso no início da execução de v_{12} que apesar de se encontrar em r_0, depende diretamente de v_9. A melhor opção é que o escalonador dinâmico do site decida reescalonar v_9 em r_0 imediatamente após a execução de v_8. Neste caso, o *makespan* de r_0 será reduzido de acordo com o algoritmo busca_tarefas().

A estratégia de reescalonamento busca_tarefas() chamada na linha 6 do Algoritmo 17 encontra uma *tarefa crítica* que pertença ao conjunto de predecessoras imediatas das tarefas $v \in r_{max}$. Seja $V_{max} = \{v_1, v_2, ..., v_n\}$ a lista de tarefas alocadas em r_{max} e u_i todas as tarefas *pendentes* predecessoras imediatas das tarefas de V_{max}, a tarefa crítica u_{crit} é aquela que estabelece o maior valor:

$$st(v_i, r_{max}) + \sum_{h=i}^{n} et(v_h, r_{max}) \tag{7.2}$$

onde $st(v_i, r_{max})$ é o tempo de início de v_i no recurso r_{max} e $et(v_h, r_{max})$ é o tempo de execução de uma tarefa v_h em r_{max}, como definido na Seção 3.3 do Capítulo 3.

O procedimento busca_tarefas() procura por toda tarefa u_{crit} que possa ter seu tempo final de execução minimizado e por consequência possa minimizar o tempo final do recurso r_{max}. Este procedimento executa enquanto for possível encontrar tarefas que satisfaçam esta condição. É possível que a tarefa u_{crit} pertença a um recurso que não faça parte do site s_k. Neste caso, o GDS é que intermediará o pedido de tarefas que ocorrerá entre recursos de sites diferentes.

Caso (b)

No exemplo apresentado Figura 7.2(b), o SDS_k detecta que o recurso r_0 oferece apenas 50% de poder computacional para a aplicação do usuário, dobrando o tempo de

execução de suas tarefas. Neste caso, como r_1 oferece 100% de poder computacional, a melhor opção é que o escalonador dinâmico do site reescalone as tarefas de r_0 em r_1 segundo a estratégia cede_tarefas(). Na estratégia cede_tarefas(), chamada na linha 7 do Algoritmo 17, são selecionadas as tarefas de r_{max} que serão cedidas para recursos mais ociosos. Esta seleção é feita através da análise de todas as tarefas pendentes $v_i \in V_{max}$, onde é verificado se o tempo final de execução de v_i pode ser minimizado em outro recurso do site s_k.

Cada uma das estratégias descritas acima cria uma lista que possuirá todas as tarefas que devem ser reescalonadas, definindo para cada tarefa v, o seu tempo inicial de execução (st), o seu recurso origem e destino. Assim, é possível determinar os conjuntos de recursos que devem ceder e receber tarefas. Outro ponto importante é que uma tarefa só será reescalonada para outro recurso se nenhuma outra tarefa da estrutura de escalonamento $aloc_l$ for atrasada. O algoritmo escalonar_pendentes_local é apresentado a seguir.

Algoritmo 17 : escalonar_pendentes_local()

```
1   cria estrutura de escalonamento do site aloc_l;
2   ListaCedidas ← ∅;
3   calcula mspan_k^aloc e determina r_max;
4   mspan_anterior ← mspan_k^aloc;
5   enquanto (existir novo r_max) faça
6       ListaCedidas ← ListaCedidas + busca_tarefas(r_max);
7       ListaCedidas ← ListaCedidas + cede_tarefas(r_max);
8       determina r_max;
    fim enquanto
9   calcula mspan_k^aloc;
10  se (mspan_k^aloc < mspan_anterior + α) então
11      tarefas_ask ← ListaCedidas;
    senão
12      tarefas_ask ← ∅;
    fim se
13  retorna aloc_l, R_ask, R_sub e tarefas_ask;
```

Todos os testes realizados para verificar se uma tarefa v deve ser reescalonada em um outro recurso, são realizados utilizando a estrutura de escalonamento $aloc_l$. Após todas as modificações realizadas, o novo *makespan* do site é calculado considerando $aloc_l$. Assim, o SDS_k só envia um pedido de tarefas se o *makespan* tiver sido reduzido (Algoritmo 17, linha 10). Um fator de tolerância, α, é usado para determinar aproximadamente o custo para o reescalonamento de tarefas ou simplesmente para ajustar os cálculos realizados durante o reescalonamento dinâmico. A utilização de α permite que não sejam realizados reescalonamentos que não impliquem em uma real melhora do *makespan* do site, já que as estimativas dos tempos de início e fim das tarefas podem não considerar com precisão

os tempos de fim de tarefas predecessoras que ainda estejam em execução. Esta falta de precisão pode ocorrer devido às variações do ambiente computacional.

Assim como no escalonamento de tarefas prontas, o SDS$_k$ envia o pedido de tarefas ($tarefas_{ask}$) para o conjunto de recursos, R_{ask}, que é determinado pelos procedimentos busca_tarefas() e cede_tarefas() chamados no Algoritmo 17. Somente após receber uma resposta de cada $r_j \in R_{ask}$ é que o SDS$_k$ irá distribuir as tarefas cedidas.

No caso de tarefas pendentes, a distribuição é feita de maneira simples, já que durante o processo de reescalonamento, a lista de tarefas cedidas é criada contendo informações úteis como o tempo inicial de execução e o recurso destino de cada tarefa. Assim, para cada recurso destino, o SDS$_k$ cria um pacote que conterá todas tarefas que devem se direcionados para este recurso. Após a etapa de empacotamento de tarefas, cada pacote é enviado pelo SDS$_k$ para cada HDS$_j$ correspondente. Ao receber o pacote cedido, o HDS$_j$ insere cada uma das tarefas na sua lista de processos pendentes, de acordo com o tempo de inicial de execução definido no escalonamento local.

7.2.4 Escalonamento Dinâmico Global

Como explicado no Capítulo 4, o escalonador dinâmico global proposto para aplicações GAD pode agir como ativador do evento de escalonamento ou mediador. Nas ações do GDS especificadas no Algoritmo 5, um escalonamento global é ativado pelo GDS somente para reescalonar tarefas prontas, enquanto que sempre que o GDS receber um pedido de tarefas de algum SDS$_k$, ele deve trabalhar como mediador, encaminhando este pedido para os sites correspondentes.

Escalonamento Global de Tarefas Prontas

Para o tratamento de tarefas prontas, o GDS é responsável por verificar se existe a necessidade de disparar um *evento de escalonamento*. Da mesma forma como no algoritmo para aplicações BoT, este evento será disparado quando o índice de desbalanceamento do sistema for maior que um limite ϕ, pré-definido. O cálculo do *tempo de execução alvo* ($gert^*$), assim como do índice de desbalanceamento do sistema (sii), definidos nas Equações 5.7 e 5.8, podem ser visualizados abaixo:

$$gert^* = \frac{\sum_{s_k \in S} \overline{ert}_k \times tcp_k}{\sum_{s_k \in S} tcp_k} \qquad sii = \frac{\max_{s_k \in S}\{\overline{ert}_k\}}{gert^*}$$

O escalonamento global de tarefas prontas segue a mesma idéia da política implementada no escalonamento do site. Na tentativa de não atrasar a execução de tarefas sucessoras, todos os sites da grade podem participar de um mesmo evento de escalonamento global. Logo, para aplicações GAD ao se verificar a necessidade de um reescalonamento dinâmico global, o GDS envia um pedido de tarefas para cada site do conjunto S_{ask}, que

representa os sites s_k sobrecarregados, ou seja que possuam $\overline{ert}_k > gert^*$. O Algoritmo 18 especifica os passos seguidos pelo GDS no escalonamento global de tarefas prontas.

Algoritmo 18 : escalonar_prontas_global()

```
1   aloc ← ∅;
2   calcula gert* e sii;
3   se (sii_k > φ) então
4       determina S_ask e S_sub;
5       tarefas_ask ← gert* + T;
    fim se
6   retorna aloc, S_ask, S_sub e tarefas_ask;
```

Diferentemente do SDS, o GDS não cria a estrutura de escalonamento *aloc*. Esta idéia foi adotada com o objetivo de diminuir a complexidade da heurística global e tornar o sistema mais escalável, onde o GDS não possui conhecimento da alocação de tarefas dentro dos recursos de cada site. Devido a esta característica, no escalonamento global, ao invés de se pedir tarefas específicas para o conjunto de sites S_{ask}, é enviado o *tempo de execução alvo* ($gert^*$) somado ao tempo T, no qual foi disparado o evento de escalonamento atual (Algoritmo 18, linha 5). Desta forma, cada SDS_{ask} ao receber o pedido enviado pelo GDS (Algoritmo 4, linha 3) deve selecionar as tarefas de $s_{ask} \in S_{ask}$ que possuam tempo de fim de execução maior que $gert^* + T$, como mostrado no Algoritmo 19.

Algoritmo 19 : selecionar_tarefas_SDS$(gert^* + T)$

```
1   pacote_SDS ← ∅;
2   R_ask ← {r_j ∈ R/ert_j > gert*};
3   para todo r_ask ∈ R_ask faça
4       tarefas_ask ← ∅;
5       para todo v ∈ r_ask faça
6           se (ft(v, r_ask) > (gert* + T) então
7               tarefas_ask ← tarefas_ask + {v};
            fim se
        fim para
8       repassa pedido de tarefas_ask para HDS_ask;
    fim para
    /* recebe tarefas de HDS_ask */
9   para cada (ack recebido de HDS) faça
10      pacote_SDS ← pacote_SDS + pacote_HDS;
    fim para
11  se (pacote_SDS ≠ ∅) então
12      envia ack e o pacote_SDS para o GDS;
    senão
13      envia nack ao GDS;
    fim se
```

No escopo do escalonamento global, cada SDS$_{ask}$ define o conjunto R_{ask} que contém os recursos do seu site que receberão um pedido de tarefas a serem cedidas para outro site (Algoritmo 19, linha 2). Para cada recurso $r_{ask} \in R_{ask}$ ($ert(r_{ask}) > gert^*$), o SDS$_{ask}$ identifica as tarefas que devem ser cedidas e envia o pedido correspondente para r_{ask} (Algoritmo 19, linhas 3 a 8).

Ao receber o pedido de tarefas do SDS$_{ask}$, o escalonador dinâmico da máquina associado ao recurso r_{ask}, seleciona as tarefas conforme o Algoritmo 13 e as envia para o SDS$_{ask}$. Cada conjunto de tarefas recebido pelo SDS$_{ask}$ é armazenado em um pacote, denotado por $pacote_{SDS}$ (Algoritmo 19, linha 10). Somente após receber todas as respostas dos HDSs correspondentes, o $pacote_{SDS}$ é enviado para o GDS (Algoritmo 19, linha 12) que deve redistribuir as tarefas prontas entre os sites subcarregados $s_k \in S_{sub}$, onde $\overline{ert}_k < gert^*$. Caso não sejam cedidas tarefas, um negação é enviada ao GDS que encerra o evento de escalonamento global.

O GDS só pode distribuir as tarefas cedidas após ter recebido respostas de todos sites $s_k \in S_{ask}$, como mostrado no Algoritmo 6. Da mesma maneira que na implementação do escalonamento global de tarefas prontas para aplicações BoT, o GDS define a carga de trabalho que cada site subcarregado deve receber de acordo com os valores de $\overline{ert}_k$ e tcp_k. Logo, seja S_{sub} o conjunto de sites subcarregados, cada $s_k \in S_{sub}$ deverá receber uma carga de trabalho ($wload_{s_k}$) calculada de acordo com as Equações 5.9, 5.10 e 5.11, discutidas na Seção 5.2.4 do Capítulo 5. O algoritmo de distribuição de tarefas realizado pelo GDS é apresentado a seguir.

Algoritmo 20 : distribuir_tarefas*($pacote_{SDS}$, S_{sub}, $wload$)*

```
1   listaTarefas ← pacote_SDS em ordem decrescente de blevel;
2   enquanto (listaTarefas ≠ ∅) faça
3       v ← first(listaTarefas);
4       maxDado ← 0;
5       para todo s_k ∈ S_sub faça
6           dados(v, s_k) ← quantidade de dados trocados por v em s_k;
7           se (dados(v, s_k) > maxDado) E (wload_{s_k} > 0) então
8               (s_min, maxDado) ← (s_k, dados(v, s_k));
            fim se
        fim para
9       adiciona v ao pacote de tarefas pacote_{s_min};
10      wload_{s_min} ← wload_{s_min} − ε(v);
11      listaTarefas ← listaTarefas −{v};
    fim enquanto
12  para todo s_k ∈ S_sub faça
13      envia pacote_{s_k} para s_k;
    fim para
```

Antes de iniciar a redistribuição de tarefas, o pacote de tarefas recebido pelo GDS é ordenado decrescentemente pelo *blevel* das tarefas (Algoritmo 20, linha 1). A lista de tarefas é percorrida, onde para cada tarefa v é designado um site s_{min} onde v deve ser escalonada. Na escolha de s_{min}, respeita-se a carga $wload_{s_k}$ que cada site s_k tem para oferecer e ainda é dada a preferência para alocar v no site onde v fará o maior número de troca de mensagens, totalizados em $dado(v, s_k)$ (Algoritmo 20, linhas 5 a 9). O objetivo desta estratégia é minimizar a troca de mensagens entre diferentes sites que compõem a grade, já que a comunicação entre os recursos de um mesmo site é menos custosa.

No último passo do escalonamento global, cada SDS_{sub} que receber tarefas cedidas deve distribuí-las entre os recursos do seu site. A escolha dos recursos é feita conforme apresentado no Algoritmo 16.

Escalonamento Global de Tarefas Pendentes

Na versão atual do algoritmo de escalonamento global proposto nesta tese, ainda não foram implementadas as funções de mediador de eventos relacionada ao GDS, sempre que um SDS realiza um pedido de tarefas *pendentes*. Até agora, o GDS apenas repassa os pedidos de tarefas de um site ao outro. O SDS que recebe o pedido é que deve verificar se a tarefa pode ser cedida sem que o seu *makespan* atual aumente.

Para aplicações GAD, a idéia é que o GDS trabalhe verificando o recebimento das tarefas pedidas e possa decidir, caso uma tarefa v não seja cedida, se é proveitoso realizar a replicação de v no site que a requisitou. Tal estratégia ainda não foi criada visto que o mecanismo que gerencia tarefas duplicadas ainda não foi implementado na última versão do *EasyGrid AMS*. Entretanto, como a replicação pode trazer inúmeros benefícios, quando os custos de comunicação entre os sites forem altos [71], este mecanismo deve ser analisado nas futuras versões do algoritmo de escalonamento dinâmico.

7.2.5 Exemplo de Escalonamento GAD

Como explicado anteriormente, o escalonador dinâmico global proposto não possui uma visão detalhada do escalonamento das tarefas dentro de cada site da grade computacional. Nesta subseção, será apresentado um exemplo de escalonamento local, onde o SDS_k constrói uma estrutura auxiliar chamada $aloc_l$, e a partir dela toma as decisões de escalonamento de tarefas no site s_k. O exemplo considera uma aplicação, representada por uma árvore *in-tree* de 63 tarefas, apresentada na Figura 7.3. O site s_k possui 8 recursos e o tempo de execução de cada tarefa é de 1 segundo, no recurso mais rápido de s_k.

O SDS_k verifica se existe a necessidade de realizar escalonamento de tarefas *prontas* a cada 1 segundo, e o limite de tolerância de desbalanceamento, θ, é definido em 10%. O escalonamento de tarefas *pendentes* é tratado sempre que a primeira tarefa de um nível topológico l da árvore terminar a sua execução. Logo, como a árvore utilizada possui suas

tarefas divididas em 6 níveis topológicos (*level*), no máximo 6 eventos de escalonamento de tarefas *pendentes* serão disparados.

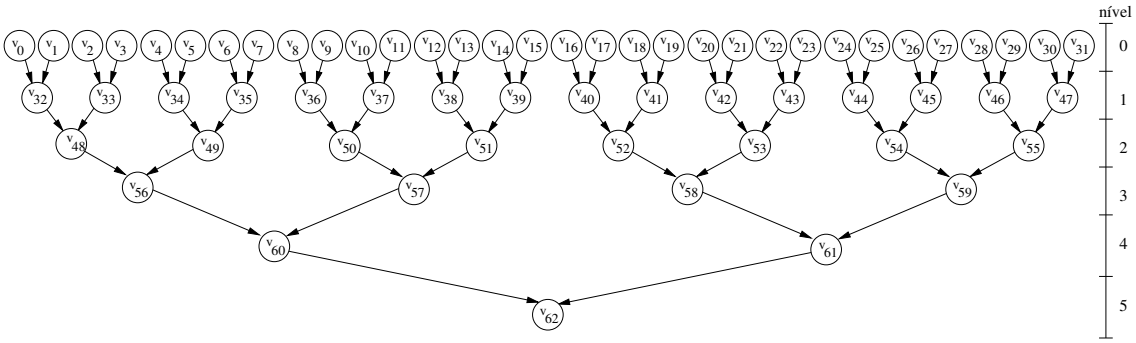

Figura 7.3: Exemplo de uma aplicação representada por uma árvore do tipo *in-tree* com 63 tarefas.

O escalonamento estático inicial é feito pelo algoritmo *HEFT*, considerando todos os recursos de s_k dedicados a execução da aplicação. Como o tempo de troca de mensagens entre os processos do site é inferior a 9 milissegundos, o que representa um valor desprezível em relação ao tempo de computação das tarefas, o custo de comunicação será negligenciado para a simplificação do exemplo. Porém, as relações de precedência entre as tarefas continuam sendo respeitadas. O escalonamento estático é apresentado na Figura 7.4, onde o *makespan* obtido é igual a 10 segundos.

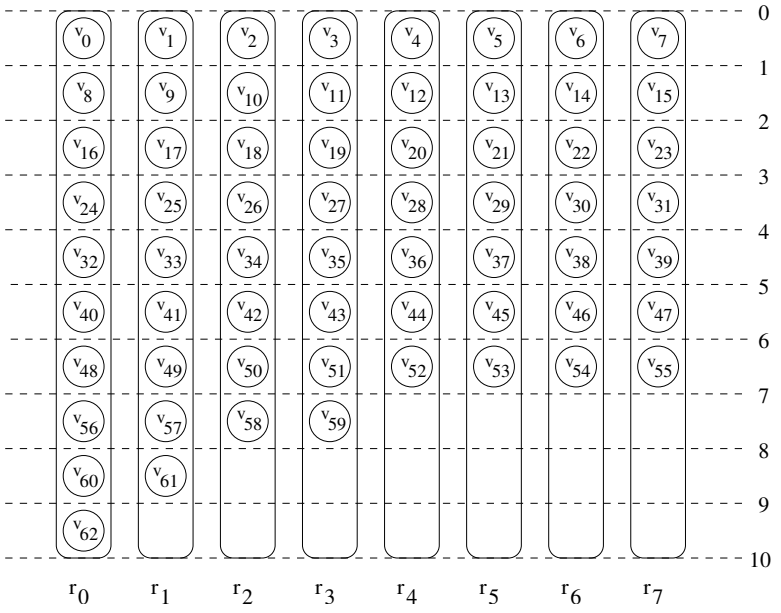

Figura 7.4: Alocação inicial realizada pelo política de escalonamento estático *HEFT*.

Através da figura acima, também é possível identificar o nível escalonado (*blevel*, Definição 4, Seção 3.3, Capítulo 3) de cada tarefa. O $blevel(v)$ de uma tarefa v é o seu tempo de execução somado ao tamanho do caminho desde v até um nó de saída do grafo.

Por exemplo, as tarefas v_0, v_1, v_2, v_3, v_4, v_5, v_6 e v_7 possuem $blevel = 10$, enquanto v_{60} e v_{61} possuem $blevel = 2$. As tarefas prontas que são cedidas para escalonamento, são ordenadas decrescentemente pelo *blevel*, assim o SDS estabelece prioridades entre as tarefas antes de reescaloná-las nos novos recursos (Algoritmo 16). Durante o reescalonamento dinâmico, o bloco B_l de tarefas tratadas em um evento de escalonamento compreenderá todas as tarefas v com $level(v) \leq l$. A estrutura de escalonamento auxiliar $aloc_l$, construída pelo SDS (Algoritmo 14), conterá as tarefas pertencentes a B_l.

Neste exemplo, assim que a aplicação do usuário começa a sua execução, logo após o tempo de execução $T = 0s$, a camada de monitoramento do *EasyGrid AMS* detecta que os recursos r_0, r_1, r_2 e r_3 oferecem apenas 50% do seu poder computacional. Desta forma, todas as tarefas alocadas nestes recursos possuirão tempo de execução igual a 2 segundos, atrasando a execução de suas tarefas sucessoras (Figura 7.5). Assim, logo após serem disparadas as primeiras tarefas de cada recurso r_j, o SDS verifica se existe a necessidade de disparar um evento de escalonamento de tarefas prontas.

Seguindo o Algoritmo 15, o primeiro passo é calcular o *tempo de execução alvo* para as tarefas prontas, $sert_k^*$, e o índice de desbalanceamento sii_k. O tempo estimado ert_j indica o tempo restante para execução de tarefas prontas em cada recurso r_j. Sabendo que $ert_0 = 6$, $ert_1 = 6$, $ert_2 = 6$, $ert_3 = 6$, $ert_4 = 3$, $ert_5 = 3$, $ert_6 = 3$ e $ert_7 = 3$, como é possível visualizar na Figura 7.5, os valores para $sert_k^*$ e sii_k são os seguintes:

$$sert_k^* = \frac{\sum_{r_j \in R_k} ert_j \times cp_j}{\sum_{r_j \in R_k} cp_j} = \frac{(6 \times 0,5) \times 4 + (3 \times 1) \times 4}{0,5 + 0,5 + 0,5 + 0,5 + 1 + 1 + 1 + 1} = 4$$

$$sii = \frac{\max_{r_j \in R_k}\{ert_j\}}{sert_k^*} = \frac{6}{4} = 1,5$$

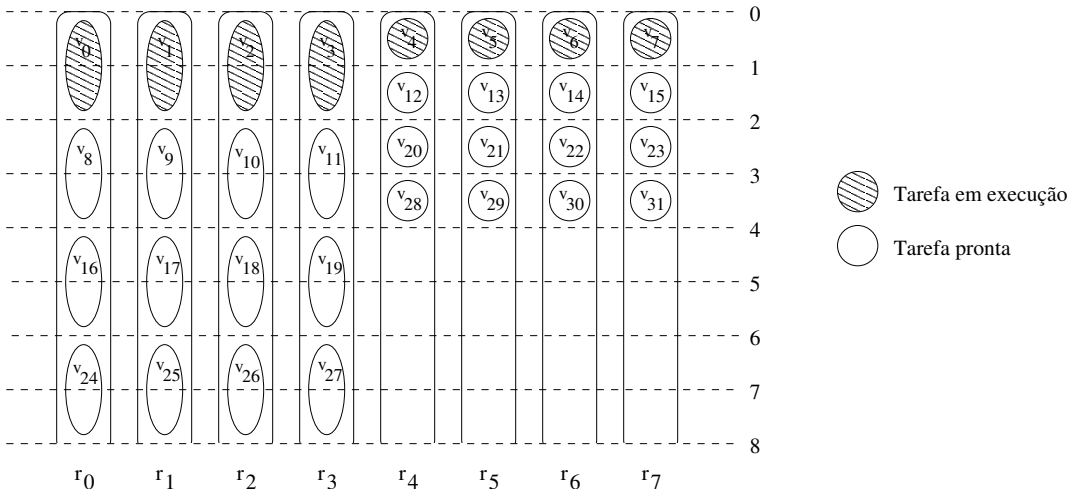

Figura 7.5: Estrutura de escalonamento $aloc_0$ para $level \leq 0$.

Como o índice de desbalanceamento tolerável estabelecido para esta aplicação é 10% e o sii_k calculado equivalente a 50%, um evento de escalonamento de prontas é disparado. Seguindo os passos do Algoritmo 15, o SDS deve criar a estrutura de escalonamento $aloc_0$, determinar o conjunto de recursos que cederá tarefas (R_{ask}), o conjunto de recursos que receberá tarefas (R_{sub}) e o conjunto de tarefas que serão reescalonadas ($tarefas_{ask}$). A estrutura auxiliar de escalonamento $aloc_0$, apresentada na Figura 7.5, conterá todas as tarefas v que possuam $level(v) \leq 0$.

O conjunto R_{ask} será formado por todos recursos r_j que possuam $ert_j > sert_k^*$, e o conjunto R_{sub} pelos recursos r_j com $ert_j < sert_k^*$. Assim, $R_{ask} = \{r_0, r_1, r_2, r_3\}$ e $R_{sub} = \{r_4, r_5, r_6, r_7\}$. De acordo com o Algoritmo 15, linhas 5 a 8, o SDS irá selecionar as tarefas v_{16}, v_{17}, v_{18}, v_{19}, v_{24}, v_{25}, v_{26} e v_{27} que possuem tempo de fim maior que $sert_k^*$, para serem reescalonadas. O SDS envia um pedido de tarefas para cada HDS_j alocado em $r_j \in R_{ask}$, especificando quais as tarefas devem ser cedidas.

Ao receber o pacote com todas as tarefas cedidas ($pacote_{HDS}$) no reescalonamento, o SDS deve distribuí-las entre os recursos $r_j \in R_{sub}$, como especificado no Algoritmo 16. Assim, uma tarefa v será escalonada no recurso r_j que minimize o seu tempo de execução final. O SDS realiza os testes na estrutura $aloc_0$, e determina um novo escalonamento para o site s_k, como pode ser visto na Figura 7.6. No último passo do escalonamento local, o SDS envia cada tarefa $v \in pacote_{HDS}$ para o recurso r_j que minimizou $ft(v, r_j)$.

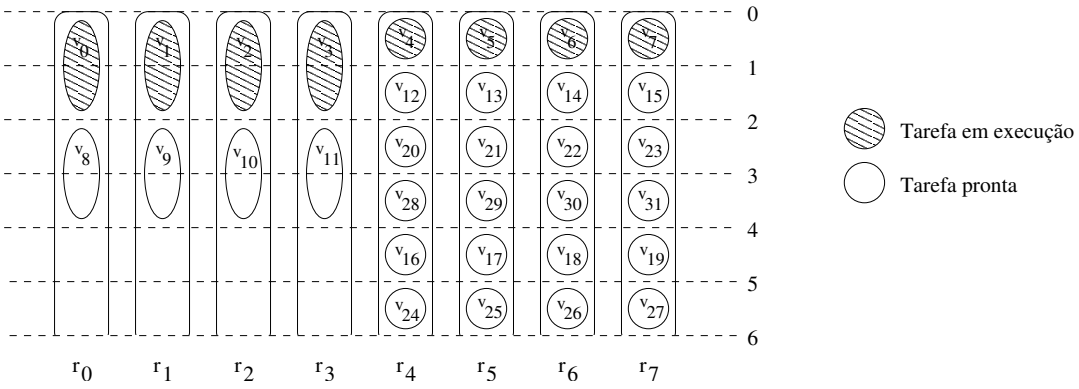

Figura 7.6: Estrutura de escalonamento $aloc_0$ após escalonamento de tarefas prontas.

Após o tempo de execução $T = 1s$, as tarefas v_4, v_5, v_6 e v_7 terminam a sua execução. Como estas são as primeiras tarefas do nível 0 ($level = 0$) a finalizar, o escalonamento de tarefas pendentes é ativado. A estrutura $aloc_1$ será criada com todas as tarefas v ainda não concluídas, que possuam $level(v) \leq 1$, como mostrado na Figura 7.7.

Após criar $aloc_1$, o SDS calcula o seu *makespan* e determina o recurso r_{max} que o define (Algoritmo 17, linha 3). Observando a Figura 7.7, inicialmente $r_{max} = r_0$, e como não existem espaços ociosos em r_0, o procedimento cede_tarefas() é executado (Algoritmo 17, linha 7). Como explicado anteriormente, cede_tarefas() verifica quais tarefas pendentes,

pertencentes a r_{max}, podem ser cedidas para outros recursos do site, de forma que seus tempos de fim sejam minimizados. Assim, o SDS tenta minimizar o tempo de fim de v_{32} e a seguir de v_{40}. Como não existem espaços ociosos em nenhum dos recursos de $aloc_1$, o tempo de r_0 não diminui e a situação permanece a mesma, sem reescalonamentos.

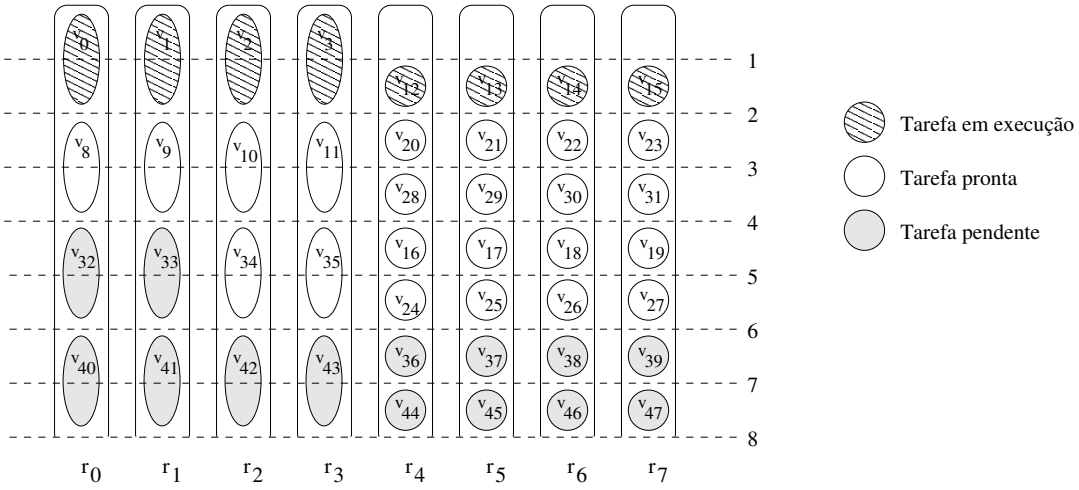

Figura 7.7: Estrutura de escalonamento $aloc_1$ para $level \leq 1$.

Durante a execução da aplicação, o SDS verifica a cada 1 segundo, se existe necessidade de reescalonamento de tarefas prontas. Através da Figura 7.8 que mostra a execução a partir de $T = 2s$, observa-se que não existem espaços ociosos entre as tarefas prontas até o tempo de execução $T = 6s$. Logo, neste intervalo de tempo, o SDS não realiza nenhum reescalonamento de tarefas prontas.

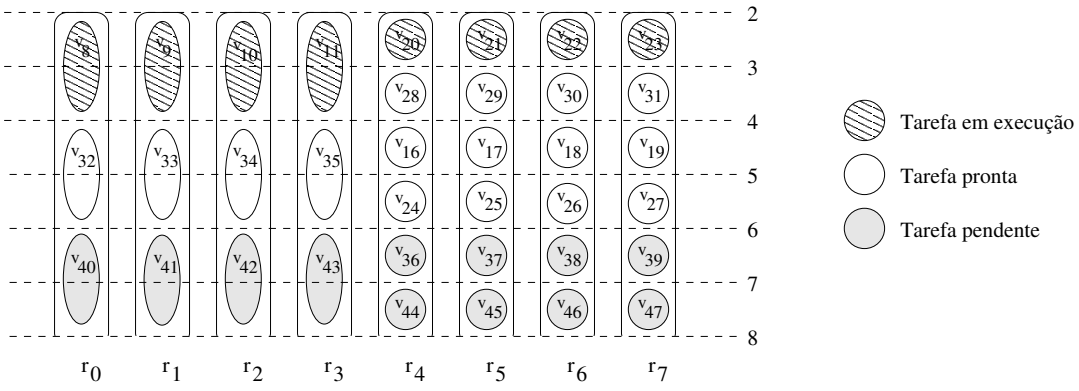

Figura 7.8: Visualização corrente da execução de tarefas até o tempo $T = 8s$.

No tempo $T = 6s$, após o término da execução das tarefas v_{32}, v_{33}, v_{34} e v_{35}, que são as primeiras tarefas pertencentes ao nível 1 a finalizar, o SDS verifica se é possível realizar um novo reescalonamento de tarefas pendentes. A estrutura $aloc_2$ é construída considerando as tarefas v, que possuem $level(v) \leq 2$, como visualizada na Figura 7.9.

Seguindo o Algoritmo 17, o SDS define r_0 como r_{max}. Porém, como não é possível minimizar o tempo de fim de r_0 cedendo tarefas pendentes, o reescalonamento não é disparado.

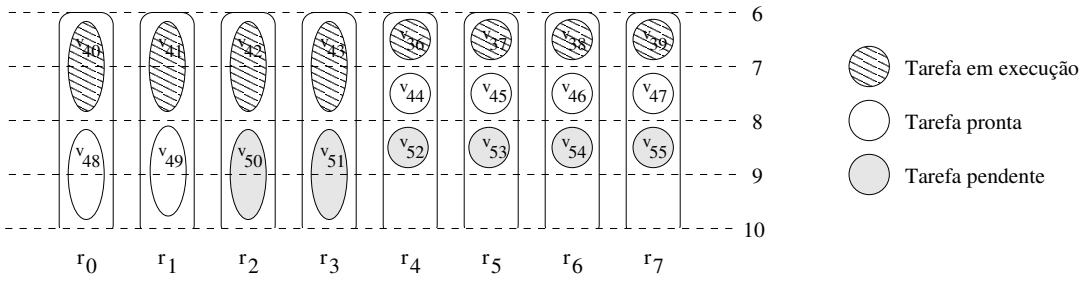

Figura 7.9: Estrutura de escalonamento $aloc_2$ para $level \leq 2$.

A execução da aplicação continua e após o tempo de execução $T = 9s$, as primeiras tarefas do nível 2 da árvore (v_{52}, v_{53}, v_{54} e v_{55}) terminam a sua execução, e além da realização do reescalonamento de tarefas prontas, um novo reescalonamento de tarefas pendentes pode ser avaliado. Para verificar se existe a necessidade de realização de um evento de escalonamento de prontas, o SDS_k calcula os valores para $sert_k^*$ e sii_k, baseado nos valores ert_j de cada recurso r_j do site s_k. Como pode ser observado na Figura 7.10, os únicos recursos que possuem tarefas prontas são r_2 e r_3, sendo os seus tempos restantes estimados iguais a 2 segundos. Os demais recursos possuem $ert = 0$.

$$sert_k^* = \frac{(2 \times 0,5) + (2 \times 0,5)}{0,5 + 0,5 + 0,5 + 0,5 + 1 + 1 + 1 + 1} = 0,33$$

$$sii = \frac{2}{0,33} = 6$$

Quando é verificado o desbalanceamento de tarefas prontas no site, o SDS cria $aloc_3$, identifica o conjunto R_{ask}, R_{sub} e o conjunto $tarefas_{ask}$ que contém as tarefas que devem ser cedidas (Algoritmo 15). De acordo com a estrutura $aloc_3$, apresentada na Figura 7.10, as tarefas v_{58} e v_{59} são selecionadas para reescalonamento por possuírem tempo de fim maior do que $sert_k^* + T$ (Algoritmo 15, linha 7).

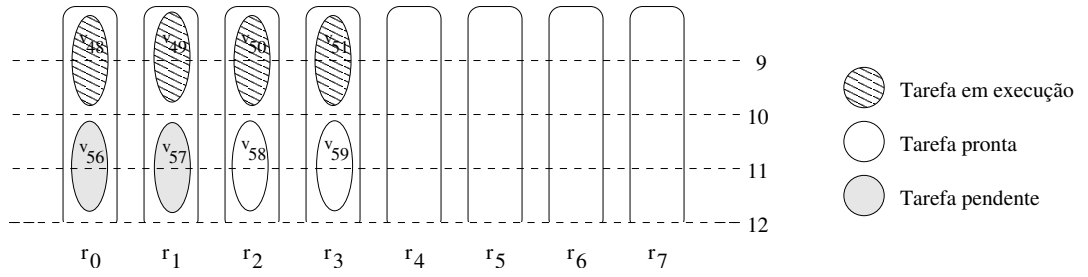

Figura 7.10: Estrutura de escalonamento $aloc_3$ para $level \leq 3$.

Após a identificação do conjunto de tarefas que devem ser reescalonadas, o SDS envia um pedido para cada HDS_j alocado em um $r_j \in R_{ask}$, que possua tarefas do conjunto $tarefas_{ask}$. O SDS aguarda até que todos os pedidos sejam respondidos e só então reescalona as tarefas cedidas ($pacote_{HDS}$) de acordo com o Algoritmo 16. As tarefas $v \in pacote_{HDS}$ são ordenadas pelo seu $blevel(v)$ e escalonadas no recurso $r_j \in R_{sub}$ que minimize o tempo de fim de v. A estrutura $aloc_3$, após o reescalonamento de tarefas prontas pode ser visualizada na Figura 7.11. Como se pode observar, as tarefas v_{58} e v_{59} não são iniciadas exatamente no tempo $T = 9s$. Dado que o evento de escalonamento de prontas ocorreu logo após $T = 9s$, deve se contabilizar um custo de reescalonamento e um pequeno atraso no início da execução das tarefas.

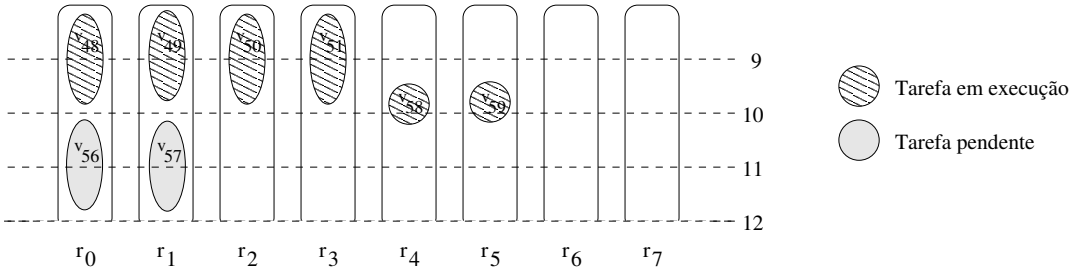

Figura 7.11: Estrutura de escalonamento $aloc_3$ após escalonamento de tarefas prontas.

Ao observar a Figura 7.11 após o escalonamento de tarefas prontas, verifica-se que o SDS irá disparar o evento de reescalonamento de tarefas pendentes relativo a estrutura $aloc_3$. O SDS identifica r_0 como o recurso que inicialmente define o *makespan* de $aloc_3$ e tenta minimizar o tempo de execução de v_{56} em outro recurso do site s_k. O procedimento cede_tarefas() é disparado e a tarefa v_{56} é associada ao recurso r_6 que diminui seu tempo final de execução em 1 segundo.

Conforme é mostrado no Algoritmo 17, linhas 5 a 8, o procedimento cede_tarefas() será executado enquanto for possível diminuir o *makespan* de um recurso r_{max}. Logo, em um segundo passo, o SDS identifica r_1 como r_{max} e a tarefa v_{57} também será cedida, sendo alocada no recurso r_7. Após estes ajustes, a estrutura auxiliar $aloc_3$ é apresentada na Figura 7.12.

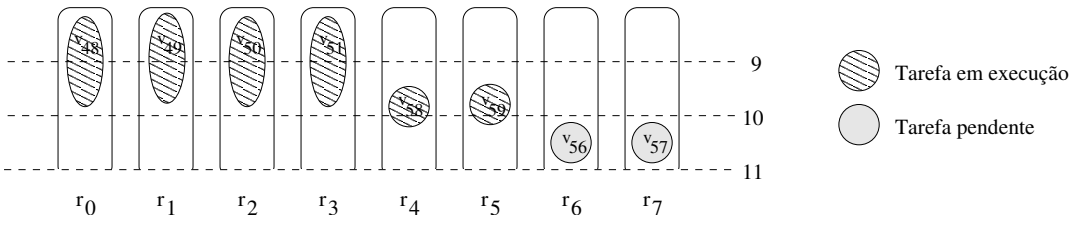

Figura 7.12: Estrutura de escalonamento $aloc_3$ após escalonamento de tarefas pendentes.

Seguindo a estratégia de escalonamento de tarefas pendentes, um novo *makespan* é calculado pelo SDS, considerando $aloc_3$ (Algoritmo 17, linha 9). Observando as Figuras 7.11 e 7.12, que representam o escalonamento do site antes e depois do reescalonamento de tarefas pendentes, verifica-se que o *makespan* do site diminuiu de 12 para 11 segundos. Assim, o evento de escalonamento de tarefas pendentes é disparado, onde as tarefas reescalonadas são pedidas para seus recursos origem e cedidas para seus recursos destino de acordo com as informações da estrutura $aloc_3$.

Até o final da execução, o SDS não realiza mais reescalonamentos de tarefas prontas. Isto ocorre devido a estrutura do grafo, onde o número de tarefas prontas a cada nível, reduz em relação ao número de recursos disponíveis. Assim, as tarefas que ficam prontas são imediatamente disparadas para execução impedindo o seu reescalonamento, pois o escalonador dinâmico proposto não trata tarefas em execução. Por outro lado, são disparados dois eventos de reescalonamento de tarefas pendentes considerando as estruturas auxiliares $aloc_4$ e $aloc_5$.

A estrutura $aloc_4$ (Figura 7.13, a partir de $T = 10s$) é criada após o término da execução das tarefas v_{58} e v_{59}. Assim, a tarefa v_{61} é disparada para execução e a tarefa pendente v_{60} é liberada para o reescalonamento local. O SDS decide colocar v_{60} no recurso r_4 (Figura 7.14) de acordo com o algoritmo de reescalonamento de tarefas pendentes explicado anteriormente.

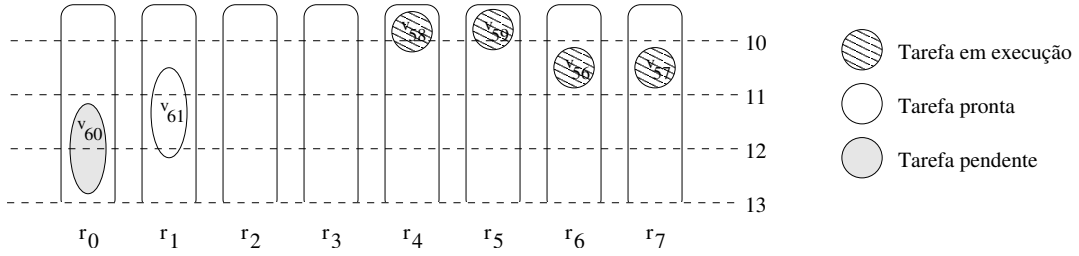

Figura 7.13: Estrutura de escalonamento $aloc_4$, a partir de $T = 10s$, para $level \leq 4$.

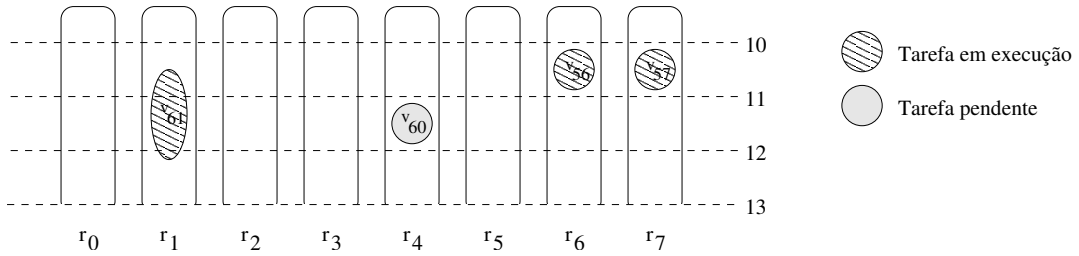

Figura 7.14: Estrutura de escalonamento $aloc_4$ após escalonamento de tarefas pendentes.

O último evento de escalonamento ocorre no tempo $T = 12s$ quando v_{60} termina sua execução (Figura 7.15). A tarefa v_{62} é liberada para reescalonamento e o SDS a aloca no recurso r_4 (Figura 7.16), minimizando o tempo final de execução da aplicação.

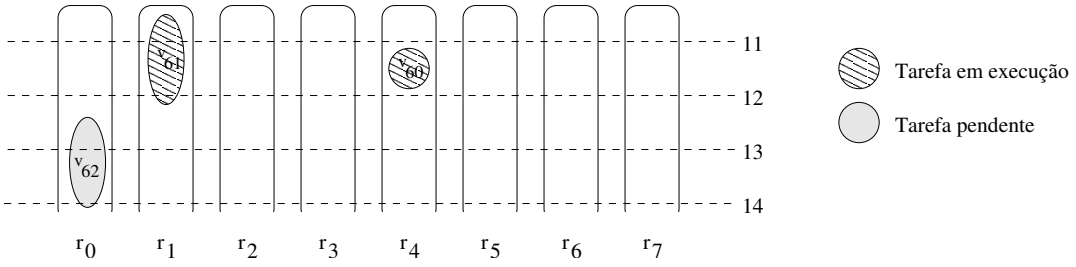

Figura 7.15: Estrutura de escalonamento $aloc_5$, a partir de $T = 12s$, para $level \leq 5$.

O *makespan* final obtido é aproximadamente 13 segundos, o que permite avaliar a eficiência do escalonador dinâmico em relação ao escalonador estático *HEFT*. Quando o algoritmo *HEFT* é executado sabendo que os 4 primeiros recursos do site oferecem apenas 50% do seu poder computacional é obtido o *makespan* de 13 segundos.

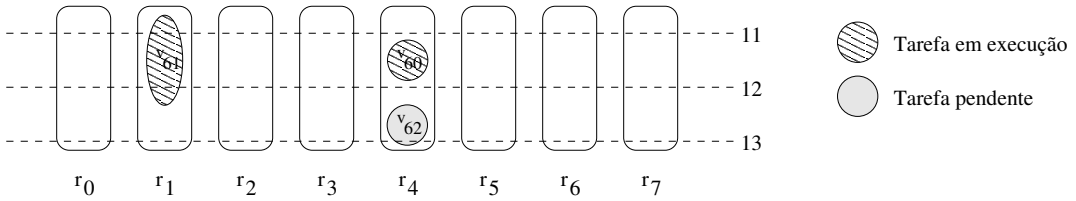

Figura 7.16: Estrutura de escalonamento $aloc_5$ após escalonamento de tarefas pendentes.

7.3 Resumo

Este capítulo apresentou a heurística de escalonamento dinâmico de tarefas para aplicações que possuem relações de precedência entre suas tarefas. Sendo este tipo de aplicação mais complexo do que as aplicações BoT, algumas considerações em relação ao escalonamento de aplicações GAD foram discutidas para que melhores alternativas fossem utilizadas na política proposta nesta tese.

A estratégia de escalonamento descrita foi implementada no contexto do *EasyGrid AMS*, e detalhes como a interação entre os diferentes escalonadores da hierarquia e suas principais funções foram especificados. A heurística apresentada torna o *EasyGrid AMS* mais abrangente, podendo este ser usado para diferentes tipos de aplicações paralelas. No próximo capítulo são apresentados os experimentos realizados, onde a heurística proposta nesta tese é comparada com outras heurísticas de escalonamento híbrido encontradas na literatura.

Capítulo 8

Análise Experimental

Este capítulo apresenta os resultados obtidos nos experimentos realizados para aplicações GAD. Para a política de escalonamento local foram realizados testes comparativos entre a heurísticas propostas nos Capítulos 5 e 7 e outras heurísticas encontradas na literatura. Avaliações para o escalonamento dinâmico global foram feitas usando a heurística local que apresentou melhores resultados.

Nos resultados apresentados no Capítulo 6 foram discutidas as características da estrutura de escalonamento proposta, onde se concluiu que a estrutura hierárquica e distribuída é eficiente e escalável. Além disso, dado que as aplicações executadas no *EasyGrid AMS* se tornam *system aware*, o escalonamento distribuído por aplicação consegue perceber a existência de outras aplicações no sistema e as características do ambiente computacional, lidando com possíveis conflitos.

O objetivo deste capítulo é avaliar a eficiência da estrutura de escalonamento proposta, onde os diferentes níveis da hierarquia de escalonamento adotam políticas distintas, que trabalham em conjunto para proporcionar uma execução eficiente das aplicações GAD em uma grade computacional. A idéia de adotar este tipo de estratégia distribuída para o escalonamento de tarefas de uma mesma aplicação é permitir que o sistema gerenciador de aplicações se torne escalável para ambientes computacionais de larga escala. Antes da análise dos resultados, são descritos o ambiente de execução, as características das aplicações e as métricas de avaliação.

8.1 Ambiente de Avaliação

Assim como os experimentos realizados para aplicações BoT, o ambiente de execução utilizado para aplicações GAD foi o oferecido pelo Grid Sinergia, descrito com maiores detalhes no Capítulo 6, Seção 6.1. Para evitar a influência da execução de outras aplicações nos resultados obtidos, os testes realizados foram executados nas máquinas do SGCLab da UFF. Desta forma, é possível obter um ambiente semi-controlado onde os recursos da grade

computacional são dedicados apenas à execução das aplicações que estão sendo avaliadas. Entretanto, como visto anteriormente, os canais de comunicação continuam compartilhados, podendo sofrer alguma interferência externa.

Diferentemente da configuração apresentada na Figura 6.1, a grade computacional do SGCLab sofreu algumas mudanças. Atualmente, a configuração disponível para testes é composta por 26 processadores Pentium IV 2.6 GHz com 512Mb RAM, rodando Linux Fedora Core 2, Globus Toolkit 2.4 e LAM/MPI 7.0.6. A Figura 8.1 mostra que as máquinas estão divididas em três sites, interconectados por switches Fast Ethernet e Gigabit Ethernet, onde o site 1 possui 7 máquinas, o site 2 possui 13 máquinas e o site 3, 6 máquinas.

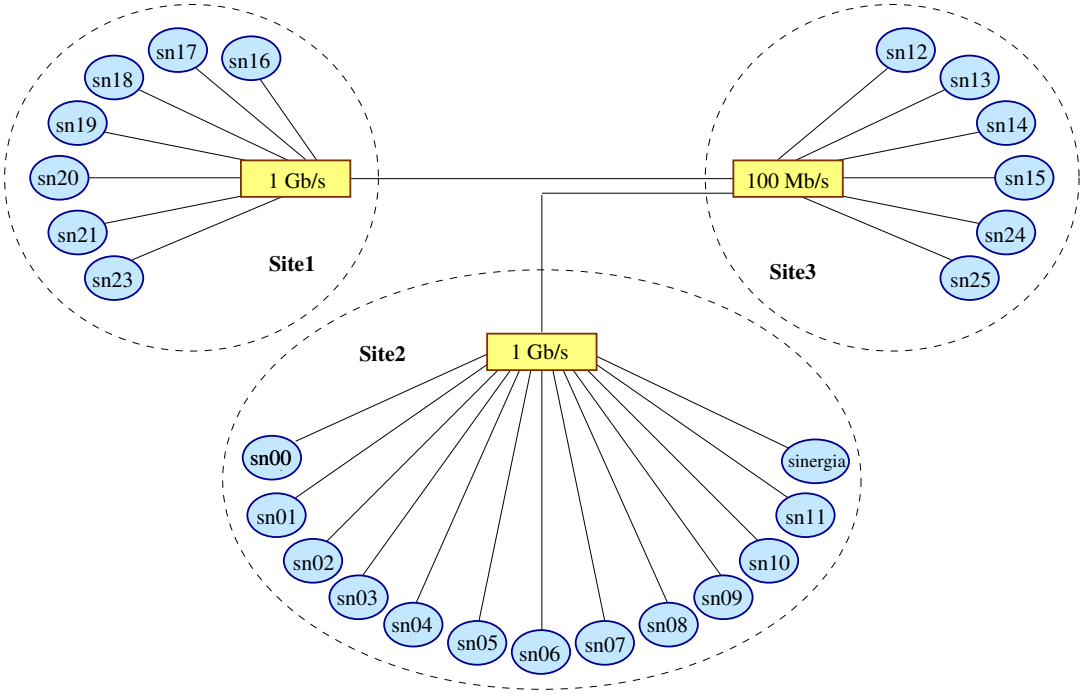

Figura 8.1: Configuração da grade computacional do SGCLab nos experimentos realizados para aplicações GAD

8.2 Aplicações Avaliadas

Para testar a política desenvolvida para aplicações GAD foram utilizadas aplicações sintéticas e uma aplicação real. O objetivo de usar aplicações sintéticas é poder controlar a duração de uma tarefa e a quantidade de *bytes* enviada entre tarefas adjacentes. Desta forma, é possível uma melhor avaliação do *makespan* obtido pela aplicação. Tais aplicações sintéticas são representadas por árvores binárias e grafos diamante.

Árvore binária é um dos paradigmas mais conhecidos na computação paralela. Algoritmos de difusão (*broadcast*) e divisão e conquista são alguns dos exemplos que se ajustam

a este tipo de grafo, aonde os dados vão da raiz até as folhas da árvore. Em outras aplicações, este caminho pode ser inverso, isto é, dados partem das folhas para a raiz, o que representa uma redução nas operações. Um exemplo clássico deste tipo de abordagem é a soma em paralelo [100]. As árvores utilizadas nestes experimentos são completas e podem ser separadas em dois grupos: (a) árvores do tipo *out-tree* e (b) árvores do tipo *in-tree* (invertidas) que são visualizadas na Figura 8.2.

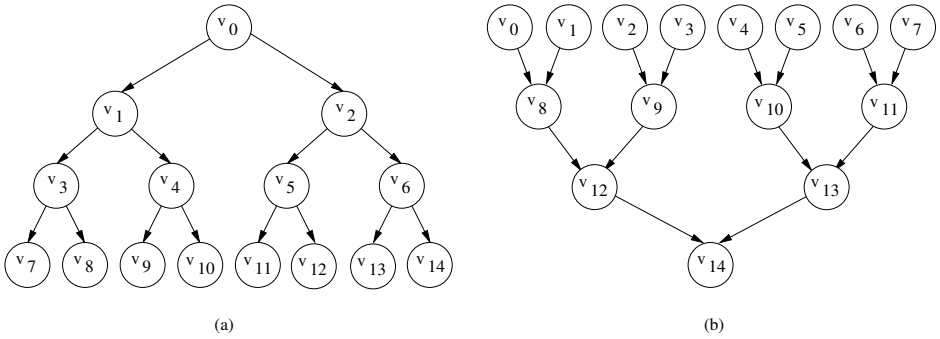

Figura 8.2: (a) Árvore binária do tipo *out-tree*. (b) Árvore binária do tipo *in-tree*.

Grafos diamante também constituem uma classe importante da computação científica. Tais grafos podem representar aplicações como: multiplicação de matrizes, *systolic arrays* e decomposição LU [92]. Esta classe de grafos também é muito usada para modelar aplicações que façam simulações de problemas da física, engenharia, meteorologia, entre outros. Um exemplo deste tipo de grafo é mostrado na figura abaixo.

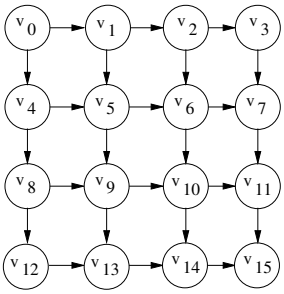

Figura 8.3: Grafo diamante

A aplicação real utilizada nestes experimentos considera um problema da astrofísica conhecido como *N-corpos* (*N-body*). Em sua implementação clássica este problema simula a evolução de um sistema composto por N corpos ou partículas que interagem entre si de acordo com a lei da gravidade de Newton. Ou seja, dado um conjunto de partículas e suas condições iniciais, o objetivo do problema é prever a evolução do sistema. Na literatura, os algoritmos para solução do problema de *N-corpos* são divididos em duas categorias principais, os que usam uma abordagem direta [1] e aqueles que adotam um esquema aproximativo [10].

Nesta tese, é utilizado o algoritmo de somatório direto (*direct summation*), onde é computada a interação computacional de um conjunto de N partículas [103]. Neste caso, a cada passo do algoritmo é calculado um total de N^2 forças entre partículas. Este código pode ser usado para resolver problemas de astrofísica como: evolução de aglomerados estelares, formação de sistemas planetários, simulações cosmológicas e a dinâmica dos buracos negros [63]. Dado que aglomerados de estrelas podem conter entre milhares e milhões de estrelas, a simulação deste problema com precisão científica necessita tipicamente de uma capacidade de processamento só disponível em um supercomputador ou em uma grade computacional. O grafo de tarefas que representa um passo do algoritmo de *N-corpos* pode ser visto na Figura 8.4.

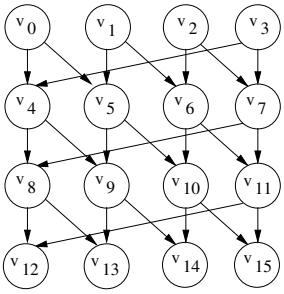

Figura 8.4: Representação do problema de *N-corpos* com 16 tarefas, onde cada tarefa v processa um total de N/W partículas. W representa a largura do grafo, neste exemplo $W = 4$.

Nos testes realizados, foram avaliadas aplicações GAD com granularidade grossa, isto é, o peso dado a uma tarefa é maior do que o tamanho de uma mensagem. De uma maneira geral, considerando duas máquinas distintas do SGCLab, cada mensagem trocada entre processos do *EasyGrid AMS* gasta menos do que 9 milissegundos entre seu envio e recebimento. Sabe-se que este valor é mantido para mensagens com tamanho até 24 *KBytes*. Por outro lado, os pesos computacionais das tarefas de uma aplicação serão definidos de acordo com os experimentos que serão avaliados a seguir.

8.3 Métricas de Avaliação e Parâmetros de Execução

Quando as execuções são realizadas em ambientes controlados nos quais é possível conhecer o poder computacional das máquinas e o tempo de execução das tarefas, o *makespan* obtido por uma heurística pode ser utilizado para avaliar a sua eficiência. Como o objetivo é minimizar o tempo final de execução da aplicação, a melhor heurística é aquela que obtiver o menor *makespan*.

Como aplicações GAD possuem comunicação entre as suas tarefas, e os canais de comunicação que interconectam as máquinas do SGCLab são compartilhados, os *makespans* obtidos podem sofrer alguma interferência relativa a variação da rede, que não é controlada

neste ambiente de execução. Logo, os *makespans* apresentados nos gráficos deste capítulo representam uma média de oito execuções de cada política de escalonamento sobre uma determinada aplicação. As análises de desempenho das heurísticas são realizadas considerando um intervalo de confiança de 95%.

Assim como no Capítulo 6, o tempo de execução de uma tarefa, em uma máquina ociosa com $csi = 1$, será denotado por t_{exe}. A Tabela 8.1 mostra os parâmetros de execução usados pelos escalonadores dinâmicos nos testes realizados nas próximas seções. Como a aplicação tratada possui relações de precedência entre suas tarefas, as execuções foram feitas considerando que apenas um processo da aplicação poderia ser disparado por vez ($nt_j = 1$). Os valores para o intervalo máximo entre mensagens de monitoramento ($I_{monitor}$) e para os índices de desbalanceamento local (θ) e global (ϕ) foram os mesmos usados para a execução de aplicações BoT. O parâmetro, I_{min}, que indica o intervalo mínimo entre cálculos de desbalanceamento de tarefas prontas estará associado a média do tempo de execução de todas as tarefas v da aplicação paralela, considerando o recurso r_j mais rápido da grade computacional, ou seja, $I_{min} = \overline{t_{exe}} = \frac{\sum_{v_i \in V} et(v_i, r_j)}{|V|}$. Assim, se a média dos tempos de execução for 1 segundo, então $I_{min} = 1s$.

Tabela 8.1: Parâmetros de execução

nt_j	I_{min}	$I_{monitor}$	θ	ϕ
1	$\overline{t_{exe}}$	5s	10%	10%

8.4 Experimentos e Avaliação dos Resultados

Os experimentos realizados para avaliação da estrutura de escalonamento e das heurísticas dinâmicas propostas no Capítulo 7 são divididos em duas partes principais. Primeiramente, é avaliada a política de escalonamento dinâmico local, onde a Subseção 8.4.1 mostra os resultados obtidos nas comparações entre várias heurísticas de escalonamento dinâmico; a Subseção 8.4.2 apresenta a intrusão das heurísticas locais implementadas; e a Subseção 8.4.3 faz a avaliação da heurística de escalonamento dinâmico local proposta com a execução da aplicação real de *N-corpos*. A segunda parte dos experimentos, apresentada nas Subseções 8.4.4 e 8.4.5, é dedicada a análise da heurística de escalonamento dinâmico global que trabalha em conjunto com as políticas de escalonamento dinâmico do SDS e HDS.

8.4.1 Avaliação de políticas de escalonamento local

Esta subseção apresenta os testes realizados considerando cinco heurísticas de escalonamento híbrido específicas para aplicações GAD: CD, CS, PS, GAD1 e GAD2. Assim como justificado no Capítulo 3, as políticas CD, CS e PS foram escolhidas por produzirem bons

resultados computacionais e principalmente por terem sido projetadas especificamente para aplicações que possuem relações de precedência entre suas tarefas [79, 83]. Os resultados obtidos na execução de cada heurística são comparados entre si, e também com a heurística proposta para aplicações BoT (denominada heurística BoT) e com a execução da aplicação sem a ativação do escalonamento dinâmico.

As heurísticas CD, CS, e PS, apresentadas em [83], consideram que um bloco B_l de tarefas pode ser reescalonado apenas uma vez. Na proposta original, este bloco B_l pode possuir tarefas de um único nível topológico do grafo ou tarefas que pertençam a vários níveis topológicos consecutivos. Se um bloco B_l, considerado para reescalonamento, possui tarefas de vários níveis e o ambiente computacional sofre variações, o desempenho do algoritmo pode ser ruim, já que as tarefas de B_l não serão reescalonadas uma segunda vez. Logo, para melhor adaptação destas heurísticas a ambientes dinâmicos como as grades, a implementação realizada nesta tese considera que cada bloco B_l de tarefas possui apenas as tarefas v de um único nível topológico, onde $level(v) = l$.

A estratégia denotada por GAD1 foi proposta e detalhada no Capítulo 7, enquanto a estratégia BoT foi apresentada no Capítulo 5. É importante ressaltar que a heurística proposta para aplicações BoT pode ser usada para aplicações com relações de precedência, porém como apenas as tarefas prontas são reescalonadas, o seu desempenho poderá ser pior, à medida que um menor número de tarefas prontas existir em um dado evento de escalonamento. Entretanto, como na heurística BoT não é necessária a criação de estruturas de escalonamento auxiliares (*aloc*, definida no Algoritmo 14) que armazenam o estado do site em um dado momento, ela se torna mais escalável quando existir um número muito grande de tarefas *prontas* para ser tratado. Como explicado no Capítulo 7, diferentemente das heurísticas CD, CS, e PS onde as tarefas *prontas* e *pendentes* são tratadas em um mesmo evento de escalonamento, a heurística GAD1 possui estratégias de escalonamento distintas para tarefas *prontas* e *pendentes*.

A heurística GAD2, também proposta nesta tese, é uma variação da heurística CD, onde para contornar a limitação de reescalonar tarefas de um bloco B_l apenas uma vez, um segundo mecanismo de escalonamento de tarefas é adicionado. Logo, além da política de escalonamento utilizada por CD, GAD2 possui a mesma estratégia de reescalonamento de tarefas *prontas* usada pela heurística GAD1. Desta forma, GAD2 possui dois mecanismos distintos de reescalonamento de tarefas. O primeiro deles, assim como implementado em CD, trata em um mesmo evento de escalonamento tarefas *prontas* e *pendentes* com $level(v) = l$. O segundo mecanismo, detalhado no Algoritmo 15 do Capítulo 7, permite que apenas o conjunto das tarefas *prontas* v, cujo $level(v) \leq l$, sejam consideradas para reescalonamento. Neste caso, o evento de escalonamento pode ser disparado sempre que o intervalo I_{min} for ultrapassado. As principais diferenças entre os tipos de tarefas tratadas a cada evento de escalonamento, nas heurísticas implementadas no *EasyGrid AMS*, podem ser visualizadas na Tabela 8.2.

Tabela 8.2: Mecanismos de escalonamento em comum das heurísticas avaliadas.

Heurísticas	Mecanismos de escalonamento comuns
CD, CS, PS e GAD2	Um mesmo evento de escalonamento trata tarefas prontas e pendentes. Tais eventos são disparados de acordo com o nível topológico l do grafo. O bloco B_l contém as tarefas v, onde $level(v) = l$.
GAD1, GAD2 e BoT	Possui eventos de escalonamento que tratam apenas as tarefas prontas. Estes eventos são disparados respeitando o intervalo I_{min}. O bloco B_l contém as tarefas prontas v, onde $level(v) \leq l$.
GAD1	Possui eventos de escalonamento específicos para as tarefas pendentes. Tais eventos são disparados de acordo com o nível topológico l do grafo. O bloco B_l contém as tarefas pendentes v, onde $level(v) \leq l$.

A configuração virtual do *EasyGrid AMS*, para os testes apresentados a seguir, é visualizada na Figura 8.5. Com o objetivo de aumentar os números de recursos de um site, considera-se que todas as máquinas do SGCLab pertencem ao mesmo site virtual. Assim, o gerenciador global (GM) e o gerenciador do site (SM) são alocados em uma mesma máquina (sinergia) e outras 24 máquinas são usadas para executar processos da aplicação, possuindo um HM cada.

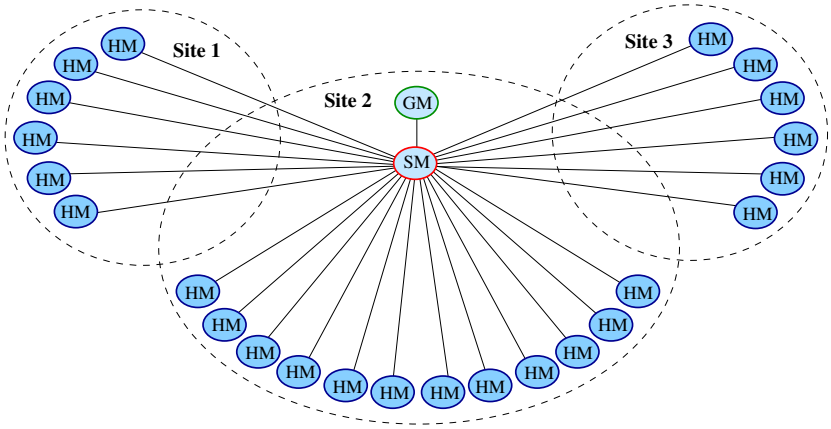

Figura 8.5: Configuração virtual da grade para a avaliação das heurísticas de escalonamento local.

Na primeira parte dos testes realizados foram consideradas apenas aplicações sintéticas. Os nomes das aplicações são compostos por duas partes: a primeira é um prefixo que indica o tipo de grafo que representa a aplicação e a segunda indica a quantidade de tarefas que a mesma possui. O prefixo OUT é usado para árvores *out-tree*, IN para árvores invertidas *in-tree* e DI para grafos diamante. Por exemplo, a aplicação IN_{4095} representa uma árvore binária invertida com 4095 tarefas.

Dentro da grade virtual apresentada na Figura 8.5, três configurações distintas foram usadas para a realização dos experimentos. Na primeira configuração foram utilizadas

apenas quatro máquinas para executar as tarefas da aplicação do usuário, na segunda 12 máquinas e por último 24 máquinas. Para cada configuração de testes foram definidos quatro cenários de execução:

- HOM: ambiente homogêneo. Todos os recursos da grade oferecem o mesmo poder computacional.
- HET1: ambiente heterogêneo. Os recursos utilizados são divididos em dois subconjuntos iguais. A primeira metade dos recursos utilizados oferece apenas 50% do seu poder computacional enquanto a segunda metade oferece 100%.
- HET2: ambiente heterogêneo. Representa o inverso do cenário HET1. Neste caso, é a segunda metade dos recursos utilizados que oferece apenas 50% do seu poder computacional.
- DIN: ambiente heterogêneo e dinâmico. Ao iniciar a execução da aplicação o ambiente oferecido é igual ao cenário HET1, porém, logo após o escalonamento das tarefas do primeiro nível da aplicação, a configuração oferecida migra para o cenário HET2. Aproximadamente na metade da execução da aplicação, a configuração dos recursos retorna ao estado inicial, representado pelo cenário HET1.

É necessário ressaltar que, em todos os testes realizados nesta subseção, independentemente do tipo de cenário usado ou da heurística avaliada, a alocação inicial feita pelo escalonador estático (*HEFT* [108]) é sempre a mesma. Isto é, o escalonamento estático produzido supõe que todas os recursos disponíveis oferecem 100% de poder computacional para a aplicação do usuário. É o escalonador dinâmico que percebe o tipo real de cenário (HOM, HET1, HET2 ou DIN) durante a execução. Esta estratégia é usada para que se possa avaliar a eficiência do escalonador dinâmico em detectar mudanças no ambiente computacional e, consequentemente, adaptar a execução da aplicação. Além disso, os escalonadores dinâmicos global e do site não competem com a aplicação do usuário, visto que estes são associados aos processos GM e SM que executam exclusivamente na máquina sinergia.

O cenário de execução HOM é usado para verificar se a heurística dinâmica utiliza de maneira eficiente as informações e a alocação inicial estabelecida pelo escalonador estático. Nestes casos, o ideal é que o escalonador dinâmico não precise reescalonar muitas tarefas. O reescalonamento deve acontecer apenas para ajustar a execução da aplicação às pequenas variações que existem em qualquer ambiente computacional.

As análises realizadas nos cenários HET1 e HET2 têm o objetivo de avaliar as heurísticas implementadas quando um ambiente heterogêneo for detectado pelo escalonador dinâmico. Estes cenários podem ser encontrados quando o escalonador estático não recebe a especificação de uma modelagem prévia do sistema, e a política dinâmica deve se eficiente para reajustar o escalonamento estático inicial para a nova configuração encontrada.

Os cenários HET1 e HET2 possuem como única diferença a posição das máquinas que oferecem menor poder computacional. Como as heurísticas testadas tendem a escalonar tarefas nos primeiros recursos encontrados que minimizem o tempo de execução de uma tarefa v, a análise dos resultados obtidos em HET1 e HET2 foi realizada para verificar o impacto nas decisões das políticas de escalonamento dinâmico. Comparativamente, as heurísticas avaliadas obtiveram o mesmo desempenho entre si, porém, *makespans* ligeiramente menores foram encontrados para o ambiente HET2. Isto se deve ao fato, principalmente, da abordagem usada pela heurística estática *HEFT*, onde em caso de empate entre recursos é dada preferência ao primeiro recurso encontrado. Desta forma, as tarefas *fontes* do grafo que são escalonadas pela heurística estática para serem disparadas no primeiro subconjunto de recursos, assim que a aplicação inicia sua execução, irão atrasar a execução de suas sucessoras no cenário HET1. Isto ocorre, pois os primeiros recursos de HET1 são os mais lentos, e as políticas dinâmicas implementadas não reescalonam tarefas em execução.

Para analisar o comportamento das heurísticas em um ambiente heterogêneo e dinâmico foi definido o cenário DIN. Neste caso, além da política de escalonamento ter que adaptar a execução da aplicação ao ambiente heterogêneo encontrado inicialmente, uma segunda mudança na capacidade de processamento dos recursos é realizada. Embora somente duas mudanças no poder computacional dos recursos tenham sido estabelecidas, o cenário DIN é importante para verificar se a heurística dinâmica é capaz de perceber as variações do ambiente computacional e ajustar eficientemente a execução da aplicação.

Apesar das mudanças que ocorrem na capacidade de processamento dos recursos no cenário DIN, é importante ressaltar que o sistema retorna para o seu estado inicial e não ocorrem variações no poder computacional total do sistema. Tal cenário pode favorecer o desempenho do escalonamento estático quando este não possuir conhecimento da configuração inicial disponível. Muitas configurações de um ambiente dinâmico podem ser criadas, e conclusões diferentes podem ser obtidas com relação à necessidade de acurácia do escalonamento estático inicial. O que se pode destacar é que quanto maior as mudanças em um ambiente, menor será a importância das decisões do escalonador estático.

Dado que os recursos do SGCLab utilizados possuem a mesma capacidade computacional, para simular a heterogenidade do ambiente, *cargas extras* de processamento (programas *CPU-bound*) são disparadas em algumas máquinas para que estas tenham seu poder computacional diminuído. Assim, os cenários HET1, HET2 e DIN são configurados utilizando este mecanismo para que o poder computacional oferecido para a aplicação do usuário seja alterado de forma controlada.

Nas três configurações da grade virtual (4, 12 e 24 máquinas) foram executadas aplicações GAD com tempo uniforme de execução e de comunicação entre as tarefas. As mensagens entre as tarefas possuem um tamanho de *128 bytes*, enquanto todas as tarefas da aplicação possuem o mesmo tempo de execução (t_{exe}) de um segundo ou de cinco segundos, relativo a execução exclusiva na máquina mais rápida da grade.

Considerando as configurações escolhidas, os cenários estabelecidos, as heurísticas comparadas e as aplicações sintéticas executadas, os testes aqui apresentados correspondem a aproximadamente 45 dias de utilização exclusiva das máquinas do SGCLab. Por ser um ambiente de estudo compartilhado entre várias pesquisas diferentes, a realização de um maior número de testes com exclusividade no uso das máquinas se torna uma dificuldade.

Configuração com 4 HMs

A primeira configuração apresentada, com apenas quatro máquinas executando as tarefas do usuário, está longe de representar uma configuração real de uma grade computacional. O objetivo é apenas verificar o comportamento das heurísticas quando um site oferecer um número reduzido de máquinas. Os testes foram realizados utilizando cinco máquinas do site 2 da grade computacional: sinergia, sn00, sn01, sn02, sn03. A configuração virtual com quatro HMs é apresentada na Figura 8.6.

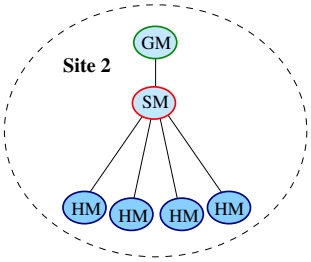

Figura 8.6: Configuração virtual com 4 HMs. O GM e o SM executam em uma mesma máquina.

Nesta configuração, as seguintes aplicações foram executadas: DI_{64}, DI_{256}, DI_{400}, OUT_{63}, OUT_{255}, OUT_{511}, IN_{63}, IN_{255} e IN_{511}. O testes realizados no cenário HOM mostraram que todas as heurísticas possuem no geral, o mesmo desempenho em relação ao *makespan* obtido, igualando a média de tempo à execução estática. Como os *makespans* obtidos são semelhantes, a comparação entre as heurísticas pode ser feita através da quantidade de tarefas reescalonadas durante a execução. A Tabela 8.3 mostra a média do número de tarefas reescalonadas por tipo de aplicação (DI, OUT e IN).

Tabela 8.3: Média do número de tarefas reescalonadas no cenário HOM (4 HMs).

t_{exe}	aplicação	GAD1	CD	CS	PS	BoT	GAD2
1s	*DI*	0	228,5	229,9	229,3	2,7	0
	OUT	0	41,8	57,5	90,5	0,4	34,5
	IN	0	41,0	43,0	43,1	0,8	41,8
5s	*DI*	4,1	179,9	186,1	185,0	12,4	230,9
	OUT	0	180,8	175,0	211,3	2,2	202,5
	IN	0	103,2	111,5	109,7	0,5	105,8

Com relação aos valores apresentados na Tabela 8.3 é possível observar que as heurísticas PS, CS, CD e GAD2 reescalonam o maior número de tarefas, já que suas decisões não consideram a alocação inicial definida pelo escalonador estático, mas somente prioridades associadas a esta alocação. Por outro lado, GAD1 é a heurística que realiza o menor esforço, mostrando assim, a importância de uma abordagem realmente híbrida.

Os resultados obtidos para os cenários HET1 e HET2 são similares, sendo discutidos apenas os *makespans* alcançados para o cenário HET1, onde as máquinas sn00 e sn01 oferecem 50% do seu poder computacional. Os resultados para cada heurística são apresentados nos gráficos das Figuras 8.7 (tarefas de 1 segundo) e 8.8 (tarefas de 5 segundos).

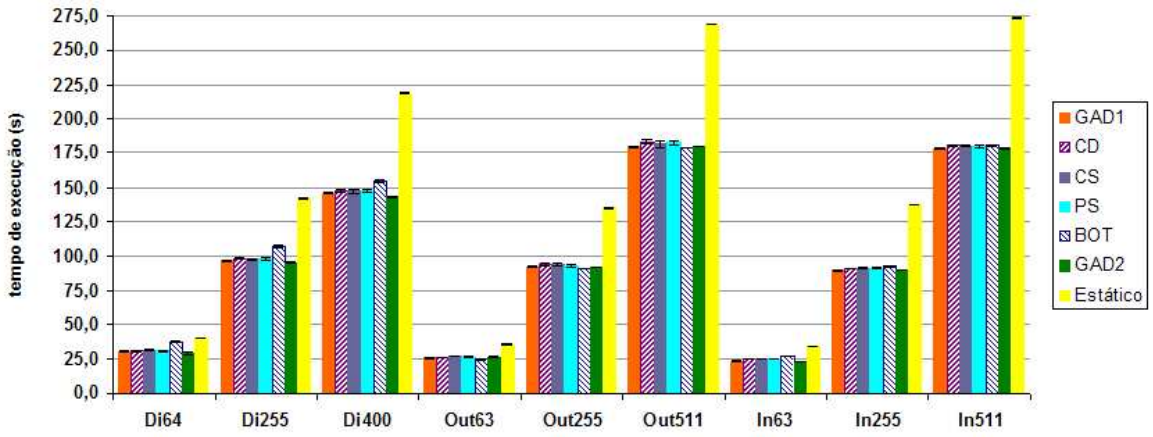

Figura 8.7: Execução em um ambiente heterogêneo (HET1) com quatro máquinas, onde $t_{exe} = 1s$. Intervalo de confiança de 95%.

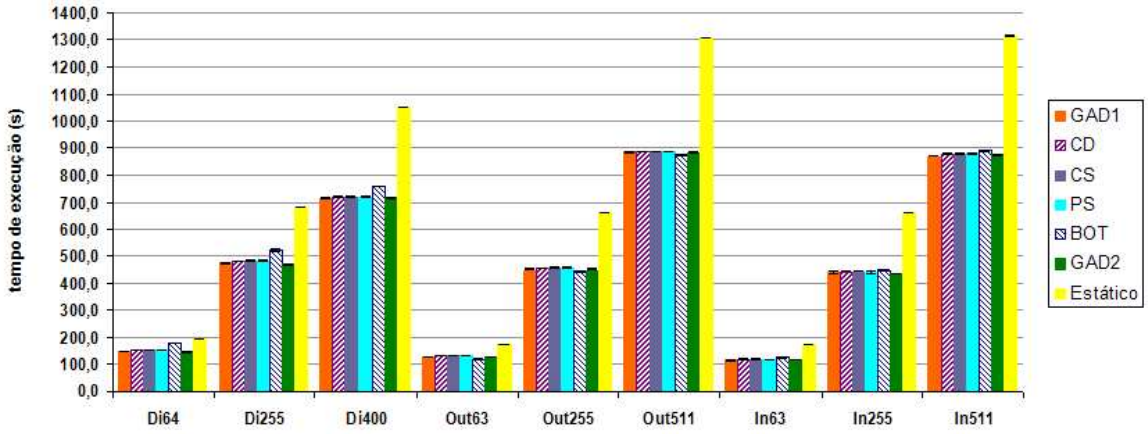

Figura 8.8: Execução em um ambiente heterogêneo (HET1) com quatro máquinas, onde $t_{exe} = 5s$. Intervalo de confiança de 95%.

Através da observação dos gráficos, verifica-se que o comportamento obtido entre as diferentes heurísticas na execução de aplicações com tarefas de duração de 1 e 5 segundos é muito semelhante. Como esperado, a execução estática obtém os piores resultados, já

que não adapta a sua alocação inicial ao ambiente heterogêneo disponível, detectado no início da execução da aplicação.

Para uma melhor comparação entre os resultados, a média dos *makespans* obtidos para cada heurística, por tipo de aplicação, são classificados de acordo com o critério de qualidade, onde a heurística mais eficiente (com melhor média) recebe valor 1. As outras heurísticas recebem valores de qualidade que indicam o quão distante o *makespan* obtido está da melhor média. Por exemplo, na Tabela 8.4, que mostra os resultados obtidos no cenário HET1, os valores apresentados na primeira linha indicam que a melhor heurística é GAD2 e que GAD1, CD, CS, PS, BoT e a execução estática, alcançaram *makespans* na média 2%, 3,6%, 3,3%, 3,3%, 12,2% e 50,6% piores do que GAD2, respectivamente.

O cálculo da qualidade de uma heurística é mostrado a seguir, onde o seu *makespan* é comparado com o menor *makespan* ($menorMspan$) obtido entre todas as heurísticas para um determinado tipo de aplicação.

$$qualidade = \frac{(makespan \times 100/menorMspan) - 100}{100} + 1 \qquad (8.1)$$

Tabela 8.4: Qualidade das médias dos *makespans* obtidos no cenário HET1 (4 HMs).

t_{exe}	aplicação	GAD1	CD	CS	PS	BoT	GAD2	Estático
1s	*DI*	1,020	1,036	1,033	1,033	1,122	**1,000**	1,506
	OUT	1,010	1,030	1,027	1,026	**1,000**	1,012	1,497
	IN	**1,000**	1,018	1,017	1,018	1,031	1,002	1,534
5s	*DI*	1,005	1,017	1,018	1,019	1,099	**1,000**	1,451
	OUT	1,020	1,025	1,026	1,025	**1,000**	1,018	1,495
	IN	**1,000**	1,012	1,011	1,011	1,029	1,004	1,513

As heurísticas CD, CS e PS obtiveram praticamente o mesmo desempenho, enquanto a heurística BoT apresenta os seus melhores resultados para aplicações do tipo *out-tree*. Este resultado é possível para uma heurística simples como BoT, devido a topologia do grafo e a quantidade de máquinas oferecidas, onde um grande número de tarefas prontas aguardam por processamento em cada recurso. Neste caso, a medida que a execução da aplicação se aproxima do final, um maior número de tarefas prontas são consideradas no evento de escalonamento. Por outro lado, para grafos diamante, a heurística BoT obtém seu pior desempenho, já que pela própria topologia do grafo existe um menor número de tarefas prontas a serem tratadas em um evento de escalonamento.

Ainda com relação aos grafos diamante, GAD2 normalmente possui *makespans* médios ligeiramente melhores do que a heurística GAD1. A política de reescalonamento de tarefas pendentes de GAD1 é mais conservadora, onde é considerada a alocação inicial definida pelo escalonador estático e não apenas prioridades associadas ao escalonamento inicial. Além

disso, as tarefas v selecionadas de um bloco B_l são reescalonadas apenas se for possível a diminuição do *makespan* atual do site em um dado evento de escalonamento, não sendo realizados testes individuais para cada tarefa v, como acontece em GAD2.

De uma maneira geral, em relação aos tempos de execução obtidos no cenário HET1, as heurísticas GAD1 e GAD2 apresentam os melhores resultados, sendo que GAD1 reescalona menos tarefas, como pode ser visto na Tabela 8.5. A heurística BoT também reescalona um número pequeno de tarefas quando comparada às outras heurísticas, pois BoT não trata tarefas pendentes.

Tabela 8.5: Média do número de tarefas reescalonadas no cenário HET1 (4 HMs).

t_{exe}	aplicação	GAD1	CD	CS	PS	BoT	GAD2
1s	*DI*	47,1	165,8	177,9	180,1	35,2	181,0
	OUT	95,1	200,7	195,0	205,0	47,6	211,3
	IN	59,5	111,2	109,5	111,1	48,0	128,0
5s	*DI*	53,6	181,9	177,1	176,4	34,1	179,9
	OUT	99,2	220,7	206,9	217,9	46,1	210,3
	IN	62,9	107,6	112,8	121,1	47,0	146,5

Os testes apresentados a seguir consideram o cenário DIN, definido anteriormente. A primeira metade das máquinas (sn00 e sn01) possuem *cargas extras* de processamento (programas *CPU-bound*), oferecendo cada uma apenas 50% do seu poder computacional à aplicação do usuário. Após serem disparadas as primeiras tarefas, e o primeiro evento de escalonamento ter ocorrido, estas cargas são transferidas para sn02 e sn03 que passam a oferecer 50% de poder computacional, enquanto sn00 e sn01 se tornam dedicadas ao usuário. Aproximadamente, na metade do tempo de execução de cada aplicação, as *cargas extras* retornam para sn00 e sn01. Os resultados obtidos podem ser visualizados nas Figuras 8.9 e 8.10.

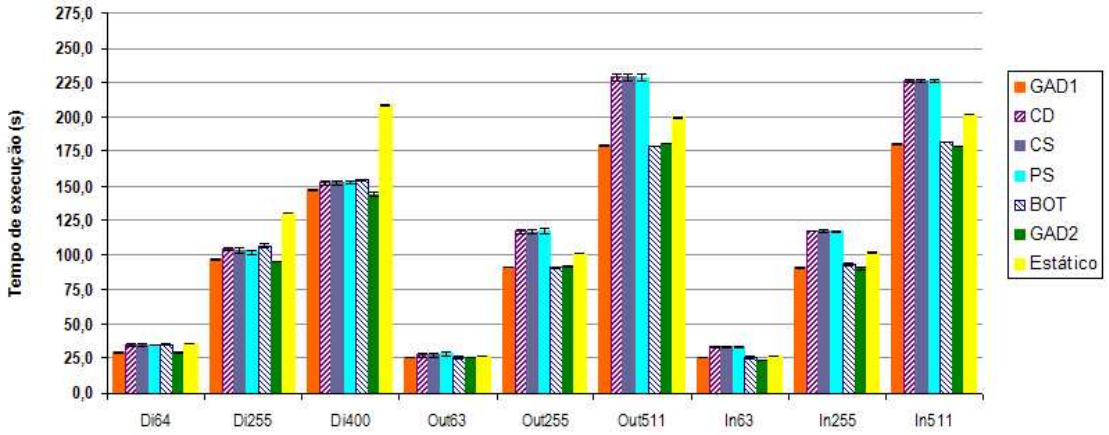

Figura 8.9: Execução em um ambiente dinâmico (DIN) com quatro máquinas, onde $t_{exe} = 1s$. Intervalo de confiança de 95%.

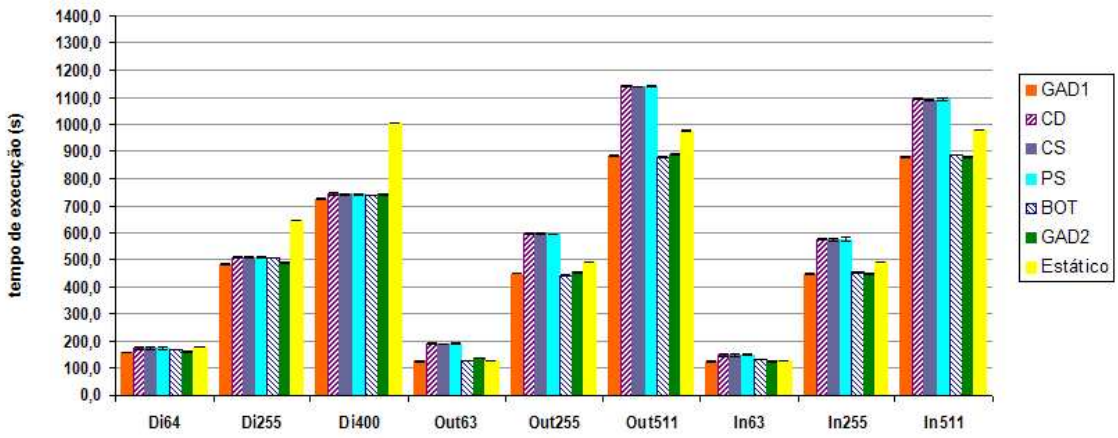

Figura 8.10: Execução em um ambiente dinâmico (DIN) com quatro máquinas, onde $t_{exe} = 5s$. Intervalo de confiança de 95%.

Ao observar os gráficos das Figuras 8.9 e 8.10, é possível verificar que as políticas CD, CS e PS possuem o pior desempenho entre as heurísticas dinâmicas. Tal comportamento é explicado pela filosofia dos algoritmos de escalonamento, onde uma tarefa só pode ser reescalonada apenas uma vez. Em ambientes dinâmicos, esta filosofia de implementação pode não ser adequada. Por exemplo, suponha que um recurso r oferece 100% de poder computacional em um instante de tempo t e passa a oferecer 50% em um instante $t+1$. Se o escalonamento das tarefas de um bloco B_l ocorrer entre os instantes t e $t+1$, as decisões do escalonador dinâmico irão considerar que o recurso r é totalmente dedicado a aplicação (100% do poder computacional). Entretanto, tal situação muda no instante $t+1$, e as tarefas já escalonadas não podem ser remanejadas. Por outro lado, se o evento de escalonamento ocorrer após $t+1$ a política será favorecida, podendo alcançar um bom desempenho. De qualquer modo, as heurísticas GAD1, GAD2 e BoT não são prejudicadas por esta situação, já que uma tarefa v, que possua $level(v) \leq l$, pode fazer parte do bloco de reescalonamento B_l, enquanto não for disparada para execução.

Com relação aos *makespans* obtidos na execução estática nos cenários HET1 e DIN, verifica-se através dos gráficos apresentados, que os valores obtidos no ambiente dinâmico foram melhores do que em HET1. Isto ocorre, devido à configuração estabelecida para o cenário DIN, onde o poder computacional total do sistema não é alterado em relação ao seu estado inicial e as *cargas extras* são distribuídas igualmente entre os recursos durante a execução da aplicação. Desta forma, diferentemente do cenário HET1, o atraso no processamento das tarefas é compensado ao longo da execução de acordo com a transferência das *cargas extras*. Assim, a execução estática pode alcançar melhores desempenhos principalmente para grafos onde existam um maior número de tarefas paralelas. Esta característica será exemplificada na configuração para 12 HMs.

A qualidade dos *makespans* obtidos no cenário de execução dinâmico (DIN) é apresentada na Tabela 8.6. As heurísticas CD, CS e PS, para grafos *out-tree* e *in-tree*, obtêm

resultados que são na média entre 26% e 33% piores que o melhor resultado alcançado. A estratégia BoT, assim como no cenário HET1, produz na média os melhores tempos de execução para grafos *out-tree*. No geral, GAD1 e GAD2 são as heurísticas com melhor desempenho, produzindo as melhores médias de *makespan*, ou resultados que se afastam em no máximo $2,2\%$ do melhor valor obtido.

Tabela 8.6: Qualidade das médias dos *makespans* obtidos no cenário DIN (4 HMs).

t_{exe}	aplicação	GAD1	CD	CS	PS	BoT	GAD2	Estático
1s	*DI*	1,020	1,087	1,080	1,079	1,104	**1,000**	1,398
	OUT	1,003	1,269	1,268	1,271	**1,000**	1,011	1,109
	IN	1,017	1,293	1,294	1,292	1,030	**1,000**	1,131
5s	*DI*	**1,000**	1,043	1,043	1,044	1,035	1,018	1,342
	OUT	1,006	1,337	1,335	1,335	**1,000**	1,022	1,103
	IN	**1,000**	1,256	1,251	1,254	1,014	1,0002	1,103

Com relação ao número de tarefas reescalonadas, principalmente as heurísticas GAD1 e GAD2 reescalonam um maior número de tarefas do que no cenário HET1. Isto ocorre, pois estas heurísticas permitem que tarefas ainda não executadas sejam reescalonadas mais de uma vez, tentando adaptar a execução da aplicação a cada mudança do ambiente computacional. Os valores apresentados na Tabela 8.7 contabilizam todas a vezes que uma mesma tarefa v foi reescalonada no cenário DIN.

Tabela 8.7: Média do número de tarefas reescalonadas no cenário DIN (4 HMs).

t_{exe}	aplicação	GAD1	CD	CS	PS	BoT	GAD2
1s	*DI*	50,0	165,8	179,7	183,2	33,7	183,3
	OUT	153,0	206,3	209,2	199,8	69,1	211,3
	IN	173,9	101,8	101,5	102,4	95,5	137,7
5s	*DI*	63,5	180,6	175,7	169,6	34,7	185,8
	OUT	154,7	216,8	202,6	199,7	68,7	286,4
	IN	225,3	94,7	99,2	95,7	95,9	237,8

Ao analisar o número de tarefas reescalonadas no cenário DIN, verifica-se que a política GAD1 possui maiores médias de reescalonamento de tarefas nos grafos do tipo *in-tree*. Os grafos *in-tree* utilizados se caracterizam por possuírem, logo de início, a metade de suas tarefas $v \in V$ prontas para executar, ou seja, com $level(v) = 0$. Como visto, a heurística GAD1 pode reescalonar tais tarefas enquanto elas não forem executadas. Como o ambiente de execução sofre uma alteração, logo após o primeiro reescalonamento, GAD1 praticamente reescalona todas estas tarefas em um segundo evento de escalonamento de tarefas prontas.

Nas próximas configurações adotadas para execução (12 e 24 HMs), não será discutido o número de tarefas reescalonadas em cada heurística, pois o mesmo comportamento analisado para a configuração com 4 HMs pode ser verificado. O que se conclui é que devido ao maior número de eventos de escalonamento que podem ser disparados pelas heurísticas GAD1 e GAD2, elas se tornam mais sensíveis às mudanças do ambiente. Entretanto, em ambientes que sofrem poucas variações, GAD1 não necessita reescalonar um grande número de tarefas, pois está baseada no escalonamento estático inicial.

A heurística BoT reescalona um número pequeno de tarefas, já que seu algoritmo não é preparado para tratar tarefas pendentes. O número reduzido de tarefas reescalonadas pode ser visto principalmente no caso de grafos diamante, onde devido a topologia do grafo que determina um grande número de dependências, existem poucas tarefas prontas a serem consideradas a cada evento de escalonamento, de maneira geral. Outro comportamento que pode ser destacado entre as heurísticas, é que quanto maior o número de tarefas em relação ao número de máquinas, mais tarefas pertencentes a um mesmo nível l se tornam disponíveis para reescalonamento em um recurso.

Configuração com 12 HMs

Nesta configuração virtual da grade, são utilizadas as 13 máquinas do site 2, como foi apresentado na Figura 8.5. A máquina sinergia é dedicada a execução dos gerenciadores global e local, e as restantes executam os HMs que são responsáveis por disparar as tarefas da aplicação do usuário. As mesmas aplicações utilizadas na configuração anterior são usadas: DI_{64}, DI_{256}, DI_{400}, OUT_{63}, OUT_{255}, OUT_{511}, IN_{63}, IN_{255} e IN_{511}.

Os resultados obtidos no cenário homogêneo (HOM) mostram que a heurística GAD1 obtém os melhores resultados, cujos valores são similares aos *makespans* associados à execução especificada pelo escalonador estático. No pior caso, a diferença entre GAD1 e a execução estática é de $0,3\%$, representando uma valor irrelevante que pode ser desconsiderado devido às flutuações da rede de interconexão e o intervalo de confiança adotado. A média das qualidades obtidas para cada heurística podem ser analisadas na Tabela 8.8.

Tabela 8.8: Qualidade das médias dos *makespans* obtidos no cenário HOM (12 HMs).

t_{exe}	aplicação	GAD1	CD	CS	PS	BoT	GAD2	Estático
1s	*DI*	1,003	1,026	1,090	1,095	1,003	1,012	**1,000**
	OUT	**1,000**	1,018	1,034	1,037	1,015	1,012	1,001
	IN	**1,000**	1,002	1,005	1,003	1,013	1,001	1,001
5s	*DI*	1,001	1,097	1,098	1,097	1,002	1,096	**1,000**
	OUT	1,001	1,035	1,026	1,025	**1,000**	1,028	**1,000**
	IN	**1,000**	1,027	1,022	1,023	1,012	1,026	**1,000**

As maiores diferenças entre as qualidades calculadas podem ser vistas para aplicações representadas por grafos diamante. Devido a topologia deste grafo ser mais complexa, as análises feitas pelo escalonador estático são de suma importância, principalmente em um cenário onde o poder computacional das máquinas não é alterado.

Assim como para a configuração anterior (4 HMs), os cenários de execução heterogêneos HET1 e HET2 obtiveram valores semelhantes, sendo reportados somente os resultados obtidos em HET1. As médias dos *makespans* alcançados na execução de cada heurística podem ser visualizadas nas Figuras 8.11 e 8.12, que são relativas às aplicações com tarefas de 1 segundo e 5 segundos, respectivamente.

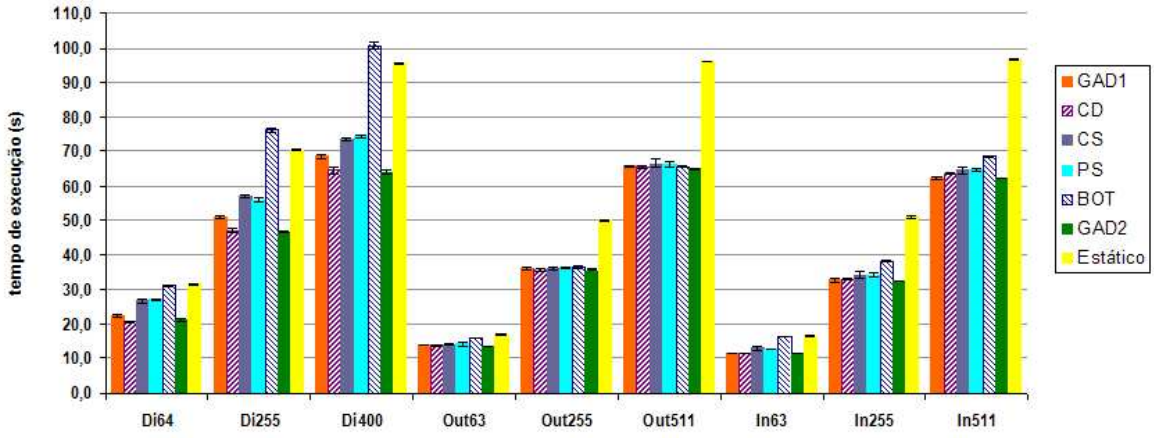

Figura 8.11: Execução em um ambiente heterogêneo (HET1) com 12 máquinas, onde $t_{exe} = 1s$. Intervalo de confiança de 95%.

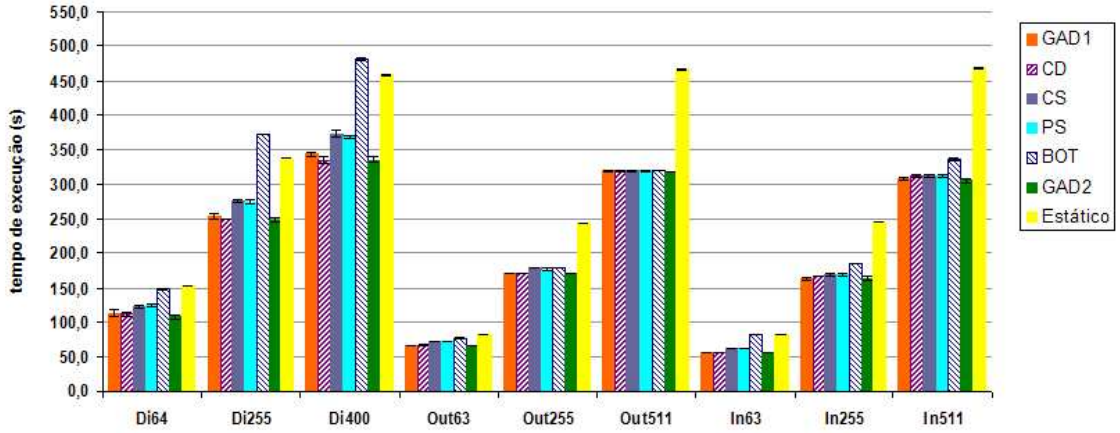

Figura 8.12: Execução em um ambiente heterogêneo (HET1) com 12 máquinas, onde $t_{exe} = 5s$. Intervalo de confiança de 95%.

Através da observação do desempenho das heurísticas CD, CS, e PS nos ambientes heterogêneos para 12 HMs, verifica-se que CD é a política que obtém os melhores resultados. Em [83], já haviam sido apresentadas simulações que mostravam que CD possuía desempe-

nho melhor do que CS e PS, pois adota prioridades de tarefas que se adequam melhor a um ambiente dinâmico (caminho-crítico escalonado, *cpath*, Definição 5, Seção 3.3, Capítulo 3). Entretanto, devido a ineficiência destas heurísticas em ambiente dinâmicos, a heurística GAD2 foi implementada nesta tese como uma variação da política CD. Como explicado anteriormente, além de GAD2 utilizar os mesmos critérios de escalonamento que CD, foi adicionado o mecanismo de reescalonamento de tarefas prontas explicado na Seção 7.2.3 do Capítulo 7.

Nos gráficos apresentados para o cenário HET1, GAD2 mostra os melhores resultados, seguida pelas heurísticas CD e GAD1. A Tabela 8.9 indica as médias das qualidades obtidas em cada heurística, por tipo de aplicação, onde GAD1 possui seu pior desempenho para grafos diamante. Estes grafos, devido a sua topologia, se caracterizam por possuírem um pequeno número de tarefas prontas aguardando por execução, principalmente pelo fato de uma maior número de máquinas estar sendo usado. Como explicado na configuração para 4 HMs, a estratégia de escalonamento de tarefas pendentes usada para GAD1 é mais conservadora quando comparada com as heurísticas CD e GAD2, o que contribui para que GAD1 não obtenha resultados tão bons para esta situação.

Tabela 8.9: Qualidade das médias dos *makespans* obtidos no cenário HET1 (12 HMs).

t_{exe}	aplicação	GAD1	CD	CS	PS	BoT	GAD2	Estático
1s	*DI*	1,076	1,002	1,191	1,193	1,575	**1,000**	1,496
	OUT	1,013	1,006	1,021	1,024	1,030	**1,000**	1,428
	IN	1,003	1,018	1,051	1,050	1,152	**1,000**	1,543
5s	*DI*	1,025	1,005	1,117	1,112	1,452	**1,000**	1,372
	OUT	1,003	1,004	1,020	1,019	1,033	**1,000**	1,410
	IN	1,004	1,018	1,033	1,032	1,143	**1,000**	1,512

Ainda de acordo com a Tabela 8.9, a estratégia BoT apresenta seus melhores resultados para grafos *out-tree*, assim como já explicado na configuração para 4 HMs. Entre as heurísticas desenvolvidas especificamente para aplicações GAD, CS e PS são as que atingem os piores *makespans*. Tal comportamento pode ser explicado pelo uso de prioridades mais simples, que não conseguem refletir o comportamento dinâmico do ambiente.

Comparando os *makespans* obtidos no cenário HET1 com os do cenário DIN, apresentados a seguir, observa-se que o desempenho da heurística GAD1 melhora em relação à CD, CS, e PS, pois como visto anteriormente estas políticas possuem algumas deficiências em ambientes dinâmicos. No cenário DIN para a configuração de 12 HMs, as *cargas extras* do sistema são inicialmente transferidas do primeiro conjunto de máquinas composto por sn00 a sn05, para o segundo conjunto composto por sn06 a sn11, e depois retornam ao primeiro conjunto. Os tempos de execução obtidos, para tarefas de 1 segundo e 5 segundos, podem ser visualizados nas Figuras 8.13 e 8.14, respectivamente.

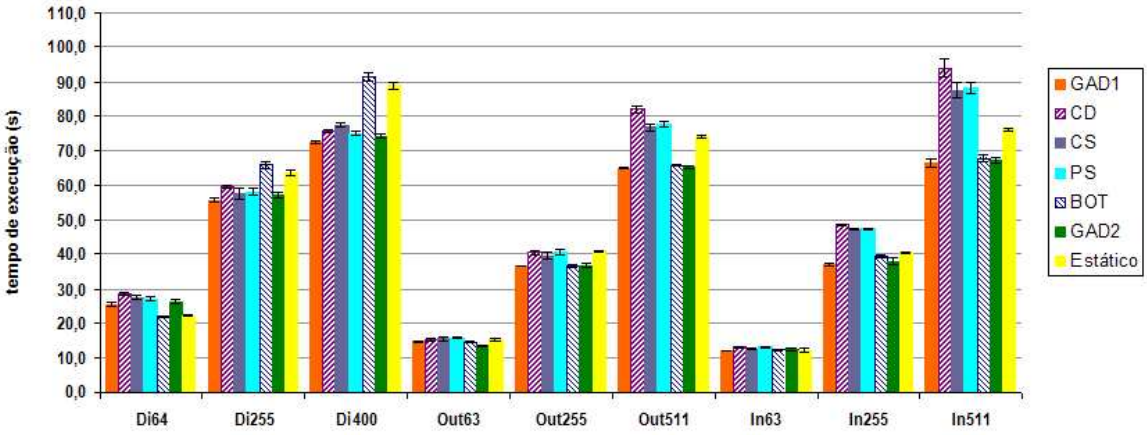

Figura 8.13: Execução em um ambiente dinâmico (DIN) com 12 máquinas, onde $t_{exe} = 1s$. Intervalo de confiança de 95%.

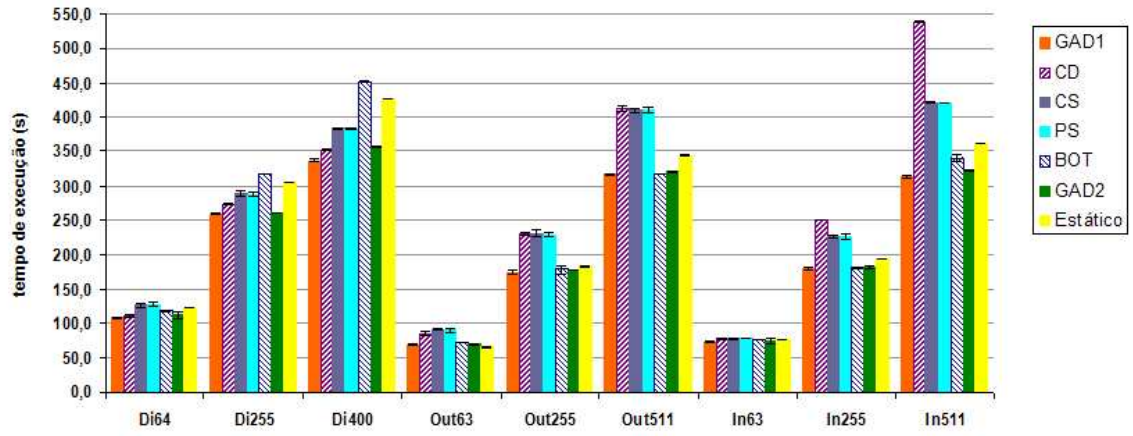

Figura 8.14: Execução em um ambiente dinâmico (DIN) com 12 máquinas, onde $t_{exe} = 5s$. Intervalo de confiança de 95%.

É interessante ressaltar que para os grafos com poucas tarefas, como DI_{64}, IN_{63} e OUT_{63}, a heurística BoT praticamente não reescalona tarefas, pois existem poucas tarefas prontas aguardando para execução, já que o número de tarefas em relação ao número de máquinas é menor. Como o sistema, apesar de possuir comportamento dinâmico, divide igualmente a *carga extra* de processamento entre os seus recursos, BoT produz bons resultados, assim como o escalonador estático. Entretanto, para grafos diamante com maior número de tarefas, este mesmo desempenho não pode ser observado. Sendo BoT uma heurística desenvolvida para tarefas *prontas* e DI um grafo com muitas relações de precedência, as decisões tomadas por BoT em um certo evento de escalonamento podem refletir em uma má escolha na execução de tarefas sucessoras.

Entre as heurísticas desenvolvidas para aplicações GAD, GAD1 e GAD2 alcançam os melhores resultados. Tal melhora pode ser percebida, principalmente para grafos com um

maior número de tarefas, onde as opções de reescalonamento se tornam maiores. Para grafos do tipo *out-tree* e *in-tree*, as heurísticas CD, CS e PS possuem um desempenho pior até mesmo que o escalonamento estático, pois apesar do escalonador dinâmico perceber as mudanças do ambiente, estas heurísticas de escalonamento não permitem que uma mesma tarefa seja reescalonada mais de uma vez.

Como já discutido, o escalonador estático se beneficia do fato da carga total do sistema não ser alterada. Para grafos *out-tree* e *in-tree*, o tipo de situação encontrada no cenário DIN não afeta muito a execução da aplicação, já que metade das tarefas das árvores usadas nestes experimentos são paralelas, pertencentes a um só nível da árvore. Assim, mesmo durante a execução da aplicação sem escalonamento dinâmico, podem não existir atrasos que prejudiquem a execução de tarefas sucessoras. Isto pode acontecer, pois o sistema volta ao seu estado inicial antes que metade das tarefas tenham sido executadas, este fato pode ser melhor explicado através das Figuras 8.15 e 8.16.

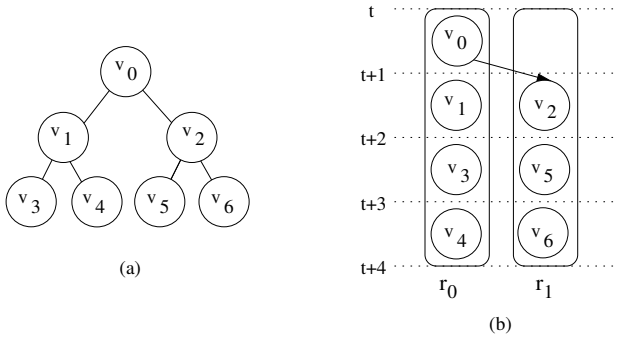

Figura 8.15: (a) Aplicação do tipo *out-tree* com 7 tarefas. (b) Alocação inicial de OUT_7.

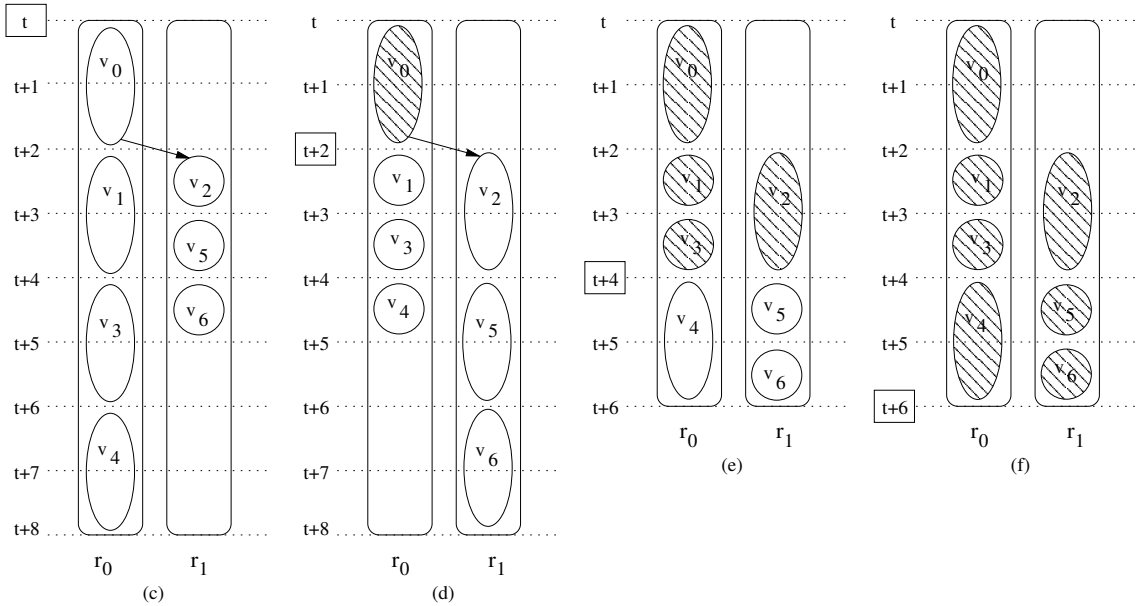

Figura 8.16: Execução estática de OUT_7 no cenário DIN.

Considere a aplicação do tipo *out-tree* apresentada na Figura 8.15(a), onde cada tarefa gasta 1 unidade de tempo (u.t.) para executar no recurso mais rápido. Na Figura 8.15(b) é proposta uma possível alocação inicial de suas tarefas entre dois recursos disponíveis, r_0 e r_1. A execução sem escalonamento dinâmico, das tarefas de OUT_7 no cenário DIN, é apresentada na Figura 8.16 a cada intervalo de 2 unidades de tempo. O início da execução é mostrado em (c), onde r_0 e r_1 oferecem 50% e 100% de poder computacional, respectivamente. Suponha que após 2 u.t. (em um tempo $t+2$), v_0 termina sua execução e as *cargas extras* existentes em r_0 migram para r_1 (Figura 8.16(d)). No próximo passo, mostrado em (e), v_1, v_2 e v_3 finalizam a execução no tempo $t+4$ e as *cargas extras* de r_1 são transferidas para r_0. No último passo (f), as últimas tarefas são executadas e o *makespan* da aplicação é de 6 u.t. É possível verificar que não existiram espaços ociosos durante a execução, logo tarefas não foram atrasadas, sendo os recursos usados da melhor maneira possível.

As médias das qualidades obtidas por tipo de aplicação no cenário DIN são mostradas na Tabela 8.10. A heurística GAD1 alcançou a melhor média de *makespans* para quase todos os casos, sendo pior 0,8% que GAD2 apenas para grafos *out-tree* com tarefas de 1 segundo. A política GAD2 possui o segundo melhor desempenho apresentando resultados que se distanciam no máximo 3,3% do melhor encontrado. Considerando o intervalo de confiança estabelecido, pode se dizer que o desempenho das heurísticas GAD1 e GAD2 é equivalente.

Tabela 8.10: Qualidade das médias dos *makespans* obtidos no cenário DIN (12 HMs).

t_{exe}	aplicação	GAD1	CD	CS	PS	BoT	GAD2	Estático
1s	*DI*	**1,000**	1,067	1,060	1,044	1,171	1,027	1,139
	OUT	1,008	1,194	1,142	1,164	1,015	**1,000**	1,128
	IN	**1,000**	1,352	1,279	1,291	1,036	1,020	1,115
5s	*DI*	**1,000**	1,043	1,135	1,131	1,259	1,033	1,213
	OUT	**1,000**	1,299	1,307	1,300	1,015	1,013	1,053
	IN	**1,000**	1,535	1,284	1,283	1,056	1,024	1,115

Configuração com 24 HMs

A configuração virtual das máquinas do SGCLab para 24 HMs é apresentada na Figura 8.5, onde a máquina sinergia executa os gerenciadores global (GM) e local (SM) e as demais executam os HMs e as tarefas da aplicação do usuário. Os resultados obtidos para as aplicações DI_{64}, DI_{256}, DI_{400}, OUT_{63}, OUT_{255}, OUT_{511}, IN_{63}, IN_{255} e IN_{511} foram similares aos da configuração com 12 HMs para os cenários HOM, HET1 e HET2, não sendo necessária a apresentação dos gráficos com os tempos de execução.

Na execução no cenário HOM, as heurísticas CS e PS também apresentam os piores resultados, principalmente para grafos do tipo diamante. Já, a heurística GAD1 alcança, para todos os grafos, a melhor média de qualidade de *makespans*, sendo no máximo 0,2% pior que a melhor qualidade obtida.

No cenário HET1, o desempenho das heurísticas também segue o mesmo padrão da execução em 12 máquinas, onde GAD2 alcança os melhores resultados, seguida por CD e GAD1. Uma diferença na execução com 24 HMs fica por conta do resultados obtidos pela política BoT para grafos *out-tree*. Neste caso, os resultados chegam a ser na média 15% piores dos que os *makespans* alcançados por GAD2, enquanto na configuração com 12 HMs esta piora ficava em torno de 3%. Isto ocorre, pois o número de máquinas dobra em relação a configuração anterior, existindo um número mais baixo de tarefas prontas para serem reescalonadas em um só recurso. Assim, a heurística BoT não tem tempo suficiente para ajustar a execução da aplicação *out-tree*.

No ambiente de execução dinâmico (DIN), a heurística GAD1 é a que possui, no geral, o melhor desempenho, seguida por GAD2 que em seu pior caso possui uma qualidade de resultados 3,1% pior que a melhor média de *makespans* alcançada. A média das qualidades é mostrada na Tabela 8.11, onde CD, CS e PS continuam mostrando um desempenho relativamente baixo em ambientes dinâmicos quando comparadas com GAD1 e GAD2. Mais uma vez, é importante destacar que os bons resultados alcançados por BoT, são devidos ao baixo número de reescalonamentos realizados, comportamento que é favorecido pelo cenário DIN, que durante a execução divide as *cargas extras* igualmente entre os recursos.

Tabela 8.11: Qualidade das médias dos *makespans* obtidos no cenário DIN (24 HMs).

t_{exe}	aplicação	GAD1	CD	CS	PS	BoT	GAD2	Estático
1s	*DI*	**1,000**	1,017	1,215	1,228	1,019	1,016	1,030
	OUT	1,037	1,168	1,224	1,193	1,076	**1,000**	1,212
	IN	**1,000**	1,182	1,189	1,170	1,063	1,008	1,030
5s	*DI*	**1,000**	1,097	1,231	1,237	1,154	1,019	1,171
	OUT	1,010	1,249	1,248	1,251	**1,000**	1,031	1,002
	IN	**1,000**	1,362	1,223	1,216	1,071	1,018	1,024

Na tentativa de avaliar o comportamento das heurísticas em um ambiente onde o poder computacional total do sistema sofre alterações durante a execução da aplicação do usuário, um novo cenário de execução (DIN2) foi utilizado nos testes com 24 HMs. De acordo com o *makespan* obtido no ambiente homogêneo (HOM), define-se dois momentos para se disparar *cargas extras* no sistema: t_1 e t_2. O tempo t_1 indica a metade do *makespan* obtido e t_2 indica 3/4. O cenário DIN2 é definido da seguinte forma:

- DIN2: Inicialmente, os 24 recursos oferecem 100% do seu poder computacional. No tempo t_1 é disparada uma *carga extra* em cada uma das 12 máquinas: sn02, sn04, sn06, sn08, sn10, sn12, sn14, sn16, sn18, sn20, sn23 e sn25, que passam a oferecer 50% de poder computacional. Em t_2, 6 destas máquinas recebem mais uma *carga extra* cada, disponibilizando para a aplicação do usuário apenas 33% do seu poder computacional.

Os *makespans* obtidos no cenário DIN2, por tipo de aplicação, são apresentados nos gráficos das Figuras 8.17 e 8.18, para tarefas de 1 segundo e 5 segundos, respectivamente.

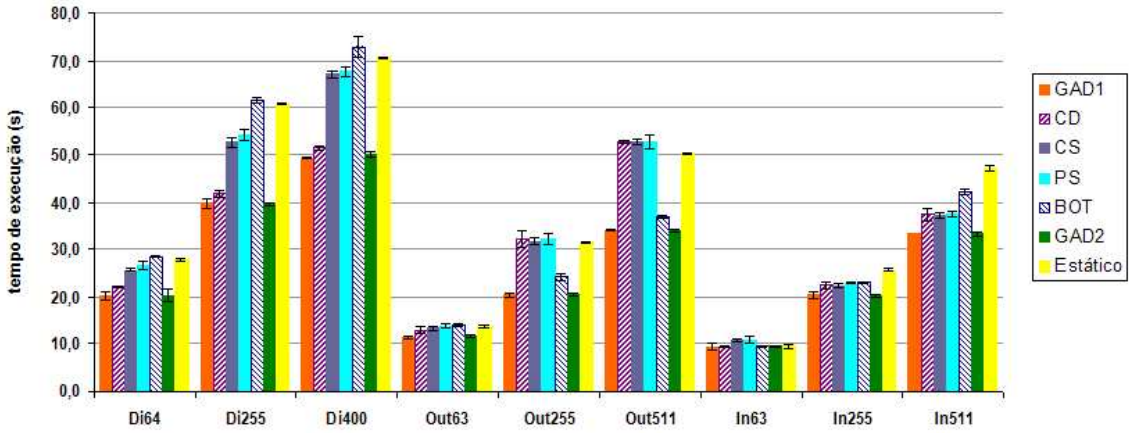

Figura 8.17: Execução em um ambiente dinâmico (DIN2) com 24 máquinas, onde $t_{exe} = 1s$. Intervalo de confiança de 95%.

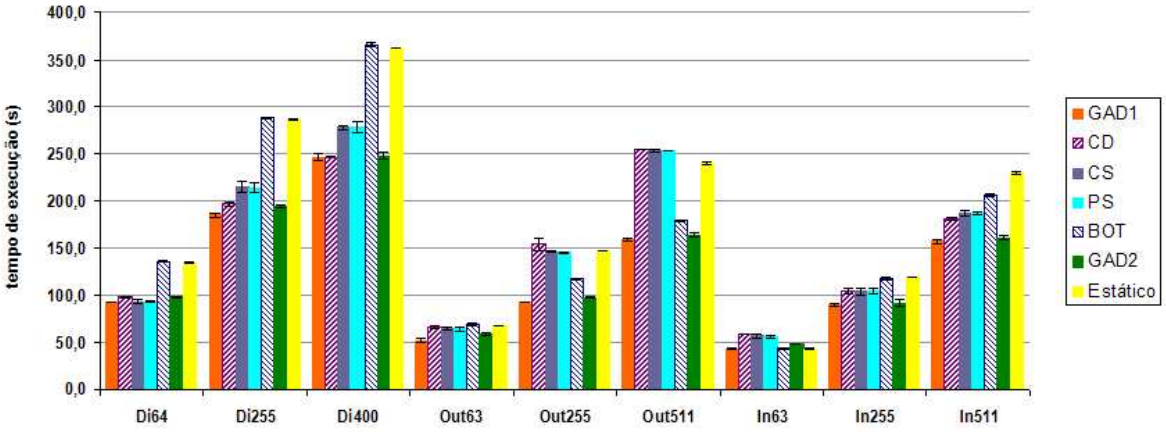

Figura 8.18: Execução em um ambiente dinâmico (DIN2) com 24 máquinas, onde $t_{exe} = 5s$. Intervalo de confiança de 95%.

No cenário DIN2, verifica-se uma piora da execução estática e das heurísticas CD, CS, PS e BoT. Para aplicações cujas tarefas possuem tempo de execução de 1 segundo, as heurísticas testadas apresentam tempos aproximadamente iguais para as aplicações OUT_{63} e

IN_{63}. Isto acontece, pois o tempo de execução da aplicação é muito pequeno (11 segundos) dificultando eventos de escalonamento, já que as tarefas logo são disparadas para execução.

Para os outros grafos, é possível observar que as heurísticas que melhor se adaptam ao ambiente dinâmico são GAD1 e GAD2. Apesar de BoT também ser uma heurística sensível às mudanças do ambiente, suas decisões não tratam a topologia do grafo, e são baseadas apenas no estado dos recursos e na alocação das tarefas prontas. Logo, além de projetar uma heurística que possa se adaptar da melhor maneira possível às variações que ocorram em uma grade, é importante que esta seja ciente das características de uma aplicação GAD.

As médias das qualidades obtidas em cada heurística, por tipo de aplicação, são mostradas na Tabela 8.12, onde a heurística GAD1 obtém os melhores resultados. GAD2 também produz bons *makespans* alcançando uma média de qualidade que no pior caso se distancia apenas 5,6% de GAD1, apresentando desempenho equivalente.

Tabela 8.12: Qualidade das médias dos *makespans* obtidos no cenário DIN2 (24 HMs).

t_{exe}	aplicação	GAD1	CD	CS	PS	BoT	GAD2	Estático
1s	*DI*	**1,000**	1,056	1,329	1,359	1,490	1,005	1,457
	OUT	**1,000**	1,480	1,481	1,496	1,144	1,003	1,445
	IN	**1,000**	1,100	1,115	1,134	1,191	1,001	1,308
5s	*DI*	**1,000**	1,035	1,120	1,118	1,505	1,032	1,495
	OUT	**1,000**	1,565	1,531	1,526	1,211	1,056	1,505
	IN	**1,000**	1,193	1,205	1,205	1,273	1,040	1,360

Para verificar a eficiência da estrutura hierárquica de escalonamento do *EasyGrid AMS* para aplicações GAD, também foram consideradas aplicações com um maior número de tarefas. Assim, nesta configuração com 24 HMs foram executadas as aplicações DI_{4096}, OUT_{4095} e IN_{4095}. As aplicações aqui utilizadas não possuem tantas tarefas como nas avaliações realizadas para a heurística BoT, apresentadas no Capítulo 6. O objetivo é que o número de tarefas por máquina não seja tão grande de forma que as precedências existentes no grafo deixem de ser importantes nas decisões da heurística de escalonamento dinâmico.

Os testes consideraram os cenários HOM, HET1, HET2, DIN1 e DIN2. Como os resultados obtidos para os cenários HET1 e HET2 são semelhantes, serão discutidos apenas as execuções realizadas em HET1. No ambiente homogêneo, todas as heurísticas obtiveram bons desempenhos. Devido ao grande número de tarefas da aplicação por recurso, é mais fácil que a heurística adapte a execução da aplicação ao ambiente, ainda mais neste caso onde não existem variações no poder computacional dos recursos.

Os gráficos apresentados na Figura 8.19 mostram os *makespans* obtidos no cenário HET1. Como explicado anteriormente, devido à aplicação possuir um maior número de tarefas, todas as heurísticas projetadas para aplicações GAD alcançam um bom desempenho. Isto é possível, pois como as aplicações têm um tempo de execução longo, as heurísticas possuem tempo de se adaptar às características do ambiente heterogêneo que são detectadas logo no início da execução.

Por outro lado, a política BoT não obtém bons resultados para grafos diamante, já que o número reduzido de tarefas prontas, a cada evento de escalonamento, prejudica o desempenho da heurística que não lida com tarefas pendentes. Mesmo com pouca diferença, GAD2 é a política que apresenta na média os melhores *makespans* no cenário HET1.

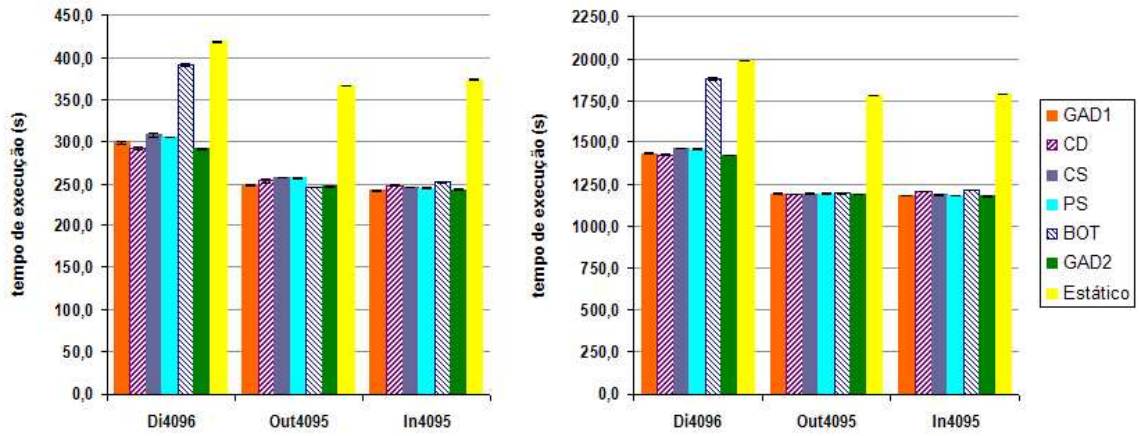

Figura 8.19: Execução em um ambiente heterogêneo (HET1) com 24 máquinas. No gráfico da esquerda $t_{exe} = 1s$ e no da direita $t_{exe} = 5s$. Intervalo de confiança de 95%.

Como as grades computacionais são ambientes de comportamento dinâmico, as avaliações realizadas nos cenários DIN e DIN2 são ainda mais interessantes no escopo deste trabalho. Os resultados obtidos nestes cenários são apresentados nas Figuras 8.20 e 8.21, respectivamente.

Considerando os experimentos realizados para árvores do tipo *out-tree* e *in-tree*, observa-se que as heurísticas CD, CS e PS obtêm os piores resultados. Assim como nos outros testes já apresentados neste capítulo, estas heurísticas não se adaptam bem a ambientes dinâmicos, já que uma dada tarefa só pode ser reescalonada uma única vez. A heurística BoT obtém bons *makespans* devido às características das árvores usadas, onde existe um grande número de tarefas prontas a serem reescalonadas em cada recurso. No geral, para as aplicações representadas por *out-trees* e *in-trees*, as heurísticas GAD1, BoT e GAD2 (nesta ordem) alcançam os melhores resultados.

Na execução das aplicações representadas por grafos diamante, a heurística GAD2 obtém os melhores resultados, seguida por GAD1. Neste caso, como já explicado, a heurística BoT produz escalonamentos que levam a um pior desempenho. Em relação às diferenças

que podem ser visualizadas entre os dois cenários de execução, observa-se que quanto mais dinâmico for o ambiente, pior será o desempenho de políticas que restringem o número de escalonamentos que podem ser realizados, como CD, CS e PS. Isto ocorre, pois um escalonamento proposto em um dado momento da execução pode se tornar ruim após algumas variações do ambiente.

Para a execução estática, o cenário DIN apresenta melhores condições do que DIN2, já que no cenário DIN2 as *cargas extras* de processamento não são divididas igualmente entre os recursos. Logo, devido às mudanças no poder de processamento total do site, os *makespans* alcançados com a execução estática no cenário DIN2 pioram em comparação às heurísticas dinâmicas.

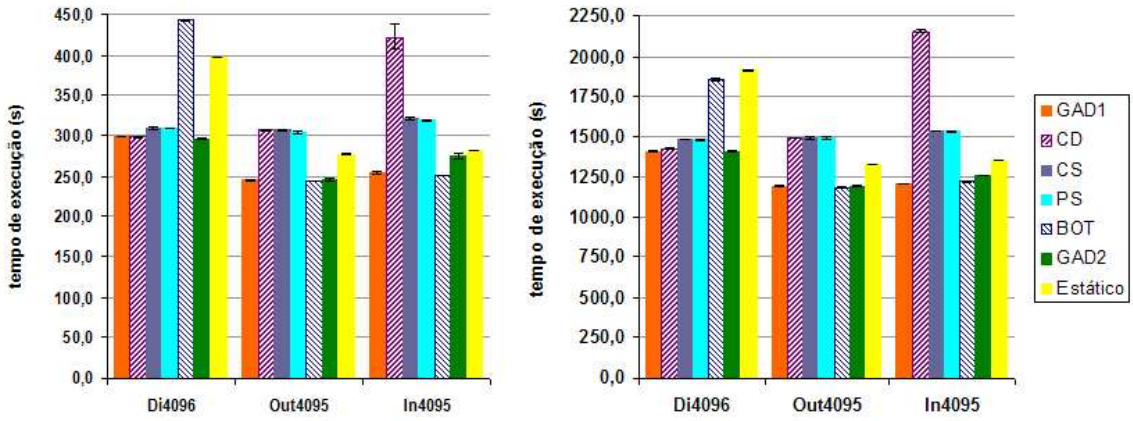

Figura 8.20: Execução em um ambiente dinâmico (DIN) com 24 máquinas. No gráfico da esquerda $t_{exe} = 1s$ e no da direita $t_{exe} = 5s$. Intervalo de confiança de 95%.

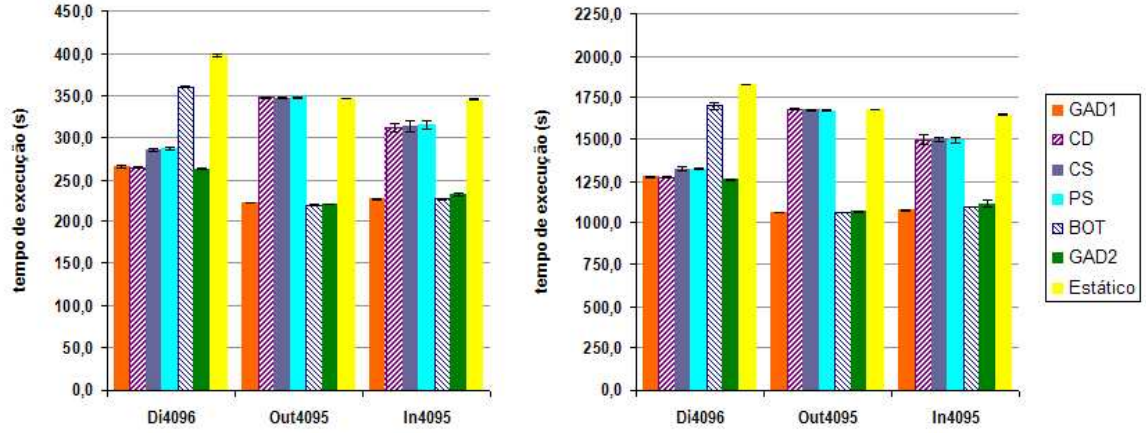

Figura 8.21: Execução em um ambiente dinâmico (DIN2) com 24 máquinas. No gráfico da esquerda $t_{exe} = 1s$ e no da direita $t_{exe} = 5s$. Intervalo de confiança de 95%.

De uma maneira geral, em todos os testes realizados nesta subseção, as heurísticas GAD1 e GAD2 obtiveram na média os melhores resultados. Em ambientes heterogêneos e dinâmicos a melhoria alcançada por estas heurísticas é incontestável, devido ao fato de uma

mesma tarefa da aplicação poder ser reescalonada mais de uma vez, sempre que necessário. Nos ambientes heterogêneos estáticos, estas heurísticas continuam obtendo bons resultados, principalmente para grafos do tipo *in-tree* e *out-tree*. Já, para grafos diamante a heurística CD também obtém um desempenho próximo da heurística GAD2.

Outro ponto importante que foi observado, é que quanto maior o número de tarefas do grafo em relação ao número de máquinas, mais próximos são os resultados encontrados pelas heurísticas estudadas. Logo, se o número de tarefas paralelas de uma aplicação é muito grande e os tempos de comunicação são pequenos, as relações de precedência entre as tarefas diminuem sua influencia no *makespan* produzido. Nestes casos, a ordenação entre as tarefas passa a ser mais flexível. Para este tipo de situação, observa-se o bom desempenho da heurística BoT, que devido ao seu baixo custo e simplicidade pode ser uma boa opção para aplicações onde existam muitas tarefas paralelas por número de máquinas disponíveis.

Com relação aos ambientes homogêneos, as heurísticas alcançam tempos de execução que se aproximam dos *makespans* produzidos pelo escalonamento estático. Entretanto, GAD1 reescalona o menor número de tarefas, já que o escalonamento estático produzido inicialmente é levado em consideração. Logo, nas avaliações realizadas até agora, as heurísticas GAD1 e GAD2 apresentam os melhores resultados. Porém, para ambientes dinâmicos como as grades computacionais, a heurística GAD1 mostra os melhores *makespans* na maioria dos testes realizados.

É importante ressaltar que GAD1 e GAD2 se diferem principalmente no escalonamento de tarefas *pendentes*. GAD1 faz uma avaliação de todo o site, procurando reescalonar várias tarefas *pendentes* de um bloco B_l na tentativa de minimizar o *makespan* do site. Entretanto, o evento de reescalonamento só é realmente disparado se o *makespan* do site for diminuído. Já GAD2, após selecionar o bloco de tarefas B_l, tenta reescalonar todas as tarefas $v \in B_l$, desde que o tempo de fim de v seja minimizado, não importando o *makespan* final produzido.

8.4.2 Intrusão da heurística de escalonamento local

Nos resultados apresentados até aqui, não foram feitas análises específicas quanto a intrusão das heurísticas de escalonamento local, já que os processos gerenciadores GM e SM executavam em uma máquina dedicada, sem dividir processamento de CPU com os processos do usuário. Logo, para que se possa verificar a sobrecarga causada pela heurística de escalonamento local, alguns experimentos foram realizados, onde os gerenciadores local (SM) e global (GM) são alocados juntamente com o HM da máquina sn00, ao invés de executarem separadamente na máquina sinergia. Assim, é possível observar o impacto destes escalonadores na execução da aplicação, já que as tarefas do usuário, o GM, o SM e o HM irão competir pela CPU da máquina sn00.

Os resultados obtidos no cenário DIN foram realizados para a configuração com 12 HMs e são mostrados nas Tabelas 8.13 e 8.14. Os resultados da Tabela 8.13 são referentes aos gráficos apresentados nas Figuras 8.13 e 8.14, onde o GM e SM executam em uma máquina separada. A Tabela 8.14 apresenta os resultados da execução, considerando que o GM e SM executam junto com a aplicação do usuário na máquina sn00.

Tabela 8.13: Médias dos *makespans* obtidos no cenário DIN (12 HMs): GM e SM executam sozinhos na máquina sinergia.

t_{exe}	aplicação	GAD1	CD	CS	PS	BoT	GAD2
1s	*DI*	**51,18**	54,60	54,26	53,42	59,94	52,57
	OUT	38,49	45,24	43,83	44,67	38,97	**38,38**
	IN	**38,36**	51,84	49,05	49,51	39,73	39,12
5s	*DI*	**234,98**	245,15	266,64	265,78	295,82	242,83
	OUT	**186,67**	243,18	243,95	243,69	189,66	189,09
	IN	**188,10**	288,71	241,49	241,26	198,70	192,61

Tabela 8.14: Médias dos *makespans* obtidos no cenário DIN (12 HMs): GM e SM executam juntamente com o HM e as tarefas do usuário na máquina sn00.

t_{exe}	aplicação	GAD1	CD	CS	PS	BoT	GAD2
1s	*DI*	**51,53**	54,50	53,79	55,31	60,78	52,78
	OUT	**38,85**	45,71	45,51	44,89	39,71	38,87
	IN	**38,55**	51,47	49,63	49,10	40,31	39,49
5s	*DI*	**237,88**	245,83	264,96	263,81	296,84	245,41
	OUT	**191,50**	248,50	248,79	249,11	192,47	192,20
	IN	**189,47**	287,16	238,70	237,78	200,92	194,10

Ao observar as médias apresentadas nas tabelas acima, verifica-se que todas as heurísticas testadas, implementadas no *EasyGrid AMS*, possuem baixa intrusão. Em muitos casos, é possível observar que não houve piora da média dos *makespans* obtidos no cenário apresentado na Tabela 8.14. As variações encontradas são maiores para as aplicações cujas tarefas possuem $t_{exe} = 5s$, nestes casos, como a tarefa do usuário possui um tempo de execução maior, ocorre um maior número de trocas de contexto entre os 4 processos alocados na máquina sn00: GM, SM, HM e o processo do usuário. Assim, quanto mais tempo o SDS ocupar a CPU para verificação ou realização de reescalonamentos, mais tempo o processo do usuário vai demorar para terminar sua execução. No geral, a intrusão máxima observada é relativa ao t_{exe} de uma tarefa.

Para todas as heurísticas testadas, considerando aplicações cujas tarefas possuem $t_{exe} = 5s$, GAD1 e GAD2 mostram o maior percentual médio de intrusão. Isto acontece,

devido a política de escalonamento usada, onde o SDS pode ser ativado em um maior número de situações. Entretanto, esta intrusão é compensada pelos melhores *makespans* obtidos, principalmente por GAD1 que atinge as melhores médias em quase todos os casos observados. Para $t_{exe} = 5s$, o percentual médio de intrusão de cada heurística foi: GAD1 = 1,49%, CD = 0,57%, CS = 0,05%, PS = 0,13%, BoT = 0,91% e GAD2 = 1,15%. Através destes valores, observa-se que quanto maior a complexidade da heurística e a quantidade de vezes que ela pode ser ativada, maior será o percentual de intrusão. Considerando o intervalo de confiança de 95%, pode-se dizer que as sobrecargas introduzidas no sistema pelas diferentes heurísticas é equivalente.

O grau de intrusão medido se refere principalmente a computação realizada pelos processos gerenciadores associados aos escalonadores dinâmicos locais. Como os testes foram realizados em uma rede local, não foram feitas avaliações em relação ao tempo gasto com o redirecionamento de mensagens das tarefas reescalonadas.

Nos próximos experimentos, a política GAD1 é escolhida por ter obtido os melhores resultados, principalmente em ambientes heterogêneos e dinâmicos que caracterizam as grades computacionais. Logo, os testes realizados para a aplicação real, *N-corpos*, e para a avaliação do escalonamento global utilizam esta heurística de escalonamento local que foi proposta no Capítulo 7.

8.4.3 Avaliação da política de escalonamento local proposta

Nesta subseção será avaliado o desempenho da heurística de escalonamento GAD1 através da execução da aplicação de *N-corpos*, utilizando o algoritmo paralelo para cálculo do somatório direto entre as partículas, denotado por *ring* [63]. O objetivo é verificar a habilidade da política proposta em adaptar a execução de uma aplicação ao ambiente disponível. A comparação de desempenho será feita entre o algoritmo *ring* implementado no *EasyGrid AMS*, chamado de AMS-ring, que segue um modelo de execução alternativo para aplicações MPI [104] e a implementação tradicional do MPI, chamada de MPI-ring, que representa um dos algoritmos mais eficientes para ambientes dedicados. O modelo do grafo de aplicação para as duas implementações é apresentado na Figura 8.22.

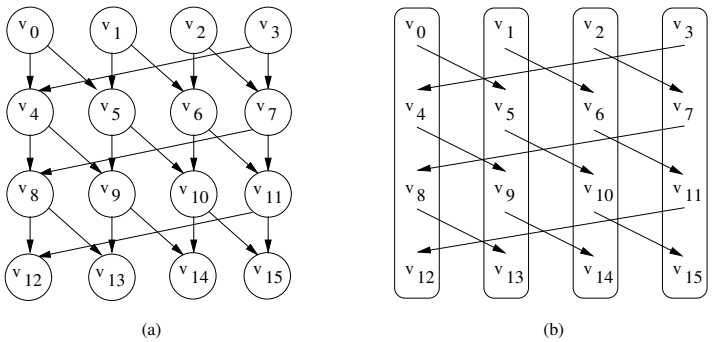

Figura 8.22: (a) Modelo de aplicação para AMS-ring. (b) Modelo de aplicação para MPI-ring.

Na modelagem apresentada para AMS-ring (a), cada tarefa v pertencente ao nível topológico l do grafo computa a força entre suas partículas e envia os seus dados para duas tarefas do nível $l+1$, como mostrado na figura. Cada nível representa uma iteração do problema e todo o grafo representa o cálculo para um *passo* do problema. Na implementação MPI-ring (b), as tarefas de uma coluna do grafo AMS-ring representam uma única tarefa, ou seja, um único processo. Assim, dentro de cada processo, não existe o conceito de tarefas, mas apenas de iterações do problema. Cada processo calcula as forças gravitacionais entre as partículas de uma iteração e envia os seus dados para o processo vizinho. Inicialmente, cada tarefa v da implementação AMS-ring recebe um total de N/W partículas, onde W representa a largura do grafo. Na implementação MPI-ring, cada iteração v de um processo recebe N/P partículas, onde P é o número de recursos usados.

O experimento é realizado utilizando 24 máquinas para a execução do problema de *N-corpos*, onde é utilizada a configuração virtual apresentada na Figura 8.5. A instância executada para o problema possui 200 mil partículas, e o objetivo é calcular a interação entre todas as partículas a cada *passo* do problema. Após cada *passo* t, é possível verificar a evolução do comportamento das partículas em um determinado espaço de tempo. A Figura 8.23 mostra um exemplo da execução de 2 *passos*, onde existe uma tarefa v_{dest}, que recebe todos os dados relativos à execução de cada *passo*.

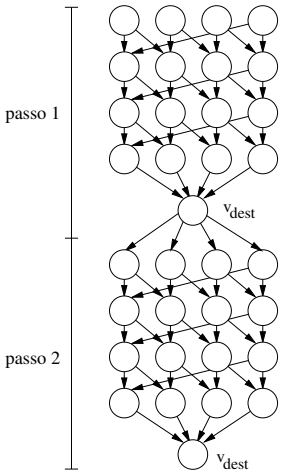

Figura 8.23: Exemplo da execução do problema de *N-corpos* com dois *passos*

A implementação AMS-ring foi executada com um o valor de W igual a 48 [103], o que para um *passo* de execução representa um grafo da aplicação com 2304 tarefas (W^2) mais a tarefa v_{dest}. Os testes foram realizados considerando 1, 2, 3, 4 e 5 *passos*. Logo, quando são executados t *passos*, o número de tarefas do grafo é $2305 \times t$. A maior aplicação utilizada, na execução com 5 *passos*, possui um total de 11525 tarefas. Os resultados obtidos são apresentados na Tabela 8.15 e na Figura 8.24.

Para definir um cenário de execução heterogêneo, considera-se a existência de uma *carga extra* de processamento que executa em alguma das máquinas disponíveis do SGCLab,

que passa a oferecer apenas 50% do seu poder computacional para a aplicação *N-corpos*. Para caracterizar o ambiente dinâmico, durante a execução da aplicação, esta *carga extra* é transferida aleatoriamente entre as máquinas de acordo com o tempo de execução total T, medido no cenário heterogêneo. Logo, a primeira transferência é feita depois que a aplicação completar aproximadamente $1/3$ do tempo T e novamente após $2/3$ deste tempo. Os resultados da terceira coluna da Tabela 8.15 mostram o percentual de melhora da execução de AMS-ring em relação a MPI-ring.

Tabela 8.15: Desempenho das implementações MPI-ring e AMS-ring, onde $N = 200.000$ partículas.

passos	MPI-ring	AMS-ring	melhora (%)
1	314,1	224,1	28,65
2	712,6	440,5	38,18
3	1136,3	659,5	41,96
4	1555,3	890,5	42,75
5	1935,6	1138,0	41,20

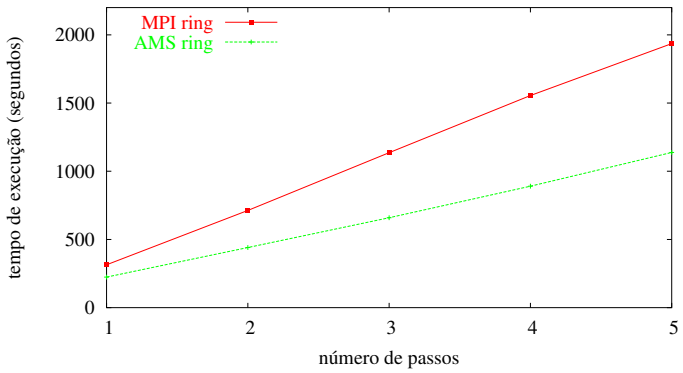

Figura 8.24: Desempenho das implementações MPI-ring e AMS-ring, onde $N = 200.000$ partículas.

Através dos resultados apresentados na Tabela 8.15 e na Figura 8.24, verifica-se que a implementação tradicional MPI-ring não se adapta às mudanças que possam ocorrer no ambiente computacional, mesmo no caso de existir apenas uma *carga extra* no sistema. Como os processos são criados estaticamente entre os recursos oferecidos, se um recurso r passa a oferecer apenas 50% do seu poder computacional, o processo alocado em r terá seu tempo de execução dobrado até que r volte a ser dedicado a execução da aplicação. Por outro lado, a implementação AMS-ring, conta com a heurística de escalonamento dinâmico GAD1 que pode reescalonar processos ainda não criados, levando em consideração o poder computacional oferecido por todos os recursos. Vale a pena relatar que ao executar em um *cluster* homogêneo que é ideal para implementações MPI tradicionais, a execução de AMS-ring alcança um desempenho similar a MPI-ring, mostrando a baixa sobrecarga do *EasyGrid AMS*.

Dado que cada tarefa da aplicação *N-corpos* utilizada neste experimento possui aproximadamente tempo de execução igual a 2,2 segundos na máquina ociosa mais rápida, é possível calcular um limite inferior teórico do *makespan* da aplicação considerando 24 recursos do SGCLab. Para isso, considera-se 23 máquinas com 100% de poder computacional e apenas uma oferecendo 50%. Desta forma, pode-se tentar avaliar o desempenho da política de escalonamento dinâmico em relação a um limite inferior mínimo, denotado por *lbound*. É importante ressaltar, que o valor de *lbound* não considera custos associados ao *EasyGrid AMS*, ao sistema operacional, nem as mensagens enviadas entre as tarefas da aplicação *N-corpos*, o que torna este valor, a princípio, impossível de ser alcançado.

A Tabela 8.16 mostra o tempo de execução obtido pelo AMS-ring em relação ao valor *lbound*. O valor calculado para o limite inferior não computa a existência da tarefa v_{dest}, porém, entre cada *passo* da execução, v_{dest} deve receber a informação de todas as tarefas pertencentes ao último nível do *passo* corrente e repassá-las para as tarefas do primeiro nível do *passo* seguinte. Logo, a comunicação entre 48 tarefas para um único processo destino causa um custo extra que não está sendo contabilizado no valor *lbound*. Por exemplo, na execução de 5 *passos* esta comunicação aumenta, ocorrendo cinco vezes, sempre ao final da execução de cada *passo*. A sobrecarga observada nestes testes é menor do que 5%, o que para um gerenciador de aplicações que trata características de aplicações GAD é considerada bem pequena.

Tabela 8.16: Desempenho do AMS-ring.

passos	*lbound*	AMS-ring	sobrecarga (%)
1	217,8	224,1	2,89
2	435,6	440,5	1,14
3	653,4	659,5	0,94
4	871,2	890,5	2,21
5	1089,0	1138,0	4,51

É importante ressaltar, que diferentemente do escalonamento de tarefas para aplicações BoT, o escalonamento dinâmico definido para aplicações GAD possui uma visão parcial do grafo de tarefas, o que permite que em alguns momentos o escalonador dinâmico possua uma visão restrita em relação a alocação de tarefas corrente. Um exemplo comum é quando o escalonador dinâmico detecta que um recurso r_i possui um maior número de tarefas prontas do que um outro recurso r_j. Neste caso, o escalonador dinâmico opta por disparar um evento de escalonamento de tarefas *prontas* para balancear a carga de trabalho dos recursos. Porém, pode ocorrer que logo em seguida alguma tarefa termine sua execução em r_j e novas tarefas passem para o estado de prontas, sobrecarregando a lista de prontos de r_j. Neste tipo de situação, o escalonador dinâmico deverá disparar um novo evento de

escalonamento para corrigir a decisão anterior. Tais características contribuem para que o problema de escalonamento de tarefas com relações de precedência se torne ainda mais difícil, principalmente em ambientes dinâmicos como as grades.

8.4.4 Avaliação da política de escalonamento global proposta

Assim como na avaliação realizada para as heurísticas locais, os experimentos apresentados nesta subseção utilizam apenas as máquinas do SGCLab. Como os testes são executados em ambientes reais, é necessário definir um ambiente distribuído e controlado, onde seja possível analisar o desempenho da heurística de escalonamento global, através do *makespan* obtido de acordo com o poder computacional disponível. Assim, os recursos são dedicados somente aos testes realizados, e como visto anteriormente é possível configurar um ambiente heterogêneo e dinâmico disparando *cargas extras* de processamento nos recursos da grade conforme os cenários de execução estabelecidos.

A configuração virtual do *EasyGrid AMS* é visualizada na Figura 8.25, onde são definidos dois sites virtuais distintos. O primeiro deles, site A, é composto pelas 13 máquinas disponíveis no site 2 da grade computacional do SGCLab, enquanto o site B é composto por 12 máquinas dos sites 1 e 3. O gerenciador global é disparado na máquina sinergia, e os gerenciadores de cada site são disparados nas máquinas sn00 e sn12. As máquinas restantes são usadas para executar os processos da aplicação do usuário, possuindo um HM cada. Logo, das 25 máquinas disponíveis, 22 são dedicadas a execução da aplicação, sendo 11 em cada site virtual.

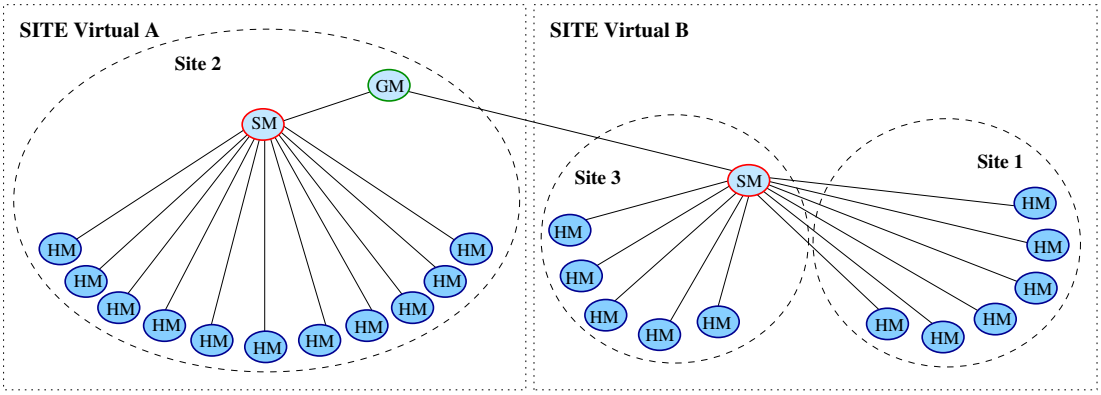

Figura 8.25: Configuração virtual da grade para a avaliação da heurística de escalonamento global.

Nesta subseção, são executadas as aplicações DI_{400}, DI_{4096}, OUT_{511}, OUT_{4095}, IN_{511} e IN_{4095}, com tempo uniforme de execução e de comunicação entre as tarefas. As mensagens possuem tamanho de *128 bytes* e todas as tarefas da aplicação possuem o mesmo tempo de execução (t_{exe}), sendo utilizadas aplicações sintéticas cujas tarefas possuem 1 segundo na máquina mais rápida da grade e aplicações com tarefas de 5 segundos. Além

disso, também é executada a aplicação real *N-corpos* utilizando o algoritmo AMS-ring. Como apresentado na Subseção 8.4.3, a instância executada possui 200 mil partículas, com o valor de W igual a 48, o que representa um grafo da aplicação com 2304 tarefas, onde cada tarefa possui aproximadamente $t_{exe} = 2,2s$.

No primeiro conjunto de testes foi avaliada a eficiência do escalonamento híbrido em um cenário heterogêneo (HET_{global}), onde os recursos do site A oferecem 50% do seu poder computacional, enquanto o site B é dedicado a execução da aplicação. As aplicações são executadas através do *EasyGrid AMS* considerando três abordagens de escalonamento distintas:

(1) Aplica uma abordagem estática onde não há ativação do escalonamento dinâmico. O escalonamento estático inicial definido pelo *HEFT* considera que o Site A oferece metade do poder computacional do Site B.

(2) Além de considerar o escalonamento estático inicial definido em (1), o escalonamento dinâmico é ativado durante a execução da aplicação.

(3) Utiliza escalonamento dinâmico, porém a alocação estática inicial definida pelo *HEFT* é imprecisa, pois considera que os dois sites disponíveis oferecem o mesmo poder computacional.

Os tempos de execução obtidos com a abordagem estática (1) definem um *valor base* para o *makespan* de acordo coma política *HEFT*, já que não existem custos associados a nenhuma política de escalonamento dinâmico e a alocação inicial estabelecida já possui conhecimento do poder computacional que cada recurso irá oferecer durante toda a execução da aplicação. Estes experimentos visam avaliar não somente o desempenho da política de escalonamento global em relação a um *valor base*, mas também verificar até que ponto um escalonamento estático preciso pode ser proveitoso em ambientes de grades computacionais.

Os *makespans* obtidos no cenário de execução HET_{global}, para cada tipo de aplicação em relação a abordagem de escalonamento definida, podem ser visualizados nas Tabelas 8.17 e 8.19. As duas últimas colunas das tabelas indicam, respectivamente, o quanto a abordagem (2) se distancia do *valor base* definido pela abordagem estática (1), e o quanto a abordagem (3) é pior do que a abordagem (2). Como definido anteriormente, a única diferença entre as abordagens (2) e (3) é com relação ao escalonamento estático inicial. O *valor base* estabelecido em (1) representa um tempo real de execução que pode ser alcançado, já que a execução estática foi realizada através do *EasyGrid AMS* e, portanto, os *makespans* obtidos incorporam a sobrecarga relativa ao gerenciamento da aplicação.

Os resultados apresentados mostram que a abordagem (2) produz bons resultados, para aplicações representadas por árvores binárias. A piora em relação à abordagem estática é de no máximo 5%, valor que diminui à medida que o número de tarefas da aplicação

aumenta. Como já explicado no escalonamento para aplicações BoT, quanto menor o custo de uma tarefa mais eficiente pode ser o escalonamento produzido, visto que as tarefas são unidades de computação indivisíveis. Logo, um melhor desempenho do escalonador dinâmico pode ser esperado para aplicações com $t_{exe} = 1s$. A abordagem (3) produz *makespans* que são até 20% piores do que os obtidos com a abordagem (2), o que mostra para este cenário, a importância da alocação inicial definida pelo escalonador estático. Os grafos com menor número de tarefas produzem os piores resultados, visto que o escalonador dinâmico possui menos opções de tarefas para reescalonar em cada passo. Além disso, a abordagem (3) é menos eficiente para grafos do tipo *in-tree*, onde várias tarefas prontas são disparadas para execução assim que aplicação inicia, o que impede que o escalonador dinâmico tenha chance de reescaloná-las para o site menos sobrecarregado.

Tabela 8.17: Execução no ambiente HET_{global} para aplicações representadas por árvores binárias

t_{exe}	aplicação	(1)	(2)	(3)	% piora (2)/(1)	% piora (3)/(2)
$1s$	OUT_{511}	37,12	37,59	43,02	1,28%	14,44%
	IN_{511}	37,26	37,53	44,55	0,72%	19,24%
	OUT_{4095}	266,81	267,26	285,63	0,17%	6,87%
	IN_{4095}	269,23	266,79	297,77	-0,91%	11,61%
$5s$	OUT_{511}	178,64	183,57	206,93	4,09%	12,72%
	IN_{511}	178,87	183,16	218,68	2,40%	19,39%
	OUT_{4095}	1281,99	1288,12	1325,92	0,48%	2,93%
	IN_{4095}	1287,44	1290,88	1451,28	0,27%	12,43%

Os resultados apresentados na Tabela 8.17 foram obtidos com a política de escalonamento dinâmico do *host* desativada. Desta forma, a CPU pode ficar ociosa mesmo quando existem tarefas prontas para executar, pois se estas tiverem sido escalonadas após alguma tarefa ainda pendente, o HDS responsável não irá dispará-las. Em várias situações pode ser proveitoso ativar a política de escalonamento do HDS. Através da análise dos resultados obtidos, observa-se que existe uma certa melhora da abordagem (3) nos grafos OUT_{4095} e IN_{4095}, como pode ser observado na Tabela 8.18. Com relação aos outros resultados os *makespans* obtidos foram semelhantes.

Tabela 8.18: Execução da abordagem (3) no ambiente HET_{global} para aplicações representadas por árvores binárias com a políticas de HDS diferentes

t_{exe}	aplicação	HDS = 0	HDS = 1	% melhora
$1s$	OUT_{4095}	285,63	275,39	3,58%
	IN_{4095}	297,77	281,86	5,34%
$5s$	OUT_{4095}	1325,92	1310,21	1,18%
	IN_{4095}	1451,28	1343,79	7,40%

Os testes realizados ativando a política de escalonamento permite que o HDS mude a ordem de execução definida pelo escalonador dinâmico local (SDS). Esta estratégia é importante para ambientes dinâmicos e também para grafos com muitas tarefas onde o SDS possui uma visão parcial do grafo. Sendo assim, como nos testes realizados a ativação da política de escalonamento do HDS trouxe vantagem nos *makespans* obtidos, os próximos experimentos foram executados considerando esta política ativada (HDS = 1).

Os tempos de execução obtidos para aplicações representadas por grafos diamante são mostrados na Tabela 8.19. Devido a topologia interna destes grafos, que determina um grande número de dependências entre as tarefas, o escalonamento dinâmico não obtém um desempenho tão bom em relação ao *valor base* estabelecido pela abordagem estática (1). Isto acontece, pois o escalonador dinâmico possui uma visão parcial do grafo o que pode levar a decisões de escalonamento que apesar de eficientes em um dado momento não contribuam para um *makespan* final menor. Este problema se torna mais evidente neste caso, onde o escalonador dinâmico trata apenas as tarefas que possuam nível menor do que l, onde l representa o nível da última tarefa que terminou a sua execução. Uma alternativa seria aumentar o tamanho do bloco B_l de forma que mais tarefas pendentes possam ser consideradas, e uma melhor análise da topologia do grafo possa ser realizada.

Visto que o ambiente de execução é ideal para a alocação inicial estabelecida pela abordagem (1), que possui uma visão total do grafo, os *makespans* obtidos em (2) são até 12% piores, enquanto os resultados da abordagem (3) são piores que (2) em até 60%. Para grafos com poucas tarefas os *makespans* obtidos pela abordagem (3) são quase o dobro da execução estática. Tal comportamento ocorre devido ao alto grau de dependência entre as tarefas e o pequeno número de tarefas por máquina, onde a estratégia dinâmica possui poucas opções de reescalonamento, não havendo tempo suficiente para corrigir a alocação inicial estabelecida. Nestes casos, uma alocação estática inicial precisa resulta em uma grande vantagem.

Tabela 8.19: Execução no ambiente HET_{global} para aplicações representadas por grafos diamante

t_{exe}	aplicação	(1)	(2)	(3)	% piora (2)/(1)	% piora (3)/(2)
$1s$	DI_{400}	48,88	51,56	81,50	5,48%	58,06%
	DI_{4096}	315,88	330,29	411,91	4,56%	24,71%
$5s$	DI_{400}	221,68	248,28	397,24	11,99%	59,99%
	DI_{4096}	1430,41	1558,75	1832,96	8,96%	17,59%

De acordo com a estrutura de escalonamento definida no Capítulo 4, as tarefas pendentes de uma aplicação só serão reescalonadas entre sites, quando algum dos sites requisitar ao GDS tarefas pendentes que pertençam a outros sites da grade computacional (Algoritmo 5). Para grafos diamante, a maioria das tarefas disponíveis para reescalona-

mento são pendentes. Neste caso, o GDS trabalha apenas como mediador entre pedidos disparados pelos escalonadores dinâmicos do site. No Algoritmo 2, é possível verificar que o escalonamento de tarefas pendentes é disparado pelo SDS de acordo com o nível topológico do grafo, e como especificado anteriormente ele ocorre com menos frequência do que o escalonamento de tarefas prontas. Logo, o GDS participa de um número bem menor de eventos de escalonamento de tarefas pendentes do que prontas. Para grafos diamante esta abordagem não gera tantos benefícios como para aplicações representadas por árvores. Uma outra alternativa que também pode ser analisada em trabalhos futuros, é a viabilidade de disparar um maior número de eventos de escalonamento de tarefas pendentes. Este estudo deve ser feito analisando tanto a topologia do grafo, quanto as variações do ambiente computacional.

Apesar da abordagem (2) possuir conhecimento da alocação estática inicial, e o cenário de execução HET_{global} não possuir variações no poder computacional dos recursos durante a execução da aplicação, os *makespans* obtidos não foram os mesmos de (1). Durante a execução de aplicações GAD, a carga de trabalho existente na fila de tarefas prontas não é previsível, pois a todo momento novas tarefas terminam a execução e outras se tornam prontas. O mesmo ocorre com as tarefas pendentes, que podem mudar de estado a qualquer momento da execução. A heurística desenvolvida para aplicações com relações de precedência não é capaz de prever estas mudanças, e como a visão do escalonador dinâmico é parcial, eventos de escalonamento podem ser disparados mesmo em um ambiente que a princípio se mostra estático. Esta particularidade no escalonamento de aplicações GAD permite que muitas pesquisas ainda sejam realizadas para que se aumente o desempenho do escalonador proposto.

Para avaliação da política de escalonamento global em um ambiente dinâmico foi definido o cenário de execução DIN_{global}. Através da execução da aplicação do usuário no cenário HET_{global} com a abordagem estática, são estabelecidos dois instantes de tempo t_1 e t_2 para alterar a carga computacional dos recursos durante a execução. O tempo t_1 indica 1/4 do *makespan* obtido e t_2 indica 3/4. Inicialmente, no cenário DIN_{global}, todos os recursos do site A disponibilizam metade do seu poder computacional, enquanto o site B oferece 100%. No instante t_1, o site A se torna dedicado a execução da aplicação e o site B passa a oferecer apenas 1/3 do seu poder computacional. No tempo t_2 os dois sites passam a oferecer 100% de sua capacidade de processamento. Neste ambiente de execução, a avaliação da política de escalonamento dinâmica foi feita através da execução das aplicações DI_{4096}, OUT_{4095}, IN_{4095} e $Ncorpos_{2304}$ (AMS-ring).

A Tabela 8.20 apresenta os resultados obtidos no cenário DIN_{global}, utilizando as mesmas abordagens de escalonamento definidas no cenário HET_{global}. As duas últimas colunas da tabela mostram, respectivamente, a porcentagem de melhora das abordagens (2) e (3) que utilizam escalonamento dinâmico em relação a abordagem estática (1).

Tabela 8.20: Execução no ambiente DIN_{global}

t_{exe}	aplicação	(1)	(2)	(3)	% melhora (2)/(1)	% melhora (3)/(1)
$1s$	DI_{4096}	415,52	373,42	375,72	10,13%	9,58%
	OUT_{4095}	357,53	268,22	270,02	33,30%	32,41%
	IN_{4095}	357,19	281,33	291,46	26,97%	22,55%
$5s$	DI_{4096}	1960,64	1546,09	1581,41	21,14%	19,34%
	OUT_{4095}	1708,44	1262,94	1280,50	35,27%	33,42%
	IN_{4095}	1710,37	1300,34	1300,46	31,53%	31,52%
$2,2s$	$Ncorpos_{2304}$	421,87	355,80	362,91	15,66%	13,98%

Como esperado, a abordagem estática produz os piores resultados, já que o cenário de execução definido é dinâmico. As abordagens (2) e (3) alcançam *makespans* menores devido à sua capacidade de perceber as variações do ambiente e adaptar a execução da aplicação. Os maiores benefícios trazidos pelo escalonamento dinâmico são para os grafos representados por árvores binárias, onde a melhora em relação à abordagem estática gira em torno de 30%. Os grafos do tipo diamante e a aplicação *N-corpos* possuem um maior número de relações de precedência entre suas tarefas, o que dificulta as decisões do escalonador dinâmico, já que apenas uma parte do grafo é tratada a cada evento de escalonamento.

Com relação à importância de se adotar um escalonamento estático inicial preciso, é possível perceber que no cenário DIN_{global}, a vantagem da abordagem (2) em relação à abordagem (3) diminui. Isto ocorre, pois como o ambiente de execução é dinâmico, as decisões tomadas pelo escalonador estático não possuem tão grande importância quanto no cenário HET_{global}. Entretanto, mesmo em um ambiente computacional sujeito a variações, uma alocação estática inteligente pode gerar resultados ligeiramente melhores, como os obtidos pela abordagem (2).

8.4.5 Grade Computacional Compartilhada e Dinâmica

Esta subseção apresenta alguns experimentos realizados em uma grade computacional geograficamente distribuída com recursos heterogêneos e compartilhados. Os testes foram executados considerando dois sites distintos com 20 HMs cada. O primeiro site é composto por máquinas do SGCLab e o segundo, localizado na PUC-Rio, é composto por processadores Pentium II 400 MHz com 256Mb RAM, rodando Linux Red Hat, Globus Toolkit 2.4 e LAM/MPI 7.0.6. Conforme já descrito anteriormente, a conexão é feita através de uma rede com capacidade de transmissão de 1 Gigabit.

Para este conjunto de testes foi usada a aplicação de *N-corpos* com 400 mil partículas representada por uma grafo com 9217 tarefas. A Figura 8.26 mostra os resultados obtidos na execução alternada da aplicação AMS-ring com e sem a estratégia de escalona-

mento dinâmico ativada. Já, a Figura 8.27 apresenta os *makespans* obtidos considerando, porém, que a execução da aplicação AMS-ring com e sem escalonamento dinâmico foi realizada simultaneamente. Como o ambiente é compartilhado, não é possível fazer análises detalhadas em relação aos tempos de execução obtidos. O que se pode perceber é que o escalonamento dinâmico permite que o desempenho da aplicação seja menos sensível às mudanças ocorridas no ambiente.

Na Figura 8.26, é importante ressaltar que nas dez primeiras rodadas de teste o escalonador estático produz uma alocação inicial que considera que as máquinas do SGCLab e da PUC possuem a mesma capacidade computacional, o que explica a piora dos valores obtidos para a abordagem estática. Já, com o escalonamento dinâmico ativado, as tarefas da aplicação *N-corpos* são reescalonadas adequadamente entre as máquinas de acordo com a capacidade de processamento disponível. É importante observar que, alocação estática inicial mais precisa melhora inclusive os valores obtidos pela abordagem dinâmica, como é mostrado nos resultados das 5 rodadas subsequentes.

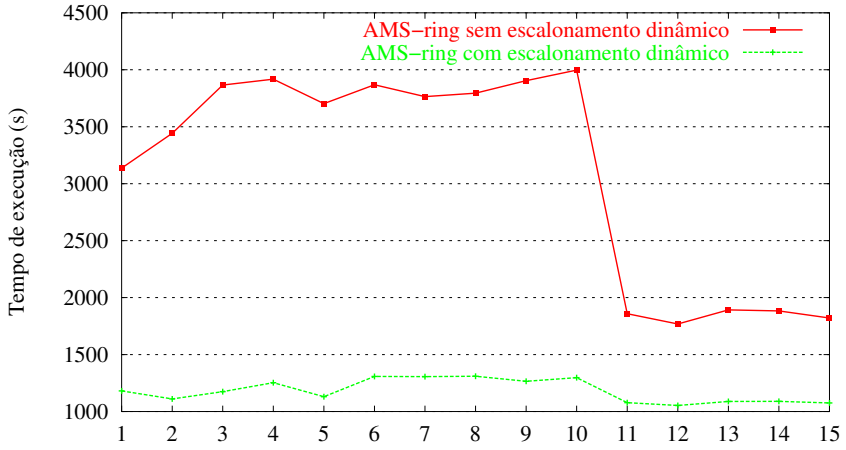

Figura 8.26: *Makespans* obtidos com a execução alternada de aplicações AMS-ring com e sem escalonamento dinâmico ativado

Os resultados apresentados na Figura 8.27 também mostram o bom desempenho da estratégia de escalonamento dinâmico implementada. Neste caso, são disparadas duas aplicações AMS-ring ao mesmo tempo, com o objetivo que cada uma delas possua aproximadamente o mesmo poder computacional disponível. Com o escalonamento dinâmico ativado não foram observados picos nos tempos de execução obtidos, como no caso da execução da aplicação que segue apenas o escalonamento estático inicial. Como é de se esperar, ao ocorrer sobrecarga de processamento em algumas das máquinas disponíveis, o escalonamento dinâmico pode corrigir a alocação de tarefas corrente, melhorando o desempenho da aplicação.

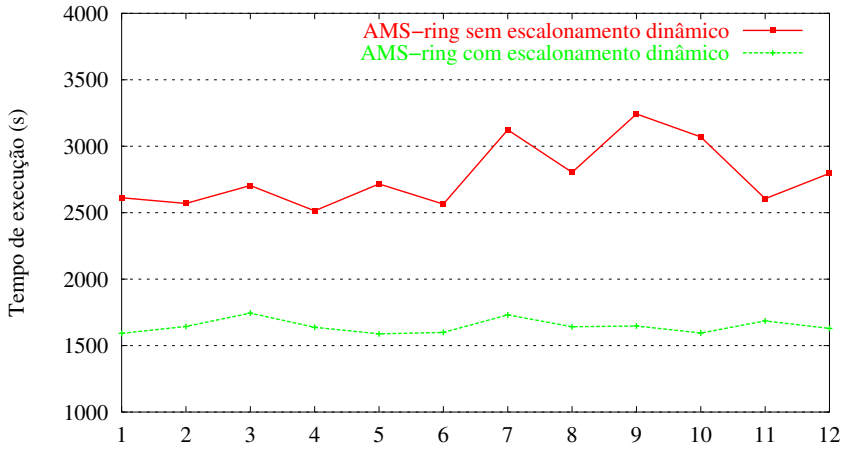

Figura 8.27: Tempos de execução obtidos com a execução simultânea de duas aplicações AMS-ring com e sem escalonamento dinâmico ativado

8.5 Resumo

Neste capítulo foi realizada uma avaliação da estrutura de escalonamento proposta com a execução de aplicações GAD. Através dos diversos experimentos realizados em um ambiente real semi-controlado, foi verificada a eficiência da estrutura criada que permite o uso de diferentes políticas de escalonamento distribuídas para uma mesma aplicação paralela. Cada escalonador dinâmico da hierarquia possui heurísticas específicas de acordo com o seu escopo de atuação, e de forma colaborativa eles trabalham para proporcionar uma execução eficiente.

As heurísticas propostas nesta tese foram comparadas com outras heurísticas de escalonamento dinâmico para aplicações GAD encontradas na literatura. Através da análise dos resultados foi verificada a eficiência das heurísticas GAD1 e GAD2, principalmente em cenários dinâmicos que retratam uma grade computacional. Além disso, devido à implementação de diversas heurísticas de escalonamento dinâmico, o *EasyGrid AMS* aumenta o seu escopo de atuação em relação às diferentes classes de aplicações que podem ser executadas em uma grade computacional.

Devido à dificuldade de se encontrar o *makespan* ótimo de uma aplicação GAD em um ambiente dinâmico, heterogêneo e distribuído, não foram feitas comparações em relação a valores ótimos, mas com relação a *valores base* que representam um *makespan* alvo eficiente dentro de um determinado cenário de execução. Entretanto, considerando as características internas da aplicação *N-corpos*, uma estimativa aproximada de um tempo de execução ótimo foi realizada, onde a sobrecarga da heurística GAD1 se mostrou pequena. Atualmente, não se conhece nenhum outro *middleware* no contexto das grades computacionais com uma sobrecarga tão pequena, que seja capaz de tratar aplicações com restrições de dependência.

Capítulo 9

Conclusões e Trabalhos Futuros

Atualmente, muita pesquisa vem sendo investida no estudo de problemas relacionados às grades computacionais, que são ambientes que se tornaram atraentes por oferecem grande capacidade de processamento a um baixo custo. Um dos principais objetivos dos pesquisadores é criar soluções para o gerenciamento das aplicações paralelas e distribuídas que compartilham os recursos de uma grade, de forma a proporcionar uma execução eficiente e segura.

Um ponto essencial para uma execução eficiente é que o sistema gerenciador de grades possua uma estrutura de escalonamento que seja capaz de perceber as variações do ambiente e assim adaptar a execução da aplicação de acordo com o poder computacional disponível. O estudo apresentado nesta tese se concentra na área de escalonamento dinâmico para grades computacionais. Para melhor entendimento das dificuldades e desafios de se trabalhar com tais ambientes, várias características relacionadas ao gerenciamento de uma aplicação na grade foram estudadas.

Inicialmente, alguns sistemas gerenciadores de grade específicos foram pesquisados com o objetivo de indicar os seus princípios e características principais. A partir desta pesquisa foi proposta uma taxonomia que classifica cada sistema de acordo com suas filosofias de implementação: *UMS*, *RMS* e *AMS*. Características típicas são definidas para cada tipo de implementação, considerando critérios como orientação, gerenciamento, escalonamento, instalação e autonomia. Assim, é possível avaliar as vantagens e desvantagens trazidas por um determinado tipo de implementação e destacar, dentro do contexto da taxonomia, as propriedades do *EasyGrid AMS*.

O *EasyGrid AMS* em desenvolvimento na Universidade Federal Fluminense [18, 21, 22, 90, 113] é um sistema gerenciador de aplicações, responsável por gerenciar e automatizar aplicações para que estas executem de forma eficiente e segura nos vários recursos disponíveis de uma grade. A criação de ferramentas que permitem que a aplicação do usuário seja autogerenciável é um ponto crucial para que se minimize o esforço de cientistas ao se utilizar uma grade computacional.

Um dos objetivos principais desta tese é prover o *EasyGrid AMS* com uma estrutura de escalonamento dinâmico eficiente e flexível que possa se adaptar às dificuldades impostas pelas grades computacionais. Assim, é criada uma estrutura de escalonamento dinâmico hierárquico para cada aplicação, onde políticas distintas podem ser empregadas nos diferentes níveis da hierarquia. Desta forma, é possível escolher heurísticas adequadas às características das aplicações e às políticas de acesso de diferentes instituições. A estrutura de escalonamento proposta permite que os escalonadores dinâmicos de diferentes níveis hierárquicos trabalhem em conjunto para definir um escalonamento eficiente. A distribuição de tarefas entre os escalonadores da hierarquia, além de permitir o uso de políticas de escalonamento distintas, proporciona maior escalabilidade ao sistema gerenciador de aplicações.

Neste trabalho, também foram criadas heurísticas de escalonamento específicas por classes de aplicações e um vasto número de experimentos foram realizados para que a estrutura de escalonamento e as políticas propostas fossem validadas. As heurísticas desenvolvidas no contexto do *EasyGrid AMS* tiveram que considerar vários aspectos importantes, como por exemplo, a distribuição das funções desempenhadas em cada escalonador dinâmico da hierarquia e a atualização constante das informações relativas ao ambiente e à execução da aplicação entre os diversos escalonadores. A interação entre cada escalonador da hierarquia foi analisada de forma que heurísticas eficientes e de baixa intrusão fossem desenvolvidas.

A estrutura de escalonamento fornecida pelo *AMS* é embutida, assim como outras funcionalidades, na própria aplicação MPI. São criadas, portanto, aplicações cientes do ambiente (*system aware*) que são capazes de se auto-otimizar de acordo com as variações do ambiente computacional. É importante ressaltar que não são feitas modificações no código da aplicação do usuário, as funcionalidades relativas ao gerenciamento da aplicação na grade são embutidas automaticamente no código, permitindo que as complexidades impostas pelo ambiente sejam transparentes ao usuário. Esta abordagem aumenta a quantidade de aplicações MPI pré-existentes que podem ser executadas de forma eficiente em um ambiente grade.

As heurísticas desenvolvidas são divididas em duas partes nesta tese, segundo o tipo de aplicação do usuário: aplicações BoT e aplicações GAD. Na primeira parte é estudado especificamente o escalonamento de aplicações do tipo *bag-of-tasks*, que são vastamente executadas em grades computacionais. Através da análise dos experimentos realizados em ambientes de grades reais, a política de escalonamento dinâmico implementada mostra baixo custo, além de definir uma alocação eficiente das tarefas nos recursos disponíveis. Apesar de existirem vários trabalhos que tratam aplicações BoT, são poucos os estudos que comparam as soluções propostas. Isto ocorre devido à dificuldade de se instalar diferentes sistemas gerenciadores, configurar parâmetros e de criar ambientes de execução reais. Logo, a análise de resultados realizada neste trabalho é de grande importância, pois os tempos

de execução obtidos são comparados com valores que representam limites inferiores de execução, o que permite avaliar a eficiência da solução proposta.

A heurística BoT apresentada no Capítulo 5, é uma política de baixo custo e intrusão, onde as decisões de escalonamento são baseadas no cálculo do índice de desbalanceamento das máquinas. O objetivo é balancear as cargas de trabalho existentes em cada recurso da grade computacional. O índice de desbalanceamento implementado é eficiente e permite que a heurística se torne mais sensível às mudanças do sistema quando a execução da aplicação se aproxima do seu final. Além disso, o uso de porcentagens ao se pedir tarefas aos recursos sobrecarregados do sistema, também permite que a heurística seja mais eficiente, já que nunca é pedido um número de tarefas maior do que um recurso realmente pode ceder. Desta forma, os recursos não ficam oscilando entre os estados de sobrecarregado e subcarregado.

A segunda parte da tese, que é dedicada à construção de uma heurística para aplicações com relações de precedência, mostra as dificuldades que surgem com este tipo de aplicação, onde detalhes como a ordenação de tarefas pode ser crucial. Apesar de não ter sido avaliado experimentalmente nesta pesquisa, o problema de ordenação é resolvido usando prioridades associadas as tarefas de forma que elas sejam escalonadas em um recurso obedecendo uma ordem de preferência. As prioridades influenciam diretamente no escalonamento produzido, e através das análises realizadas, observa-se que a utilização do *blevel* pode ser uma boa estratégia. Porém, com o objetivo de ainda melhorar o escalonamento baseado no *blevel* das tarefas, foi implementado, na política dinâmica, o mecanismo onde é possível inserir tarefas em espaços ociosos de um recurso sem que haja necessidade de respeitar o *blevel*.

Algoritmos desenvolvidos para aplicações GAD possuem melhor desempenho se um maior número de tarefas do grafo for tratado em cada evento de escalonamento. Isto acontece devido às dependências existentes entre as tarefas. Com o objetivo de diminuir a complexidade dos algoritmos dinâmicos, alguns estudos consideram apenas blocos de tarefas a cada evento de escalonamento. Nestes casos, como o escalonador dinâmico possui uma visão restrita do grafo é importante que informações do escalonamento estático possam ser usadas para que se alcance melhores resultados. Uma abordagem de escalonamento híbrida traz benefícios principalmente para aplicações com relações de precedência, onde o escalonador dinâmico pode basear suas decisões no escalonamento estático inicial e, ainda, nas prioridades associadas a ele. Por outro lado, quanto maior as variações ocorridas no poder computacional dos recursos ao longo da execução da aplicação, menor será a importância de uma alocação estática inicial precisa.

Outro aspecto analisado é a importância de se reescalonar uma mesma tarefa, mais de uma vez, principalmente em ambientes dinâmicos como as grades computacionais. Os testes realizados para esta classe de aplicações, também foram executados em ambientes reais e comparados com outras heurísticas híbridas existentes na literatura. Além de prover

o *EasyGrid AMS* com diferentes políticas de aplicações GAD, os experimentos realizados mostraram a eficiência das heurísticas propostas.

Nas heurísticas para aplicações GAD descritas no Capítulo 7, é proposto um escalonamento dinâmico que permite que uma mesma tarefa da aplicação do usuário seja reescalonada várias vezes, sempre que necessário. Esta abordagem se mostra mais eficiente quando os ambientes de execução possuem características dinâmicas. Para diminuir a complexidade do algoritmo e aumentar a sua eficiência, as tarefas da aplicação são divididas em dois grupos distintos: *prontas* e *pendentes*. O reescalonamento de tarefas *prontas* é realizado com mais frequência, pois além destas tarefas possuírem maior urgência de execução, sua heurística de escalonamento tem uma complexidade menor, onde todas as tarefas *prontas* constituem um grupo de tarefas independentes.

O escalonamento de tarefas *pendentes* ocorre de acordo com a topologia do grafo, sendo disparado um número menor de vezes. Como estas tarefas ainda não estão prontas para executar, não existe a necessidade que o seu reescalonamento seja disparado frequentemente. Isto é importante, pois esta heurística possui cálculos mais elaborados para avaliação do *makespan* atual. Para isso, é necessário que se considere a comunicação entre as tarefas e os tempos estimados de fim de todos seus predecessores. O escalonamento de tarefas *pendentes* é importante principalmente para grafos que possuem poucas tarefas prontas a cada passo da execução, como por exemplo, os grafos do tipo diamante. Nestes casos, as tarefas podem ficar prontas e logo serem disparadas, não permitindo o seu reescalonamento, já que o escalonamento dinâmico implementado não trata tarefas em execução.

O problema de escalonamento de tarefas para aplicações com relações de precedência ainda é pouco estudado na área de grades computacionais. A maioria dos sistemas gerenciadores existentes que são capazes de executar este tipo de aplicação não possui políticas de escalonamento específicas que lidem com as dependências existentes entre as tarefas da aplicação. Nestes casos, a política de escalonamento adotada se limita a tratar apenas as tarefas prontas para execução, que se caracterizam por não possuírem dependências entre si. Como demonstrado nos experimentos realizados, esta abordagem simplista pode comprometer a execução eficiente da aplicação.

Outro ponto que deve ser destacado é a existência de um *AMS* por aplicação, o que permite que a tarefa de escalonamento seja distribuída entre as diversas aplicações que compartilham a grade. Assim, não existe nenhum módulo centralizador que comprometa a escalabilidade do sistema. Porém, o escalonamento descentralizado, deve cuidar para que conflitos sejam evitados, e para isso, os escalonadores distribuídos coordenam as suas decisões através do monitoramento do sistema e da aplicação. A ausência de conflitos foi verificada nos experimentos realizados, onde cada aplicação se ajusta automaticamente às mudanças do sistema, tomando conhecimento de outras aplicações que executam ao mesmo tempo na grade computacional. Além disso, o escalonamento proposto possui baixo custo

de execução, podendo ser ativado frequentemente na busca de melhores escalonamentos. Tal fator pode ser decisivo quando o sistema não possuir informações precisas em relação às características internas da aplicação ou do poder computacional da grade. Vale a pena lembrar que a utilização de mensagens de monitoramento e as próprias mensagens disparadas pelo escalonador dinâmico causam uma sobrecarga irrisória no sistema, já que enquanto as mensagens circulam entre os gerenciadores distribuídos, as tarefas da aplicação do usuário continuam a sua execução.

É importante ressaltar, que o objetivo deste trabalho não é somente propor técnicas de escalonamento eficientes, mas estudar maneiras de usar tais heurísticas dentro da abordagem do *EasyGrid AMS* e, assim, validar o modelo de escalonamento proposto: descentralizado e hierárquico. Como detalhado no Capítulo 1, as seguintes contribuições deste trabalho podem ser destacadas:

- proposta de uma estrutura de escalonamento hierárquico, descentralizado por aplicação, eficiente e de baixa intrusão;
- criação de novas heurísticas de escalonamento dinâmico para aplicações *bag-of-tasks*;
- criação de novas heurísticas de escalonamento dinâmico para aplicações com relações de precedência;
- integração de várias heurísticas de escalonamento distribuídas que trabalham em conjunto para proporcionar, de maneira escalável, a execução eficiente de uma aplicação em uma grade computacional;
- validação do modelo de escalonamento no contexto do *EasyGrid AMS*, avaliando um grande número de resultados experimentais;
- suporte às características de auto-otimização e auto-ajuste;
- suporte para que o *EasyGrid AMS* trate diferentes tipo de aplicações paralelas que possam ser representadas por grafos acíclicos direcionados.
- classificação de sistemas gerenciadores de grade segundo a taxonomia descrita.

Com relação aos trabalhos futuros, uma vasta pesquisa pode ser feita na área de escalonamento dinâmico para grades computacionais. Ao se trabalhar com aplicações que possuam relações de precedência, critérios como a replicação de tarefas podem trazer grandes benefícios quando o custo de comunicação for grande. Desta forma, a replicação de tarefas deve ser implementada principalmente entre sites diferentes. Como as grades são ambientes dinâmicos também devem ser propostos mecanismos que permitam que cópias de tarefas possam ser eliminadas de acordo com as condições do sistema. Além disso,

devem ser adotados critérios que permitam avaliar a quantidade de replicação que pode ser realizada e onde esta replicação deve ser feita.

Ainda com relação às aplicações GAD, análises futuras podem ser realizadas com relação ao tamanho do bloco de tarefas que deve ser tratado pelo escalonador dinâmico. É importante ressaltar, que a utilização de um bloco que compreenda tarefas de mais níveis topológicos permite que o escalonador dinâmico obtenha mais informações sobre a topologia do grafo, já que é possível tratar um maior número de tarefas e, consequentemente, as precedências existentes entre elas. Entretanto, é necessário identificar um tamanho de bloco que não comprometa o desempenho do sistema, já que à medida que o número de tarefas tratado em um evento de escalonamento aumenta, maior será a sobrecarga causada pelo escalonador dinâmico.

Técnicas para migração de tarefas também podem ser implementadas para que a heurística proposta se torne ainda mais abrangente. Entretanto, os custos associados a migração devem ser analisados para que as decisões de escalonamento possam se basear em informações precisas, onde as sobrecargas da implementação sejam consideradas. Além disso, é importante que futuras pesquisas avaliem a execução de aplicações que possuam granularidade mais fina do que as estudadas nesta tese. Nestes casos, os custos de comunicação devem ser tratados com eficiência, e para isso o modelo de comunicação do *EasyGrid* deve ser especificado detalhadamente.

Um outro ponto que merece atenção é o desenvolvimento de mecanismos que diminuam a sobrecarga do *EasyGrid AMS* quando um número de tarefas muito grande for considerado para execução. Devido a estrutura de dados empregada atualmente, durante a execução da aplicação do usuário pode ocorrer um alto uso de memória que comprometa o seu desempenho. Porém, soluções para este problema já vem sendo analisadas, através da especificação de estruturas de dados otimizadas.

Referências Bibliográficas

[1] Aarseth, S. J. From NBODY1 to NBODY6: The growth of an industry. *Publications of the Astronomical Society of the Pacific 111*, 765 (Novembro 1999), 1333–1346.

[2] Adam, T. L., Chandy, K. M., Dickson, J. R. A comparison of list schedules for parallel processing systems. *Communications of the ACM 17*, 12 (1974), 685–690.

[3] Ali, S., Siegel, H. J., Maheswaran, M., Ali, S., Hensgen, D. Task execution time modeling for heterogeneous computing systems. *Proceedings of the 9th Heterogeneous Computing Workshop (HCW'00)* (Cancun, México, Maio 2000), IEEE Computer Society, p. 185–199.

[4] Allen, G., Angulo, D., Foster, I., Lanfermann, G., Liu, C., Radke, T., Seidel, E., Shalf, J. The cactus worm: Experiments with dynamic resource discovery and allocation in a grid environment. *International Journal of High Performance Computing Applications 15*, 4 (Novembro 2001), 345–358.

[5] Allen, G., Benger, W., Dramlitsch, T., Goodale, T., Hege, H.-C., Lanfermann, G., Merzky, A., Radke, T., Seidel, E., Shalf, J. Cactus tools for grid applications. *Journal of Cluster Computing 4*, 3 (Julho 2001), 179–188.

[6] Allen, G., Dramlitsch, T., Foster, I., Karonis, N. T., Ripeanu, M., Seidel, E., Toonen, B. Supporting efficient execution in heterogeneous distributed computing environments with Cactus and Globus. *Proceedings of the ACM/IEEE Supercomputing 2001 Conference (SC'01)* (Denver, EUA, Novembro 2001), IEEE Computer Society Press, p. 52–76.

[7] Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M., Werthimer, D. SETI@home: An experiment in public-resource computing. *Communications of the ACM 45*, 11 (2002), 56–61.

[8] Andrade, N., Costa, L., Germoglio, G., Cirne, W. Peer-to-peer grid computing with the ourgrid community. *Proceedings of the 23rd Brazilian Symposium on Computer Networks (SBRC 2005) - IV Special Tools Session* (Fortaleza, Brasil, Maio 2005).

[9] Arora, M., Das, S. K., Biswas, R. A de-centralized scheduling and load balancing algorithm for heterogeneous grid environments. *Proceedings of the International Conference on Parallel Processing Workshops (ICPPW'02)* (Vancouver, Canadá, Agosto 2002), IEEE Computer Society Press, p. 499–505.

[10] Barnes, J., Hut, P. A hierarchical O(N log N) force-calculation algorithm. *Nature 324*, 4 (Dezembro 1986), 446–449.

[11] Beaumont, O., Boudet, V., Robert, Y. A realistic model and efficient heuristic for scheduling with heterogeneous processors. *Technical Report RR2001-37* (Setembro 2001).

[12] Beaumont, O., Legrand, A., Robert, Y. Static scheduling strategies for heterogeneous systems. *Computing and Informatics 21* (2002), 413–430.

[13] Berman, F., Casanova, H., Chien, A., Cooper, K., Dail, H., Dasgupta, A., Deng, W., Dongarra, J., Johnsson, L., Kennedy, K., Koelbel, C., Liu, B., Liu, X., Mandal, A., Marin, G., Mazina, M., Mellor-Crummey, J., Mendes, C., Olugbile, A., Patel, J. M., Reed, D., Shi, Z., Sievert, O., Xia, H., YarKhan, A. New grid scheduling and rescheduling methods in the GrADS project. *International Journal of Parallel Programming 33*, 2 (2005), 209–229.

[14] Berman, F., Chien, A., Cooper, K., Dongarra, J., Foster, I., Gannon, D., Johnsson, L., Kennedy, K., Kesselman, C., Mellor-Crummey, J., Reed, D., Torczon, L., Wolski, R. The GrADS Project: Software support for high-level Grid application development. *The International Journal of High Performance Computing Applications 15*, 4 (2001), 327–344.

[15] Berman, F., Wolski, R. The AppLeS project: A status report. *Proceedings of the 8th NEC Research Symposium* (Berlim, Alemanha, Maio 1997), Springer-Verlag.

[16] Blythe, J., Jain, S., Deelman, E., Gil, Y., Vahi, K., Mandal, A., Kennedy, K. Task scheduling strategies for workflow-based applications in grids. *Proceedings of 5th International Symposium on Cluster Computing and the Grid (CCGrid 2005)* (Cardiff, Reino Unido, Maio 2005), IEEE Computer Society Press, p. 759–767.

[17] Boeres, C., Filho, J. V., Rebello, V. E. F. A cluster-based strategy for scheduling task on heterogeneous processors. *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2004)* (Foz do Iguaçu, Brasil, Outubro 2004), IEEE Computer Society Press, p. 214–221.

[18] Boeres, C., Fonseca, A. A., Mendes, H. A., Menezes, L. T., Moura, N. T., Silva, J. A., Vianna, B. A., Rebello, V. E. F. An EasyGrid Portal for scheduling system-aware applications on computational grids. *Concurrency and Computation: Practice and Experience 18*, 6 (2005), 553–566.

[19] Boeres, C., Lima, A., Rebello, V. Hybrid task scheduling: Integrating static and dynamic heuristics. *Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2003)* (São Paulo, Brasil, Novembro 2003), IEEE Computer Society Press, p. 199–206.

[20] Boeres, C., Nascimento, A., Rebello, V. Cluster-based task scheduling for LogP model. *International Journal of Foundations of Computer Science 10*, 4 (1999), 405–424.

[21] Boeres, C., Nascimento, A. P., Rebello, V. E. F., Sena, A. C. Efficient hierarchical self-scheduling for MPI applications executing in computational grids. *Proceedings of the 3rd international workshop on Middleware for Grid Computing* (Grenoble, França, Novembro 2005), ACM Press, p. 1–6.

[22] Boeres, C., Rebello, V. E. F. EasyGrid: Towards a framework for the automatic grid enabling of legacy MPI applications. *Concurrency and Computation: Practice and Experience 16*, 5 (Abril 2004), 425–432.

[23] Braun, T., Siegel, H., Boloni, L., Maheswaran, M., Reuther, A., Robertson, J., Theys, M., Yao, B. A taxonomy for describing matching and scheduling heuristics for mixed-machine heterogeneous computing systems. *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems* (Ohio, EUA, Outubro 1998), IEEE Computer Society Press, p. 330–335.

[24] Brooks, R. A. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation 2*, 1 (Março 1986), 14–23.

[25] Buyya, R., Abramson, D., Giddy, J. Economy driven resource management architecture for computational power grids. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA2000)* (Las Vegas, EUA, Junho 2000).

[26] Buyya, R., Abramson, D., Giddy, J. Nimrod/G: An architecture for a resource management and scheduling system in a global computational grid. *Proceedings of the 4th International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region (HPC ASIA'2000)* (Pequim, China, Maio 2000), vol. 1, IEEE Computer Society Press, p. 283–289.

[27] Buyya, R., Abramson, D., Venugopal, S. The grid economy. *Proceedings of the IEEE, Special Issue on Grid Computing 93*, 3 (Março 2005), 698–714.

[28] Buyya, R., Murshed, M., Abramson, D., Venugopal, S. Scheduling parameter sweep applications on global grids: a deadline and budget constrained cost-time optimization algorithm. *Software Practice Experience 35*, 5 (2005), 491–512.

[29] Buyya, R., Venugopal, S. The Gridbus toolkit for service oriented grid and utility computing: An overview and status report. *Proceedings of the First IEEE International Workshop on Grid Economics and Business Models (GECON 2004)* (Seul, Coréia do Sul, Abril 2004), IEEE Computer Society Press, p. 19–66.

[30] Cao, J., Jarvis, S., Saini, S., Nudd, G. Gridflow: Workflow management for grid computing. *Proceedings of 3rd International Symposium on Cluster Computing and the Grid (CCGrid 2003)* (Tóquio, Japão, Maio 2003), IEEE Computer Society Press, p. 198–205.

[31] Cao, J., Jarvis, S. A., Saini, S. ARMS: An agent-based resource management system for grid computing. *Scientific Programming, Special Issue for Grid Computing 10*, 2 (2002), 135–148.

[32] Carter, B. R., Watson, D. W., Freund, R. F., Keith, E., casca Mirabile, F., Siegel, H. J. Generational scheduling for dynamic task management in heterogeneous computing systems. *Journal of Information Sciences 106*, 3-4 (1998), 219–236.

[33] Casanova, H., Legrand, A., Zagorodnov, D., Berman, F. Heuristics for scheduling parameter sweep applications in grid environments. *Proceedings 9th Heterogeneous Computing Workshop (HCW'00)* (Cancun, México, Maio 2000), IEEE Computer Society Press, p. 349–363.

[34] Casanova, H., Obertelli, G., Berman, F., Wolski, R. The AppLeS parameter sweep template: User-level middleware for the grid. *Proceedings of the ACM/IEEE Supercomputing 2000 Conference (SC'00)* (Dallas, EUA, Novembro 2000), IEEE Computer Society Press, p. 60–78.

[35] Casanova, H., Zagorodnov, D., Legrand, A., Berman, F. Using simulation to evaluate scheduling heuristics for a class of applications in grid environments. *Technical Report RR1999-46* (Setembro 1999).

[36] Chapin, S., Katramatos, D., Karpovich, J., Grimshaw, A. S. The Legion resource management system. *Proceedings of the 5th Workshop Job Scheduling Strategies for Parallel Processing* (San Juan, Porto Rico, Abril 1999), Springer Verlag, p. 162–178.

[37] Chervenak, A., Foster, I., Kesselman, C., Salisbury, C., Tuecke, S. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications 33*, 3 (2000), 187–200.

[38] Cirne, W., Paranhos, D., Costa, L.and Santos-Neto, E., Brasileiro, F., Sauve, J., Silva, F., Barros, C., Silveira, C. Running Bag-of-Tasks applications on computational grids: The MyGrid approach. *Proceedings of the 32nd International Conference on Parallel Processing (ICPP03)* (Kaohsiung, Taiwan, 2003), IEEE Computer Society Press, p. 407–416.

[39] Cosnard, M., Jeannot, E., Rougeot, L. Low memory cost dynamic scheduling of large coarse grain task graphs. *Proceedings of the 12th. International Parallel Processing Symposium (IPPS'98)* (Orlando, EUA, Março 1998), IEEE Computer Society, p. 524–530.

[40] Cosnard, M., Trystram, D. *Parallel Algorithms and Architectures.* Thomson Learning, 1994.

[41] Costa, L. B., Feitosa, L., Araújo, Mendes, G. W. D., Coelho, R., Cirne, W., Fireman, D. Mygrid: A complete solution for running Bag-of-Tasks applications. *Proceedings of the 22nd Brazilian Symposium on Computer Networks - III Special Tools Session* (Gramado, Brasil, 2004).

[42] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K., Santos, E., Subramonian, R., von Eicken., T. LogP: Towards a realistic model of parallel computation. *Proceedings of the 4th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPOPP'93)* (San Diego, EUA, Julho 1993), ACM Press, p. 1–12.

[43] da Silva, D. P., Cirne, W., Brasileiro, F. V. Trading cycles for information: Using replication to schedule Bag-of-Tasks applications on computational grids. *Proceedings of the 9^{th} International Euro-Par Conference on Parallel Computing (EuroPar 2003)* (Klagenfurt, Áustria, Agosto 2003), Springer-Verlag, p. 169–180.

[44] da Silva, J. A., Rebello, V. E. F. Low cost self-healing in MPI applications. *Proceedings of the 14th European PVM/MPI User's Group Meeting* (Paris, França, Outubro 2007), Springer, p. 144–152.

[45] Directed acyclic graph manager. `http://www.cs.wisc.edu/condor/dagman`, last access 08/03/2007.

[46] de O. Lucchese, F., Yero, E. J. H., Sambatti, F. S., Henriques, M. A. A. An adaptive scheduler for grids. *Journal of Grid Computing 4*, 1 (Março 2006), 1–17.

[47] Dogan, A., Özgüner, F. LDBS: A duplication based scheduling algorithm for heterogeneous computing systems. *Proceedings of the International Conference on Parallel Processing (ICPP'02)* (Vancouver, Canadá, Agosto 2002), IEEE Computer Society Press, p. 352–359.

[48] Dongarra, J., Foster, I., Fox, G., Gropp, W., Kennedy, K., Torczon, L., White, A., Eds. *Source Book of Parallel Computing*. Morgan Kaufmann, 2003.

[49] Foster, I. *Designing and Programming Parallel Programs*. Addison-Wesley, 1995.

[50] Foster, I., Kesselman, C. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing 11*, 2 (Summer 1997), 115–128.

[51] Foster, I., Kesselman, C., Eds. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[52] Foster, I., Kesselman, C., Eds. *The GRID: Blueprint for a New Computing Infrastructure*. 2^a edição. Morgan Kaufmann, 2004.

[53] Foster, I., Kesselman, C., Tuecke, S. The anatomy of the Grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications 15*, 3 (Agosto 2001), 200–222.

[54] Frey, J., Tannenbaum, T., Livny, M., Foster, I., Tuecke, S. Condor-G: A computational management agent for multi-institutional grids. *Journal of Cluster Computing 3*, 5 (Julho 2002), 237–246.

[55] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., Woodall, T. S. Open MPI: Goals, concept, and design of a next generation MPI implementation. *Proceedings, 11th European PVM/MPI Users' Group Meeting* (Budapeste, Hungria, Setembro 2004), p. 97–104.

[56] Garey, M. R., Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., Nova Iorque, EUA, 1979.

[57] Gerasoulis, A., Yang, T. On the granularity and clustering of directed acyclic task graphs. *IEEE Transactions on Parallel and Distributed Systems 4*, 6 (1993), 686–701.

[58] Graham, R. L., Shipman, G. M., Barrett, B. W., Castain, R. H., Bosilca, G., Lumsdaine, A. Open MPI: A high-performance, heterogeneous MPI. *Proce-*

edings, Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (Barcelona, Espanha, Setembro 2006).

[59] The GRIDS Lab and the Gridbus Project. `http://www.gridbus.org/middleware`, last access 04/01/2008.

[60] Gridway metascheduler. `http://www.gridway.org/doku.php?id=about:schedcapab`, last access 27/01/2008.

[61] Grimshaw, A. S., Natrajan, A. Legion: Lessons learned building a grid operating system. *Proceedings of the IEEE, Special Issue on Grid Computing 93*, 3 (Março 2005), 589–603.

[62] Grimshaw, A. S., Wulf, W. A., Team, C. T. L. The Legion vision of a worldwide virtual computer. *Communications of the ACM 40*, 1 (Janeiro 1997), 39–45.

[63] Gualandris, A., Zwart, S. P., Tirado-Ramos, A. Performance analysis of direct N-body algorithms for astrophysical simulations on distributed systems. *Parallel Computing 33*, 3 (2007), 159–173.

[64] He, L., Jarvis, S., Spooner, D., Bacigalupo, D., Tan, G., Nudd, G. Mapping DAG-based applications to multiclusters with background workload. *Proceedings of 5th International Symposium on Cluster Computing and the Grid (CCGrid 2005)* (Cardiff, Reino Unido, Maio 2005), IEEE Computer Society Press, p. 855–862.

[65] Hong, B., Prasanna, V. Adaptive allocation of independent tasks to maximize throughput. *IEEE Transactions on Parallel and Distributed Systems 18*, 10 (2007), 1420–1435.

[66] Horn, P. Autonomic computing: IBM's perspective on the state of information technology. `http://www.research.ibm.com/autonomic`, Outubro 2001.

[67] Horowitz, E., Sahni, S. Exact and approximate algorithms for scheduling non-identical processors. *Journal of the ACM (JACM) 23*, 2 (Abril 1976), 317–327.

[68] Huedo, E., Montero, R. S., Llorente, I. M. Experiences on adaptive grid scheduling of parameter sweep applications. *Proceedings of the 12th Euromicro Conference of Parallel, Distributed and Network-Based Processing (PDP 2004)* (La Coruña, Espanha, Fevereiro 2004), IEEE Computer Society Press, p. 28–33.

[69] Huedo, E., Montero, R. S., Llorente, I. M. The GridWay framework for adaptive scheduling and execution on grids. *Scalable Computing: Practice and Experience 6*, 3 (Setembro 2005), 1–8.

[70] Ibarra, O. H., Kim, C. E. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Journal of the ACM 24*, 2 (1977), 280–289.

[71] Jung, H., Kirousis, L. M., Spirakis, P. Lower bounds and efficient algorithms for multiprocessor scheduling of directed acyclic graphs with communication delays. *Information and Computation 105*, 1 (1993), 94–104.

[72] Kalinowski, T., Kort, I., Trystram, D. List scheduling of general task graphs under LogP. *Parallel Computing 26*, 9 (2000), 1109–1128.

[73] Karniadakis, G. E., Kirby, R. M., Eds. *Parallel Scientific Computing in C++ and MPI*. Cambridge University Press, 2003.

[74] Kephart, J., Chess, D. The vision of autonomic computing. *IEEE Computer Magazine 36*, 1 (Janeiro 2003), 41–50.

[75] Krauter, K., Buyya, R., Maheswaran, M. A taxonomy and survey of grid resource management systems for distributed computing. *Software: Practice and Experience 32*, 2 (2002), 135–164.

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

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

[78] LAM/MPI Parallel Computing. `http://www.lam-mpi.org/`, last access 01/11/2007.

[79] Lima, A. Escalonamento híbrido de tarefas: Integração de heurísticas estáticas e dinâmicas. Dissertação de Mestrado, Instituto de Computação, Universidade Federal Fluminense, 2003.

[80] Litzkow, M., Livny, M., Mutka, M. Condor - a hunter of idle workstations. *Proceedings 8th International Conference on Distributed Computing Systems* (San José, EUA, Junho 1988), IEEE Computer Society Press, p. 104–111.

[81] Ma, T., Buyya, R. Critical-path and priority based algorithms for scheduling workflows with parameter sweep tasks on global grids. *Proceedings of the 17th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2005)* (Rio de Janeiro, Brasil, Outubro 2005), IEEE Computer Society, p. 251–258.

[82] Maheswaran, M., Ali, S., Siegel, H., Hensgen, D., Freund, R. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. *Proceedings of the 8th Heterogeneous computing Workshop (HCW´99)* (San Juan, Porto Rico, Abril 1999), IEEE Computer Society Press, p. 30–40.

[83] Maheswaran, M., Siegel, H. A dynamic matching and scheduling algorithm for heterogeneous computing systems. *Proceedings of the 7th Heterogeneous Computing Workshop (HCW98)* (Orlando, EUA, Março 1998), IEEE Computer Society Press, p. 57–69.

[84] Maki-Turja, J., Hanninen, K., Nolin, M. Efficient development of real-time systems using hybrid scheduling. *International conference on Embedded Systems and Applications (ESA'05)* (Las Vegas, EUA, Junho 2005), p. 53–59.

[85] Massie, M. L., Chun, B. N., Culler, D. E. The Ganglia distributed monitoring system: Design, implementation and experience. *Parallel Computing 30*, 7 (2004), 817–840.

[86] Mendes, H. A. HLogP: Um modelo de escalonamento para a execução de aplicações MPI em grades computacionais. Dissertação de Mestrado, Instituto de Computação, Universidade Federal Fluminense, 2004.

[87] Message Passing Forum. MPI: A Message Passing Interface. Technical report, University of Tennessee, 1995.

[88] MPICH-G2. `http://www3.niu.edu/mpi/`, last access 01/11/2007.

[89] Nascimento, A. P., Sena, A. C., Boeres, C., Rebello, V. E. F. Distributed and dynamic self-scheduling of parallel MPI grid applications. *Concurrency and Computation: Practice and Experience 19*, 14 (2007), 1955–1974.

[90] Nascimento, A. P., Sena, A. C., da Silva, J. A., Vianna, D. Q. C., Boeres, C., Rebello, V. Managing the execution of large scale MPI applications on computational grids. *Proceedings of the 17th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2005)* (Rio de Janeiro, Brasil, Outubro 2005), IEEE Computer Society, p. 69–76.

[91] Nascimento, A. P., Sena, A. C., da Silva, J. A., Vianna, D. Q. C., Boeres, C., Rebello, V. Autonomic application management for large scale MPI programs. *International Journal of High Performance Computing and Networking* (2008). (A ser publicado).

[92] Norman, M., Chochia, G., Thanisch, P., Issman, E. Predicting the performance of the diamond dag computation. *Technical Report EPCC-TR92-07* (1992).

[93] Nudd, G. R., Kerbyson, D. J., Papaefstathiou, E., Perry, S. C., Harper, J. S., Wilcox, D. V. PACE – A toolset for the performance prediction of parallel and distributed systems. *The International Journal of High Performance Computing Applications 14*, 3 (Agosto 2000), 228–251.

[94] Pacheco, P. S., Ed. *Parallel Programming with MPI.* Morgan Kaufmann, 1997.

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

[96] Parashar, M., Salim, H. Autonomic computing: An overview. *Proceedings of the Unconventional Programming Paradigms: International Workshop (UPP 2004)* (Le Mont St-Michel, França, 2005), Springer, p. 247–259.

[97] Plastino, A. *Balanceamento de Carga para Aplicações Paralelas SPMD.* Tese de Doutorado, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, 2000.

[98] Prodan, R., Fahringer, T. Dynamic scheduling of scientific workflow applications on the grid: A case study. *Proceedings of the 2005 ACM symposium on Applied computing (SAC'05)* (Santa Fé, EUA, Março 2005), ACM Press, p. 687–694.

[99] Qin, X., Jiang, H. A dynamic and reliability-driven scheduling algorithm for parallel real-time jobs executing on heterogeneous clusters. *Journal of Parallel and Distributed Computing 65*, 8 (2005), 885–900.

[100] Quinn, M. J. *Parallel computing: theory and practice.* 2^a edição. McGraw-Hill, Inc., Nova Iorque, EUA, 1994.

[101] Ranaweera, S., Agrawal, D. A task duplication based scheduling algorithm for heterogeneos systems. *International Parallel and Distributed Processing Symposium* (Cancun, México, Maio 2000), p. 445–450.

[102] Rodrigues, H., Trannin, H., Sena, A. C., Rebello, V. Usando grades computacionais para avaliação de heurísticas de escalonamento de tarefas. *Proceedings of the VI Workshop em Sistemas Computaionais de Alto Desempenho (WSCAD 2005)* (Rio de Janeiro, Brasil, Outubro 2005), M. C. S. de Castro e E. Midorikawa, Eds., p. 218–221.

[103] Sena, A. *Um Modelo de Execução para Aplicações MPI em Ambientes Grid.* Tese de Doutorado, Instituto de Computação, Universidade Federal Fluminense, 2008. (A ser apresentada).

[104] Sena, A. C., Nascimento, A. P., da Silva, J. A., Vianna, D. Q. C., Boeres, C., Rebello, V. E. F. On the advantages of an alternative MPI execution model for grids. *Proceedings of 7th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007)* (Rio de Janeiro, Brasil, Maio 2007), IEEE Computer Society, p. 575–582.

[105] Sinnen, O., Ed. *Task Scheduling for Parallel Systems*. Wiley, 2007.

[106] Souto, H., da Silveira Filho, O., Moyne, C., Didierjean, S. Thermal dispersion in porous media: Computations by the random walk method. *Journal of Computational and Applied Mathematics 21*, 2 (2002), 513–544.

[107] Sterritt, R., Parashar, M., Tianfield, H., Unland, R. A concise introduction to autonomic computing. *Advanced Engineering Informatics 19*, 3 (Julho 2005), 181–187.

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

[109] Ullman, J. D. NP-complete scheduling problems. *Journal of Computer and System Sciences 10*, 3 (1975), 384–393.

[110] Vadhiyar, S., Dongarra, J. Self-adaptivity in grid computing. *Concurrency and Computation: Practice and Experience 17*, 2-4 (Fevereiro 2005), 235–257.

[111] Venugopal, S., Buyya, R., Winton, L. A grid service broker for scheduling e-science applications on global data grids. *Concurrency and Computation: Practice and Experience 18*, 6 (2006), 685–699.

[112] Venugopal, S., Nadiminti, K., Gibbins, H., Buyya, R. Designing a resource broker for heterogeneous grids. *CoRR - The Computer Research Repository abs/cs/0702145* (Fevereiro 2007).

[113] Vianna, D. Q. C. Um sistema de gerenciamento de aplicações MPI para ambientes grid. Dissertação de Mestrado, Instituto de Computação, Universidade Federal Fluminense, 2005.

[114] Wolski, R., Spring, N., Hayes, J. Predicting the CPU availability of time-shared unix systems on the computational grid. *Journal of Cluster Computing 3*, 4 (Outubro 2000), 293–301.

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

[116] YANG, T., GERASOULIS, A. List scheduling with and without communication delays. *Parallel Computing 19*, 12 (1993), 1321–1344.

[117] YU, J., BUYYA, R. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing 3*, 3-4 (2005), 171–200.

[118] ZOMAYA, A. Y., TEH, Y.-H. Observations on using genetic algorithms for dynamic load-balancing. *IEEE Transactions on Parallel and Distributed Systems 12*, 9 (Setembro 2001), 899–911.