**UNIVERSIDADE FEDERAL DE SÃO CARLOS**
**CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA**
**PROGRAMA DE PÓS-GRADUAÇÃO EM**
**CIÊNCIA DA COMPUTAÇÃO**

# SLOT: uma ferramenta dinâmica para escalonamento global de aplicações em Grades Computacionais

Ricardo Araújo Rios

**São Carlos – SP**
**Maio/2008**

**Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar**

R586sf

Rios, Ricardo Araújo.
SLOT: uma ferramenta dinâmica para escalonamento global de aplicações em grades computacionais / Ricardo Araújo Rios. -- São Carlos : UFSCar, 2008.
115 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2008.

1. Grade computacional. 2. Escalonamento. 3. Gerenciador de recursos e aplicações. I. Título.

CDD: 004.36 (20ª)

# Universidade Federal de São Carlos

## Centro de Ciências Exatas e de Tecnologia

### Programa de Pós-Graduação em Ciência da Computação

# *"SLOT: uma ferramenta dinâmica para escalonamento global de Aplicações em Grades Computacionais"*

RICARDO ARAÚJO RIOS

**Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.**

**Membros da Banca:**

Hélio Crestana Guardia

**Prof. Dr. Hélio Crestana Guardia**
**(Orientador – DC/UFSCar)**

V. Rebello

**Prof. Dr. Eugene Francis Vinod Rebello**
**(IC/UFF)**

**Prof. Dr. Luis Carlos Trevelin**
**(DC/UFSCar)**

**São Carlos**
**Maio/2008**

# Agradecimentos

Agradeço inicialmente à Deus pela força e pela disposição para concluir o mestrado.

Agradeço ao meu orientador, prof. Hélio Guardia, pela oportunidade de fazer o mestrado, pela paciência, por me ensinar a pesquisar, ler, implementar, analisar e escrever como pesquisador e, principalmente, pela amizade que ultrapassaram limites entre orientador e aluno.

Agradeço à minha noiva Tati por compreender os tantos finais de semana que deixamos de nos ver, sendo que enquanto deveríamos estar juntos como um casal normal, eu estava longe em companhia da dupla Linux e gcc. Gostaria de agradecê-la ainda pelas revisões dos textos da qualificação, da dissertação e dos artigos, pela sabedoria nos momentos em que eu apenas reclamava das coisas que não davam certo, pela calma nos estressantes momentos antes dos "deadlines". Puxa, obrigado por ainda estarmos juntos!

Agradeço ao meus amados pais e irmãos, que mesmo sem compreender completamente as dificuldades enfrentadas no mestrado, estiveram ao meu lado sempre me dando força. Agradeço aos meus queridos avós pelas orações e, nos momentos difíceis, a certeza que torciam por mim. Ao meu tio Jair pela preocupação, empenho e suporte financeiro e aos demais familiares por estarem disponíveis sempre que precisei e torcendo sempre.

Agradeço também à amiga Daniele Santini pela parceria no desenvolver do projeto e no companheirismo nas implementações, nas instalações de programas, nas escritas de artigos e da dissertação. Aos grandes amigos do laboratório GSDR e de disciplinas Bruno Kimura, Paulo Cereda e Reginaldo Gotardo. Aos incomparáveis

amigos do Grupo de Estudos: Márcio Roberto, Alexandre Mello, Cristiane Oliveira (e ao Gui!), Simone Borges, Vinícius Durelli, Leonardo Botega, Eliane Nascimento, Débora Corrêa, e, os já citados, Reginaldo Gotardo e Tatiane Nogueira.

Agradeço ainda aos amigos da república Cássio Prazeres, Maycon Leone, Leonardo Campos, Hélio Beloto e Fernando Panont pelas calorosas discussões sobre livros, filmes, política, futebol e etc, que serviram para desviar um pouco a pressão do mestrado.

Ah! Por fim, gostaria de agradecer ainda à máquina 'xeon', que apesar de dar muito trabalho no começo foi bastante cooperativa no final do projeto, e à todos os nós da grade um grande: cluster-fork "echo 'Thanks!' ". À máquina 'rembrandt' um muitíssimo obrigado e ao seu teclado: 'Sinceramente, me desculpe'.

"Que grande livro se poderia
escrever com o que se sabe.
Outro muito maior se escreveria
com o que não se sabe."
Jules Verne

# Resumo

A melhoria constante de desempenho que os computadores e as redes de interconexão vêm apresentando favoreceu o uso de recursos computacionais distribuídos, dando origem à Computação em Grade. Esta nova abordagem utiliza recursos heterogêneos e geograficamente distribuídos, a fim de resolver problemas de grande custo computacional. A execução de aplicações neste ambiente geralmente é realizada por meio de mecanismos de escalonamento que manipulam os conjuntos de tarefas e suas interdependências, mapeando-as de forma eficiente nos recursos. Contudo, os escalonadores existentes atualmente realizam o escalonamento de cada aplicação individualmente, deixando de avaliar o impacto na execução de aplicações previamente escalonadas. Neste sentido, este trabalho apresenta uma ferramenta de escalonamento global das tarefas submetidas para a Grade e apresenta ainda um algoritmo de escalonamento que aloca as tarefas em fatias de tempo livre entre tarefas previamente escalonadas. A utilização da ferramenta e do algoritmo propostos permite a redução dos períodos de tempo ociosos nos processadores e a execução das aplicações de forma mais eficiente quando comparado com algoritmos tradicionais.

**Palavras-chave**: Grade Computacional, Escalonamento Dinâmico, Gerenciador de Recursos e Aplicações.

# Abstract

The constant improvement in performance that computers and interconnection networks present has favored the use of distributed computational resources, and given rise to Grid Computing. This new approach uses heterogeneous and geographically distributed resources to resolve problems with high computational costs. The execution of applications in this environment is generally achieved with scheduling mechanisms that manipulate the task set and its interdependences, mapping the tasks on to the resources. However, existing schedulers generate the schedule of each application individually, not evaluating the impact on the execution of previously scheduled applications. In this sense, this work presents a global scheduling tool for the tasks submitted to the Grid, and also presents a scheduling algorithm that allocates tasks between previously scheduled slots of time. The use of the proposed tool and algorithm permit a reduction in the amount of time processors remain idle and therefore a more efficient execution of the applications.

**Keywords**: Grid Computing, Dynamic Scheduling, Resources and Applications Management Systems.

# Sumário

# Lista de Figuras

# Lista de Tabelas

# Lista de Abreviaturas

**AEST**: *Absolute Earliest Start Time*

**ALST**: *Absolute Latest Start Time*

**AMS**: *Application Management System*

**AppLeS**: *Application Level Scheduling*

**BoT**: *Bag-of-Tasks*

**CA**: *Certification Authority*

**CCR**: *Communication-to-Computation Ratio*

**CPOP**: *Critical-Path-on-a-Processor*

**DAG**: *Directed Acyclic Graph*

**DCP**: *Dynamic Critical Path*

**DLS**: *Dynamic Level Scheduling*

**DNS**: *Domain Name Service*

**DOROC**: *Dynamically-Updated Request Online Coallocator*

**DRS**: *Data Replication Service*

**EFT**: *Earliest Finish Time*

**FCP**: *Fast Critical Path*

**FIFO**: *First-In-First-Out*

**FTP**: *File Transfer Protocol*

**GIIS**: *Grid Index Information Service*

**GrADS**: *Grid Application Development Software*

**GRAM**: *Grid Resource Allocation and Management*

**GRIS**: *Grid Resource Information Service*

**GSI**: *Grid Security Infrastructure*

**HEFT**: *Heterogenous Earliest Finish Time*

**HTTP**: *HyperText Transfer Protocol*

**LAN**: *Local Area Network*

**LDAP**: *Lightweight Directory Access Protocol*

**MDS**: *Monitoring and Discovery Service*

**MPI**: *Message-Passing Interface*

**P2P**: *peer-to-peer*

**PKI**: *Public Key Infrastructure*

**QoS**: *Quality of Service*

**RFT**: *Reliable File Transfer*

**RLS**: *Replica Location Service*

**RMS**: *Resource Management System*

**RSL**: *Resource Specification Language*

**SLOT**: *Scheduling Lots Of Tasks*

**SOAP**: *Simple Object Access Protocol*

**SRS**: *Stop Restart Software*

**SSL**: *Secure Socket Layer*

**TLS**: *Transport Layer Securit*

**VO**: *Virtual Organizations*

**WAN**: *Wide-Area Network*

**XML**: *eXtensible Markup Language*

CAPÍTULO 1

# Introdução

## 1.1 Contextualização

Pesquisadores de diversas áreas têm implementado aplicações mais robustas, que necessitam de computadores com alto poder computacional para obter resultados com menores tempos de execução possíveis. São exemplos de programas que exigem alto desempenho computacional: aplicações que movimentam grandes quantidades de dados, realizam simulações financeiras ou executam cálculos científicos complexos.

Muitas pesquisas foram direcionadas para o desenvolvimento de supercomputadores com o objetivo de tentar reduzir o tempo de execução das aplicações. Estas máquinas eram compostas por diversos processadores interconectados por mecanismos de comunicação de alta velocidade. Alguns dos problemas encontrados na utilização destas máquinas são o alto custo financeiro para implantação das mesmas e a dificuldade em adicionar novos elementos de processamento, caso sejam executadas aplicações que exijam maior poder computacional do que o existente.

Sendo assim, os sistemas distribuídos ganharam destaque, principalmente, por serem mais viáveis financeiramente e por possuírem uma maior facilidade em adicionar ou remover recursos. Nestes sistemas, aplicações são executadas em diferentes máquinas interconectadas através de uma rede de computadores, com o objetivo de resolver um determinado problema em comum.

Sistemas formados por computadores distribuídos, interconectados através de uma rede local são chamados de *Clusters*. A utilização deste meio de comunicação impôs uma limitação, impedindo o uso de computadores externos, restringindo o número de computadores disponíveis e, conseqüentemente, a capacidade de processamento total.

Devido ao grande aumento da capacidade de processamento e de armazenamento das máquinas convencionais e ao aumento da largura de banda das redes de computadores, pesquisadores têm dedicado esforços para desenvolver uma infra-estrutura de compartilhamento para permitir um melhor aproveitamento destes recursos. Essa infra-estrutura é conhecida pelo termo de **Grade Computacional** ou simplesmente **Grade**.

Uma das diferenças entre o *cluster* e a Grade Computacional está no meio de comunicação utilizado entre os recursos. A Grade, geralmente, interconecta computadores através de redes não locais, permitindo que recursos de diversas localidades possam ser adicionados ou removidos, aumentando a heterogeneidade. Outra diferença significativa está relacionada à existência de múltiplos domínios administrativos, de forma que é preciso criar uma unidade lógica que defina os participantes da Grade.

As máquinas que fazem parte da infra-estrutura de Grade são agrupadas dinamicamente em **Organizações Virtuais**, regidas por políticas de acesso, que permitem ceder e utilizar recursos computacionais, como por exemplo, processamento e espaço disponível em disco.

A principal vantagem em implementar o ambiente de Grade reside na possibilidade de agregar *poder* computacional independente da localização, podendo obter

um desempenho equivalente ou superior ao uso de supercomputadores para certas classes de aplicações e com custos mais baixos.

## 1.2 Motivação

A heterogeneidade e o dinamismo dos recursos de processamento e dos canais de comunicação que os interligam tornam o escalonamento de aplicações em ambientes de Grade Computacional uma atividade complexa. Para resolver este problema, é necessária a utilização de mecanismos de *escalonamento*, que tem por objetivo gerenciar os recursos e os programas desenvolvidos para a Grade.

De forma geral, estes mecanismos de escalonamento visam a alocar o máximo de tarefas possíveis nos recursos de forma eficiente para diminuir o tempo de resposta dos programas.

O escalonamento na Grade pode ser dividido em dois grupos. O primeiro, chamado **Sistema Gerenciador de Recursos**, realiza o escalonamento de aplicações de acordo com as características dos recursos, fornecendo um balanceamento de carga e maximizando a utilização dos recursos. O segundo grupo é chamado de **Sistema Gerenciador de Aplicações** e realiza o escalonamento de acordo com as características de cada aplicação. AMS pode trabalhar juntamente com RMS, submetendo e gerenciando as aplicações na Grade (Boeres et al., 2005).

O escalonamento é realizado tomando como base as características dos recursos e das aplicações, realizando um mapeamento entre as tarefas e os recursos disponíveis. Os diversos algoritmos de escalonamento podem ser classificados como **estáticos**, **dinâmicos** ou **híbridos**.

Os algoritmos de escalonamento são ditos estáticos quando as informações obtidas a respeito dos recursos e das aplicações são coletadas antes da execução. São modelos mais simples de serem implementados, porém estão sujeitos a sofrer maiores interferências, porque não são levadas em consideração as alterações ocorridas no sistema em tempo de execução.

Os algoritmos de escalonamento dinâmico analisam mudanças ocorridas nos recursos durante a execução das tarefas e, portanto, têm potencial para fornecer melhores resultados. São sistemas mais complexos de serem implementados e requerem informações atualizadas e específicas de cada recurso e de cada aplicação, podendo efetuar medidas corretivas no escalonamento realizado, a fim de obter um melhor desempenho na execução das aplicações.

Existem ainda algoritmos que realizam inicialmente um escalonamento estático e, durante a execução da aplicação, utilizam técnicas dinâmicas, usufruindo das vantagens oferecidas pelos dois métodos. Esse tipo de algoritmo é chamado de algoritmo de escalonamento híbrido.

Além dos algoritmos de escalonamento, diversas ferramentas existem para realizar o gerenciamento de aplicações e recursos na Grade. Contudo, tanto os algoritmos quanto as ferramentas, geralmente, gerenciam cada aplicação individualmente. Além disto, durante o escalonamento são levados em consideração apenas o estado atual dos recursos ou a carga média em cada processador.

Nesse sentido, este trabalho está relacionado com o desenvolvimento de uma ferramenta de escalonamento, chamada **SLOT** (*Scheduling Lots Of Tasks*), que usa um algoritmo de escalonamento dinâmico, capaz de adaptar-se às mudanças ocorridas no ambiente, a fim de realizar, de forma eficiente, um escalonamento global de todas as tarefas submetidas para a Grade.

Para alcançar esse objetivo, a ferramenta inicialmente monitora e coleta informações sobre os recursos disponíveis, criando uma topologia dinâmica da Grade em tempo de execução. Em seguida, tendo conhecimento de características das aplicações a serem executadas, a ferramenta realiza um escalonamento considerando o instante de início (*start time*) e de conclusão (*finish time*) das tarefas de cada aplicação a ser executada. Para cada tarefa é realizada uma busca por um espaço livre (*slots*) nos processadores entre o *start time* e o *finish time* de duas tarefas previamente escalonadas e, então, é realizado o mapeamento desta tarefa para o processador que possua o *slot* capaz de executar a tarefa no menor tempo

possível. Por considerar o instante entre cada tarefa para alocação de uma nova tarefa, esse algoritmo foi denominado ***slot***.

Ainda considerando o escalonamento de aplicações na Grade, este trabalho apresenta um estudo sobre uma variação do algoritmo proposto. Esta variação permite uma flexibilidade na utilização dos *slots*, para que, por um determinado instante, duas tarefas compartilhem o uso de um processador. Para isto, o algoritmo de escalonamento realiza uma sobreposição de *slots* ocupados por outras tarefas até um limite máximo definido previamente.

Para validação da proposta, a ferramenta foi executada com aplicações desenvolvidas para a Grade, no ambiente de Grade real, e com aplicações simuladas, em ambientes de Grade simulados.

Pretende-se, com a execução da ferramenta, do algoritmo de escalonamento *slot* e do algoritmo de sobreposição de *slots*, aumentar o desempenho na execução das aplicações na Grade e tentar eliminar ao máximo a ociosidade dos recursos, diminuindo as fatias de tempo inutilizadas entre a finalização de uma tarefa e o início de outra.

## 1.3 Organização da Dissertação

A dissertação está organizada em outros sete capítulos. No próximo capítulo será apresentada uma visão geral do conceito de Grade, explicando os principais componentes e a arquitetura dos protocolos utilizados.

No terceiro capítulo serão apresentados modelos de escalonamento em Grades, apresentando alguns dos principais algoritmos de escalonamento na Grade, as políticas de seleção de recursos e aplicações e algumas das principais ferramentas utilizadas atualmente na Grade.

No quarto capítulo são apresentados os algoritmos de escalonamento *slot* e o algoritmo de escalonamento baseado na sobreposição de *slots* propostos neste trabalho.

No quinto capítulo são apresentadas as etapas do funcionamento da ferramenta proposta neste trabalho que serviu de base para execução dos algoritmos propostos e os algoritmos utilizados na validação.

No sexto capítulo são apresentados o simulador e as ferramentas desenvolvidos para verificar o comportamento dos algoritmos propostos em diferentes situações.

O sétimo capítulo apresenta os experimentos realizados para validar toda a proposta. Por fim, no capítulo 8 são apresentadas as conclusões obtidas pelos experimentos realizados, as contribuições e os possíveis trabalhos futuros.

CAPÍTULO

2

# Grades Computacionais

O Conceito de Grade Computacional (Grid Computing), ou simplesmente Grade, teve origem em um trabalho pioneiro desenvolvido no início da década de 90 nos Estados Unidos que visava conectar, por meio de uma rede de alta velocidade, instituições localizadas na Califórnia e no Novo México. Para isto, foram desenvolvidos programas de alto custo computacional, aplicações maciçamente paralelas e bibliotecas de multimídia digital, para testar e demonstrar a capacidade de execução deste novo ambiente (Foster, 2002).

## 2.1 Conceito

O aprimoramento dos recursos computacionais[1], a popularização da Internet e a redução dos custos das redes de computadores de alta largura de banda têm favorecido o surgimento de uma nova subárea dentro de Sistemas Distribuídos. Chamada de Computação em Grade, essa forma de computação tem como principal

[1]Um recurso pode ser definido como uma entidade que representa, por exemplo, um processador, um canal de comunicação ou um disco rígido.

objetivo interconectar recursos remotos, de acordo com a sua disponibilidade.

Segundo Lathia (2005), o conceito de *Grade* pode ser definido como um conjunto de padrões e tecnologias que acadêmicos, pesquisadores e cientistas em todo o mundo estão desenvolvendo para permitir que organizações tenham um aumento do poder de processamento e da capacidade de armazenamento.

O ambiente de Grade é caracterizado por um conjunto de recursos, que podem estar dispersos geograficamente, com uma grande variedade de tipos de *hardware*, de Sistemas Operacionais e de canais de comunicação que conectam cada recurso. Diante deste ambiente altamente heterogêneo, o principal objetivo desta metodologia é transmitir para o usuário final a impressão de que está sendo utilizado um grande e poderoso computador virtual homogêneo (Ferreira et al., 2003).

Utilizando estes conceitos, Baker et al. (2002) definem Grade ainda como sendo uma infra-estrutura de *hardware* e *software*, que fornece acesso confiável, consistente e transparente, permitindo o compartilhamento, a busca e a descoberta dos mais variados recursos distribuídos, sejam eles computacionais ou de armazenamento.

Segundo Ferreira et al. (2003), na maioria das empresas, uma grande quantidade de computadores ficam a maior parte do tempo ociosas, inutilizadas por mais de 95% do tempo total. Durante este tempo, tarefas que estão na fila pronto para serem executadas poderiam utilizar estas máquinas.

Para solucionar este problema, existem ferramentas capazes de localizar e explorar recursos ociosos, agregando o poder computacional na execução de uma aplicação submetida para a Grade.

Uma outra razão para utilização da Grade é a alta disponibilidade dos sistemas, garantindo que um determinado serviço estará sempre disponível e não rejeitará uma solicitação cliente. Para esse fim, a Grade pode ser implementada de tal forma que seja tolerante a falhas. Quando qualquer problema for detectado, as tarefas submetidas para um recurso faltoso podem ser reenviadas para um outro geograficamente separado.

Diante das situações supracitadas, a Grade pode ser subdividida em diferentes categorias (Ferreira et al., 2003). Ela é dita **computacional** quando explora a capacidade de processamento dos recursos para resolver problemas de grande custo computacional. Para este fim, uma máquina pode submeter uma aplicação para uma outra máquina com maior poder computacional, ou ainda subdividir a aplicação em tarefas e enviá-las a recursos distintos, explorando a escalabilidade e o paralelismo da aplicação.

Uma Grade pode ser categorizada ainda como **Grade de Dados** (*Data Grid*) ou **Grade de Comunicação**. No primeiro caso, cada máquina disponibiliza uma quantidade de espaço disponível em disco para armazenagem dos dados. Essa abordagem é utilizada tanto quando se tem uma grande quantidade de dados para armazenar quanto para garantir um nível de segurança maior, replicando dados em diversos repositórios (recursos). Já a Grade de Comunicação pode ser utilizada para fornecer diversos canais de comunicação a fim de melhorar o tráfego de dados e solucionar problemas de falhas na rede.

Nas seções posteriores serão apresentados outros conceitos relacionados e necessários para o desenvolvimento de aplicações e ferramentas que executem sobre a Grade.

## 2.2 Sistemas de Gerenciamento

Conforme dito anteriormente, o grande número de máquinas e os diferentes meios de comunicação fazem da Grade um ambiente altamente heterogêneo, dificultando o gerenciamento de recursos e aplicações. Por esta razão, foram desenvolvidos sistemas de gerenciamento com o objetivo de tornar o ambiente de Grade Computacional mais simples de ser utilizado por parte dos usuários. Esses sistemas baseiam-se em funções específicas para realizar o escalonamento de recursos e aplicações e dividem-se basicamente em **Sistemas Gerenciadores de Recursos** (*Resource Management Systems* - RMS) e em **Sistemas Gerenciadores de Aplica-**

**ções** (*Application Management Systems* - AMS).

Os Sistemas Gerenciadores de Recursos são mecanismos que visam a otimização da operação dos recursos e fornecem três serviços básicos: (1) disseminação de informações dos recursos na Grade, (2) descoberta dos recursos disponíveis na Grade e (3) escalonamento de tarefas para a execução nos recursos (Krauter et al., 2002).

De forma geral, RMS gerenciam a autenticação e a autorização no acesso aos recursos disponíveis. Para que o gerenciamento possa ser realizado de forma eficiente, é importante que os RMS possam ser capazes de predizer o impacto da alocação de uma tarefa em uma determinada máquina, evitando que as exigências de Qualidade de Serviço (*Quality of Service* - QoS[2]) sejam afetadas. Uma solução para atingir este objetivo é garantir que o RMS tenha conhecimento do histórico das tarefas submetidas para a Grade (Krauter et al., 2002).

Quanto ao escalonamento dos recursos, RMS podem ser classificados como **centralizados**, **hierárquicos** ou **descentralizados** (Krauter et al., 2002). No modelo centralizado, todas as requisições de execução são enviadas para uma máquina controladora que gerencia todos os recursos. No modelo hierárquico, o ambiente é composto por mais de uma máquina controladora que são organizadas hierarquicamente. Neste modelo, controladores localizados em um nível hierárquico mais alto, chamados de controladores globais, controlam o escalonamento em um grande número de máquinas. Controladores que se encontram em um nível mais baixo escalonam um subgrupo destas máquinas e esta subdivisão acontece até que uma máquina individual seja escalonada. Na abordagem descentralizada, não existe uma máquina controladora, cada máquina determina a alocação e o escalonamento dos seus recursos.

Sendo assim, um escalonador de recursos na Grade tem por objetivo selecionar para execução as tarefas a ele atribuídas, de acordo com a carga nos recursos, a

[2]Neste caso, o conceito de QoS refere-se às redes de comunicação e aos mecanismos de processamento e de armazenagem em disco.

fim de obter um desempenho adequado na execução das aplicações (Legrand et al., 2003).

Por outro lado, os AMS utilizam algoritmos de escalonamento que baseiam-se nas características de cada aplicação. Estes algoritmos analisam cada tarefa, considerando características como, dependências de comunicação, tempo previsto para conclusão da execução da tarefa e custo computacional, com o objetivo de diminuir o tempo de resposta das aplicações (Dong;Akl, 2006).

Os AMS são compostos por ferramentas independentes que são utilizadas para realizar a alocação de tarefas nos recursos. Porém, Boeres et al. (2005) apresentam um AMS que é embutido na aplicação em tempo de execução. Este AMS substitui as funções de processamento paralelo em Grade convencionais e realiza a alocação das tarefas para os recursos de forma transparente para o usuário. Este tipo de sistema oferece uma maior portabilidade, pois são eliminadas as dependências de outros *middlewares* específicos (Vianna, 2005).

A Figura 2.1 mostra os níveis em que cada sistema gerenciador, RMS e AMS, se encontra na arquitetura da Grade. No nível mais elevado, e em contato direto com o usuário, estão os gerenciadores de aplicação. Como pode ser visto, os AMS, geralmente, utilizam RMS para descobrir e monitorar os recursos disponíveis.

Ainda em relação à Figura 2.1, no nível abaixo em relação aos AMS estão os gerenciadores de recursos. Os RMS manipulam um ou mais recursos na Grade permitido a utilização de acordo com as políticas definidas por cada organização, conforme será apresentado na Seção 2.3.

## 2.3 Organização Virtual

Assim como citado anteriormente, um dos principais focos da Grade é gerenciar recursos em larga escala. Porém, esta tarefa deve ser feita de forma flexível e segura. A flexibilidade no gerenciamento dos recursos é importante, principalmente, porque os recursos não são dedicados, ou seja, não são de uso exclusivo da Grade.

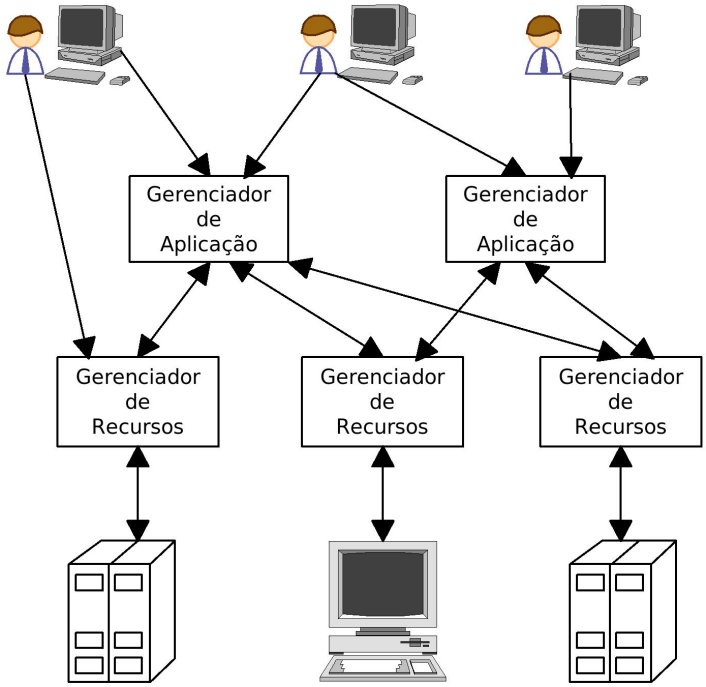


**Figura 2.1:** Processo de escalonamento utilizando RMS e AMS (Jacinto, 2006).

Geralmente, são recursos individuais (por exemplo, computadores pessoais) ou fazem parte de uma determinada instituição.

Portanto, Foster et al. (2001) apresentam o conceito de **Organização Virtual** (*Virtual Organization* - VO), que possui mecanismos de descoberta de recursos remotos e de autenticação e autorização de usuários para utilização dos recursos. O principal objetivo de uma VO é coordenar recursos individuais e/ou recursos disponibilizados por uma determinada instituição, que estão definidos pela mesma política de acesso independente da localização geográfica.

O conjunto de recursos pertencentes a uma VO podem ser gerenciado por um ou mais RMS, assim como um RMS pode gerenciar uma ou mais VOs (Foster et al., 2001). Quando um novo recurso junta-se a uma VO, os RMS devem ser capazes de determinar a disponibilidade desse recurso e as políticas que regem seu acesso (Foster et al., 2001).

Organizações Virtuais podem ser implementadas estaticamente, quando os recursos são disponibilizados de acordo com o conhecimento prévio dos usuários de

uma organização, ou de forma dinâmica, que permite que recursos juntem-se a VOs existentes e automaticamente informem qual recurso desejam compartilhar e sob qual política de acesso.

Levando em consideração a necessidade de ferramentas e mecanismos que auxiliem na criação de VOs e de mecanismos de gerenciamento dos recursos e das aplicações, Foster et al. (2001) apresentam uma arquitetura para a Grade como pode ser vista na Seção 2.4.

## 2.4 Arquitetura

Foster et al. (2001) sugerem uma arquitetura para a Grade, que agrupa componentes com características comuns em camadas, as quais fornecem comportamentos e capacidades para as camadas superiores. A Figura 2.2, apresenta essa arquitetura que, como pode ser visto, divide-se em 5 camadas.

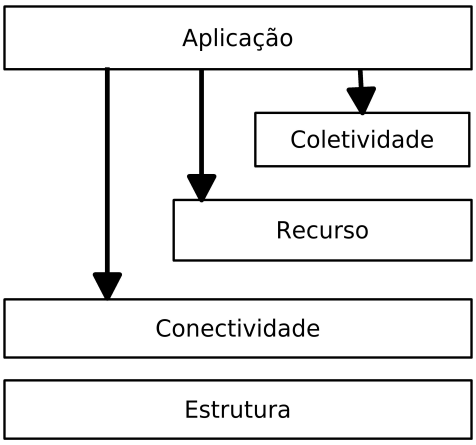


**Figura 2.2:** Arquitetura da Grade definida por Foster et al. (2001).

A camada de **Estrutura** constitui a base da arquitetura e fornece mecanismos que possibilitam o compartilhamento de recursos pela Grade. Para este fim, existe um mecanismo que possibilita a disseminação e a descoberta dos recursos e um mecanismo de gerenciamento que monitora e garante o controle sobre a Qualidade de Serviço na utilização de cada recurso.

A camada de **conectividade** define o protocolo de comunicação e autenticação realizado durante a troca de dados entre recursos da camada de estrutura. Os protocolos utilizados nesta camada são desenvolvidos sobre a pilha de protocolos TCP/IP, utilizando o IP e o ICMP na comunicação na internet, TCP e UDP no transporte de dados, e na camada de aplicação o DNS, por exemplo.

A camada de **Recurso**, construída sobre as camadas citadas anteriormente, fornece um mecanismo de obtenção de informações a respeito da estrutura e do estado dos recursos, como, por exemplo, a carga total e a política utilizada, e possibilita ainda a realização de uma negociação segura, respeitando as definições de Qualidade de Serviço e monitorando o estado de uma operação.

A camada de **Coletividade** não trata cada recurso individualmente mas, em contrapartida, coordena os recursos globalmente no nível das Organizações Virtuais. Isso permite, por exemplo, que VOs descubram a existência e as propriedades de recursos compartilhados. Esta camada fornece ainda um serviço que reserva recursos para garantir que as regras de QoS definidas no escalonamento sejam respeitadas.

A última camada da arquitetura é a camada de **Aplicação**, que gerencia aplicações de usuário que operam em uma Organização Virtual. As aplicações são desenvolvidas sobre os protocolos das camadas anteriores.

Geralmente, em um ambiente de Grade, as aplicações que executam sobre esta plataforma são desenvolvidas com base em conceitos de programação paralela, o que implica em um maior grau de dificuldade quando comparado à construção de um programa concorrente convencional. Para execução eficiente destas aplicações é importante ter o controle sobre as interações entre os serviços remotos, as estruturas de dados e os recursos de hardware. Estas aplicações geralmente são executadas em diferentes componentes de hardware e sistemas operacionais, cujas configurações podem mudar em tempo de execução (Lee et al., 2001).

Além do ambiente heterogêneo e dinâmico em que as aplicações na Grade são executadas, o desenvolvedor deve preocupar-se em desenvolver um código que seja

confiável, tolerante a falhas, e que possua um modelo de segurança e privacidade delimitado pelas políticas de acesso de uma VO.

A Figura 2.3 ilustra a visão de um programador de aplicação em ambiente de Grade. Como pode ser visto, em cada uma destas camadas, são fornecidas APIs e ambientes de desenvolvimento que permitem a utilização dos serviços disponibilizados por cada camada.

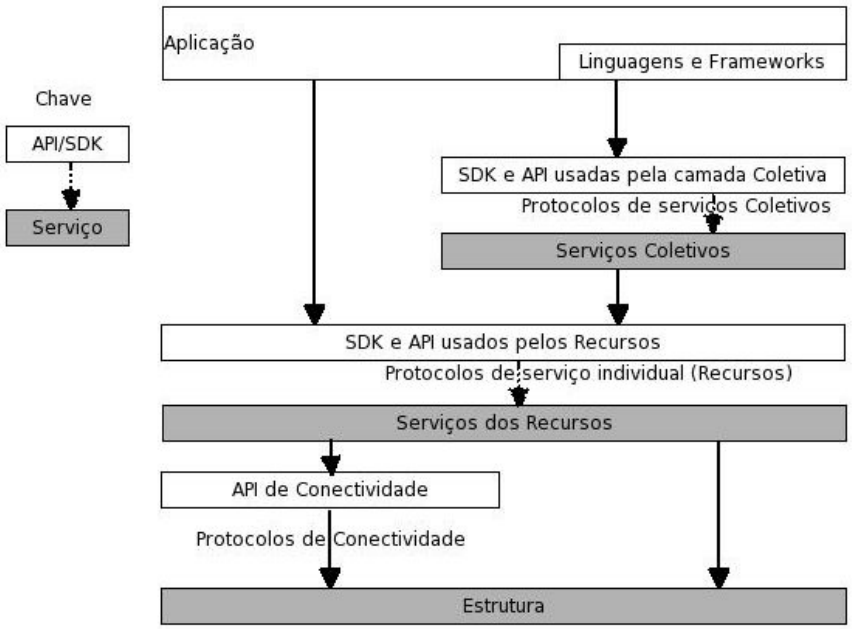


**Figura 2.3:** Em cada camada são fornecidas APIs que facilitam a implementação de aplicações.

## 2.5 Conclusão do Capítulo

Conforme descrito neste capítulo, o gerenciamento dos recursos e das aplicações de forma eficiente é um desafio devido à heterogeneidade dos recursos e ao grande conjunto de aplicações desenvolvidas na Grade.

Diante disto, utilizando a arquitetura da Grade computacional e as funcionalidades disponibilizadas em cada camada, este trabalho apresenta uma ferramenta

que possui um mecanismo de gerenciamento de recursos e um mecanismo de gerenciamento de aplicações, cujo o objetivo é facilitar a execução de aplicações na Grade e executá-las de forma eficiente.

No próximo capítulo serão apresentados ferramentas e mecanismos relacionados a este trabalho e que serviram de motivação para desenvolvimento deste trabalho.

CAPÍTULO

3

# Escalonamento em Grade

Diante do comportamento dinâmico e da instabilidade dos recursos na Grade, a tomada de decisão por parte dos usuário de quais recursos utilizar para cada tarefa de uma aplicação não é uma tarefa fácil, podendo consumir muito tempo e ainda não obter o melhor escalonamento possível.

Sendo assim, cabe ao mecanismo de escalonamento manipular os conjuntos de tarefas das aplicações e suas interdependências, e mapeá-los de forma eficiente aos recursos. Para alcançar estes objetivos, o escalonador pode usar conhecimento sobre as aplicações, sobre os recursos e a forma como são feitas as comunicações, sobre os atrasos, as latências, a reserva de recursos e as larguras de banda (Aggarwal;Aggarwal, 2006).

De forma geral, o escalonamento pode ser dividido em três fases: a descoberta e filtragem de recursos, a seleção dos recursos de acordo com uma política de escalonamento, e a submissão de tarefas (Dong;Akl, 2006). Todas essas fases são realizadas a fim de obter um menor tempo de resposta na execução de uma aplicação (Aggarwal;Aggarwal, 2006) e de forma transparente para o usuário (Vianna

et al., 2004).

Para medir o nível de satisfação do usuário, Casavant;Kuhl (1988) determinam dois aspectos. O primeiro analisa o desempenho medindo a satisfação do cliente em relação à alocação de recursos. O segundo mede a eficiência de acordo com o nível de satisfação do cliente em relação às dificuldades encontradas ao gerenciar recursos.

Casavant;Kuhl (1988) definem ainda uma taxonomia para o problema de escalonamento com o objetivo de fornecer mecanismos e terminologias comuns. Esta taxonomia visa a classificação das metodologias usadas no escalonamento em níveis hierárquicos.

No nível mais alto desta hierarquia, os escalonadores podem ser divididos em locais e globais. No primeiro caso, é feita apenas a alocação de tarefas residentes localmente, com base em fatias de tempo de uso do processador. Os escalonadores globais devem decidir em qual dos múltiplos recursos disponíveis devem ser executadas as aplicações, sendo que, nesta abordagem, as tarefas locais ficam a cargo do Sistema Operacional.

No próximo nível da hierarquia, o escalonamento é distinguido entre Estático e Dinâmico. A diferença entre eles pode ser resumida de acordo com o momento em que o escalonador decidirá qual mapeamento deverá ser utilizado. No escalonamento Estático as informações a respeito dos diversos recursos são conhecidas antecipadamente. Antes da execução da aplicação, todas as tarefas estão mapeadas para os recursos adequados. Essas informações são desconhecidas no caso do escalonamento dinâmico e a alocação das tarefas é realizada no momento da execução da aplicação.

A grande variedade de recursos, que encontram-se distribuídos, dificulta a seleção daqueles que irão prover melhores desempenhos para as aplicações. Por essa razão, algoritmos tendem a classificar os resultados como sendo satisfatórios quando uma solução aproximada é encontrada ou, baseando-se em alguma heurística, tentam obter suposições mais realísticas a respeito da execução de cada

aplicação.

Assim, algoritmos de escalonamento podem utilizar funções que categorizam o resultado, segundo algum critério, como sendo ótimo. Exemplos de critérios de medida de otimização são: redução do tempo de execução total de uma aplicação, utilização máxima dos recursos disponíveis, ou diminuição do tempo de resposta.

Os algoritmos de escalonamento, quando implementados em sua abordagem dinâmica, podem ser utilizados em um cenário centralizado ou podem ser executados em múltiplas máquinas distribuídas. A vantagem do primeiro cenário é a facilidade na implementação. Porém, o escalonador pode vir a ser um *gargalo* no desempenho de todo o sistema e, caso ele falhe, todo o sistema é prejudicado (Dong;Akl, 2006).

Quando forem utilizados algoritmos de escalonamento dinâmicos e distribuídos, mecanismos podem ser implementados com o objetivo de prover uma cooperação entre os escalonadores. Uma outra opção é fazer com que cada escalonador realize o mapeamento entre recursos e processos autonomamente.

Contudo, na maioria dos sistemas de escalonamento, o custo de execução do escalonador não é considerado tampouco o volume de dados trafegados na rede. Por essa razão, Shan et al. (2004) propõem um outro modelo de escalonamento, diferente do modelo hierárquico, para o ambiente de Grade.

Esta arquitetura é composta por filas e escalonadores locais, e filas e escalonadores compartilhados na Grade, conforme Figura 3.1. Cada tarefa enviada para o escalonador local é enfileirada localmente. Em seguida, estas são submetidas para o escalonador compartilhado da Grade que os analisa e os coloca em sua fila. Um escalonador global da Grade coleta as tarefas dos escalonadores compartilhados e realiza a submissão para outros escalonadores compartilhados, que farão o processo inverso, submetendo as tarefas para as filas locais e estas para os recursos. Quando é recebido uma tarefa de um outro escalonador distribuído, ele é analisado de acordo com uma política de escalonamento local. O canal de comunicação entre os escalonadores locais e globais, de acordo com a proposta, é estabelecido

por meio do modelo de comunicação *peer-to-peer* (P2P).

A abordagem hierárquica fornece uma taxonomia padrão para classificação. Porém, a vantagem da última abordagem apresentada é a possibilidade de análise do custo de migração entre as tarefas.

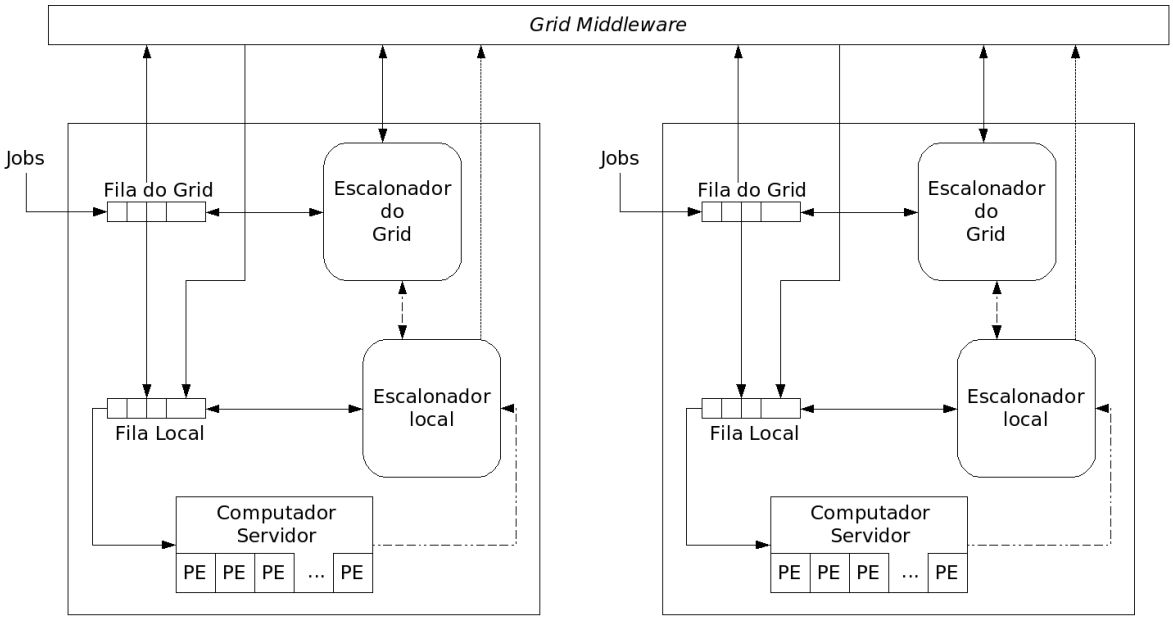


**Figura 3.1:** Modelo Arquitetural de Escalonamento em Grade proposto por Shan et al. (2004). As linhas pontilhadas representam a transferência de informações e as linhas contínuas representam a troca de dados.

## 3.1 Modelo de Aplicação e Modelo Arquitetural

As aplicações que são executadas na Grade podem ser classificadas em modelo de aplicações com dependência de tarefas e sem dependência de tarefas. Quando o escalonador recebe um conjunto de tarefas, se elas forem independentes, deve preocupar-se apenas em alocá-las de acordo com a carga disponível nos recursos (Dong;Akl, 2006). Aplicações *Bag-of-Tasks* (BoT) são um exemplo de grupos de tarefas que devem ser executadas seqüencialmente e sem ter necessariamente dependência entre elas.

Por outro lado, conjuntos de tarefas podem ser dependentes, devendo ser res-

peitada a ordem de precedência durante a execução. Aplicações em que uma tarefa deve terminar a execução antes que a sucessora comece a executar são chamadas de workflow (Andrieux et al., 2003).

O Modelo de Aplicação com dependência de tarefas pode ser representado via uma estrutura de Grafo Direcionado Acíclico (*Directed Acyclic Graph - DAG*), onde cada nó do grafo representa uma tarefa, ou seja, um conjunto de instruções que devem ser executadas seqüencialmente em um processador. Associado a cada nó, há o seu Custo Computacional. As arestas podem representar dependências de comunicação (por exemplo, aplicações *workflow*), ou a seqüência na execução das tarefas (por exemplo, aplicações BoT). Quando representam a comunicação existente, o tempo requerido para troca de mensagens (Custo de Comunicação) vem associado com a aresta (Kwok;Ahmad, 1996).

Grande parte dos algoritmos e ferramentas utilizam o modelo de aplicação baseado em DAG para escalonamento das tarefas [(Aggarwal;Aggarwal, 2006), (Fahringer et al., 2005), (Yang;Gerasoulis, 1994), (Yu;Buyya, 2005b), (Topcuouglu et al., 2002), (Vianna et al., 2004), (Prodan;Fahringer, 2005)] , assim como o algoritmo e a ferramenta apresentados nesse trabalho.

Quanto ao Modelo Arquitetural, a coordenação dos recursos a serem utilizados tornou-se um desafio devido à grande quantidade de máquinas que participam de uma ou mais VOs. Com isso, descobrir e monitorar os recursos em um ambiente computacional distribuído e de larga escala tornou-se uma tarefa crítica e que influencia no desempenho da aplicação Kee et al. (2006).

O Modelo Arquitetural consiste em informações sobre os recursos como, por exemplo, capacidade de processamento, carga em execução no processador, total de memória, total de memória utilizada, espaço em disco, e em informações sobre os canais de comunicação como, por exemplo, latência e largura de banda entre os recursos. Este modelo, assim como o Modelo de Aplicação pode ser caracterizado como um grafo, sendo os recursos representados pelos vértices e os canais de comunicação representados pelas arestas.

## 3.2 Algoritmos Estáticos

De maneira geral, os algoritmos de escalonamento tentam minimizar o tempo de execução de uma aplicação, diminuindo o *makespan*[1]. Quando algumas características da aplicação, como tempo de execução, tamanho das mensagens trocadas e dependência entre tarefas, são conhecidas antecipadamente, podem ser utilizados algoritmos de escalonamento estáticos (Topcuouglu et al., 2002).

Os principais algoritmos de escalonamento tentam enviar as tarefas para os processadores, visando a redução do tempo de execução das aplicações. Porém, o custo de comunicação na troca excessiva de informações pode degradar o desempenho da aplicação. Portanto, a submissão de diversas tarefas simultaneamente para um número máximo de recursos, utilizando o menor custo possível na comunicação é chamado de problema *max-min* [(El-Rewini et al., 1994),(Dong;Akl, 2006)]. Os algoritmos desenvolvidos para solucionar este problema foram divididos em 3 classes: *List Scheduling*, Algoritmos de Duplicação de Tarefas e Algoritmos de Agrupamento.

Os Algoritmos de *List Scheduling* definem prioridades para cada tarefa e as adicionam em uma lista ordenada (Dong;Akl, 2006). Os cálculos das prioridades de cada tarefa podem ser realizados tanto antes quanto durante o escalonamento. A cada iteração do escalonamento é selecionada uma tarefa com a maior prioridade da lista e, em seguida, escolhido o processador que possobilite o inicio da sua execução mais rapidamente e/ou minimize o seu tempo de execução (Kwok;Ahmad, 1996).

Os Algoritmos de Duplicação de Tarefas tentam reduzir o *makespan* de uma aplicação por intermédio da duplicação de tarefas em diferentes recursos. Com a duplicação de tarefas predecessoras, utilizando o tempo ocioso no recurso, evita-se a transferência do resultado de um predecessor para o sucessor.

[1] *Makespan* é o tempo total para execução de todas as tarefas escalonadas no sistema. Topcuouglu et al. (2002) definem o makespan como sendo: $makespan = max\{ AFT(n) \}$, onde *AFT* é uma função que calcula qual processador pode finalizar mais rapidamente uma tarefa *n*.

Os Algoritmos de Agrupamento são eficientes mecanismos para redução do problema *max-min*. Nesta abordagem, os custos de comunicação são reduzidos com agrupamento de tarefas para execução no mesmo processador.

A seguir, serão apresentados algoritmos clássicos de escalonamento baseados em *List Scheduling* e Agrupamento de tarefas que serviram de base para implementação do trabalho.

### 3.2.1 DCP

O algoritmo DCP (Dynamic Critical Path) é um exemplo de algoritmo de *List Scheduling*. Este algoritmo utiliza o caminho crítico para priorizar as tarefas. O caminho crítico é o caminho mais longo em um DAG, que corresponde a um conjunto de nós e arestas que possui o maior valor somando o custo computacional e de comunicação.

No DCP há duas fases, sendo a primeira a seleção das tarefas (ou nós), em que o caminho crítico determina parcialmente o tamanho do escalonamento. Enquanto executa esse procedimento, o escalonador define as prioridades de cada tarefa, que podem ser alteradas à medida que o algoritmo for executado, ou seja, enquanto existirem tarefas para serem escalonadas.

A próxima etapa é a seleção do processador que executa a tarefa no menor tempo possível. Para não violar a regra de precedência entre tarefas, uma tarefa não deve ser escalonada antes de seus predecessores, nem depois de seus sucessores.

Além do DCP, existem diversos outros algoritmos que utilizam a abordagem de Caminho Crítico para selecionar, em uma lista de escalonamento, nós com maiores prioridades. Por exemplo, Radulescu;van Gemund (1999) apresentam o algoritmo *Fast Critical Path* (FCP), que reduz a complexidade de algoritmos de *List Scheduling*.

Este algoritmo não ordena todas as tarefas no início, mas mantém um número limitado de tarefas ordenadas em um intervalo de tempo. O algoritmo DCP possui ordem de complexidade $O(v^3)$ e o FCP reduz para $O(v\ log\ p + e)$, onde $v$ é o número

de tarefas, *p* é o número de recursos e *e* as arestas que conectam as tarefas.

Uma outra abordagem apresentada por Ma;Buyya (2005), propõe um algoritmo DCP estendido para escalonamento de aplicações com trocas de parâmetros na Grade.

### 3.2.2 HEFT

Hwang et al. (1989), apresentam um algoritmo de escalonamento chamado ETF (*Earliest Time First*), que mapeia as tarefas para um conjunto de processadores ociosos, calculando o tempo de finalização mais cedo para cada tarefa. A abordagem apresentada considera que o algoritmo será executado em um ambiente homogêneo e com um número de processadores limitado.

Diante das limitações do algoritmo anterior, Topcuouglu et al. (2002), propõem uma extensão, chamado HEFT (*Heterogenous Earliest Finish Time*). O algoritmo, além de superar as dificuldades encontradas pelo EFT, adiciona novas estratégias. Assim como os demais algoritmos de escalonamento que utilizam a abordagem de *List Scheduling*, o HEFT prioriza as tarefas a serem escalonadas e seleciona o melhor processador para cada tarefa.

A fase de priorização das tarefas, em grande parte dos algoritmos existentes, leva em consideração apenas o custo computacional de cada tarefa. O HEFT configura a prioridade de uma tarefa com base no resultado da função *upward*. Esta função calcula o custo computacional médio de uma tarefa somando com a maior média de custo de comunicação da tarefa analisada até a tarefa de saída[2]. Resumindo, *upward* é o tamanho do caminho crítico de uma tarefa $n_i$ até a tarefa de saída $n_j$ somado com o custo computacional da tarefa $n_i$.

A partir do cálculo de prioridade, as tarefas são ordenadas em uma lista. Na fase de seleção de processadores, o algoritmo seleciona uma tarefa que ainda não foi escalonada e verifica qual processador finaliza a execução da tarefa no menor tempo possível.

[2]Tarefas de saída são as que não possuem tarefas filhas.

### 3.2.3 CPOP

Topcuouglu et al. (2002) propõem também um outro algoritmo baseado em Caminho Crítico chamado *Critical-Path-on-a-Processor* (CPOP), que utiliza diferentes estratégias para priorização dos nós e seleção dos processadores. Inicialmente, o algoritmo determina através das funções *upward* e *downward* a prioridade de cada tarefa. A função *upward* utilizada pelo CPOP é a mesma definida no algoritmo HEFT e a função *downward*, ao contrário da função *upward*, calcula o tamanho do caminho crítico dos predecessores de uma tarefa $n$ até a tarefa inicial.

Em seguida, as tarefas que possuem prioridade igual ao custo do caminho crítico são agrupadas e submetidas para escalonar no melhor processador do ambiente. As demais tarefas, que não fazem parte do caminho crítico, são escalonadas individualmente nos processadores que reduzam o seu instante de finalização, ou seja, de acordo com o EFT da tarefa.

De maneira geral, CPOP é um algoritmo que combina as heurísticas de *List Scheduling* e de Agrupamento.

### 3.2.4 DSC

O algoritmo DSC (*Dominant Sequence Clustering*), apresentado por Yang;Gerasoulis (1994) é um algoritmo de agrupamento que utiliza uma função chamada *Dominant Sequence*, que é equivalente ao cálculo do caminho crítico.

Este algoritmo é iniciado definindo prioridades para cada tarefa. A prioridade de uma tarefa $n$, inicialmente, é o valor da soma dos custos de comunicação e computação das tarefas que estão no mesmo caminho de $n$ até a tarefa de saída (*bottom level*). Em seguida, o valor da prioridade de cada tarefa é incrementado com o decorrer do algoritmo que soma os custos de comunicação e computação das tarefas que estão no mesmo caminho até a tarefa de entrada (*top level*).

Desta forma, a tarefa com mais alta prioridade é selecionada e agrupada no mesmo processador de uma tarefa predecessora que miniminize o seu o valor de *top*

*level*. Com isso, o algoritmo executa paralelamente grupos de tarefas que possuem maior custo de comunicação entre si.

## 3.3 Algoritmos Dinâmicos

Diferentemente do escalonamento estático, o escalonamento dinâmico é aplicado em situações em que é difícil estimar o comportamento de uma aplicação previamente, ou em situações em que o estado corrente do ambiente muda constantemente. Por essa razão, este tipo de algoritmo faz pouca suposição a respeito das características e informações obtidas das tarefas antes da execução (Rotithor, 1994), disparando eventos de escalonamento a cada alteração no comportamento geral da aplicação ou do recurso (Boeres et al., 2003).

O escalonamento dinâmico utiliza informações obtidas em tempo de execução para tomar uma decisão, procurando reduzir o tempo restante de uma tarefa no sistema (Vallée et al., 2003). Por essa razão, é chamado também de escalonamento *online*.

### 3.3.1 Taxonomia para escalonamento dinâmico

Rotithor (1994) define uma taxonomia para escalonamento dinâmico de tarefas propondo um componente para estimar o estado do sistema e um outro componente para realizar a tomada de decisão. O componente que estima o estado do sistema fica responsável por transmitir informações e construir estimativas a respeito dos recursos. Ao processo de tomada de decisão cabe escolher um recurso para uma tarefa com base na estimativa do estado do sistema.

O componente que realiza a estimativa do sistema pode ser estruturado de forma centralizada, possuindo um agente central que coleta as informações de estado do sistema e constrói a estimativa geral. Pode ser utilizada ainda uma abordagem descentralizada, em que cada recurso coleta informações individualmente e troca informações com os demais recursos, realizando uma estimativa do estado

global do sistema.

### 3.3.2 Abordagens

Os algoritmos de escalonamento dinâmicos podem ser classificados de acordo com a abordagem utilizada no gerenciamento das tarefas. A seguir serão apresentadas algumas destas abordagens.

#### FIFO

Algoritmos de escalonamento que utilizam a abordagem conhecida como FIFO (*First-In-First-Out*)[3] procuram escalonar tarefas que primeiro chegaram no escalonador. A grande vantagem desta abordagem é a simplicidade de implementação (Dong;Akl, 2006).

Iverson;Ozguner (1998) apresentam um *framework* que permite que cada aplicação realize seu próprio escalonamento. As aplicações não necessitam obter informações de outras aplicações, exceto a sua previsão de execução sistema.

Entretanto, no ambiente heterogêneo em que são executadas as aplicações, torna-se muito complexo determinar o tempo de execução com exatidão, já que várias tarefas estão executando em um mesmo recurso. Além disto, o cálculo do custo de uma aplicação em sistemas que realizam migração de tarefas entre os recursos, geralmente, é imprecisa e difícil de ser implementado.

Diante das dificuldades encontradas, Iverson;Ozguner (1998) implementam uma fila, que utiliza a metodologia FIFO, em cada máquina. As tarefas que estão prontas para serem executadas devem aguardar na fila até que o recurso esteja disponível. As tarefas são dependentes entre si e, portanto, além da disponibilidade dos recursos para serem executadas, as tarefas antecessoras devem ter sido executadas, os dados necessários devem estar prontos e a tarefa deve estar no início da fila. Esta metodologia fica comprometida se uma tarefa estiver pronta para ser executada e o recurso estiver disponível mas os dados de que ela depende não

[3]Essa abordagem também é conhecida na literatura como *First-Come-First-Served*

estão prontos. Neste caso, a tarefa fica ociosa bloqueando todas as outras tarefas da fila.

Para minimizar este problema, o algoritmo de escalonamento FIFO pode estimar a quantidade de tempo que uma tarefa pode ficar bloqueada, a partir do tamanho da fila e do tempo que a tarefa gasta para chegar.

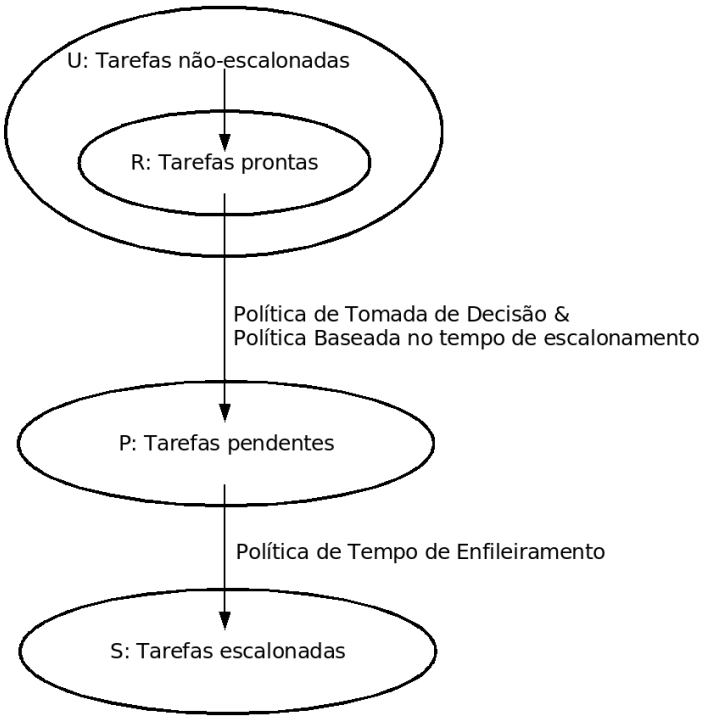


**Figura 3.2:** *Framework* proposto por Iverson;Ozguner (1998).

A estrutura do *framework* pode ser vista na Figura 3.2. O topo da estrutura é povoado por tarefas que ainda não foram escalonadas, representadas por *U*. No início da execução do algoritmo, todas as tarefas se encontram neste conjunto. Contidas neste grupo, estão as tarefas, representadas por *R*, que estão prontas para serem executadas, ou seja, estão prontas para serem alocadas aos recursos e cujas tarefas antecedentes já foram executadas.

A próxima fase do *framework* utiliza uma política de tomada de decisão, que decide qual função de escalonamento deve ser realizada. O instante em que uma decisão deve ser tomada é determinado por uma política baseada no tempo de

escalonamento.

A política de tomada de decisão foi implementada adaptando o algoritmo de escalonamento estático DLS (*Dynamic Level Scheduling*) (Sih;Lee, 1993). O algoritmo DLS é uma estratégia de escalonamento não-preemptiva, executado em tempo de compilação, que mapeia tarefas para processadores heterogêneos por meio de um grafo direcionado acíclico (Sih;Lee, 1993).

A adaptação realizada no algoritmo permite que uma tarefa seja escalonada não levando em conta apenas o instante em que os dados e o recurso serão disponibilizados, mas o tempo que a tarefa gastaria na fila do recurso em que foi alocada e o tempo de execução total da tarefa no processador.

Com a combinação das duas políticas, o algoritmo envia as tarefas do conjunto $R$ para o próximo grupo de tarefas pendentes, representado por $P$. O grupo de tarefas pendentes é formado por tarefas que foram alocadas para um determinado recurso, mas encontram-se na fila aguardando o momento para serem executadas.

Uma outra política gerencia as tarefas que entram na fila, determinando o instante em que devem mudar para o próximo estado. As tarefas que são selecionadas por esta política passam para o estado $S$, que é formado por tarefas escalonadas.

### Carga Restrita

Essa abordagem tenta rebalancear a carga de todos os recursos periodicamente, trocando tarefas que estão na fila esperando para serem executadas. Chen;Maheswaran (2002) definem um algoritmo de Carga Restrita que executa uma fase interna e uma fase externa. Na fase externa, executada em uma WAN (*Wide-Area Network*), é utilizado um escalonador distribuído. E na fase interna, que é executada em uma LAN (*Local Area Network*), é utilizado um escalonador centralizado.

O escalonador externo recebe uma tarefa submetida por um recurso da LAN, envia uma solicitação para todos os escalonadores internos e, baseado na resposta deles, escolhe o mais adequado para executar a tarefa. Por fim, fica coordenando os escalonadores internos e coletando os resultados da execução das tarefas.

O escalonador interno primeiramente responde uma solicitação do escalonador externo. Se for selecionado, recebe do escalonador externo uma tarefa para executar. O escalonador interno, ao receber uma tarefa, fica responsável por distribuir e administrar sub-tarefas, bem como por coletar seus resultados e enviá-los para o escalonador externo.

Uma outra abordagem, diferente da apresentada por Chen;Maheswaran (2002), preocupa-se não apenas com o balanceamento de carga entre os recursos, mas também com o custo de comunicação entre as tarefas. Essa abordagem é conhecida como Custo Restrito, e tem um melhor desempenho do que o de Carga Restrita em ambientes onde o custo de comunicação entres os recursos é heterogêneo.

#### Híbrido

Vianna et al. (2004) apresentam uma abordagem que efetua um escalonamento dinâmico em um conjunto de tarefas que foram previamente alocadas por um algoritmo de escalonamento estático. O escalonamento primeiro define prioridades para as tarefas e, em seguida, escolhe o processador, como é feito no *list scheduling*.

A escolha da tarefa pode ser realizada por um algoritmo de escalonamento estático ou por meio de uma heurística que prioriza uma tarefa com base no custo do caminho crítico associado. A fase de seleção do processador pode ser realizada em tempo de execução. Essa fase pode realocar uma tarefa a fim de minimizar o tempo de execução total ou para reduzir o caminho crítico de uma tarefa.

Essa abordagem utiliza as principais vantagens do escalonamento estático e considera as alterações no comportamento das aplicações e dos recursos em tempo de execução.

## 3.4 Gerenciadores de recursos e aplicações

Apesar da Grade disponibilizar diversos recursos, aumentando o poder computacional disponível para execução de uma aplicação, o usuário depara-se com diversos

problemas como, por exemplo, ambientes altamente heterogêneos, diferentes políticas de escalonamento e possíveis falhas de comunicação.

Para solucionar esses problemas, foram desenvolvidos diversos *middlewares* que visam a formação de uma camada de *software* com o objetivo de facilitar a utilização deste ambiente. Nascimento et al. (2005) subdividem esses *middlewares* em dois subconjuntos: *Middleware Básico*, cujo principal objetivo é facilitar o mapeamento das tarefas para os recursos individuais, e *Middleware de Serviços*, que fornece ferramentas e serviços para aplicações, abstraindo a complexidade da Grade.

O *Middleware* de Serviço pode ser utilizado para gerenciar os recursos disponíveis na Grade. Neste caso, são conhecidos como RMS, conforme descrito no capítulo anterior, e fornecem um serviço confiável, capaz de descobrir e gerenciar recursos dinamicamente. Além disto, o usuário não deve preocupar-se com a localização, com os mecanismos requeridos para utilizá-los, com a carga em cada recurso ou como reagem no caso de ocorrer algum tipo de falha (Frey et al., 2002).

Um outro tipo de *Middleware de Serviços* é o AMS, também descrito no capítulo anterior, que permite que tarefas utilizem a Grade de acordo com a disponibilidade dos recursos e as características individuais de cada aplicação.

As subseções seguintes descrevem alguns *middlewares* e ferramentas que provêem funcionalidades de gerenciamento de recursos e de aplicações da Grade.

### 3.4.1 Globus Toolkit

O Globus (Foster, 2005) é um *middleware* que cria um nível de abstração no uso da Grade, fazendo com que as aplicações visualizem um conjunto de máquinas heterogêneas como sendo uma única máquina virtual. O principal elemento do Globus é o *Globus Toolkit* que fornece uma implementação de serviços e protocolos necessários para a construção de um ambiente de Grade Computacional.

O *Globus Toolkit* é composto por um conjunto de componentes que implementam alguns serviços como: segurança, por meio do protocolo GSI; gerenciamento

e alocação de recursos, com o protocolo GRAM/DUROC; descoberta e divulgação de informações, através do MDS; e mecanismos para transferência de dados entre os recursos remotos (Foster, 2005). Os tópicos a seguir explicam em detalhes cada um destes serviços.

### GSI

O GSI (*Grid Security Infrastructure*) fornece um serviço de autenticação e autorização utilizando uma infra-estrutura de chave pública (PKI - *Public Key Infrastructure*). Além disso, este módulo cria um canal de comunicação seguro entre os elementos de uma Organização Virtual.

Para um *host* e/ou um usuário ter acesso a um ambiente de Grade é preciso inicialmente criar um conjunto de chaves criptografadas e requisitar um certificado a uma Autoridade Certificadora (*Certification Authority* - CA).

Como pode ser visto na Figura 3.3, para um *host* ter acesso a um ambiente de Grade gerenciado pelo Globus é preciso inicialmente que o *host* solicite à Autoridade Certificadora um certificado que contém a chave pública da CA.

Em seguida, são gerados uma solicitação para assinatura do certificado (*Certificate Signing Request* - CSR) e a chave privada do *host* com base nas informações transmitidas pela CA. Então, o *host* requisita à CA a assinatura do seu certificado gerado. A CA por sua vez, utilizando sua chave privada, assina e devolve o certificado para o *host*.

O processo de autenticação do usuário funciona de forma semelhante. Inicialmente é gerado pelo usuário uma CSR e submetido para que seja assinado pela CA. Cada usuário na Grade é identificado e autenticado por uma chave, mantida em um certificado assinado pela Autoridade Certificadora. O conteúdo do certificado é formado por um nome que identifica o usuário, a chave pública, a identificação da Autoridade Certificadora que assinou o certificado e outros dados necessários para autenticação.

A seguir são apresentados os dois tipos de certificados. O primeiro é utilizado

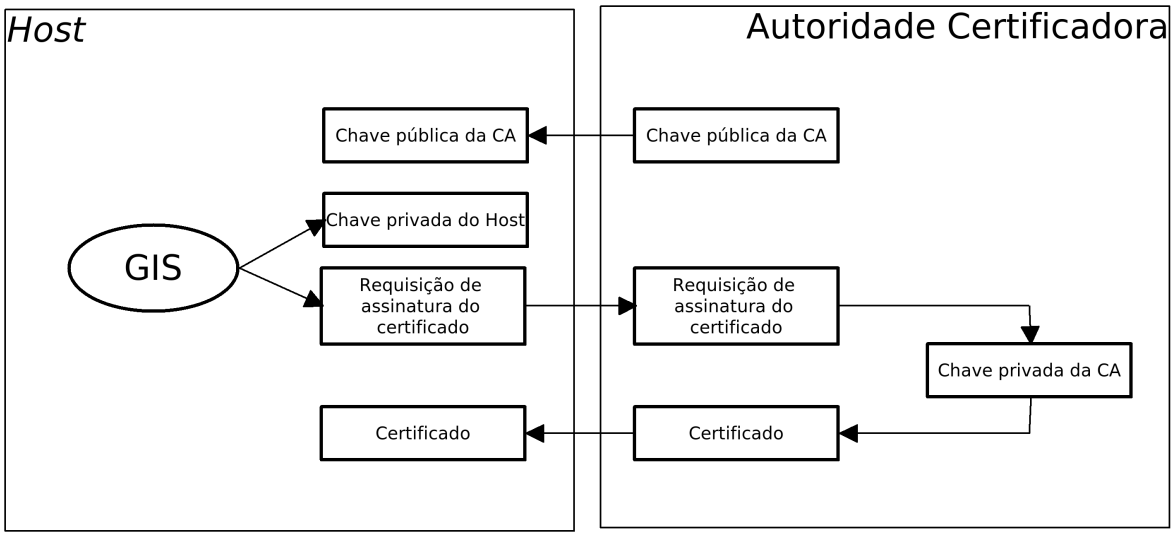


**Figura 3.3:** Processo de autenticação e autorização de usuários e máquinas na Grade (Ferreira et al., 2003).

para identificar um usuário na Grade.

Certificate Subject:

"/O=Grid/OU=UFSCar/OU=xeon.dc.ufscar.br/OU=dc.ufscar.br/CN=Ricardo Araujo Rios"ricardo

O segundo tipo de certificado é utilizado para autenticar um *hosts*.

Certificate Subject:

/O=Grid/OU=UFSCar/OU=xeon.dc.ufscar.br/CN=host/compute-0-0.local

Depois de assinado o certificado, o usuário tem permissão para executar tarefas apenas nas organizações virtuais em que ele tem acesso e para as máquinas as quais ele tem um certificado autorizado. A submissão de tarefas na Grade, geralmente, é feita com a geração de um Certificado *Proxy* de curta duração que é assinado pelo usuário e é baseado no padrão X.509 e nos protocolos de comunicação SSL (*Secure Socket Layer*) e TLS (*Transport Layer Security*) (Jacob et al., 2005).

### GRAM/DUROC

O protocolo GRAM (*Grid Resource Allocation and Management*) permite a submissão de uma tarefa para um recurso remoto, monitorando e controlando o resultado da

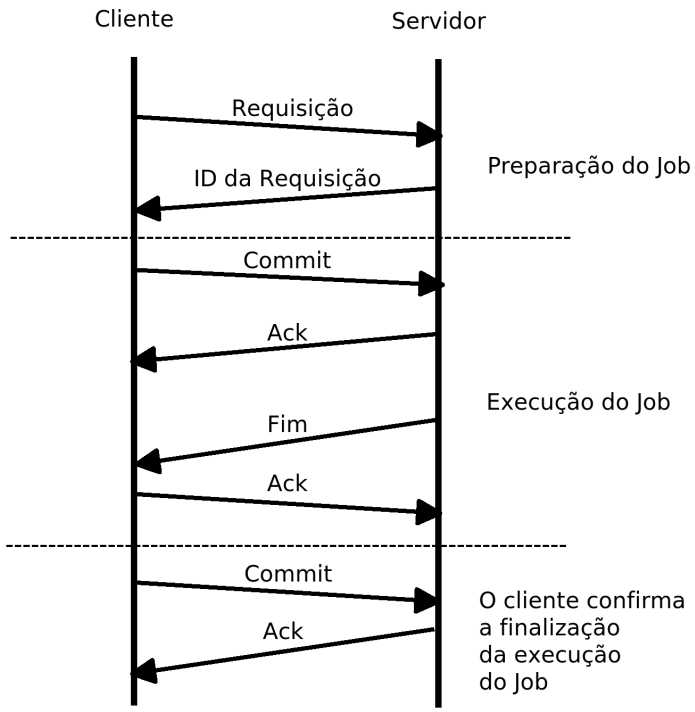


**Figura 3.4:** Modelo de troca de mensagens do protocolo GRAM (Frey et al., 2002).

execução. A execução do protocolo é dividida em uma fase de segurança, duas fases de *commit* e uma fase de tolerância a falhas (Frey et al., 2002), como demonstra a Figura 3.4.

A Figura 3.4, apresentada por Frey et al. (2002), descreve o modelo de troca de mensagens do protocolo GRAM. Inicialmente, o cliente conecta-se ao servidor submetendo uma requisição para execução de um *job*. O *job* submetido pelo cliente é manipulado por um *daemon* chamado **gatekeeper**, o qual cria um gerenciador de *job* que inicializa e manipula o *job* (Ferreira et al., 2003).

O servidor responde à requisição do usuário, devolvendo o identificador usado para alocar o *job*. O identificador é armazenado no cliente, que devolve para o servidor uma mensagem de *commit*. Após a execução do *job*, o servidor envia uma mensagem informando que a execução foi completada. O cliente analisa a mensagem recebida e, em seguida, envia uma mensagem confirmando a finalização do *job*.

De uma forma geral, o GRAM é ativado por um comando **globusrun** (ou **globus-job-submit**), que submete e gerencia um *job* remoto e coleta os dados de saída que representam o resultado da execução. A submissão do *job* é realizada por meio de uma Linguagem de Especificação de Recursos (*Resource Specification Language* - RSL) (Ferreira et al., 2003). A Tabela 3.1 demonstra um exemplo de um RSL.

**Tabela 3.1:** Arquivo RSL.

```
+(
 &(resourceManagerContact="compute-0-0.local")
 (count=1)
 (label="mpi-mergesort2-8 0")
 (environment=(GLOBUS_DUROC_SUBJOB_INDEX 0)(LD_LIBRARY_PATH /opt/globus/lib))
 (arguments="8")
 (stdout=/home/ricardo/saida.log)
 (stderr=/home/ricardo/saida.log)
 (directory="/home/ricardo/programas-GRID/mpi/mergesort/less/v0.1.4/")
 (executable="/home/ricardo/programas-GRID/mpi/mergesort/less/v0.1.4/mpi-mergesort2")
)
...
(
 &(resourceManagerContact="compute-0-15.local")
 (count=1)
 (label="mpi-mergesort2-8 7")
 (environment=(GLOBUS_DUROC_SUBJOB_INDEX 7)(LD_LIBRARY_PATH /opt/globus/lib))
 (arguments="8")
 (stdout=/home/ricardo/saida.log)
 (stderr=/home/ricardo/saida.log)
 (directory="/home/ricardo/programas-GRID/mpi/mergesort/less/v0.1.4/")
 (executable="/home/ricardo/programas-GRID/mpi/mergesort/less/v0.1.4/mpi-mergesort2")
)
```

Um outro elemento que compõe o GRAM, e que foi citado anteriormente, é o *gatekeeper*. Este *daemon* é responsável pela criação de um canal seguro de comunicação entre clientes e servidores. Após estabelecer uma conexão segura, o *gatekeeper* cria um gerenciador de *job*. Este gerenciador fica responsável por validar o RSL, e realizar todas as funções de monitoramento e manipulação dos *jobs* (Ferreira et al., 2003).

Um cliente GRAM pode ainda utilizar o DUROC (*Dynamically-Updated Request Online Coallocator*), que o capacita para submeter múltiplas tarefas para diferentes gerenciadores de tarefas (servidores GRAM) (Ferreira et al., 2003), como demonstra a Figura 3.5.

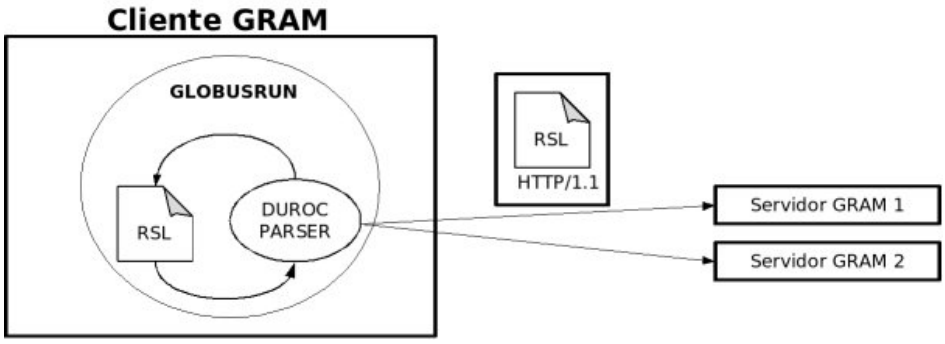


**Figura 3.5:** Visão geral do DUROC (Ferreira et al., 2003).

### MDS

O componente MDS (*Monitoring and Discovery Service*) fornece mecanismos para publicar e descobrir informações sobre o *status* e as configurações dos recursos. O MDS possui uma estrutura descentralizada que favorece a escalabilidade, podendo ser estruturado hierarquicamente, de forma semelhante ao DNS (*Domain Name Service*) (Ferreira et al., 2003). Este componente é implementado utilizando os protocolos GRIS, GIIS e LDAP.

O GRIS (*Grid Resource Information Service*) é um servidor que fornece um repositório de informações sobre os recursos locais. O GIIS (*Grid Index Information Service*) é um repositório que indexa informações sobre os recursos registrados no GRIS e em outros GIIS. O cliente MDS utiliza o LDAP (*Lightweight Directory Access Protocol*) para buscar informações sobre recursos na grade.

### Gerenciamento de dados

O Globus Toolkit oferece diferentes componentes para transferir e manipular dados. Um destes componentes é o GridFTP que estende funcionalidades do FTP (*File Transfer Protocol*), e inclui suporte ao GSI. A principal característica do GridFTP é permitir, além da troca simples de arquivos entre cliente e servidor, que um cliente transfira, diretamente, um arquivo de um servidor remoto para outro (Ferreira et al., 2003), como ilustrado na Figura 3.6.

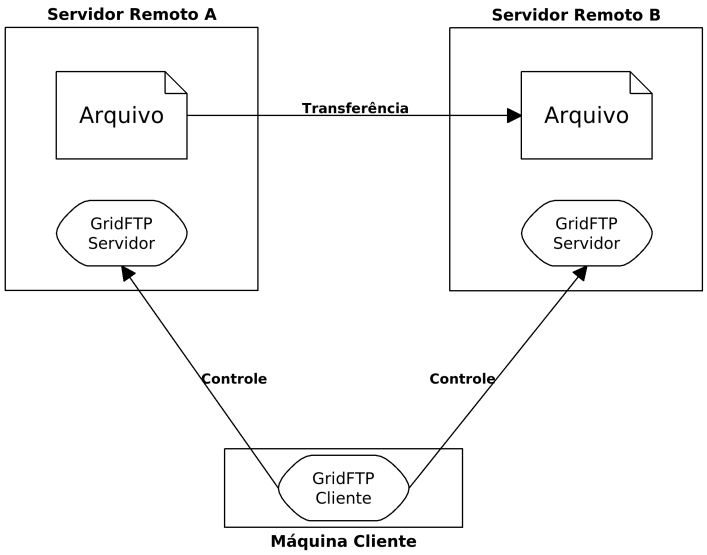


**Figura 3.6:** Transferência de arquivos entre servidores, também chamado *Third-part file transfer* (Ferreira et al., 2003).

Além do GridFTP, existe um outro serviço de transferência de arquivos, chamado RFT (*Reliable File Transfer*). Esse serviço oferece uma interface para *Web Services*. O RFT recebe uma requisição por intermédio de uma mensagem para transferir ou excluir um arquivo e então utiliza o GridFTP. A requisição é feita através de uma mensagem SOAP (*Simple Object Access Protocol*), que é um protocolo que permite a troca de informações em um ambiente descentralizado e distribuído. O objetivo do SOAP é transmitir dados XML (*eXtensible Markup Language*) através do Protocolo HTTP (*HyperText Transfer Protocol*).

Os mecanismos mostrados são utilizados para transferência de arquivos, sendo que, além destes existem outros dois mecanismos para replicação de arquivos na Grade. RLS (*Replica Location Service*) é um serviço que armazena informações sobre dados replicados em diferentes recursos da Grade e apresenta como se fosse um único arquivo lógico. Um outro mecanismo é o DRS (*Data Replication Service*) que cria réplicas de arquivos e as registra no RLS. O DRS utiliza o RFT e o GridFTP para transferir arquivos e o RLS para localizar e registrar réplicas.

### 3.4.2 NWS

Conforme visto anteriormente, a ferramenta MDS provê um mecanismo de descoberta e monitoramento dos recursos na Grade. Porém, as informações fornecidas por esta ferramenta restringem-se ao estado do recurso computacional, ou seja, são fornecidas informações sobre as máquinas disponíveis na Grade.

Contudo, no escalonamento, o gerenciamento dos canais de comunicação entre os recursos computacionais é importante para a execução eficiente do escalonamento, principalmente quando se pretende minimizar o problema *max-min* descrito anteriormente.

Uma forma de solucionar este problema é integrar no monitoramento as ferramentas MDS e NWS (*Network Weather Service*) (Wolski et al., 1997). A ferramenta NWS fornece meios para periodicamente monitorar os recursos computacionais disponíveis e os canais de comunicação. Além disso, o NWS realiza uma previsão dos comportamentos futuros dos recursos monitorados.

O NWS é uma ferramenta modular e sua arquitetura divide-se em três camadas. A primeira camada, *Sensory SubSystem*, realiza o monitoramento efetivo dos recursos. A segunda camada, *Forecasting SubSystem*, realiza a previsão de condições futuras dos recursos monitorados. E por fim, a camada *Reporting SubSytem*, responsável por disseminar as informações sobre os recursos.

De forma geral, a ferramenta NWS permite a criação do modelo arquitetural utilizado pelos algoritmos de escalonamento fornecendo dados sobre a previsão e o estado corrente dos recursos

### 3.4.3 Condor-G

O *middleware* Condor-G, apresentado por Frey et al. (2002), é uma ferramenta que gerencia a execução de tarefas nos recursos da Grade. O Condor-G permite que o usuário especifique um conjunto de tarefas e as relações e dependências entre elas.

Condor-G pode ser dividido em duas partes. A primeira é formada por uma interface utilizada pelo usuário, que facilita a execução de aplicações na Grade. A interface é composta por ferramentas de linha de comando que executam funções como submissão de *jobs*, passagem de parâmetros, obtém informações sobre o *status* dos *jobs*, acesso a *logs* e um completo histórico de execução dos *jobs*.

A segunda parte do Condor-G, utiliza os protocolos do Globus Toolkit citados na Seção 3.4.1. A Figura 3.7 apresenta a estrutura de submissão de tarefas no Condor-G. Inicialmente, o usuário realiza uma submissão ao escalonador do Condor-G. O escalonador cria um processo *GridManager* que fica responsável pela submissão e gerenciamento dos *jobs*. Para criar este processo gerenciador, o Condor-G utiliza o protocolo GRAM do Globus.

Toda submissão realizada pelo *GridManager* resulta na criação de um *Globus JobManager* pelo *Globus GateKeeper*. O *GridManager* conecta-se com o *JobManager* utilizando os protocolos de transferência de arquivos do Globus. Por meio destes protocolos são transferidos arquivos executáveis, arquivos de entrada padrão e arquivos de saída padrão e de erro. Por fim, o *JobManager* envia os *jobs* para os escalonadores locais de uma Organização Virtual. Os *status* de execução dos *jobs* são enviados do *JobManager* para o *GridManager*, o qual atualiza o escalonador do Condor-G.

Condor-G utiliza outro mecanismo de gerenciamento que permite a união temporariamente de máquinas remotas ao grupo de recursos disponíveis. Essa técnica, chamada *Glide In*, inicia processos em computadores remotos, que informam a disponibilidade de recursos nas máquinas.

### 3.4.4 AppLeS

AppLeS (*Application Level Scheduling*) é um *middleware* que aplica técnicas de escalonamento adaptativo em ambiente de Grade Computacional, a fim de obter um bom desempenho para o usuário final. Para alcançar esses objetivos, foram utilizadas abordagens de obtenção estática e dinâmica de informações sobre os recursos,

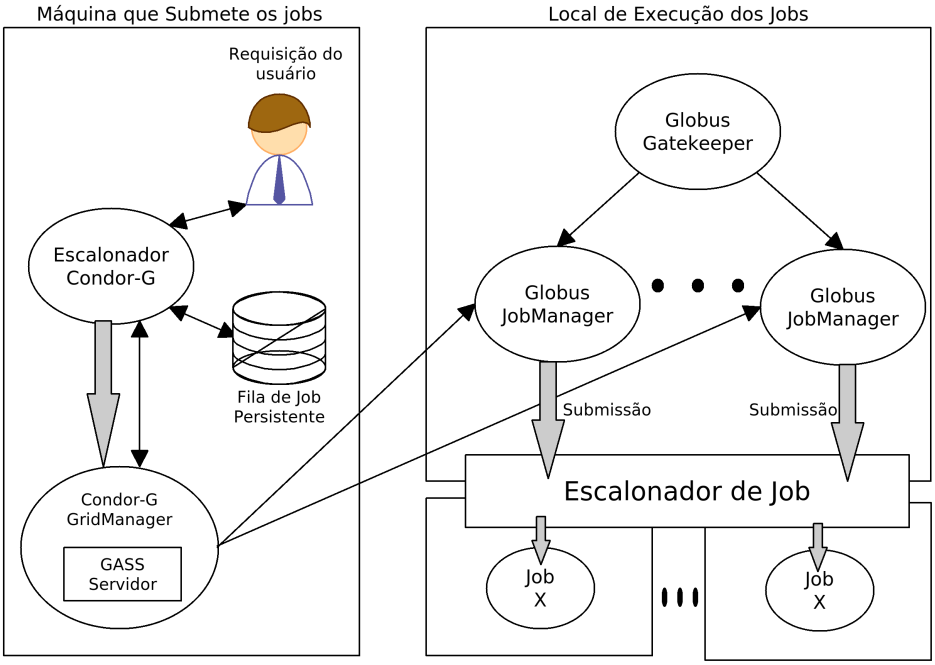


**Figura 3.7:** Arquitetura de execução remota de *Jobs* no Condor-G (Frey et al., 2002).

predição de desempenho e técnicas de escalonamento que adaptam-se às aplicações em tempo de execução (Berman et al., 2003).

O *middleware* AppLeS não é um sistema gerenciador de recursos propriamente dito, ele utiliza outros sistemas como por exemplo o Globus e o Legion para executar essa tarefa (Vianna, 2005). O primeiro passo para escalonamento de tarefas usando o AppLeS é a descoberta de recursos que são potencialmente úteis para a aplicação (Berman et al., 2003).

Em seguida, o agente AppLeS identifica e seleciona conjuntos de recursos descobertos no passo anterior. No terceiro passo, o agente gera uma lista ordenada de possíveis recursos e, aplicando um modelo de desempenho, determina um conjunto de escalonadores candidatos para aplicação. Em seguida, é escolhido o melhor escalonador com base no critério de desempenho estabelecido pelo usuário. O melhor escalonador é então utilizado para distribuição e execução das tarefas da aplicação (Berman et al., 2003).

O agente AppLeS pode optar por voltar ao passo inicial para refinar o escalonamento ou quando são executadas aplicações que consomem muito tempo de processamento. Neste caso, outras tarefas podem chegar aos recursos degradando o desempenho da tarefa escalonada, obrigando o agente a realizar um escalonamento adaptativo (Berman et al., 2003).

### 3.4.5 Nimrod/G

O Nimrod/G é uma extensão do projeto Nimrod (Abramson et al., 1995), que fornece uma linguagem de modelagem para declarar parâmetros para execução de uma tarefa. O Nimrod é executado com um bom desempenho em ambientes que possuem um conjunto estático de recursos. Porém, em ambientes altamente heterogêneos e dinâmicos a sua implementação é inviável, por causa da dificuldade em modelar estes recursos (Buyya et al., 2000).

Para superar a dificuldade encontrada, Buyya et al. (2000) criaram o Nimrod/G, que utiliza o protocolo MDS do Globus Toolkit para descobrir recursos da grade.

A arquitetura do Nimrod/G é organizada em cinco componentes. O primeiro componente atua como uma interface que permite que o usuário controle e supervisione um determinado experimento. Essa interface funciona como um console de monitoramento, listando o *status* de todos os processos. Esse componente permite ainda que um experimento seja iniciado em uma máquina, mas monitorado e controlado de uma outra máquina, por um usuário diferente.

O segundo componente atua como um agente que controla os *jobs*, sendo responsável pelo gerenciamento e pela manutenção do experimento. Este componente declara os parâmetros do experimento, cria os *jobs*, mantém o *status* dos *jobs*, interage com os clientes e outros componentes da arquitetura.

O terceiro componente é o escalonador. Esse componente é responsável por descobrir e selecionar os recursos e por alocar os *jobs*. O serviço de descoberta dos recursos é realizado por meio do protocolo MDS do Globus Toolkit. O algoritmo para seleção dos recursos tenta minimizar o custo de computação escolhendo o

recurso que executa a tarefa com o menor tempo de resposta (*deadline*).

O quarto componente, chamado de *Dispatcher*, inicia a execução da tarefa no recurso selecionado e mantém atualizado o agente que controla o *status* da tarefa. O último componente da arquitetura é responsável por executar os comandos de execução do *Dispatcher* e enviar os resultados de volta para o componente que monitora a execução da tarefa.

### 3.4.6 Legion

O Legion é um ambiente de metacomputação, orientado a objetos, que tem por objetivo garantir tolerância a falhas, segurança, tornando o ambiente de grade computacional altamente eficiente e fácil de programar. Neste *software* todos os componentes de interesse do sistema são representados por objetos e todos os objetos são instâncias de classes definidas (Grimshaw;Wulf, 1997).

A comunicação entre os objetos é estabelecida por meio de uma API, que independe de linguagem de programação ou protocolo de comunicação. Objetos que não estão executando nenhuma função no sistema são desativados e armazenados. Um objeto é automaticamente reativado quando um outro objeto quer comunicar-se com ele.

Um dos principais serviços fornecidos pelo Legion é composto por objetos essenciais que realizam funções de invocação, eventos de processamento e, descoberta e gerenciamento de metadados. Esses objetos são chamados de *vaults* e *hosts*. Os *hosts* são objetos que abstraem o conceito de capacidade das máquinas e são responsáveis por criar instâncias de objetos nos processadores. Os *vaults* são objetos que encapsulam o conceito de armazenamento persistente.

O escalonamento no ambiente Legion pode ser realizado utilizando desde algoritmos simples até heurísticas mais avançadas. Existem três componentes envolvidos no escalonamento: O *scheduler*, que realiza o mapeamento dos objetos para os recursos; o *enactor*, que é responsável por negociar a utilização dos recursos; e o *monitor*, que analisa a execução da aplicação e solicita o reescalonamento, se

necessário (Vianna, 2005).

Legion possui um método chamado *Collection* que permite que novas funcionalidades sejam adicionadas dinamicamente. Além disso, possui implementações de bibliotecas paralelas, MPI e PVM, e suporte nativo a linguagens de programação que permitem a implementação de aplicações paralelas, como por exemplo Java.

### 3.4.7 NetSolve

NetSolve é uma aplicação cliente-servidor projetada para resolver problemas computacionais de natureza científica. Para tanto, utiliza heurísticas de balanceamento de carga para fazer uso dos melhores recursos interconectados pela rede, a fim de obter um desempenho melhor na execução das tarefas (Casanova;Dongarra, 1995).

O sistema NetSolve é composto por máquinas heterogêneas fracamente conectadas, ou seja, máquinas que fazem parte de uma mesma rede local ou conectadas através da internet. O principal objetivo deste sistema é criar um conjunto formado por sistemas NetSolve independentes, em diferentes localizações, fornecendo diferentes serviços.

Na prática, um cliente NetSolve conecta-se a um agente externo submetendo uma determinada tarefa para execução. O agente externo fica com a responsabilidade de encontrar o melhor recurso NetSolve de acordo com o tamanho e a natureza do problema, e a localização do cliente. A comunicação existente entre os diversos recursos do sistema NetSolve é estabelecida através de uma camada de *socket* que utiliza o protocolo TCP/IP (Casanova;Dongarra, 1995).

Para facilitar a utilização por parte do usuário, NetSolve disponibiliza um *script* CGI para listar os recursos computacionais e agentes disponíveis no sistema, e um outro *script* com uma lista de problemas que podem ser resolvidos pelo sistema.

Para obter um melhor desempenho e confiança, NetSolve implementa técnicas de balanceamento de carga e tolerância a falhas. Para resolver o problema do balanceamento de carga, NetSolve tenta escolher o melhor recurso que, segundo

algum critério, executa melhor um determinado problema.

O problema do balanceamento de carga é resolvido calculando inicialmente qual é a melhor máquina que possui o menor tempo de execução para uma determinada tarefa. Em seguida, o desempenho da máquina é estimado tomando como base o modelo matemático descrito na Equação 3.1. Onde $p$ é o desempenho estimado, $w$ a carga de trabalho, $n$ o número de processadores na máquina e $P$ significa o desempenho da máquina quando nenhum outro processo está executando na CPU. Para o balanceamento de carga, também são levados em consideração critérios como características da rede, latência e largura de banda, e complexidade dos algoritmos (Casanova;Dongarra, 1995).

$$p = \frac{P \times 100 \times n}{100 \times n + max(w - 100 \times (n - 1), 0)} \tag{3.1}$$

NetSolve oferece um mecanismo de tolerância a falhas dividido em diferentes níveis. A detecção de falhas mais simples no NetSolve ocorre quando um cliente ou servidor não consegue estabelecer uma conexão com outro servidor. Quando isso ocorre, um relatório contendo os erros é enviado para o agente NetSolve e, se após um determinado tempo ele não for reativado, será removido do sistema.

Quando um agente recebe um relatório contendo erros de falha na comunicação, ele devolve para o cliente uma lista contendo todos os recursos, ordenada de acordo com a capacidade de cada um deles para resolver o problema. Se ainda assim nenhum destes servidores resolver o problema, o cliente solicita uma nova lista, enviando para o servidor os erros ocorridos com cada servidor da lista anterior. Este processo deve ser realizado de forma transparente para o usuário. A principal vantagem do uso da lista é a redução da comunicação entre cliente e servidor, porém, todo esse processo aumenta o tempo de execução total de uma tarefa (Casanova;Dongarra, 1995).

Atualmente o projeto NetSolve passou a chamar-se GridSolve e utiliza um coleção de diversos outros *middlewares* existentes para executar os mecanismos des-

critos nesta seção.

### 3.4.8 EasyGrid

Alguns dos *middlewares* apresentados nas seções anteriores visam a construir um modelo de escalonamento baseado na disponibilidade dos recursos da Grade, utilizando Sistemas Gerenciadores de Recursos (RMS) para monitorar e analisar as informações de cada recurso com o objetivo de permitir uma utilização mais eficiente.

Porém, apenas o monitoramento dos recursos não é suficiente para que aplicações sejam executadas com alto desempenho na Grade. As características de cada aplicação devem ser consideradas para que adaptações mais eficientes possam ser realizadas no escalonamento. Informações de cada aplicação podem ser obtidas por intermédio do sistema Gerenciador de Aplicações (AMS) (Vianna, 2005).

Um dos *middlewares* que utiliza o AMS para escalonamento de aplicações em Grade, é a ferramenta EasyGrid. Essa ferramenta gera automaticamente, a partir de uma aplicação paralela, uma aplicação capaz de ser executada eficientemente na Grade. Esse *software* é composto por ferramentas de escalonamento, tolerância a falhas e gerenciamento de tarefas, levando em consideração o ambiente heterogêneo da Grade e as características de cada aplicação (Mendes, 2004).

O projeto EasyGrid propõe a transformação de aplicações MPI (*Message-Passing Interface*) em aplicações *system-aware*, ou seja, que são capazes de adaptar-se às alterações do ambiente. Para alcançar esse objetivo, EasyGrid realiza o escalonamento das aplicações em duas etapas. Na primeira etapa é realizado um escalonamento estático por meio de algoritmos de escalonamento *list scheduling*. Na segunda etapa, com base nas informações obtidas do escalonamento estático e nas alterações ocorridas no ambiente durante a execução da aplicação, é realizado o escalonamento dinâmico (Vianna et al., 2004).

O escalonamento das tarefas é realizado com base na representação da aplicação por meio do DAG. Como o objetivo da ferramenta é investigar os efeitos e a

granularidade da estrutura de um programa, o EasyGrid utiliza um mecanismo, chamado *Task Graph Generator*, para criar representações de DAG e a partir da representação são geradas aplicações MPI sintéticas (Vianna, 2005).

### 3.4.9 GrADS

O projeto **GrADS** (Berman et al., 2005) é um meta-escalonador responsável por gerenciar os recursos disponíveis na Grade e as aplicações submetidas pelos usuários. O principal objetivo desta ferramenta é fornecer mecanismos de desenvolvimento que facilitem a implementação dos algoritmos e a execução das aplicações paralelas. As ferramentas NWS e MDS são utilizadas pelo GrADS para descobrir, monitorar e criar uma lista com os recursos disponíveis no sistema. Este meta-escalonador realiza a submissão de aplicações monitorando as que foram previamente mapeadas para os recursos, a fim de minimizar o impacto de novas submissões e, se necessário, executar a migração de tarefas entre os recursos para melhorar o desempenho, prevenir uma possível degradação do sistema ou para garantir tolerância a falhas. Apesar do meta-escalonador facilitar o desenvolvimento de aplicações para a Grade, dependendo do número de tarefas e das comunicações existentes entre elas, a criação do modelo de aplicação pode ser extremamente complexa. Além disto, o escalonamento sem considerar tarefas alocadas anteriormente pode aumentar o custo de execução devido ao processo de reescalonamento.

### 3.4.10 GridWay

A ferramenta **GridWay** (E. Huedo;Llorente, 2005) gerencia os processos na Grade seguindo os seguintes passos: descoberta e seleção dos recursos, e preparação das tarefas. Na etapa de descoberta e seleção dos recursos, a ferramenta possui um *script* que realiza uma consulta no MDS. A fase de preparação das tarefas envolve o escalonamento, a submissão, o monitoramento e a migração de tarefas. Porém, assim como nas outras ferramentas, cada aplicação é escalonada individualmente e, em caso de redução no desempenho da execução, é realizado um processo de

migração de tarefas para outros recursos.

## 3.5 Conclusão do Capítulo

Neste capítulo foram apresentados conceitos, ferramentas e características relacionadas ao escalonamento em Grade, cujo principal objetivo foi fazer uma breve discussão acerca das tecnologias que favorecem ou facilitam o escalonamento de tarefas.

Com isto, foi possível absorver os pontos relevantes considerados durante a implementação da ferramenta apresentada neste trabalho, destacando a importância e a dificuldade no gerenciamento de recursos e aplicações.

CAPÍTULO

4

# Algoritmo de Escalonamento baseado em *Slots*

No capítulo 3 foram apresentados diversos algoritmos que compõem o estado atual dos escalonadores mais freqüentemente utilizados no ambiente de Grade Computacional.

Conforme foi visto, estes algoritmos dividem-se, principalmente, em uma fase de priorização das tarefas e uma fase de seleção dos processadores.

Baseando-se no estudo apresentado destes algoritmos, este capítulo apresenta algumas políticas de priorização de tarefas e de seleção de processadores que conduziram à proposta de um novo algoritmo de escalonamento global e com informações de estado (*stateful*) das aplicações e dos recursos, e que é utilizado na ferramenta SLOT.

As principais contribuições esperadas com o uso do algoritmo abrangem: reduzir o tempo de execução das aplicações quando comparado com algoritmos tradicionais, evitar que a execução de uma nova aplicação submetida ao sistema

comprometa a execução de tarefas previamente escalonadas, e prover um melhor aproveitamento dos recursos com a diminuição dos intervalos de tempo ociosos.

## 4.1 Política de Priorização das Tarefas

Nesta fase, os algoritmos de escalonamento ordenam as tarefas que ainda não foram escalonadas em uma fila de prioridade, de acordo com alguns critérios como, por exemplo, custo de execução e custo de comunicação na troca de dados.

Com o intuito de facilitar a compreensão destas políticas, utilizaremos o modelo formal apresentado em Topcuouglu et al. (2002). Neste modelo, uma tarefa é denotada por $n$. O custo computacional associado a cada tarefa $n_i$ é representado por $w_i$, enquanto o custo de comunicação dos dados enviados de uma tarefa $n_i$ a uma tarefa $n_j$ é formalizado através da variável $c_{i,j}$.

Definido o modelo, a primeira política a ser apresentada, considerada a mais simples, ordena as tarefas de acordo com seus custos computacionais:

$$prioridade(n_i) = \langle w_{i-1} > w_i > w_{i+1} \rangle \tag{4.1}$$

Porém, esta política não utiliza outras variáveis como o custo da troca de dados entre as tarefas, tornando-se menos eficiente quando utilizada em aplicações com alto custo de comunicação. Por outro lado, Topcuouglu et al. (2002) apresentam uma política que ordena a fila de prioridades de acordo com o $rank_u$ (*Rank Up*) de cada tarefa. O $rank_u$ de uma tarefa $n_i$ é o custo do caminho crítico começando da tarefa $n_i$ até a tarefa de saída ($n_{saida}$) e pode ser representado de acordo com a função a seguir:

$$rank_u(n_i) = \begin{cases} \overline{w_i}, & w_i \text{ igual à tarefa final} \\ \overline{w_i} + max_{n_j \in suc(n_i)}(\overline{c_{i,j}} + rank_u(n_j)), & w_i \text{ diferente da tarefa final} \end{cases} \tag{4.2}$$

Nesta equação, $suc(n_i)$ é o conjunto de todas as tarefas sucessoras da tarefa $n_i$, e

$\overline{w_i}$ representa o custo médio de execução da tarefa $n_i$ nos processadores disponíveis no ambiente e é obtido com a equação

$$\overline{w_i} = \frac{1}{q} \sum_{m=1}^{q} \left( \frac{\varepsilon_i}{\rho_m} \right), \tag{4.3}$$

sendo $q$ o número de processadores, $\varepsilon_i$ o custo da tarefa $n_i$ e $\rho_m$ a capacidade de processamento do processador $m$.

Ainda em relação à equação 4.2, $\overline{c_{i,j}}$ é o custo médio dos dados transferidos entre as tarefas $n_i$ e $n_j$ e pode ser calculado segundo a equação 4.4. Nesta equação, $\overline{L}$ representa a latência média e $\overline{B}$ a largura de banda média entre os canais de comunicação que interligam os recursos.

$$\overline{c_{i,j}} = \overline{L} + \left( \frac{data_{i,j}}{\overline{B}} \right) \tag{4.4}$$

Topcuouglu et al. (2002) apresentam também uma outra técnica similar que é o cálculo do $rank_d$ (*Rank Down*), onde o caminho crítico de uma tarefa $n_i$ é computada considerando os predecessores desta tarefa até a tarefa de entrada. A equação 4.5 define o cálculo do $rank_d$.

$$rank_d(n_i) = \begin{cases} 0, & w_i \; igual \; a \; tarefa \; de \; entrada \\ max_{n_j \in pred(n_i)}(rank_d(n_j) + \overline{w_i} + \overline{c_{i,j}}), & w_i \; diferente \; da \; tarefa \; de \; entrada \end{cases} \tag{4.5}$$

Ainda uma outra política apresentada por Topcuouglu et al. (2002), calcula a prioridade de uma tarefa a partir da soma do $rank_u$ e do $rank_d$.

$$prioridade(n_i) = rank_u(n_i) + rank_d(n_i) \tag{4.6}$$

Como foi dito anteriormente, as políticas de priorização de tarefas utilizadas no algoritmo de escalonamento definem a ordem com que cada tarefa é escalonada, podendo afetar o desempenho final da execução das tarefas. Portanto, no Capítulo

7 foram avaliadas cada uma destas políticas e a mais adequada para a classe de problemas testados foi utilizada na composição o algoritmo de escalonamento proposto.

## 4.2 Política de Seleção dos Processadores via *Slots*

No capítulo 3 são apresentadas algumas políticas de seleção de processadores. Nestas políticas o algoritmo de escalonamento seleciona uma tarefa que não foi escalonada a partir da fila de prioridade e realiza o mapeamento para o recurso que a execute com o maior desempenho.

Nesta seção será apresentada uma política de seleção de processadores que se baseia no algoritmo HEFT e utiliza a política de inserção por *slots* (ou inserção por fatias de tempo). Este algoritmo mantém uma lista de todas as tarefas submetidas aos processadores. A cada nova submissão, o algoritmo consulta esta lista procurando intervalos entre duas tarefas previamente escalonadas que sejam suficientemente grandes para executar a nova tarefa.

A Figura 4.1, ilustra o escalonamento de três aplicações (A, B e C) com o algoritmo baseado nos *slots* de tempo livre dos processadores. Supondo-se que exista entre os processadores a relação P1 $>$ P2 $>$ P3, quanto ao poder computacional, e considerando que as aplicações A e B foram escalonadas primeiro. No momento do escalonamento da tarefa C1 da aplicação C, algoritmos de escalonamento normalmente tendem a optar pelo uso do processador P3, por analisarem apenas o poder computacional individual de cada recurso ou o próximo tempo livre de cada processador. Com o uso do escalonamento global, que permite conhecer as fatias de tempo livre de todos os recursos, pode-se optar pelo uso de um recurso com menos poder computacional, mas que, por minimizar o início da execução e o instante de conclusão da tarefa, consegue concluir a execução em um tempo inferior em relação aos demais processadores

Este modelo de escalonamento de tarefas, formalmente apresentado no Algo-

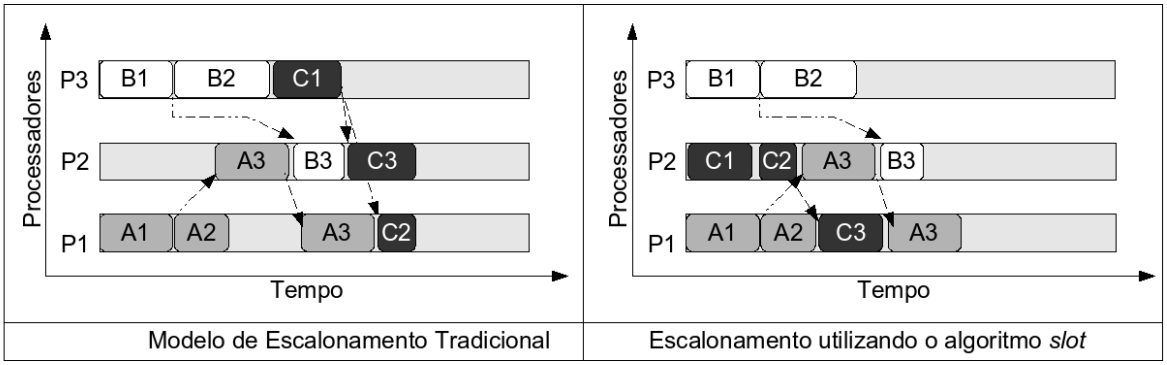


**Figura 4.1:** Resultado final do escalonamento das aplicações A, B e C nos processadores P1, P2 e P3 com algoritmos tradicionais e com o algoritmo *slot*, respectivamente. As setas representam dependência entre as tarefas.

ritmo 1, possui custo computacional maior que os algoritmos de escalonamento tradicionais, porém, tem como principais objetivos: (1) escalonar todas as tarefas de forma eficiente, não importando qual o processador que executa cada tarefa mais rapidamente, mas obtendo um menor *makespan* de cada aplicação e, conseqüentemente, do conjunto total de aplicações, e (2) diminuir os intervalos em que os recursos ficam ociosos, aproveitando assim, melhor o poder computacional de cada máquina.

Como pode ser observado no Algoritmo 1, inicialmente um DAG é submetido para a execução e, em seguida, cada tarefa do grafo receberá um valor de prioridade de acordo com uma das políticas apresentadas na seção 4.1.

Na fase de **Escolha do Processador**, uma tarefa $n_i$ que ainda não foi escalonada é selecionada e é verificado se todos os seus predecessores foram previamente escalonados. Em seguida, para cada máquina $q$ disponível, é estimado o custo computacional através da divisão do custo da tarefa pelo poder de processamento (passo 9), conforme apresentado na equação 4.7.

$$w_{i,m} = \left(\frac{\varepsilon_i}{\rho_m}\right) \tag{4.7}$$

O custo computacional definirá o tamanho da tarefa no processador e será uti-

**Algoritmo 1** Algoritmo de escalonamento baseado em *slots*

1: **while** ∃ aplicações pendentes **do**
2: Obtenção do DAG da aplicação;
3: Priorização das tarefas;
4: **while** ∃ tarefa $n_i$ não escalonada **do**
5: **if** ∃ predecessor da tarefa $n_i$ não escalonada **then**
6: Escalona todos os predecessores de $n_i$;
7: **end if**
8: **while** ∃ processador $q$ não avaliado para escalonamento de $n_i$ **do**
9: Cálculo do custo de computação da tarefa $n_i$ na máquina $q$ (Equação 4.7);
10: Cálculo do custo de comunicação entre a máquina que executará a tarefa predecessora de $n_i$ mais lenta e a máquina $q$ (Equação 4.10);
11: Escolha do *slot* a partir do instante em que chegam os dados da última tarefa predecessora e que seja maior ou igual ao "tamanho" da tarefa ($w_{i,q}$) (Equação 4.8);
12: **if** $q$ garante o menor tempo de resposta para $n_i$ **then**
13: Seleciona $q$ para executar $n_i$;
14: **end if**
15: **end while**
16: **end while**
17: **end while**

lizado para encontrar *slots* livres com o mesmo tamanho ou superior, ou seja, onde o tempo de execução da tarefa $n_i$, $t_i^E$, é menor ou igual que o valor resultante da subtração entre o instante de início do próximo *slot* ocupado pela tarefa $n_k$, $t_k^I$, e o instante final de execução do *slot* predecessor ocupado pela tarefa $n_j$, $t_j^F$, como na Fórmula 4.8:

$$t_i^E \leq t_k^I - t_j^F \tag{4.8}$$

O tempo de execução $t_i^E$ de uma tarefa em um processador $q$ é o somatório entre o custo da tarefa($w_i$) no processador $q$ e o tempo de execução da tarefa predecessora mais lenta:

$$t_i^E = (w_{i,q}) + max_{j \epsilon pred(i)} \left( t_j^F \right) \tag{4.9}$$

O passo 10 obtém o instante em que todos os dados enviados pelas tarefas

predecessoras estão disponíveis, para evitar que uma determinada tarefa seja escalonada em um *slot* e inicie o processamento antes da chegada dos pré-requisitos necessários. Este instante é obtido a partir da equação 4.10, sendo que $lat(i,j)$ e $band(i,j)$ é, respectivamente, a latência e a largura de banda do canal de comunicação que conecta as máquinas que irão executar as tarefas $n_i$ e $n_j$. $dados(i,j)$ é a quantidade de dados transferidos entre da tarefa $n_i$ à $n_j$.

$$I_{i,q} = max_{j \epsilon pred(i)} \left( lat(i,j) + \frac{dados(i,j)}{band(i,j)} \right) \tag{4.10}$$

Diversos algoritmos de escalonamento como HEFT, CPOP e DSC, analisam os canais de comunicação considerando uma média da latência e uma média da largura de banda entre os recursos. Com o auxílio de ferramentas de monitoramento, o escalonador SLOT trata a rede como um recurso da Grade. O conhecimento em tempo de execução das cargas e da largura de banda entre dois recursos permite um cálculo mais preciso e próximo de valores reais do que em relação aos demais algoritmos.

Então, tendo conhecimento de uma estimativa do custo de execução de uma tarefa e do instante de chegada do último dado enviado pelos predecessores, uma busca é realizada para encontrar um *slot* disponível para execução da tarefa.

Essa fase é repetida enquanto existirem máquinas não avaliadas sendo selecionada a máquina que apresentar o menor *finish time* (instante da conclusão da execução da tarefa).

A execução do algoritmo de escalonamento determina uma estimativa dos valores de início e conclusão da execução de cada tarefa para serem analisados com os valores reais obtidos durante o monitoramento da execução em cada recurso.

Contudo, a busca por *slots* livres em cada máquina disponível na Grade pode ser trabalhosa, variando de acordo com a quantidade de recursos. Uma solução para este problema consiste em limitar o espaço de busca no conjunto de recursos disponíveis de acordo com as características da aplicação.

## 4.3 Política de Seleção dos Processadores com Sobreposição de *Slots*

Nesta seção será apresentada uma variação do Algoritmo 1 obtida através da substituição da política de inserção.

No algoritmo de escalonamento original proposto, um *slot* era considerado satisfatório quando o tempo de execução de uma tarefa era igual ou inferior ao tamanho do *slot*. Porém, é possível que mesmo aplicando esta política ainda permaneçam fatias de tempo ociosas nos processadores que não foram preenchidas, ou seja, *slots* de tempo insuficientes para execução de uma tarefa.

Baseando-se nisto, esta seção apresenta uma política de inserção mais flexível que permite o escalonamento de uma tarefa sobrepondo, até um limite máximo permitido, um *slot* ocupado. O principal objetivo desta abordagem é permitir explorar o máximo da capacidade de execução de aplicações no ambiente real. De maneira geral, a sobreposição de *slots* significa o compartilhamento de tempo no processador alvo para execução de 2 tarefas.

Durante a realização dos testes, a sobreposição de *slots* apresentou um ganho no desempenho das aplicações, apesar de algumas tarefas destas aplicações sofrerem um aumento do tempo de execução. Isso pôde ser observado em situações onde uma tarefa pronta para execução tinha o seu *start time* adiado devido à falta de *slot* de tempo suficiente para sua execução. Com o uso da sobreposição este problema foi atenuado, já que o instante de início da tarefa foi reduzido. Além disso, essa abordagem pode permitir uma redução no *makespan* de todas as aplicações submetidas para a Grade, apesar de haver casos em que o *makespan* de uma aplicação foi aumentado.

A Figura 4.2 exemplifica esta situação. Inicialmente é importante saber que as tarefas 1, 2, 3 e 4 não necessariamente fazem parte da mesma aplicação e os números representam a ordem de chegada no escalonador, podendo não existir

relação entre elas.

Sendo assim, a situação **A** da figura 4.2 representa um processador com as tarefas 1, 2 e 3 alocadas para execução e o instante da chegada da tarefa 4. A tarefa 4 chega no instante $t_0$, porém, neste momento a tarefa 1 está sendo executada. O algoritmo de escalonamento com a política de inserção baseado em *slots* tentaria alocar a tarefa recém chegada em um intervalo de ociosidade do processador, mas como não há, a tarefa 4 é alocada no final, depois da execução da tarefa 3, como pode ser visto na situação **B**. Porém, com o uso da política de sobreposição, o escalonador tenta alocar a tarefa 4 em um *slot*, sobrepondo, até um limite máximo permitido, tarefas previamente alocadas, representada pela situação **C**. A parte hachurada no gráfico representa o instante em que duas tarefas estão compartilhando o uso do processador. Como pode ser visto, a sobreposição da tarefa 4 nas tarefas 1 e 2 causa uma perturbação no sistema. A perturbação na tarefa 1 resultou em um aumento $\delta_1$ no tempo total de execução desta tarefa. O mesmo acontece com as tarefas 4 e 2, onde $\delta_2$ e $\delta_3$ representam, respectivamente, o aumento do tempo de execução destas tarefas quando comparado com os tempos na situação **B**. Porém, a situação **C** levaria vantagem quando analisado o tempo de execução total de todas as tarefas.

A alteração na política de inserção para realização da sobreposição foi realizada da linha 11 do Algoritmo 1, com a substituição da equação 4.8 pela equação 4.11. Nesta nova equação, $\Delta$ representa a variação da sobreposição máxima permitida, ou seja, se $\Delta = 0.05$, significa que é permitido sobrepôr até no máximo 5% de um *slot*.

$$t_i^E \leq \left(t_k^I - t_j^F\right) + \left(\frac{\Delta * t_i^E}{100}\right) \tag{4.11}$$

O escalonamento com sobreposição de *slots* visa a redução maior no tempo de resposta de diversas aplicações submetidas para execução do que em relação à política original de inserção baseado em *slots*. Contudo, pode existir situações em

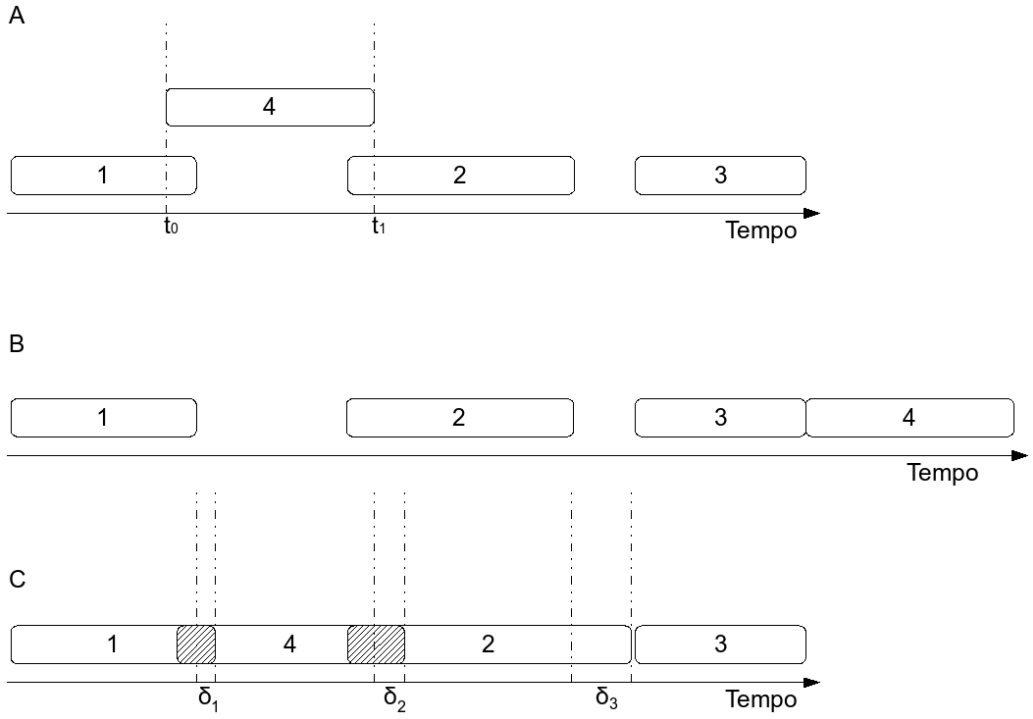


**Figura 4.2:** Representação da sobreposição de *slots*. A situação **A** representa o instante da chegada de uma tarefa 4, enquanto as tarefas 1, 2 e 3 já se encontram no sistema. A situação **B** representa o escalonamento usando *slots*. A situação **C** representa o uso da sobreposição de *slots*.

que a sobreposição de *slots* proporcione um aumento no tempo de execução de uma tarefa $n_i$ sobreposta e, conseqüentemente, o início da execução de uma tarefa $n_j$ sucessora que dependia dos dados enviados por $n_i$ é atrasado. Esta situação pode desencadear um efeito "cascata" degradando o tempo de execução de todas as tarefas no sistema.

Experimentos realizados na seção de resultados apresentam o efeito do comportamento desta abordagem através de um estudo do limite máximo de sobreposição para algumas classes de aplicações.

## 4.4 Conclusão do Capítulo

Este capítulo apresentou algumas políticas de priorização de tarefas e duas políticas de inserção baseadas em *slots*. Estas políticas foram testadas e simuladas

como apresentado no capítulo 7, e as que obtiveram melhor desempenho foram adicionadas à ferramenta SLOT, que será apresentada no capítulo a seguir.

Quanto ao algoritmo de sobreposição apresentado, uma variação mais rígida analisaria o impacto sofrido pelas tarefas sucessoras de uma tarefa sobreposta. Esta variação recalcularia e atualizaria o *start time* e o *finish time* de cada tarefa que depende da execução da tarefa sobreposta.

Uma outra análise que pode ser feita consiste em utilizar o algoritmo de sobreposição em ambientes de Grade compostos por máquinas *multicores* e/ou HT (*Hyper-Threading Technology*). Como a grande maioria dos algoritmos de escalonamento consideram uma máquina como um único recurso sem considerar o número de processadores, a sobreposição pode trazer ganhos significativos. Para isto, cada processador, e não cada nó, poderia ser considerado um recurso e o escalonamento poderia mapear uma tarefa para cada processador, usufruindo da vantagem do baixo custo de comunicação entre os processadores de uma mesma máquina.

CAPÍTULO

5

# A ferramenta SLOT

Diante da heterogeneidade dos recursos distribuídos que compõem um ambiente de Grade Computacional e da instabilidade de suas operações, a escolha por parte dos usuários de onde executar cada aplicação tornou-se uma atividade complexa, principalmente no caso de aplicações paralelas, que são executadas em múltiplos processadores simultaneamente (Vianna et al., 2004).

De maneira geral, aplicações paralelas são normalmente compostas de coleções de tarefas que necessitam ser executadas em uma ordem determinada pelas dependências de controle e dados (Berman et al., 2005). Essa relação de dependência caracteriza um *workflow* (Yu;Buyya, 2005a), o qual pode ser estruturado em forma de DAG (*Directed Acyclic Graph* - Grafo Acíclico Direcionado), caso não existam ciclos de dependências.

A representação do modelo de aplicação através de DAGs é amplamente utilizada no escalonamento de aplicações paralelas por parte dos principais algoritmos como, por exemplo, DSC (Yang;Gerasoulis, 1994), HEFT e CPOP (Topcuouglu et al., 2002), e de ferramentas de escalonamento, como Condor DAGMan (DAGMan,

2007), EasyGrid (Vianna et al., 2004), GridWay (E. Huedo;Llorente, 2005) e ASKALON (Fahringer et al., 2005). Porém, a criação/obtenção de grafos das aplicações em formato apropriado para o escalonamento ainda é um problema significativo (Saad et al., 2006).

O conhecimento do ambiente computacional disponível também é necessário para o escalonamento das aplicações, especialmente no ambiente heterogêneo e dinâmico da Grade.

Cabe ao algoritmo de escalonamento, usando informações sobre características das aplicações e dos recursos disponíveis, realizar um mapeamento eficiente de tarefas para os recursos e de forma transparente para o usuário.

O escalonamento individualizado de cada aplicação, como é geralmente realizado, leva em consideração as características da aplicação e o estado dos recursos em um dado instante, tal como coletado de ferramentas de monitoramento do estado da Grade. Esta abordagem, contudo, pode não ser adequada, uma vez que os estados dos recursos na Grade podem variar rapidamente com o fim da execução de tarefas que os ocupavam, ou com a ativação de tarefas que estavam pendentes à espera de recursos.

Neste sentido, este trabalho apresenta uma ferramenta de escalonamento centralizado, denominada **SLOT** (*Scheduling Lots Of Tasks*), que realiza o escalonamento de aplicações paralelas de forma global, gerenciando todas as tarefas submetidas para a Grade e procurando eliminar a ociosidade dos recursos, reduzindo os períodos de tempo inutilizados entre a finalização de uma tarefa e início de uma outra.

Embora a utilização do modelo de escalonamento centralizado possa ser um fator limitante à escalabilidade do sistema, observa-se que o conhecimento do estado global da Grade, incluindo seus recursos e cargas, pode propiciar uma melhor utilização dos recursos e um menor tempo de execução para as tarefas.

Para alcançar este objetivo, o ambiente de Grade vislumbrado como cenário na criação da ferramenta caracteriza-se pelo dinamismo dos recursos, pela execução

de aplicações que demandam alto custo de processamento e pela submissão de diversas aplicações simultaneamente.

Baseando-se neste cenário de Grade, a arquitetura geral da ferramenta desenvolvida pode ser observada na Figura 5.1. Na parte inferior estão os recursos disponíveis na Grade. Em um nível acima, estão o **NWS** e o **Globus Toolkit**, com os serviços para descoberta e monitoramento de recursos, autenticação dos usuários, transferência de dados e gerenciamento de tarefas. A camada acima é composta pelo escalonador proposto na capítulo anterior. Por fim, na camada superior tem-se uma interface utilizada pelos usuários para autenticação e submissão de tarefas.

A execução de uma aplicação na Grade usando esta ferramenta subdivide-se em 3 etapas:

- Etapa 1: Autenticação dos usuários, descoberta e monitoramento dos recursos:
  - **(a)** Criação do modelo arquitetural, que representa os recursos da Grade, usando o **MDS** (*Monitoring and Discovery Service*) (Foster, 2005) do **Globus** e o **NWS**;
  - **(b)** Autenticação do usuário na Grade via GSI (*Grid Security Infrastructure*) (Foster, 2005) para ter permissão para executar as tarefas nos recursos;
- Etapa 2: Escalonamento da aplicação:
  - **(a)** Obtenção do grafo e geração do DAG da aplicação;
  - **(b)** Escalonamento utilizando *slots*;
- Etapa 3: Monitoramento de cada tarefa individualmente, com a ferramenta **DUROC** do **Globus** (Foster, 2005), que também informa sobre os instantes de ativação e de finalização da execução nos recursos;

Com o intuito de facilitar a compreensão do mecanismo de execução e gerenciamento de tarefas e recursos, a camada que representa a ferramenta SLOT na

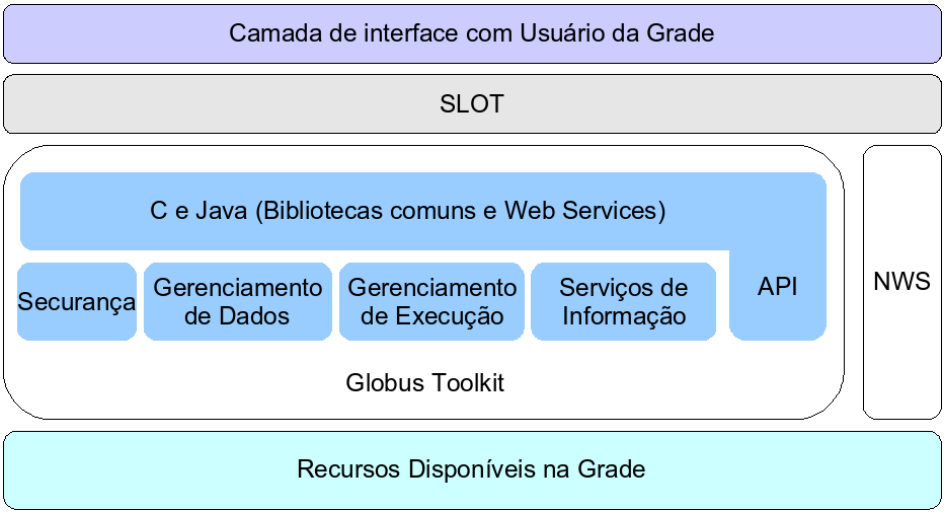


**Figura 5.1:** Visão geral da Arquitetura do sistema.

arquitetura foi expandida, conforme mostra o Diagrama UML (*Unified Modeling Language*) na figura 5.2. Este Diagrama apresenta o funcionamento da ferramenta, passando desde os estados de monitoramento dos recursos até a geração do DAG e o escalonamento da aplicação.

A ferramenta SLOT foi desenvolvida para funcionar como um processo *daemon*, ou seja, após a inicialização, a ferramenta permanece executando em *background*, aguardando por uma requisição do usuário. No instante em que a ferramenta é inicializada, duas *threads* são criadas. A primeira *thread* realiza, com as ferramentas NWS e MDS, o monitoramento e a descoberta dos recursos disponíveis na Grade, criando e atualizando o modelo arquitetural do ambiente em tempo de execução. A segunda *thread* permanece aguardando até que o usuário submeta uma aplicação para execução na Grade.

O modelo arquitetural pode ser obtido tanto com o uso das ferramentas NWS e MDS quanto através de um arquivo XML fornecido pelo usuário. O modelo de aplicação também pode ser obtido de duas formas: ou usando um grafo fornecido pelo usuário juntamente com a aplicação para execução ou através do monitoramento e da geração automática do grafo da aplicação.

Depois da obtenção do modelo de aplicação e do modelo arquitetural, uma bar-

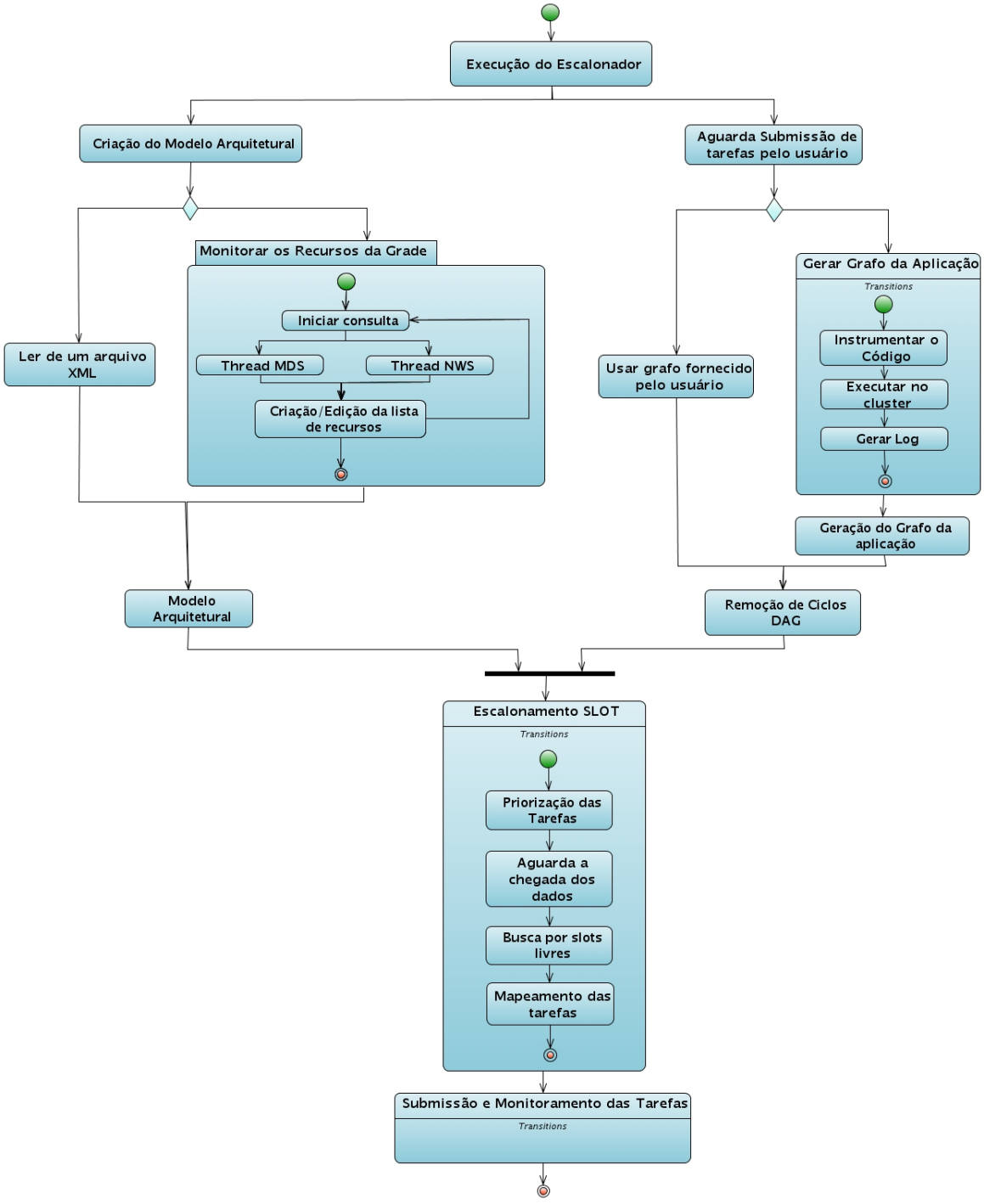


**Figura 5.2:** Diagrama de Estado da Ferramenta SLOT.

reira foi implementada para sincronizar a chegada dos dados resultantes destes dois processos.

Tendo o escalonador informações sobre cada tarefa do DAG e sobre cada recurso ativo na Grade, os próximos estados do diagrama descrevem o escalonamento das tarefas, a submissão para os recursos e o monitoramento de cada tarefa. Todos os estados serão descritos em mais detalhes nas seções a seguir.

## 5.1 Construção do Modelo Arquitetural

As máquinas que fazem parte da infra-estrutura de Grade são agrupadas dinamicamente em Organizações Virtuais (Foster et al., 2001), regidas por políticas de acesso, que permitem ceder e utilizar recursos computacionais, como por exemplo, processamento e espaço disponível em disco.

A coordenação dos recursos a serem utilizados tornou-se um desafio devido à grande quantidade de máquinas que participam de uma ou mais VOs. Com isso, descobrir e monitorar os recursos em um ambiente computacional distribuído e de larga escala tornou-se uma tarefa crítica e que influencia no desempenho da aplicação (Kee et al., 2006).

Para solucionar este problema, foram implementadas duas bibliotecas estáticas[1] que, ao serem associadas à ferramenta SLOT, fornecem mecanismos para monitorar o ambiente e montar, em tempo de execução, o modelo arquitetural dos recursos que compõem o sistema.

Uma das bibliotecas, chamada *libmds*, utilizada no monitoramento do ambiente real, foi implementada utilizando a API do componente MDS e fornece mecanismos para descobrir e monitorar informações sobre o estado e as configurações dos recursos computacionais.

A outra biblioteca, chamada *libnws*, foi implementada utilizando funções do NWS e coleta informações do canal de comunicação que interliga os recursos des-

[1] Uma biblioteca estática é uma coleção de objetos que podem ser associadas (*linking*) durante a compilação.

cobertos pelo MDS, como a latência e a largura de banda.

O modelo arquitetural é mantido durante toda a execução do escalonador e atualizado em intervalos de tempo definidos em arquivos de configuração do escalonador. Esta atualização pode ser efetuada por intermédio de uma nova consulta ao sistema ou através de predições fornecidas pelo NWS.

## 5.2 Construção do Modelo de Aplicação

Muitos algoritmos e ferramentas de escalonamento requerem um modelo de aplicação representado por DAG. Contudo, a criação de um modelo realístico do comportamento da aplicação pode ser uma tarefa complexa e geralmente é realizada pelo próprio desenvolvedor da aplicação, o que torna este procedimento exaustivo e suscetível a erros.

Para facilitar o processo de criação manual de DAG, Fahringer et al. (2005) apresenta uma linguagem de marcação baseada em XML, chamada AGWL (*Abstract Grid Workflow Language*), usada para descrever aplicações paralelas. Deelman (2003) apresenta um outro gerador de *workflow* que tenta criar uma ponte entre usuários da Grade e os mecanismos de execução de aplicações. Esta ferramenta gera um escalonamento a partir da descrição de uma aplicação de acordo com as tarefas e as suas comunicações. Porém, nenhuma delas realiza o processo de criação do DAG automaticamente sem intervenção do usuário.

Neste sentido, a ferramenta SLOT utiliza um mecanismo para obter as características da aplicação e, então, prover a geração automática do DAG. Este mecanismo obtém informações sobre cada aplicação através de técnicas de monitoramento apresentado em (Jacinto et al., 2007). A aplicação é executada em um *Cluster* composto por máquinas dedicadas e, através do monitoramento, são coletadas, dentre outras informações, o custo computacional para execução de cada tarefa e a quantidade total de dados transferidos.

Esse processo é realizado através da execução da aplicação MPI usando bibli-

otecas da ferramenta *Trace Collector* (GmbH., 2005). Os arquivos de *log* gerados pelo *Trace Collector* são analisados e informações relevantes, como por exemplo, tempo total de execução de cada tarefa, mensagens trocadas e quantidade de dados transmitidos, são extraídos (Jacinto et al., 2007).

A determinação das características da aplicação através do monitoramento de sua execução resulta em um grafo, porém, com possíveis ciclos de comunicação entre os nós. Estes ciclos representam um impecílio para execução dos algoritmos de escalonamento, devido à busca recursiva nos grafos para seleção das tarefas. A Figura 5.3 apresenta a execução da aplicação *Interger Sort* (IS) classe C do *Benchmark* paralelo NAS (NPB), desenvolvido pela divisão de Supercomputação Avançada da NASA (Bailey et al., 1994).

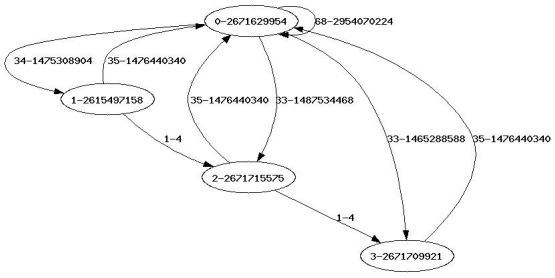


**Figura 5.3:** Grafo que representa a execução da aplicação real NAS classe C.

Com o intuito de resolver o problema dos ciclos nos grafos, foi desenvolvida uma solução baseada no algoritmo de busca em profundidade (*Depth-First Search* - DFS). Em Jacinto et al. (2007), o algoritmo original foi estendido para pesquisar por quaisquer arestas que levam a nós anteriormente visitados. Se isto ocorrer, um novo nó é criado, chamado nó virtual, e todas as arestas que anteriormente chegavam ao nó original serão redirecionadas para o nó virtual, removendo assim os ciclos.

Na Figura 5.4 podemos ver o DAG correspondente ao grafo da Figura 5.3, depois da remoção de ciclos.

Baseado no DAG criado pela remoção de ciclos do grafo gerado, a aplicação pode ser definida por $G = (V, \beta, E, \varepsilon, \omega)$, onde $V$ é um conjunto formado por $n$ nós (tare-

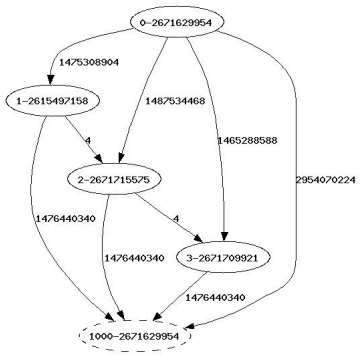


**Figura 5.4:** DAG que representa a execução da aplicação real NAS classe C. O nó tracejado representa um nó virtual

fas), $\beta$ é o conjunto de nós virtuais, e *E* é o conjunto de arestas (comunicação). $\varepsilon(i)$ representa o custo de execução do nó *i*, e $\omega(i, j)$ representa a quantidade de dados enviados entre os nós *i* e *j*. Antes da remoção de ciclos, o conjunto representado por $\beta$ é vazio.

Como resultado, SLOT pode realizar o escalonamento usando DAGs das aplicações obtidos automaticamente.

## 5.3 Escalonamento, submissão e monitoramento das tarefa

As seções anteriores apresentaram mecanismos presentes na ferramenta SLOT que são executados paralelamente para obter informações sobre cada tarefa da aplicação e sobre cada recurso disponível na Grade. Concluídas estas etapas, estas informações são utilizadas no escalonamento, como pode ser visto na Figura 5.2.

Para isto, a ferramenta SLOT utiliza os algoritmos de escalonamento *slot* e sobreposição de *slots*, os quais foram descritos no Capítulo 4. Estes algoritmos levam em consideração não apenas as tarefas que compõem a aplicação a ser escalonada, mas também todas as tarefas previamente alocadas nos recursos. Quanto à carga momentânea nos recursos, a ferramenta SLOT usa as ferramentas de monitoramento descritas na seção anterior para detectá-las e parte do pressuposto que as aplicações submetidas para os recursos passam por ela, ou seja, SLOT tem o co-

nhecimento global de todas as tarefas submetidas para execução em um recurso da Grade. Na Figura 5.2, a sub-máquina de estados "Escalonamento SLOT" apresenta os estados e as transições implementadas na ferramenta SLOT para execução do algoritmo de escalonamento mencionado anteriormente.

No primeiro estado deste diagrama, cada tarefa recebe um valor de priorização que irá determinar a sua ordem de execução (seção 4.1). Após este passo, as tarefas encontram-se prontas para serem escalonadas. Entretanto, se durante o escalonamento de uma tarefa $n_i$ existir alguma tarefa predecessora que ainda não foi escalonada, a tarefa vai para um estado de espera até que todas as suas predecessoras tenham sido escalonadas, e só então volta a ficar pronta para ser escalonada. Este processo pode ser observado nas linhas 5-7 do Algoritmo 1 na seção 4.2.

Por fim, no terceiro e no quarto estados desta sub-máquina, o algoritmo de escalonamento busca na lista de *slots* ocupados de cada recurso, fatias de tempo inutilizadas que minimizem o tempo de execução de cada tarefa, conforme descrito na seção 4.3.

Concluído o escalonamento das tarefas, o próximo estado da ferramenta é a submissão e o monitoramento de cada tarefa nos recursos da Grade. Para este fim, um arquivo RSL (*Resource Specification Language*) (Foster, 2005), como foi exemplificado na Tabela 3.1, é gerado. Além das informações passadas pelo escalonador são utilizadas informações fornecidas pelo usuário no instante da requisição para execução da aplicação, como por exemplo, o diretório com os arquivos necessários para a aplicação, as bibliotecas Globus, e o caminho completo onde encontra-se o programa executável.

A principal informação passada pelo escalonador é armazenada na variável *"resourceManagerContact"*. Esta variável armazena o recurso para qual uma determinada tarefa deve ser mapeada.

Então, logo que criado o arquivo RSL, a ferramenta inicia uma nova *thread* que utiliza a API DUROC (Foster, 2005) e fica responsável por atualizar o estado de

cada tarefa à medida que estas forem sendo monitoradas e, no final, armazena o instante exato de início e de conclusão da execução no recurso remoto. Para simplificar o uso destas funções, foi implementada uma outra biblioteca estática, chamada *libduroc*.

A obtenção dos instantes de início/conclusão através do monitoramento de cada tarefa da aplicação, permite uma análise entre os tempos de execução real nos recursos e os tempos obtidos através do processo de escalonamento, antes da submissão. Isso permite, uma análise entre o *speedup*/*makespan* da aplicação fornecido pela ferramenta e o *speedup*/*makespan* da aplicação obtido do ambiente real.

## 5.4 Conclusão do Capítulo

A construção da ferramenta SLOT permitiu que todas as técnicas apresentadas no capítulo 4 fossem testadas e validadas no ambiente real, visto que foi possível disponibilizar recursos dinamicamente e utilizar aplicações reais. Além disto, a implementação separada das bibliotecas possibilita que as ferramentas NWS, MDS e DUROC sejam utilizadas a partir de qualquer programa sem a necessidade de utilizar o algoritmo de escalonamento proposto neste trabalho.

Além disto, com a utilização da ferramenta SLOT, o usuário da Grade preocupa-se apenas com o desenvolvimento da aplicação, sendo todo o processo de busca por otimização e gerenciamento das tarefas realizado pela ferramenta.

CAPÍTULO

6

# Construção do Simulador

Conforme descrito nos capítulos 3 e 4, diversos algoritmos de escalonamento em Grade foram propostos com o objetivo de melhorar algumas métricas de análises de desempenho como, por exemplo, tempo de execução, *throughput*, utilização dos recursos e *speed up*. Porém, muitos problemas em escalonamento são complexos (*NP-hard*) e os algoritmos não são capazes de encontrar uma solução ótima em tempo polinomial. Sendo assim, geralmente, não é possível avaliar e quantificar com eficácia as heurísticas adotadas pelos algoritmos (Legrand et al., 2003).

Para solucionar este problema, os algoritmos de escalonamento desenvolvidos para a Grade são avaliados através da execução em diversos cenários, ou seja, os algoritmos são testados variando as classes de aplicações e a quantidade e as características dos recursos computacionais e dos canais de comunicação.

Todavia, a análise da eficiência dos algoritmos de escalonamento em ambiente real possui algumas dificuldades, como: (1) a execução de aplicações paralelas em ambiente real geralmente demanda muito tempo, o que inviabiliza a repetição dos experimentos em todos os cenários, (2) a execução em ambiente real necessita de

uma grande variedade de recursos para explorar a escalabilidade do sistema e, por fim, (3) as aplicações podem ter comportamentos diferentes como, por exemplo, alto custo computacional ou alto custo de comunicação, sendo necessário possuir uma grande quantidade de aplicações para testes.

Para solucionar estes problemas foram desenvolvidas ferramentas de simulação que permitem aos desenvolvedores de aplicações e de algoritmos de escalonamento validar os métodos utilizados sem se preocupar com as limitações impostas pelos ambientes e pelas aplicações reais. Dentre as principais ferramentas de escalonamento disponíveis, destacam-se NS2 (ns2, 2008), OMNeT++ (Varga, 2001) e MicroGrid (Song et al., 2000).

O principal problema da maioria das ferramentas de escalonamento existentes é que foram desenvolvidas para funcionar de forma genérica, simulando cenários gerais na área de Redes de Computadores. Com isso, necessitam de uma implementação de baixo nível, focando-se mais na troca de mensagem e transmissão dos pacotes entre os recursos do que no comportamento das aplicações (Legrand et al., 2003).

Porém, quando se pretende analisar o comportamento de algoritmos de escalonamento na Grade, estas ferramentas não satisfazem os requisitos necessários (Legrand et al., 2003). Então, para suprir estas necessidades foram desenvolvidas as ferramentas SimGrid (Legrand et al., 2003) e GridSim (Murshed et al., 2001).

Apesar das duas ferramentas serem muito semelhantes, neste projeto foi optado pelo uso da ferramenta SimGrid, que possui um módulo que manipula diretamente classes de aplicações representadas por DAG.

## 6.1 A ferramenta SimGrid

SimGrid é uma ferramenta que fornece funcionalidades que facilitam a simulação de aplicações paralelas e distribuídas em ambientes heterogêneos. A figura 6.1 apresenta a arquitetura da ferramenta e, como pode ser visto, divide-se basica-

mente em três camadas.

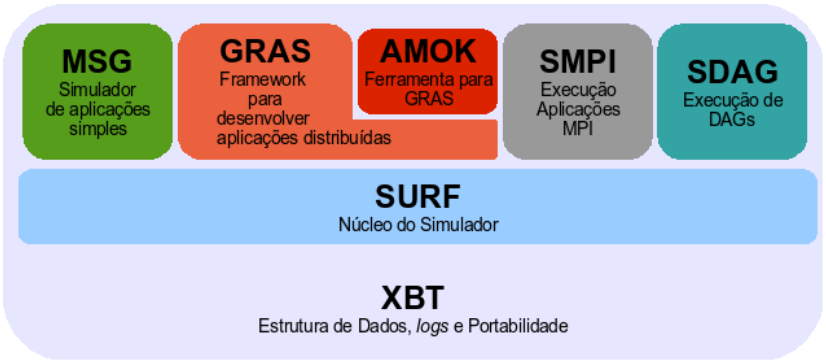


**Figura 6.1:** Arquitetura do simulador SimGrid.

A primeira camada da arquitetura é formada pela ferramenta XBT (*eXtended Bundle of Tools*), a qual fornece as principais estruturas de dados para o funcionamento do simulador. Além disso, nesta camada são implementados mecanismos de tratamento de exceção, de portabilidade e de geração de *logs*.

A segunda camada, SURF, é composta pelo *kernel*, ou núcleo, dos simuladores implementados no SimGrid e possui mecanismos para criar e manipular a plataforma virtual que representa o ambiente (processador, canais de comunicação e etc). Acima, na última camada, encontram-se módulos que implementam funções de simulação e que manipulam as aplicações através de diversos paradigmas.

Com base nestas informações sobre o SimGrid e visando a validação do algoritmo descrito na seção 4 e da ferramenta descrita na seção 5, foi desenvolvido um simulador usando o módulo SimDag (SDAG).

O módulo SimDAG gerencia cada tarefa individualmente, armazenando seus os instantes de início e de conclusão à medida que são executadas no simulador. Porém, para validar o algoritmo *slot* foi necessário alterar o código fonte deste módulo para que o instante de início de cada tarefa fosse determinado pelo escalonamento. Essencialmente, a alteração permitiu determinar, antes da execução do simulador, o início de execução de cada tarefa.

Com base nisto, o simulador implementado sobre o SimDAG recebe dois arqui-

vos de entrada: o primeiro é um arquivo *XML* que determina o ambiente onde serão realizadas as simulações e o segundo é um arquivo de texto contendo o DAG com o mapeamento de cada tarefa para os recursos e os instantes de início de cada tarefa no simulador.

Estes dois arquivos serão descritos em mais detalhes nas seções 6.3 e 6.2 a seguir.

## 6.2 Modelo Arquitetural Simulado

O modelo arquitetural utilizado na simulação é definido através de um arquivo XML, que contém informações sobre a capacidade de cada recurso que compõe o ambiente, ou seja, sobre o processador e a latência e largura de banda dos canais de comunicação.

A criação deste arquivo, contudo, não é simples e pode se tornar impraticável dependendo do número de máquinas que irá compor a Grade simulada e do número de canais de comunicação.

Logo, para facilitar este processo, foi desenvolvida uma ferramenta chamada *CreateEnv* que cria o modelo arquitetural automaticamente a partir de informações fornecidas pelo usuário.

Nesta ferramenta, o número $n$ de máquinas é definido pelo usuário e os canais de comunicação são criados de acordo com a equação $(n^2)$. As informações sobre a capacidade dos recursos e dos canais de comunicação são obtidas a partir de variáveis aleatórias denotadas por $F : |X \rightarrow [\Gamma_{min}, \Gamma_{max}]$, sendo que $\Gamma_{min}$ é o limite inferior e $\Gamma_{max}$ é o limite superior do intervalo definido pelo usuário. Assim, o poder de processamento de uma máquina é definido por intermédio da equação 6.1 e a latência e largura de banda pela equação 6.2, considerando $f(random)$ uma função que gera números aleatórios e $e = (2^{31}) - 1$.

$$\Omega = \Gamma_{min} + (f(random)\%(\Gamma_{max} - \Gamma_{min})) \tag{6.1}$$

$$\Omega = \Gamma_{min} + (\Gamma_{max} - \Gamma_{min}) * \left( \frac{f(random)}{e} \right) \quad (6.2)$$

Além da possibilidade de criar um ambiente simulado e aleatório com a utilização da ferramenta *createEnv*, o usuário pode usar o ambiente da Grade real no simulador. Para isso, as bibliotecas *libmds* e *libnws*, apresentadas na seção 5.1, geram um arquivo de entrada do simulador com base nas informações coletadas em tempo de execução.

Sendo assim, com a utilização da ferramenta *createEnv* ou das bibliotecas de monitoramento de recursos reais da Grade, *libnws* e *libmds* , torna-se possível criar diversos ambientes para testes no simulador.

## 6.3 Grafos Sintéticos

Conforme descrito anteriormente, a implementação do simulador para realização dos testes da ferramenta e dos algoritmos apresentados neste trabalho foi desenvolvida utilizando o módulo SIMDAG, o qual simula a execução de aplicações modeladas em forma de DAG. O processo de criação automática do DAG de uma aplicação real foi descrito na seção 5.2.

Visando a obtenção de melhores resultados nos testes de escalabilidade para validar o comportamento de um algoritmo ou de uma ferramenta, fez-se necessário criar um grande conjunto de DAGs com variados modelos de comunicação e números de tarefas. Porém, nem sempre é possível encontrar essas características em aplicações reais, por isso, uma das formas de resolver este problema é com a utilização de grafos simulados ou sintéticos.

Neste sentido, foi utilizada a ferramenta GTgraph (Bader;Madduri, 2006) que gera grafos sintéticos. No entanto, como a ferramenta SLOT precisava ser testada com lotes de aplicações, o código fonte da ferramenta foi alterado para que os grafos resultantes seguissem o padrão adotado pelos escalonadores implementados e pela ferramenta, e para determinar aleatoriamente valores que representam o

instante de chegada no sistema. Além disso, foi necessário fazer uma outra modificação na ferramenta para que os grafos gerados correspondessem com o modelo de aplicações tradicionais em MPI, onde a tarefa principal de uma aplicação inicia a execução e gerencia a submissão e a execução dos dados nas demais tarefas.

Desta forma, tornou-se possível a construção de conjuntos de testes com diversas variações de grafos de aplicações reais e sintéticas para o processo de validação.

## 6.4 Conclusão do Capítulo

Neste capítulo foram apresentados mecanismos que possibilitaram a construção de um ambiente simulado completo para aplicações paralelas desenvolvidas para a Grade, visando a validação ds mecanismos de escalonamento apresentados na seção 3 e da ferramenta apresentada na seção 4.

A figura 6.2 apresenta uma visão geral da ferramenta SLOT integrada com o simulador. Como pode ser visto nesta figura e de acordo com a descrição feita em seções anteriores, o modelo arquitetural que serve de entrada para o simulador e para o escalonador (setas 1 e 4) pode ser obtido de duas formas: ou através da ferramenta *createEnv*, ou através do monitoramento com as bibliotecas *libmds* e *libnws*. O modelo de aplicação, por sua vez, pode ser obtido tanto a partir do monitoramento de aplicações reais, quanto a partir da ferramenta GTgraph.

O resultado do escalonamento do modelo de aplicação (seta 3) pode ser executado em ambiente simulado ou em ambiente real, caso o modelo de aplicação seja de uma aplicação real.

Definido o processo de simulação, no próximo capítulo serão apresentados os testes implementados sobre este cenário, nos quais resultaram na validação da proposta.

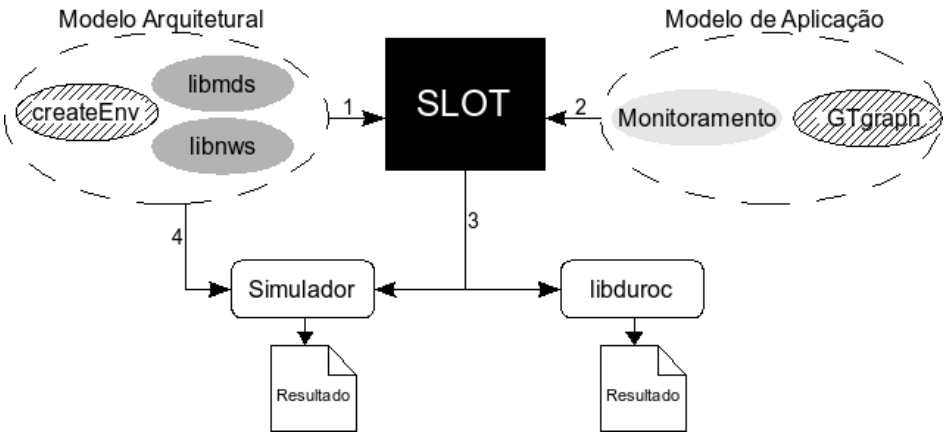


**Figura 6.2:** SLOT: Real e Simulado.

CAPÍTULO 7

# Avaliação do Trabalho

Neste capítulo são apresentados os experimentos e as análises realizados para validar o Algoritmo ***slot*** e o Algoritmo de **Sobreposição de *slots***, descritos no Capítulo 4. Além disto, são apresentados os resultados obtidos com a variação das políticas de priorização de tarefas, também apresentadas no Capítulo 4.

Para tanto, a metodologia usada na validação baseia-se nas técnicas de avaliação de desempenho apresentadas por Ferrari (1978) e divide-se em:

1. Elementar (ou direta): testes são realizados com aplicações reais e com *benchmarks*, e os resultados são obtidos através do monitoramento da execução em ambientes reais;

2. Indireta: testes são realizados em um nível de abstração maior com modelos simulados e com modelos analíticos.

Os testes realizados neste trabalho foram feitos usando o monitoramento de aplicações reais e de *benchmarks*, e o simulador apresentado no Capítulo 6. Com estas técnicas foram comparados os tempos de execução e o comportamento das

aplicações com o uso dos algoritmos de escalonamento CPOP e HEFT, e da ferramenta SLOT.

O ambiente real utilizado nos testes é composto por 16 máquinas com processadores Intel® Xeon® Dual Core com 2.40GHz, 2 GB RAM, canais de comunicação utilizando a tecnologia Fast-Ethernet e o Globus® Toolkit versão 4.0.3 instalado. Quanto ao ambiente simulado, foram construídas 2 arquiteturas contendo 50 e 100 máquinas com processadores variando aleatoriamente entre 24.5e+7 e 32.5e+7 Flops e a largura de banda dos canais de comunicação que envolvem estes recursos simulados foi estabelecida entre 70 e 90 Mbits/segundo e a latência dos canais de comunicação entre os recursos foi definida no intervalo de 0,002 a 0,015 segundos.

Os valores adotados na criação do modelo arquitetural simulado correspondem à média observada nos processadores e nas tecnologias *Fast/Gigabit Ethernet* atuais.

Quanto às aplicações utilizadas na validação, foram implementadas em MPI uma aplicação de cálculo do PI, uma aplicação de Multiplicação de Matrizes e uma aplicação de ordenação de vetores com o algoritmo MergeSort. Estas aplicações são comumente utilizadas em avaliação de algoritmos de escalonamento devido ao custo de computação e comunicação gerado durante a execução.

Além destas aplicações, foi utilizada a aplicação IS (*Interger Sort*) do *benchmak* NPB (*NAS Parallel Benchmark*). Este *benchmark* foi desenvolvido pelo programa NAS (*Numerical Aerodynamic Simulation*) da NASA e permite comparar o desempenho de sistemas computacionais altamente paralelos. O *benchmark* NAS divide-se em 3 classes (A, B e C) de acordo com o tamanho nominal dos vetores passados como parâmetro da aplicação. O tamanho nominal para a classe A é $2^{23}$x$2^{19}$, para a classe B é $2^{25}$x$2^{21}$ e para a classe C é $2^{27}$ (Bailey et al., 1994).

Para a realização dos testes com estas aplicações reais, uma base de dados foi montada com diversos DAGs, variando o número de processos e o tamanho do conjunto de dados utilizados na execução de cada uma destas aplicações.

Além dos modelos de aplicações reais, DAGs sintéticos foram gerados através da ferramenta GTgraph (Seção 6.3). Para os testes com aplicações sintéticas, uma outra base de dados foi formada por 100 aplicações com 10 tarefas cada, 10 aplicações com 100 tarefas cada e 10 aplicações com 200 tarefas cada, totalizando 4000 tarefas.

Supondo ser $N$ o número de tarefas de uma aplicação, o modelo de comunicação, ou seja, o número de arestas de cada DAG sintético foi definido baseado nas seguintes relações: $N+1$, $2N$ e $\frac{N(1+N)}{2}$. Estas relações foram determinadas através da análise de modelos de aplicações reais e foram constituídas para formalização dos testes.

O custo de execução de cada uma das tarefas das aplicações sintéticas foi definido de tal forma que a execução prévia de cada aplicação no simulador permitiu identificar que cerca de 80% das aplicações sintéticas possuem custo de execução em torno de 20 e 40 minutos e cerca de 20% das aplicações possuem custo de execução entre 4 e 8 horas.

Definidos a metodologia, os ambientes e as aplicações utilizados nos testes, nas próximas seções são apresentados os resultados e as análises obtidos do monitoramento nos ambientes reais e simulados. Para cada teste realizado no ambiente real a execução foi repetida cerca de 5 vezes, exceto nas seções em que o número de repetição estiver explicitado. Isto foi feito para que o estado momentâneo dos recursos e da rede interferissem menos no resultado final da análise.

## 7.1 Análise das Políticas de Priorização de Tarefas

Nesta seção são apresentadas análises desenvolvidas no ambiente real e no simulador, usando o algoritmo de escalonamento baseado em *slots* com diferentes políticas de priorização de tarefas.

São avaliadas três políticas apresentadas na seção 4.1. A primeira política ordena as tarefas de acordo com o *Rank Up*. A segunda ordena as tarefas de acordo

com o *Rank down* e a terceira combina as duas políticas anteriores, ou seja, as tarefas são ordenadas somando o *Rank Up* e o *Rank Down*.

No primeiro experimento, Figura 7.1, foram escolhidas aleatoriamente 15 DAGs da base de dados de aplicações reais. Cada um destes DAGs, representados pelo eixo $x$ do gráfico, foi escalonado com cada uma das políticas de priorização de tarefas e executados individualmente na Grade real.

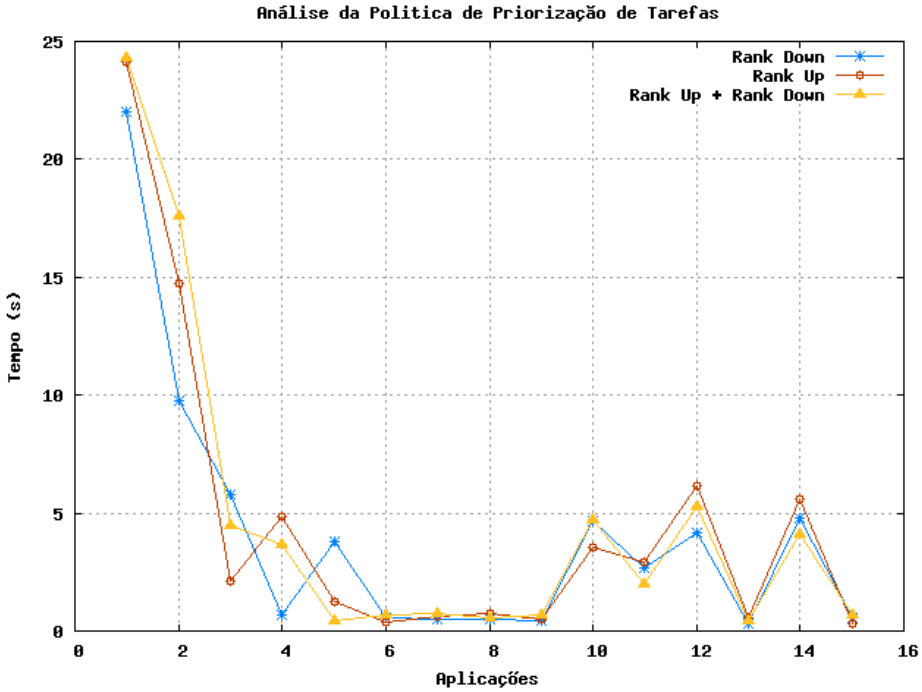


**Figura 7.1:** Testes com políticas de priorização de tarefas no ambiente real.

Já a Figura 7.2 apresenta o resultado do experimento realizado na Grade simulada, para o qual também foram escolhidas aleatoriamente 15 DAGs da base de dados de aplicações sintéticas.

Com o resultado dos experimentos apresentado na Figura 7.1 não foi possível determinar qual política de priorização de tarefas retornou melhor resultado, pois os desempenhos das aplicações foram muito semelhantes. No entanto, variando o ambiente e as aplicações no simulador, o experimento demonstrado na Figura 7.2 apresentou um ganho de desempenho na execução das aplicações quando utilizado

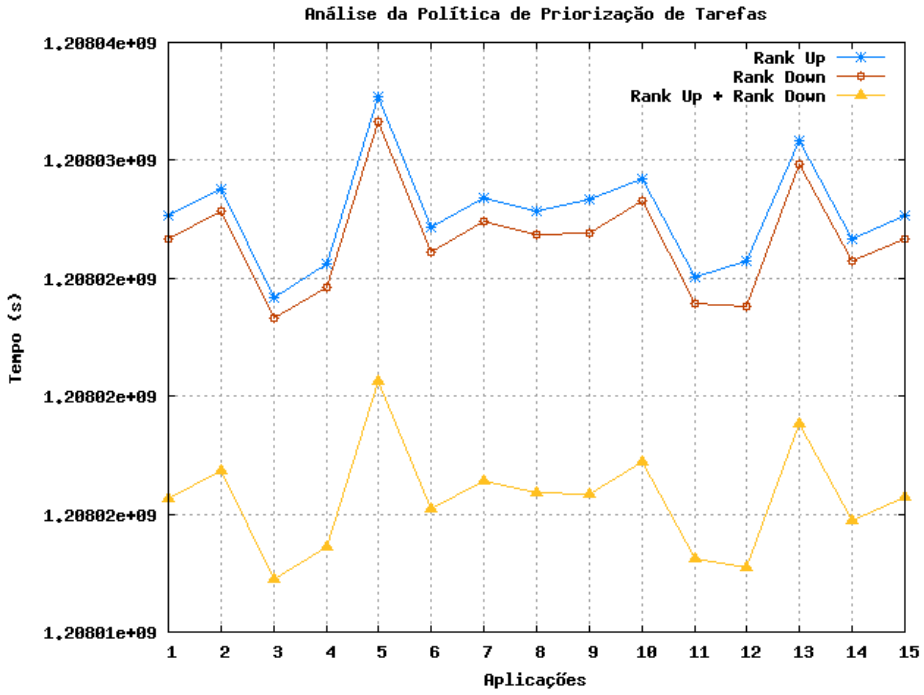


**Figura 7.2:** Testes com políticas de priorização de tarefas no ambiente simulado.

o *Rank Up* combinado com o *Rank Down* para ordenar as tarefas.

Os *Gráficos de Gantt*, Figuras 7.3 e 7.4, são apresentados para facilitar a visualização do efeito da alocação das tarefas nos recursos com a variação das políticas de priorização. Nestes gráficos, o eixo $x$ representa o tempo em segundos e o eixo $y$ representa os processadores. Os retângulos representam a ocupação em segundos de uma tarefa em um processador, sendo que tarefas de uma mesma aplicação são da mesma cor.

Nestes experimentos, quatro aplicações sintéticas foram submetidas para o escalonamento na Ferramenta SLOT na ordem apresentada nos gráficos, primeiro a Aplicação 1 seguida das Aplicações 2, 3 e 4. Foram escolhidas quatro aplicações apenas para facilitar a visualização do resultado do escalonamento e estas aplicações foram escolhidas aleatoriamente na base de dados de aplicações sintéticas. Além disto, estes gráficos foram gerados automaticamente pela Ferramenta SLOT, portanto, os tempos expressados nos gráficos não equivalem nem ao tempo de exe-

cução no ambiente real, nem no ambiente simulado, mas representam os tempos obtidos pelo escalonamento.

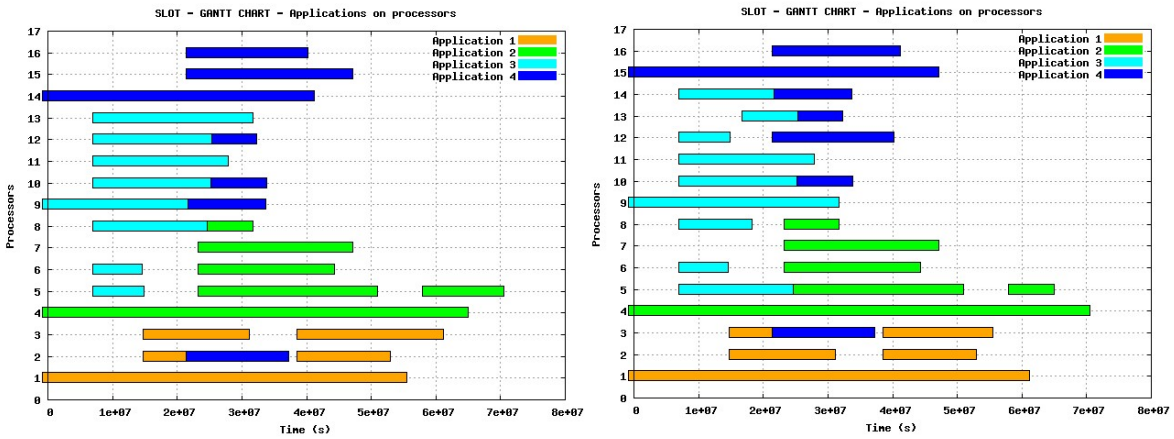


**Figura 7.3:** Esquerda, Gráfico de Gantt gerado pela Ferramenta SLOT usando a política de priorização de tarefas baseado no Rank Up. Direita, Gráfico de Gantt gerado pela Ferramenta SLOT usando a política de priorização de tarefas baseado no Rank Down

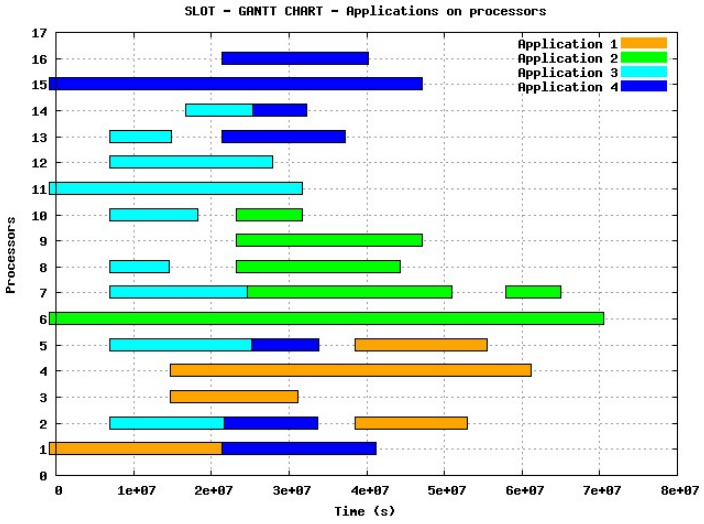


**Figura 7.4:** Gráfico de Gantt gerado pela Ferramenta SLOT usando a política de priorização de tarefas somando o Rank Up e o Rank Down.

Conforme pode ser visto nos *Gráficos de Gantt*, a política utilizada para definir a prioridade das tarefas influencia na alocação das mesmas nos processadores, alterando o tempo de execução das aplicações. Apesar da alteração na alocação das tarefas ficar clara no gráfico, a influência nos tempos de execução em relação à utilização das políticas não é visível devido à variação ser baixa nestas aplicações e o nível de visualização do gráfico não permitir tirar tais conclusões como as apresentadas na Figura 7.2.

Com base nos experimentos apresentados, a política de priorização de tarefas com o *Rank Up* e o *Rank Down* foi escolhida para ser utilizada no Algoritmo *slot*.

## 7.2 Análise do Algoritmo *slot*

A análise do Algoritmo *slot* foi realizada comparando os resultados obtidos com este algoritmo na execução das aplicações, sintéticas e reais, com os resultados obtidos com o uso dos algoritmos HEFT e CPOP. Os escalonamentos e as submissões das aplicações foram realizados com a Ferramenta SLOT. O ambiente utilizado na execução das aplicações reais e sintéticas foi a Grade real e a Grade simulada, respectivamente.

Para obtenção dos resultados apresentados na Figura 7.5, foram submetidas todas as aplicações sintéticas (120 aplicações com cerca de 4000 tarefas) para os ambientes de 50 e 100 máquinas. A taxa de chegada das aplicações foi definida através de uma distribuição de Poisson, isto é, definida exponencialmente em uma seqüência de valores aleatórios Independentes e Identicamente Distribuídos (IDD) (Kleinrock, 1976).

Como pode ser visto na Figura 7.5, o algoritmo *slot* obteve um ganho de desempenho na execução das aplicações quando comparado com os algoritmos de escalonamento tradicionais. Outro aspecto interessante observado durante os testes com aplicações sintéticas foi que o algoritmo HEFT apresentou um desempenho melhor que o algoritmo CPOP quando as aplicações são escalonadas individualmente

sem sofrer interferência das demais, porém apresentou uma perda de desempenho quando submetido a testes com diversas aplicações.

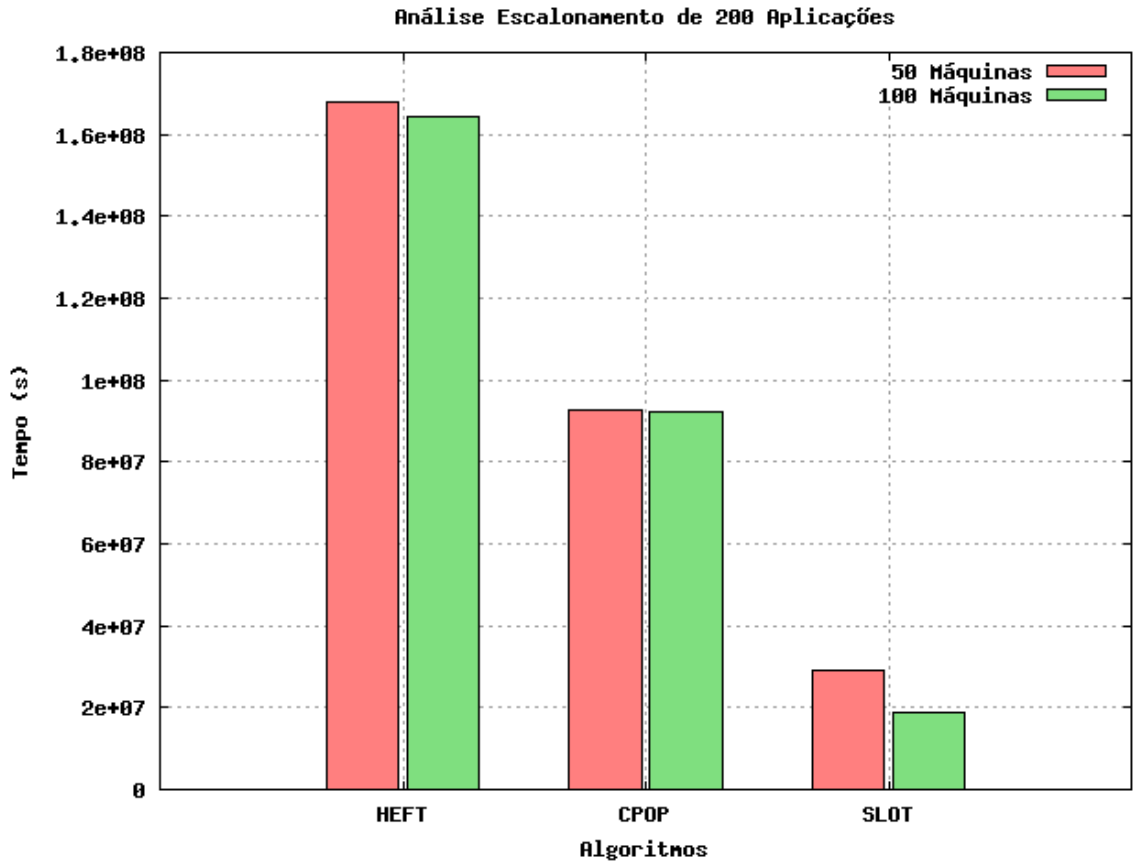


**Figura 7.5:** Análise do *makespan* de todas as aplicações sintéticas com taxa de chegada segundo uma distribuição de Poisson e em ambientes com 50 e 100 máquinas.

Um segundo teste foi realizado nestes ambientes de 50 e 100 máquinas, sem as aplicações com maior custo computacional. Foram gerados dois subconjuntos de testes: o primeiro, composto por 106 aplicações, que representam todas as aplicações com custo de execução em minutos, e o segundo, composto por uma seleção aleatória de 50 aplicações dentre as aplicações que compunham o primeiro subconjunto.

Estes subconjuntos foram criados para testar o comportamento dos algoritmos no escalonamento de aplicações com tarefas de baixo custo de execução, evitando

que a sobrecarga de tarefas nos recursos afetassem o *makespan* final. Os resultados obtidos a partir destes testes podem ser visualizados na Figura 7.6.

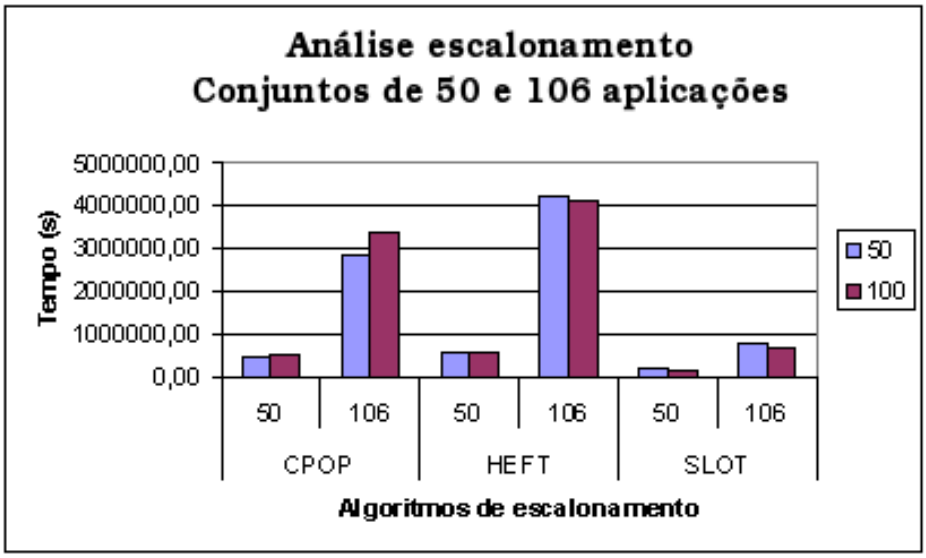


**Figura 7.6:** Análise do *makespan* das tarefas submetidas em grupos de 50 e 106 aplicações em ambientes com 50 e 100 máquinas.

Considerando que SLOT é um escalonador que obtém informações das aplicações e dos recursos dinamicamente, a ordem de chegada e, conseqüentemente, a priorização das tarefas de cada aplicação é relevante.

Nos testes anteriores, os instantes de chegada de cada aplicação foram definidos de acordo com uma distribuição exponencial. Estes instantes poderiam influenciar a execução de algum algoritmo apresentado. Por isso, foram realizados novos testes, onde as aplicações sintéticas foram submetidas para os ambientes com 50 e 100 máquinas, variando aleatoriamente os instantes de chegada no sistema. Em cada ambiente, as aplicações foram executadas 20 vezes. Os resultados destes testes podem ser vistos nas Figuras 7.7 e 7.8.

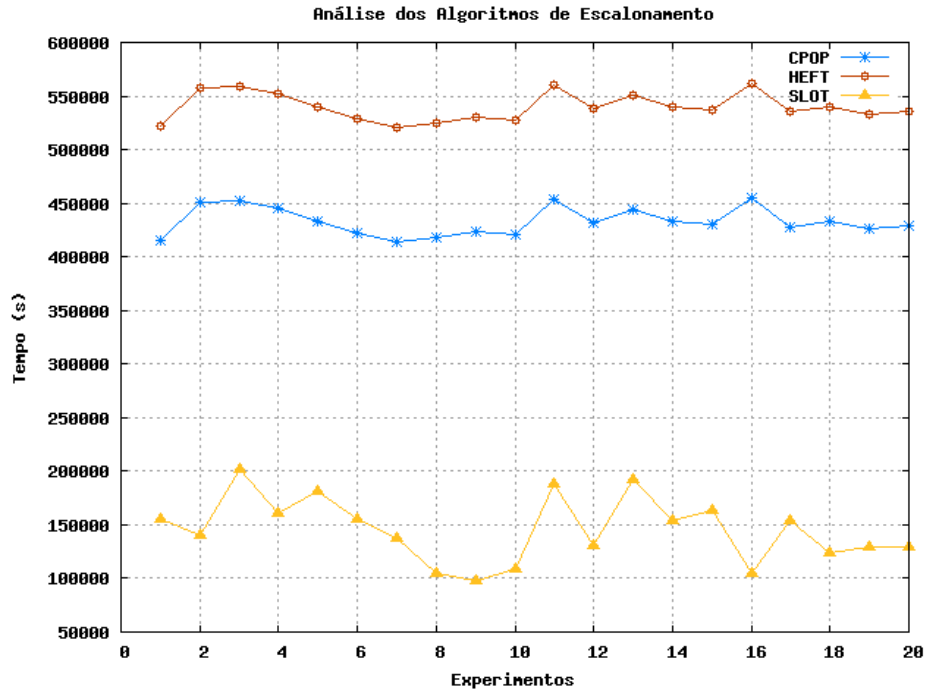


**Figura 7.7:** Análise da execução das aplicações sintéticas no ambiente com 50 máquinas, variando o instante de chegada das aplicações.

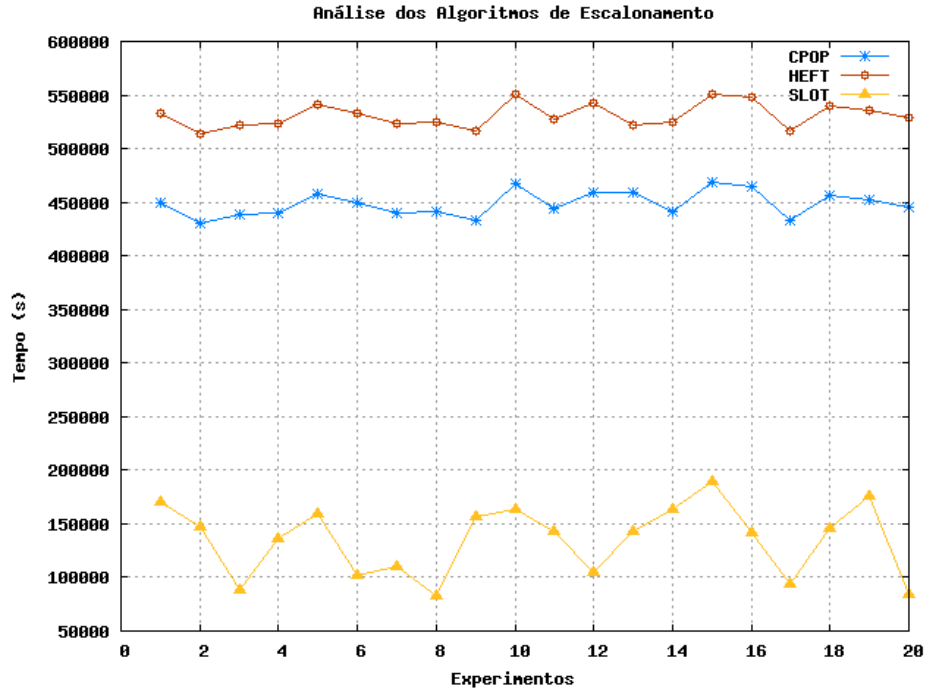


**Figura 7.8:** Análise da execução das aplicações sintéticas no ambiente com 100 máquinas, variando o instante de chegada das aplicações.

Os Gráficos de Barras, Figuras 7.9 e 7.10, apresentam a relação entre as médias e os desvios padrão dos tempos de execução entre os algoritmos nos ambientes com 50 e 100 máquinas, respectivamente.

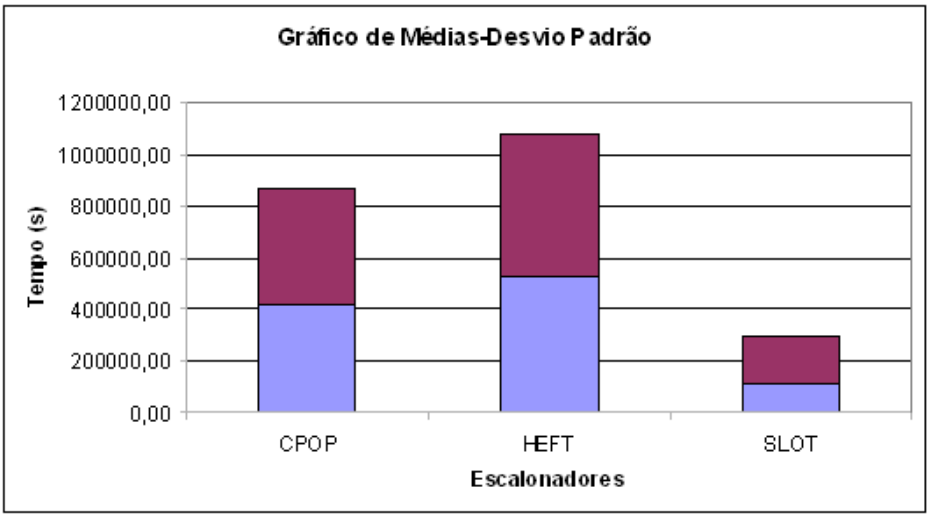


**Figura 7.9:** Análise da execução das aplicações sintéticas no ambiente com 50 máquinas, variando o instante de chegada das aplicações.

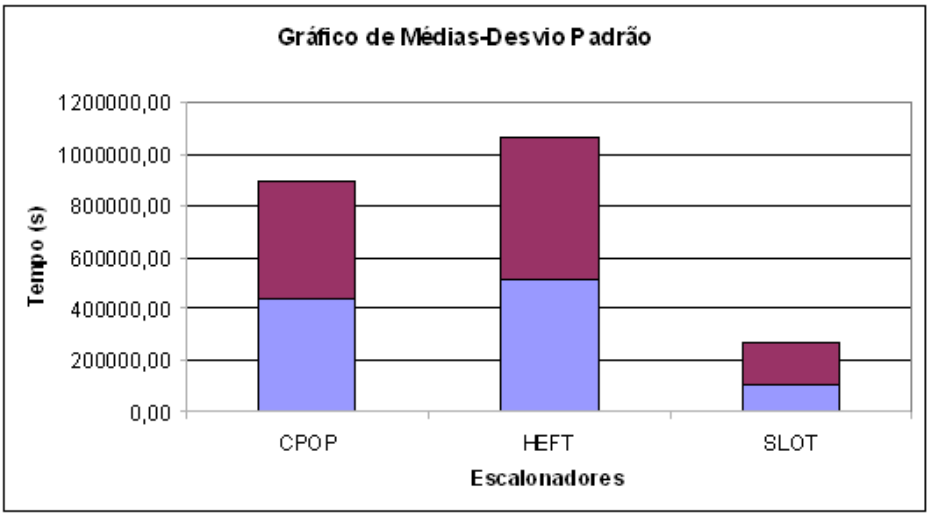


**Figura 7.10:** Análise da execução das aplicações sintéticas no ambiente com 100 máquinas, variando o instante de chegada das aplicações.

Por fim, foram realizados testes para validar a ferramenta no ambiente de Grade real. Para isto, foi utilizada a biblioteca desenvolvida baseada na Ferramenta DUROC que submete as tarefas para os recursos e obtém os instantes de iní-

cio/conclusão de cada tarefa submetida à Grade.

No primeiro experimento, Figura 7.11, foram submetidos para a ferramenta SLOT e para os algoritmos HEFT e CPOP modelos de aplicações reais com baixo custo de computação.

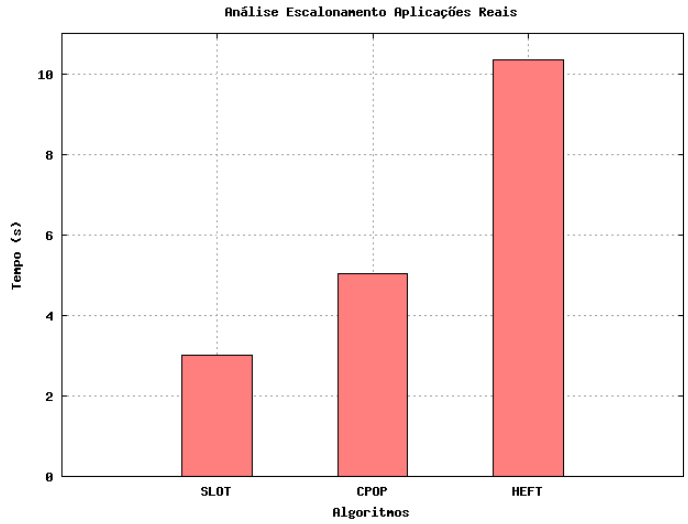


**Figura 7.11:** Resultado do *makespan* das tarefas quando submetidas para a Grade Real com os algoritmos HEFT e CPOP, e a ferramenta SLOT.

Em outro experimento realizado, apresentado na Figura 7.12, modelos de aplicações com maior custo computacional foram submetidos aos mesmos algoritmos de escalonamento do experimento anterior.

Os experimentos realizados com modelos de aplicações com custo variado serviram para demonstrar o comportamento da ferramenta SLOT, a qual executou o conjunto de aplicações com maior ganho de desempenho em relação aos demais algoritmos avaliados.

Em um outro teste, foi feita a submissão de instâncias da aplicação MergeSort, com variações no número de processos utilizados e no conjunto de dados de entrada. Esse conjunto de aplicações foi escolhido por ser o que possuía DAGs com menores custos computacionais. Com o monitoramento da execução desses diversos modelos, observou-se um ganho médio de desempenho de 15.79% no tempo total de execução quando executada com a ferramenta SLOT.

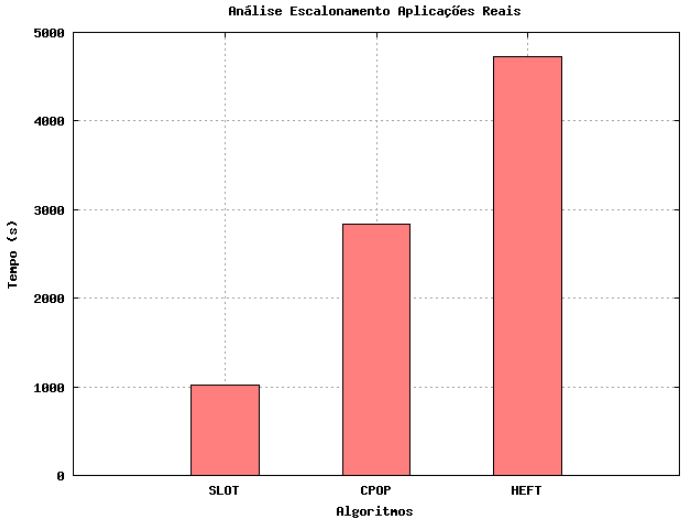


**Figura 7.12:** Resultado do *makespan* das tarefas quando submetidas para a Grade Real com os algoritmos HEFT e CPOP, e a ferramenta SLOT.

Podemos concluir que o ganho de desempenho na execução das aplicações ocorre devido à melhor ocupação das tarefas nos processadores. Para exemplificar esta afirmação, o Gráfico de Gantt, Figura 7.13, gerado pela Ferramenta SLOT apresenta o comportamento partir da submissão sucessiva de 3 aplicações para o ambiente real.

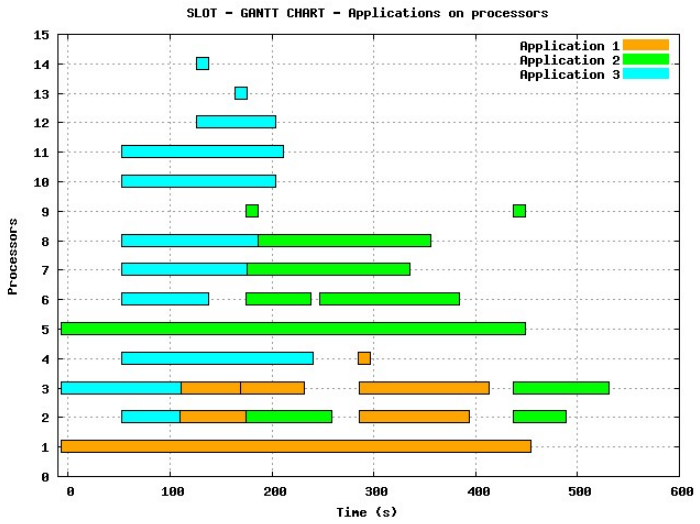


**Figura 7.13:** Gráfico de Gantt gerado pela Ferramenta SLOT que representa a ocupação das aplicações reais nos processadores.

Como pode ser visto no Gráfico de Gantt, o principal objetivo da execução da Ferramenta SLOT com o Algoritmo de escalonamento *slot* é reduzir as fatias de tempo inutilizadas nos recursos. Como resultado, a alocação de uma tarefa a um recurso não deve gerar impacto no desempenho esperado para as tarefas já atribuídas. Nos algoritmos tradicionais, no caso o HEFT e o CPOP utilizados neste capítulo para validação, os intervalos inutilizados nos processadores entre as tarefas não são considerados.

Tomando como exemplo a Figura 7.13, o Algoritmo *slot* submeteu uma tarefa da Aplicação 2 (verde) para execução no Processador 2 entre as submissões de duas tarefas da Aplicação 1 (laranja), que havia sido escalonada previamente, no mesmo processador. Na execução dos Algoritmos HEFT e CPOP, no escalonamento de uma tarefa ou de um grupo de tarefas é analisada a taxa média de ocupação em cada processador e cada submissão é realizada de forma seqüencial e independente, ou seja, estes algoritmos não fariam o escalonamento de uma tarefa considerando os *slots* livres.

Além do exemplo citado, outros casos de utilização de *slot* ociosos entre tarefas puderam ser observados na figura como, por exemplo, tarefas da Aplicação 3 que são escalonadas antes de tarefas da Aplicação 2.

Conforme os resultados obtidos com os experimentos realizados no simulador e com a execução efetiva na Grade real, observou-se que a ferramenta SLOT com o algoritmo de escalonamento baseado em *slots* apresentou um ganho de desempenho significativo quando comparado com outros algoritmos tradicionais.

Na próxima seção serão apresentados os experimentos realizados com a variação na política de inserção do algoritmo *slot* para validar o uso da sobreposição de *slots*.

## 7.3 Análise do Algoritmo de Sobreposição de *slots*

Nesta seção é apresentado um estudo sobre a variação da política de inserção do Algoritmo *slot* que foi apresentada na Seção 4.3.

Para realização dos experimentos, o Algoritmo *slot* foi executado no ambiente real e no ambiente simulado. Os modelos de aplicações sintéticos e reais utilizados nos testes foram selecionados a partir das bases de dados elaboradas para avaliação da proposta.

As Tabelas 7.1 e 7.2 apresentam os valores médios dos resultados obtidos com 15 experimentos realizados com a variação da sobreposição dos *slots* entre 0% e 100%.

Na Tabela 7.1 estão os resultados obtidos com a utilização do ambiente simulado. A partir destes resultados observou-se que houve um ganho de desempenho da sobreposição com relação à política de inserção com *slots* proposta neste trabalho. Porém, à medida que a sobreposição é aumentada, o tempo de execução tende a diminuir devido ao aumento do tempo em que duas tarefas compartilham o mesmo processador.

**Tabela 7.1:** Análise da sobreposição de *slots* no simulador.

| Sobreposição (%) | Tempo (s) | Ganho com Sobreposição (%) |
|---|---|---|
| 0 | 432292,459156 | - |
| 10 | 410210,339887 | 4,33 |
| 20 | 411286,960289 | 4,20 |
| 30 | 410756,848203 | 4,51 |
| 40 | 411397,940916 | 4,56 |
| 50 | 412796,873494 | 4,57 |
| 60 | 412026,087317 | 4,69 |
| 70 | 412579,881337 | 4,86 |
| 80 | 413576,281248 | 4,98 |
| 90 | 414120,998263 | 4,83 |
| 100 | 412543,972750 | 5,11 |

Na Tabela 7.2 são apresentados os resultados obtidos a partir da execução no

ambiente real. Ao contrário dos resultados obtidos pelo simulador houve um ganho de desempenho para as aplicações testadas apenas quando a sobreposição foi de 20%. Este comportamento se repetiu durante todos os experimentos realizados. Devido aos atrasos ocorridos na ativação das aplicações nos recursos com o globais, os testes no ambiente real não foram conclusivos sobre a influência da sobreposição dos *slots*. Uma vez que os recursos utilizados nos testes possuíam 2 processadores HT, o efeito da sobreposição não pode também ser avaliado precisamente. Enquanto no simulador, não considerou-se múltiplos *cores* por recursos. Novos estudos de alocação precisam ser realizados para esses casos.

**Tabela 7.2:** Análise da sobreposição de *slots* em ambiente real.

| Sobreposição (%) | Tempo (s) | Ganho com Sobreposição (%) |
|---|---|---|
| 0 | 3,600240 | - |
| 10 | 4,269324 | -18,58 |
| 20 | 3,525132 | 2,09 |
| 30 | 3,927504 | -9,09 |
| 40 | 4,228224 | -17,44 |
| 50 | 4,469003 | -24,13 |
| 60 | 4,558888 | -26,63 |
| 70 | 4,944088 | -37,33 |
| 80 | 4,950125 | -37,49 |
| 90 | 5,685717 | -57,93 |
| 100 | 7,110036 | -97,49 |

Portanto, pode-se concluir que a utilização dos processadores, para os testes realizados, apresentou melhores resultados quando comparado o escalonamento usando sobreposição de *slots* e o escalonamento baseado em *slots*.

## 7.4 Conclusão do Capítulo

Neste capítulo foram apresentados os experimentos realizados com a Ferramenta SLOT (Capítulo 5) para validar o Algoritmo de escalonamento baseado em *slots* e na sobreposição de *slots* (Capítulo 4). Para este fim, foram executadas aplicações no ambiente real de Grade e no ambiente simulado (Capítulo 6).

Apesar do ganho de desempenho na execução final das aplicações, alguns fatores limitam utilização da Ferramenta SLOT. Um destes fatores é o custo para executar o escalonamento de uma aplicação. No caso da Ferramenta SLOT estar sendo executada em um ambiente com poucas máquinas, o tempo de escalonamento é relativamente baixo diante do tempo de execução da aplicação. Porém, em situações em que existe um grande número de máquinas, a busca por *slots* desocupados capazes de executar uma dada tarefa certamente aumentará.

Uma maneira de verificar isso é analisando o tempo de execução do Algoritmo 1 (Seção 4.2). Esta análise não avalia o tempo de execução de forma exata, apenas tenta determinar o tempo de execução do algoritmo conforme os dados são variados.

Cada instrução única (por exemplo, as linhas 9 e 10) foi considerada como sendo uma unidade de execução, cujo valor é igual a 1. A parte do algoritmo que desejamos avaliar encontra-se entre as linhas 4 e 17 e supondo que o número de tarefas é igual a $n$, o número de recursos é igual a $m$ e o número de *slots* em cada recurso é igual a $q$, podemos afirmar que:

- A busca por *slots* em um processador, linha 11, possui custo de execução no pior caso igual a $(q + 1)$;
- Como os *slots* são avaliados em cada processador ($m$) e considerando o custo de execução de cada unidade de execução (linhas 8, 9, 10, 12 e 13) podemos afirmar que o laço mais interno (*while*, linha 8) possui custo de execução igual a $m(5 + (q + 1))$ ou $m(q + 6)$;
- Por fim, analisando o laço *while* na linha 4, que realiza os cálculos anteriores de acordo com o número de tarefas ($n$) e com as unidade de execução das linhas 4, 5 e 6, podemos afirmar que o custo de execução deste laço é igual a $n(m(q + 6) + 3)$

Com base nisto podemos afirmar que a complexidade do tempo de execução

deste algoritmo é $f(n) = \mathcal{O}(qmn)$, ou seja, o tempo de execução é diretamente proporcional às variáveis $q$, $m$ e/ou $n$.

Um outro questionamento que pode ser feito com relação à abordagem centralizada usada na Ferramenta SLOT. Como já foi citado anteriormente, os principais problemas da abordagem centralizada é a possibilidade do escalonador tornar-se um *gargalo* quando houver muitas requisições. Além disto, no caso de falha do escalonador as submissões de aplicações para a Grade ficaria comprometida.

Uma forma de atenuar os problemas existentes no escalonamento centralizado e reduzir o tempo de execução do algoritmo de escalonamento é implementar uma abordagem hierárquica usando a Ferramenta SLOT, como pode ser visto na Figura 7.14.

Nesta abordagem, existiriam um ou mais *Escalonadores Principais* e a Ferramenta SLOT ficaria responsável pelo gerenciamento das máquinas de uma Organização Virtual (*VO*). Com isto, o número de máquinas avaliadas por cada *Escalonador SLOT* seria reduzido e caso um dos *Escalonadores SLOT* falhasse afetaria apenas a execução em uma VO, sem comprometer as demais tarefas.

A submissão de uma aplicação para execução poderia ser realizada para um Escalonador Principal onde o escopo de execução seria toda a Grade ou diretamente para para o Escalonador SLOT, porém seria executada apenas na VO gerenciada pelo escalonador.

Uma outra abordagem que poderia ser utilizada visando a redução do tempo de execução do Algoritmo *slot* é combiná-lo com heurísticas de Algoritmos de Agrupamento. Neste caso, inicialmente o algoritmo separaria as tarefas em grupos e cada grupo seria enviado para máquinas que fazem parte de uma mesma organização virtual.

Com isto, reduz-se o número de máquinas pesquisadas, além de poder obter um ganho no tempo de execução das tarefas, reduzindo o custo de comunicação com a utilização de máquinas que fazem parte de uma mesma rede, na qual, geralmente, a latência entre essas máquinas é menor e a largura de banda é maior. Sendo

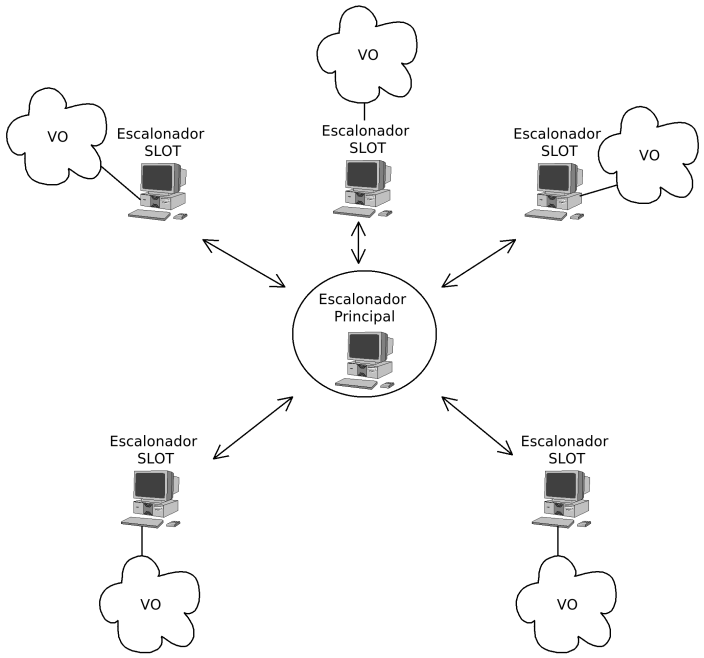


**Figura 7.14:** Utilização da Ferramenta de escalonamento SLOT em uma abordagem hierárquica.

assim, o Algoritmo *slot* poderia ser utilizado para alocar tarefas de um determinado grupo em máquinas de uma mesma VO.

CAPÍTULO

8

# Conclusões

Os algoritmos de escalonamento desenvolvidos para o ambiente de Grade computacional visam a geração de um mapeamento eficiente das tarefas nos recursos. Para tanto, características dinâmicas dos recursos e aspectos específicos das aplicações são levados em consideração.

Sendo assim, foi apresentado neste trabalho um algoritmo de escalonamento e uma variação deste que permitiram a utilização mais eficiente por parte das tarefas em cada recurso compartilhado na Grade.

Com a utilização destes algoritmos propostos, observou-se que o conhecimento global das aplicações submetidas para execução pode trazer ganhos de desempenho, pois permite uma visão da carga não só presente, mas também futura dos recursos. O conhecimento das aplicações já escalonadas possibilita a atribuição de cargas aos *slots* de tempo ociosos conhecidos para os processadores em que tarefas aguardam a conclusão de suas dependências.

Para ambientes reais, em que as taxas de chegadas das aplicações variam dinamicamente, o uso destes algoritmos é ainda mais relevante. Diferentemente dos

escalonadores independentes, o escalonamento baseado em *slots* propicia ganhos de desempenho (menor *makespan*) para o conjunto de tarefas e para cada aplicação, alocada de acordo com a real disponibilidade dos recursos.

De maneira geral, a idéia é fazer o escalonamento de novas aplicações sem comprometer o desempenho daquelas já submetidas para execução. Trata-se de ampliar a funcionalidade do escalonador de aplicações, fazendo com que ele interaja com os escalonadores dos recursos aos quais as tarefas foram atribuídas. Esta interação foi obtida com a criação da Ferramenta SLOT, apresentada neste trabalho, que utiliza mecanismos de gerenciamento de recursos como dados de entrada para o algoritmo de escalonamento *slot*, também apresentado neste trabalho.

Levando-se em consideração estes aspectos apresentados e os resultados experimentais descritos no capítulo anterior, verificou-se que o uso de uma ferramenta global que gerencia os recursos e as aplicações com os algoritmos apresentados permite explorar o capacidade máxima de processamento dos recursos e obter um ganho de desempenho na execução de aplicações submetidas para a Grade. Além disto, a ferramenta SLOT retira a responsabilidade do usuário submeter e gerenciar as aplicações nos recursos.

Os resultados apresentados neste trabalho serviram como motivação para sugestão de trabalhos futuros. Dentre estas sugestões, podemos destacar um mecanismo que utiliza os valores obtidos no monitoramento das tarefas durante a execução no ambiente real para retroalimentar o escalonador e as informações sobre as tarefas pendentes e obter resultados mais precisos no escalonamento.

Uma função que pode ser utilizada na retroalimentação é o método dos mínimos quadrados, apresentada na Equação 8.1.

$$S = \sum_{i=1}^{n} (y_i - f(x_i))^2 \tag{8.1}$$

Esta função busca ajustar o erro encontrado entre os tempos previstos no escalonamento ($y_i$) e os tempos obtidos com o monitoramento no ambiente real ($f(x_i)$).

Um outro trabalho relacionado sugerido é a realização do escalonamento das tarefas considerando o número de processadores, ou núcleos (*cores*), existentes em cada máquina. Grande parte dos algoritmos atuais tratam cada máquina como uma única unidade de processamento. Neste sentido, o gerenciador de recursos da ferramenta SLOT poderia definir o modelo arquitetural como sendo um grafo, em que cada processador seria representado como sendo um nodo e as arestas seriam o meio de comunicação entre os processadores, por exemplo, a rede, no caso de máquinas distintas, ou o barramento de comunicação, no caso de múltiplos *cores* em uma mesma máquina. Um ponto de partida para este estudo é o trabalho apresentado por (Casanova;Dongarra, 1995) com a Fórmula 3.1. Conforme descrito na Seção 3.4.7, esta fórmula calcula o desempenho estimado de uma carga de trabalho em um recurso, considerando o número de processadores existentes.

Conforme citado anteriormente, um dos grandes problemas na validação dos algoritmos desenvolvidos para a Grade é a quantidade de recursos. Uma alternativa utilizada neste trabalho foi a implementação de um simulador. Porém, uma outra alternativa seria a utilização de máquinas virtuais instaladas nos recursos. Para isto, um estudo deveria ser realizado para avaliar o comportamento dos processadores e o impacto da submissão de duas tarefas para um recursos, uma para a máquina real e outra para a máquina virtual.

APÊNDICE

A

# Implementação e Uso Efetivo do SLOT com o Globus

Neste capítulo são apresentadas algumas características da Ferramenta SLOT e das demais ferramentas utilizadas no desenvolvimento do trabalho. A implementação de todas as ferramentas foi realizada utilizando a Linguagem de Programação C.

## A.1 A ferramenta SLOT

A ferramenta SLOT foi desenvolvida utilizando as APIs disponibilizadas pelo Globus® e as bibliotecas desenvolvidas neste trabalho: *libMDS* para gerenciamento dos recursos, *libMDS* para gerenciamento dos canais de comunicação e *libDUROC* para submissão e monitoramento de aplicações. A figura A.1 apresenta o processo de compilação da ferramenta SLOT.

A ferramenta SLOT é inicializada de acordo com o comando abaixo:

```
[ricardo@xeon scheduler-central]$ make
gcc -Wall  -c grafo.c
gcc -Wall -c link_list.c
gcc -Wall -c fileio.c
gcc -Wall -O -Wall -I/opt/globus/include/gcc32 -c pon.c -L/opt/nws/lib/ -I/opt/nws/include/ \
-L./libNWS -I./libNWS -L./libMDS -I./libMDS
gcc -Wall -c priority_queue.c
gcc -Wall -c scheduler_socket.c
/usr/bin/gcc -O -Wall -I/opt/globus/include/gcc32 -c main.c -L./libMDS -I./libMDS \
-L/opt/nws/lib/ -I/opt/nws/include/ -L./libNWS -I./libNWS -L./libDUROC -I./libDUROC
/usr/bin/gcc -o schedulerd main.o scheduler_socket.o fileio.o grafo.o link_list.o priority_queue.o pon.o \
-L/opt/globus/lib -L/opt/globus/lib \
-lglobus_duroc_bootstrap_gcc32 -lglobus_duroc_runtime_gcc32 -lglobus_duroc_control_gcc32 -lglobus_gram_myjob_gcc32 -lglobus_d
uroc_common_gcc32 -lglobus_duct_runtime_gcc32 -lglobus_gram_client_gcc32 -lglobus_duct_common_gcc32 -lglobus_nexus_gcc32 -lgl
obus_gram_protocol_gcc32 -lglobus_io_gcc32 -lglobus_xio_gcc32 -lgssapi_error_gcc32 -lglobus_gss_assist_gcc32 -lglobus_gssapi_
gsi_gcc32 -lglobus_gsi_proxy_core_gcc32 -lglobus_gsi_credential_gcc32 -lglobus_gsi_callback_gcc32 -lglobus_oldgaa_gcc32 -lglo
bus_gsi_sysconfig_gcc32 -lglobus_gsi_cert_utils_gcc32 -lglobus_openssl_gcc32 -lglobus_rsl_gcc32 -lglobus_mp_gcc32 -lglobus_op
enssl_error_gcc32 -lglobus_callout_gcc32 -lglobus_proxy_ssl_gcc32 -lglobus_common_gcc32 -lssl_gcc32 -lcrypto_gcc32 -lltdl_gcc
32 -lglobus_dc_gcc32 -lm -ldl -ldl \
 -lldap -lmds -L./libMDS -I./libMDS -L./libDUROC -I./libDUROC \
        -lcollector -lnws -lduroc -L/opt/nws/lib/ -I/opt/nws/include/ -L./libNWS -I./libNWS -lm `xml2-config --libs`  -lpthre
ad
```

**Figura A.1:** Integração com o Globus e NWS na compilação.

/home/ricardo/mestrado/escalonadores/schedulerd/scheduler-central/schedulerd 3000

A ferramenta SLOT pode criar um modelo arquitetural dos recursos disponíveis usando um arquivo XML definido pelo usuário (Figura A.2) ou usando as informações obtidas pelas bibliotecas de descoberta e monitoramento dos recursos (libMDS e libNWS) como pode ser visto na Figura A.3. Logo que SLOT começa a executar, uma *thread* é inicializada e fica responsável por criar o ambiente arquitetural e atualizar as informações dos recursos.

```
Apr 25 12:30:14 xeon schedulerd[10724]: [SCHEDULER-CENTRAL] Criado modelo arquitetural usando o arquivo /tmp/arch.xml
Apr 25 12:30:14 xeon schedulerd[10724]: [SCHEDULER-CENTRAL] ***DUROC ATIVADO!!!!!!!!!
Apr 25 12:30:14 xeon schedulerd[10724]: [SCHEDULER-CENTRAL] Inicializado DUROC control
```

**Figura A.2:** Inicialização da Ferramenta SLOT usando um arquivo XML da arquitetura.

```
Apr 25 14:47:13 xeon schedulerd[12375]: [SCHEDULER-CENTRAL] ***DUROC ATIVADO!!!!!!!!!
Apr 25 14:47:13 xeon schedulerd[12375]: [SCHEDULER-CENTRAL] Inicializado DUROC control
Apr 25 14:47:18 xeon schedulerd[12375]: [SCHEDULER-CENTRAL] Modelo arquitetural criado(NWS/MDS). Nova consulta em 3 min
```

**Figura A.3:** Inicialização da Ferramenta SLOT, utilizando o NWS e o MDS.

Conforme foi explicado anteriormente, a Ferramenta SLOT funciona como um processo no Linux, ou seja, não é associada a um *shell* e permanece em *background*. A submissão de uma aplicação é realizada através de um programa cliente que abre uma conexão via *socket* e envia as informações necessárias para a execu-

ção na Grade.

A sintaxe do programa cliente pode ser vista na Figura A.4.

```
[ricardo@xeon client-scheduler]$ ./client-sched -h
Usage: ./client-sched options [inputfile]
        -h      --help                                  Apresenta esta informação de uso.
        -p      --port          port                    Porta para acessar o servidor
        -n      --hostname      host                    hostname do servidor.
        -d      --dag           "/DIRECTORY/dag_file"   DAG.
        -f      --folder        "/DIRECTORY/path_files" Diretório dos arquivos.
        -e      --exe           "/DIRECTORY/exec_file"  Arquivo executável.
        -l      --lib           "/DIRECTORY/lib_path"   Diretório das bibliotecas necessárias para execução.
        -a      --arg           "argument"              Lista de argumentos passados para o executável.
        -v      --verbose                               Imprime as etapas da execução.
```

**Figura A.4:** Ajuda do comando *client-sched* ao ser executado com a opção *-h*.

A Figura A.5 apresenta um exemplo de submissão de uma aplicação usando o programa cliente para a Ferramenta SLOT.

```
[ricardo@xeon create_env_sim]$ ./client-sched -p 3000 -n xeon -d /home/ricardo/acmsac/appmodel/MergeSort/mpi-mergesort-16-327
68.txt.dot.sched -f /home/ricardo/programas-GRID/mpi/mergesort/less/v0.1.4/ -e mpi-mergesort2 -l "/opt/globus/lib" -a 32768
```

**Figura A.5:** Execução de uma aplicação no SLOT a partir do programa cliente.

Assim que a ferramenta SLOT recebe uma requisição para execução de uma aplicação, *logs* são gerados à medida que cada etapa do algoritmo *slot* é executado.

```
Apr 25 14:51:11 xeon schedulerd[12375]: **/home/ricardo/acmsac/appmodel/MergeSort/mpi-mergesort-16-32768.txt.dot.sched
Apr 25 14:51:11 xeon schedulerd[12375]: [SCHEDULER-CENTRAL] Modelo de aplicacao: /home/ricardo/acmsac/appmodel/MergeSort/mpi-
mergesort-16-32768.txt.dot.sched
Apr 25 14:51:11 xeon schedulerd[12375]: [SCHEDULER-CENTRAL] Numero de ciclos removidos utilizando o algoritmo HDR: 5
Apr 25 14:51:11 xeon schedulerd[12375]: [SCHEDULER-CENTRAL] Latencia media: 9.444312
Apr 25 14:51:11 xeon schedulerd[12375]: [SCHEDULER-CENTRAL] Largura de banda media: 53.917228
Apr 25 14:51:11 xeon schedulerd[12375]: [SCHEDULER-CENTRAL] Criado lista de predecessores e sucessores
Apr 25 14:51:11 xeon schedulerd[12375]: [SCHEDULER-CENTRAL] Criado rankup e rankdown das tarefas
Apr 25 14:51:11 xeon schedulerd[12375]: [SCHEDULER-CENTRAL] Criada fila de prioridade
Apr 25 14:51:11 xeon schedulerd[12375]: [SCHEDULER-CENTRAL] Fim de escalonamento
Apr 25 14:51:11 xeon schedulerd[12375]: [SCHEDULER-CENTRAL] Custo total de escalonamento: 0.012135
Apr 25 14:51:17 xeon schedulerd[12375]: [SCHEDULER-CENTRAL] jobcontact: 1
Apr 25 14:51:17 xeon schedulerd[12375]: [SCHEDULER-CENTRAL] Fim da execucao da aplicacao: /tmp/mpi-mergesort2-0.rsl
```

**Figura A.6:** *Log* do escalonamento de uma aplicação na Ferramenta SLOT.

Como pode ser visto na Figura A.6, depois do escalonamento da aplicação a Ferramenta SLOT inicia uma nova *thread* que, utilizando a biblioteca libDUROC, fica responsável por submeter e monitorar a execução de cada tarefa nos recursos. Esta *thread* indica o rótulo que identifica a aplicação (ex. *jobcontact: 1*) no ambiente de execução da Grade.

Para visualização da ocupação das tarefas no recursos a ferramenta SLOT possui um comando *print* que é enviado com o programa cliente que gera relatórios e

um Gráfico de Gantt que representa um *snapshot* do ambiente.

```
./client-sched -p 3000 -n xeon -d print
```

Uma interface WEB foi desenvolvida usando a linguagem PHP para gerenciar o acesso dos usuários e facilitar o processo de submissão de tarefas e geração de modelo de Aplicação automaticamente (Jacinto et al., 2007). A Figura A.7 apresenta a tela de *login* no sistema.

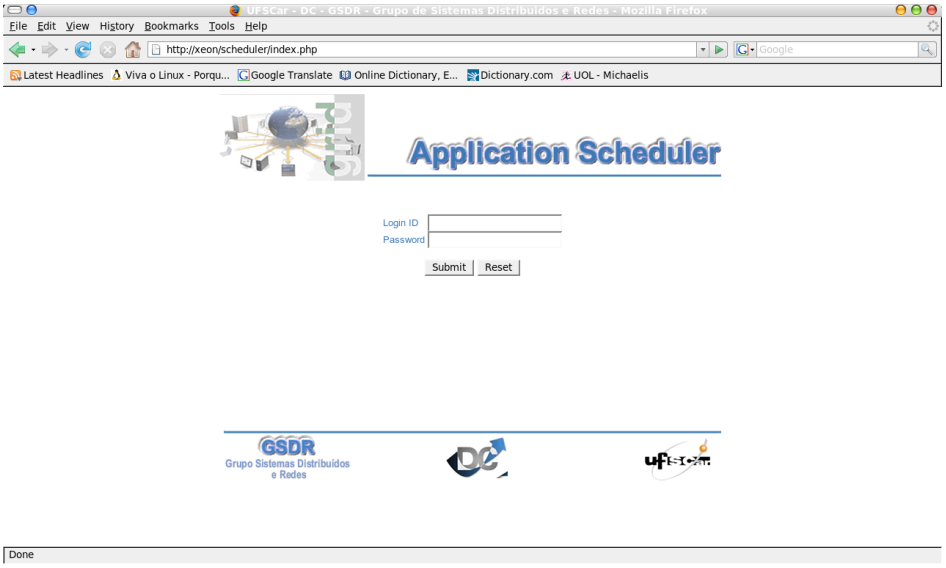


**Figura A.7:** Tela inicial do sistema WEB para execução de aplicações na Grade.

A Figura A.8 apresenta a tela inicial de boas-vindas logo após autorizar o acesso do usuário na Grade. Até o momento, o sistema WEB permite a geração automática do modelo de aplicação e a submissão de aplicações para a Grade.

O processo de criação automática do modelo de aplicação pode ser visto na Figura A.9. E o resultado da criação do modelo de aplicação pode ser visto na Figura A.10. Como pode ser visto, são gerados modelos de aplicações com o custo de execução das aplicações em **Ciclos**, **mips** e **segundos**, e uma representação gráfica do modelo de aplicação como pode ser visto na Figura A.11.

O sistema WEB permite ainda a submissão de aplicações para a Grade através de um formulário que permite usar algoritmos de escalonamento disponíveis no sistema. A Figura A.12 exemplifica o escalonamento de uma aplicação no ambiente


**Figura A.8:** Tela inicial do sistema WEB para execução de aplicações na Grade.

WEB usando o algoritmo HEFT.

## A.2 Ferramentas Auxiliares

Uma das ferramentas apresentadas que foram utilizadas para o desenvolvimento do trabalho é ferramenta GTgraph. A geração de modelos de aplicações sintéticas com essa ferramenta é feita através do comando abaixo:

```
./GTgraph-random -c config -o sint.app.100.sched.dot
```

O arquivo de configuração passado como parâmetro possui informações como valor mínimo e máximo para escolha aleatória do custo de cada tarefa, número total de comunicações entre os recursos e se é permitido *loops*.

Uma outra ferramenta desenvolvida no projeto foi chamada de createEnv. Esta ferramenta gera um modelo arquitetural aleatório de acordo com variáveis fornecidas por linha de comando como pode ser visto na Figura A.13.

Abaixo segue um exemplo da execução desta ferramenta para geração de um ambiente aleatório para o simulador.

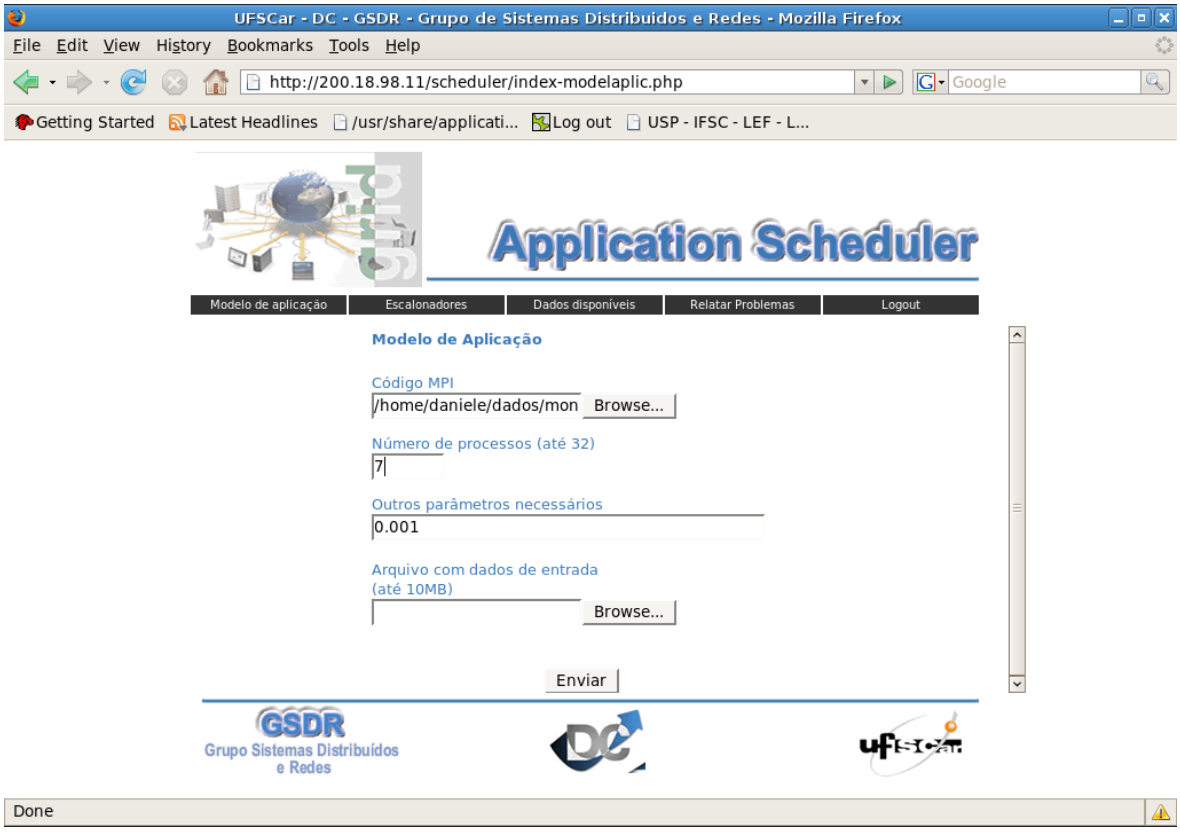


**Figura A.9:** Tela inicial do sistema WEB para execução de aplicações na Grade.

```
./arch-sim -o arch.xml -p 100 -P 1000 -b 80 -B 90 -l 0.005 -L 0.050
```

Por fim, uma outra ferramenta desenvolvida durante o projeto foi o simulador usado na validação da proposta. Este simulador foi desenvolvido com base na API da ferramenta de simulação SimGrid apresentado anteriormente. A Figura A.14 apresenta uma ajuda para execução de simulações.

A execução por linha de comando do simulador pode ser vista abaixo:

```
sd_test -v -p arch.xml -g dag-result.dot -s algo -l 0
```

Apesar de todas as telas apresentarem o ambiente desenvolvido, existem restrições que ainda estão em desenvolvimento. Uma destas restrições é o número limitado de algoritmos de escalonamento na ferramenta WEB. Uma outra restrição é o instante de ativação das tarefas nos recursos pelo Globus. Este instante de ativação é o tempo necessário para o Globus iniciar todos os processos necessários

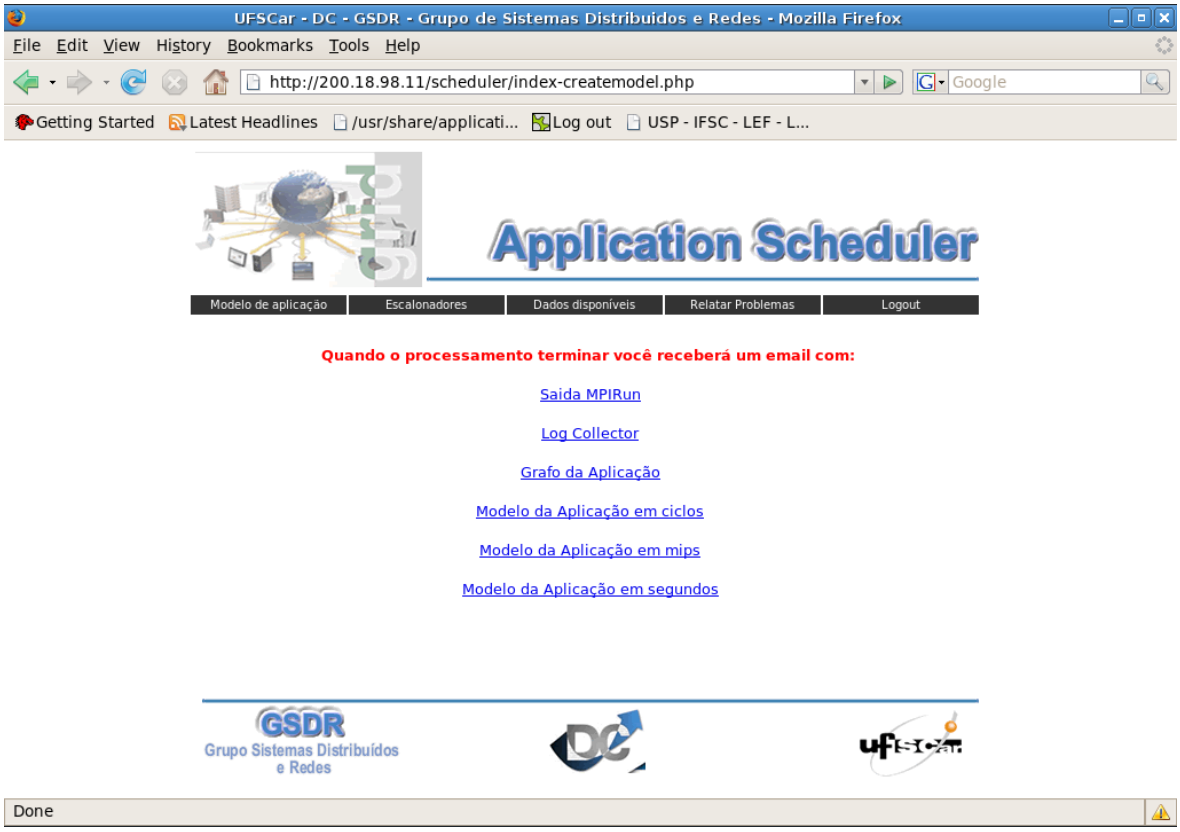


**Figura A.10:** Tela inicial do sistema WEB para execução de aplicações na Grade.

para execução das tarefas e ainda não foi completamente previsto no escalonamento.

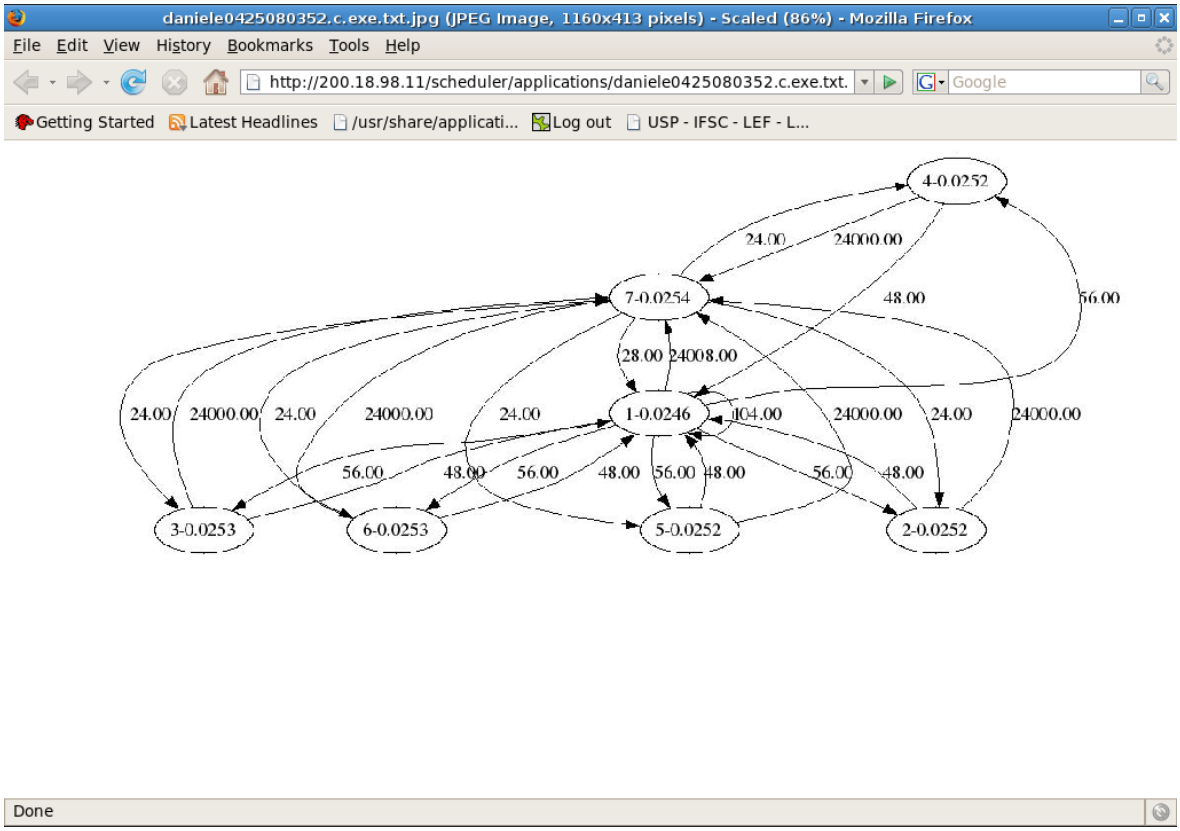


**Figura A.11:** Resultado gráfico do modelo de aplicação.

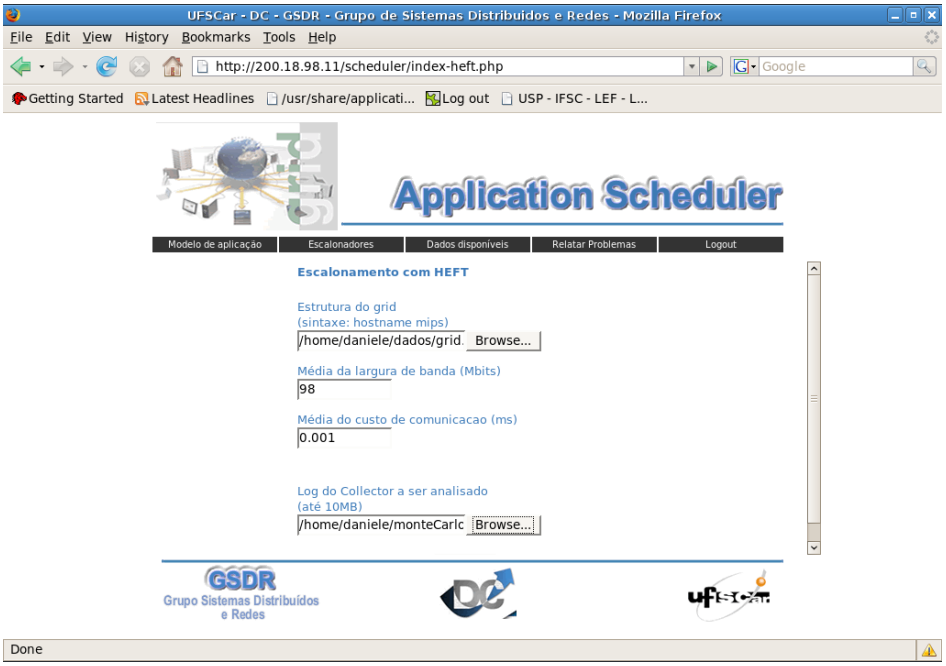


**Figura A.12:** Escalonamento de uma aplicação usando HEFT.

```
[ricardo@xeon create_env_sim]$ ./arch-sim --help
Usage: ./arch-sim options [inputfile]
      -h      --help                Apresenta este help.
      -o      --output filename     Arquivo gerado com a arquitetura.
      -n      --nodes INT            Quantidade de elementos da arquitetura.
      -p     --minproc INT           Capacidade minima de processamento de cada elemento.
      -P     --maxproc INT           Capacidade maxima de processamento de cada elemento.
      -b     --minband INT           Capacidade minima da largura de banda entre os elementos.
      -B     --maxband INT           Capacidade maxima de largura de banda entre os elementos.
      -l     --minlat  INT           Latencia minima entre os elementos.
      -L     --maxlat  INT           latencia maxima entre os elementos.
      -v      --verbose             Imprime cada detalhe da apresentacao.
```

**Figura A.13:** Criação do Modelo Arquitetural para o simulador.

```
ricardo@rembrandt:~/simgrid-3.2/examples/simdag$ ./sd_test -h
Usage: /home/ricardo/simgrid-3.2/examples/simdag/.libs/lt-sd_test options [inputfile]
   -h  --help                  Apresenta informações de utilização.
   -p  --platform filename     Arquivo do modelo arquitetural.
   -g  --graphfile filename    Arquivo que contém o restultado do escalonamento.
   -s  --schedalgo string      Algoritmo de escalonamento [opcional].
   -l  --load DOUBLE           Carga inicial no sistema.
   -v  --verbose               Imprime etapas da simulação.
```

**Figura A.14:** Ajuda de utilização do simulador.

# Referências Bibliográficas

(2008). The Network Simulator - NS2. http://www.isi.edu/nsnam/ns/.

Abramson, D.; Sosic, R.; Giddy, J.; Hall, B. (1995). Nimrod: A tool for performing parametized simulations using distributed workstations. *The 4th IEEE Symposium on High Performance Distributed Computing*.

Aggarwal, A. K.; Aggarwal, M. (2006). A unified scheduling algorithm for grid applications. In *HPCS '06: Proceedings of the 20th International Symposium on High-Performance Computing in an Advanced Collaborative Environment*, page 1, Washington, DC, USA. IEEE Computer Society.

Andrieux, A.; Berry, D.; Garibaldi, J.; Jarvis, S.; McLaren, J.; Ouelhadj, D.; Snelling, D. (2003). Open issues in Grid Scheduling. Technical report, UK e-Science Institute.

Bader, D. A.; Madduri, K. (2006). GTGraph: A synthetic graph generator suite. Oct 3 2006 http://www-static.cc.gatech.edu/ kamesh/GTgraph/.

Bailey, D.; Barscz, E.; Barton, J. (1994). The NAS parallel benchmarks. NAS Technical ReportRNR-94-007, NASA Ames Research Center.

Baker, M.; Buyya, R.; Laforenza, D. (2002). Grids and Grid technologies for wide-area distributed computing. *Softw. Pract. Exper.*, 32(15):1437–1466.

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, M.; Reed, D.; Shi, Z.; Sievert, O.; Xia, H.; YarKhan, A. (2005). New grid scheduling and rescheduling methods in the grads project. *International Journal of Parallel Programming (IJPP)*, Volume 33(2-3):209–229.

Berman, F.; Wolski, R.; Casanova, H.; Cirne, W.; Dail, H.; Faerman, M.; Figueira, S.; Hayes, J.; Obertelli, G.; Schopf, J.; Shao, G.; Smallen, S.; Spring, S.; Su, A.; Zagorodnov, D. (2003). Adaptive computing on the grid using AppLeS.

Boeres, C.; Lima, A.; Rebello, V. E. F. (2003). Hybrid task scheduling: Integrating static and dynamic heuristics. *IEEE. Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing.*

Boeres, C.; Nascimento, A. P.; Rebello, V. E. F.; Sena, A. C. (2005). Efficient hierarchical self-scheduling for mpi applications executing in computational grids. In *MGC '05: Proceedings of the 3rd international workshop on Middleware for grid computing*, pages 1–6, New York, NY, USA. ACM.

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

Casanova, H.; Dongarra, J. (1995). Netsolve: A network server for solving computational science problems. Technical report, Knoxville, TN, USA.

Casavant, T. L.; Kuhl, J. G. (1988). A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. Softw. Eng.*, 14(2):141–154.

Chen, H.; Maheswaran, M. (2002). Distributed dynamic scheduling of composite tasks on grid computing systems. *Proceedings of the International Parallel and Distributed Processing Symposium*.

DAGMan (2007). Condor dagman. http://www.cs.wisc.edu/condor/dagman/.

Deelman, E. (2003). Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing, 1:25*.

Dong, F.; Akl, S. G. (2006). Scheduling algorithms for grid computing: State of the art and open problems. Technical Report 2006-504, Queen's University School of Computing, Kingston, Ontario, Canada.

E. Huedo, R. M.; Llorente, I. (2005). The gridway framework for adaptive scheduling and execution on grids. volume 6, pages 1–8. Scalable Computing - Practice and Experience, Nova Science.

El-Rewini, H.; Lewis, T. G.; Ali, H. H. (1994). *Task scheduling in parallel and distributed systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Fahringer, T.; Prodan, R.; Duan, R.; Nerieri, F.; Podlipnig, S.; Qin, J.; Siddiqui, M.; Truong, H.-L.; Villazon, A.; Wieczorek, M. (2005). ASKALON: A grid application development and computing environment. In *GRID '05: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 122–131, Washington, DC, USA. IEEE Computer Society.

Ferrari, D. (1978). *Computer Systems Performance Evaluation*. Prentice Hall.

Ferreira, L.; Berstis, V.; Armstrong, J.; Kendzierski, M.; Neukoetter, A.; MasanobuTakagi; Bing-Wo, R.; Amir, A.; Murakawa, R.; Hernandez, O.; Magowan, J.; Bieberstein, N. (2003). *Introduction to Grid Computing with Globus*. IBM Corporation, 2 edition.

Foster, I. (2002). The grid: A new infrastructure for 21st century science. *American Institute of Physics*, pages 42–47.

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

Foster, I. T. (2005). Globus toolkit version 4: Software for service-oriented systems. In Jin, H.; Reed, D. A.; Jiang, W., editors, *IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779*, volume 3779 of *Lecture Notes in Computer Science*, pages 2–13. Springer.

Frey, J.; Tannenbaum, T.; Livny, M.; Foster, I.; Tuecke, S. (2002). Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246.

GmbH., I. (2005). Intel Trace Collector 6.0.1.0 - User's Guide 6.0.1.0.1.

Grimshaw, A. S.; Wulf, W. A. (1997). The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1):39–45.

Hwang, J.-J.; Chow, Y.-C.; Anger, F. D.; Lee, C.-Y. (1989). Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comput.*, 18(2):244–257.

Iverson, M.; Ozguner, F. (1998). Dynamic, competitive scheduling of multiple dags in a distributed heterogeneous environment. In *HCW '98: Proceedings of the Seventh Heterogeneous Computing Workshop*, page 70, Washington, DC, USA. IEEE Computer Society.

Jacinto, D. S. (2006). Adequao de aplicaes de clusters para grades computacionais. Master's thesis, Universidade Federal de So Carlos. Exame de Qualificao.

Jacinto, D. S.; Rios, R. A.; Guardia, H. C. (Jun 2007). Automatic modeling and grid scheduling MPI applications. *Anais do* V *Workshop de Computação em Grade e Aplicações (WCGA) - SBRC 2007*.

Jacob, B.; Brown, M.; Fukui, K.; Trived, N. (2005). *Introduction to Grid Computing*. IBM Corporation, 1 edition.

Kee, Y.-S.; Yocum, K.; Chien, A. A.; Casanova, H. (2006). Improving grid resource allocation via integrated selection and binding. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 99, New York, NY, USA. ACM Press.

Kleinrock, L. (1976). *Queueing Systems*, volume Vol. 2: Computer Applications. John Wiley and Sons.

Krauter, K.; Buyya, R.; Maheswaran, M. (2002). A taxonomy and survey of grid resource management systems for distributed computing. *Softw. Pract. Exper.*, 32(2):135–164.

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

Lathia, M. (2005). Advantages of grid computing. *IEEE Distributed Systems Online*, 6(2).

Lee, C.; Matsuoka, S.; Talia, D.; Sussman, A.; Karonis, N.; Allen, G.; Thomas, M. (2001). A grid programming primer: Programming models working group. *Global Grid Forum 1*.

Legrand, A.; Marchal, L.; Casanova, H. (2003). Scheduling distributed applications: the simgrid simulation framework. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 138, Washington, DC, USA. IEEE Computer Society. http://simgrid.gforge.inria.fr/.

Ma, T.; Buyya, R. (2005). Critical-path and priority based algorithms for scheduling workflows with parameter sweep tasks on global grids. In *SBAC-PAD '05: Proceedings of the 17th International Symposium on Computer Architecture on High*

*Performance Computing*, pages 251–258, Washington, DC, USA. IEEE Computer Society.

Mendes, H. A. (2004). HLogP: um modelo de escalonamento para a execução de aplicções MPI em Grades Computacionais. Master's thesis, Universidade Federal Fluminense.

Murshed, M.; Buyya, R.; Abramson, D. A. (2001). Gridsim: A toolkit for the modeling and simulation of global grids. Technical Report 2001/102, Monash University.

Nascimento, A. D. P.; Sena, A. D. C.; Silva, J. A. D.; Vianna, D. Q. C.; Boeres, C.; Rebello, V. E. F. (2005). Managing the execution of large scale MPI applications on computational grids. In *SBAC-PAD '05: Proceedings of the 17th International Symposium on Computer Architecture on High Performance Computing*, pages 69–76. IEEE Computer Society.

Prodan, R.; Fahringer, T. (2005). Dynamic scheduling of scientific workflow applications on the grid: a case study. In *SAC '05: Proceedings of the 2005 ACM Symposium on Applied Computing*, pages 687–694, New York, NY, USA. ACM Press.

Radulescu, A.; van Gemund, A. J. C. (1999). On the complexity of list scheduling algorithms for distributed memory systems. *Proc. of 13th International Conference on Supercomputing*, pages 68–75.

Rotithor, H. G. (1994). Taxonomy of dynamic task scheduling schemes in distributed computing systems. *IEE Proc. Digit. Tech.*, 141(1).

Saad, E. M.; Adawy, M. E.; Keshk, H. A.; Habashy, S. M. (2006). Task graph generation. In *Proceedings of the Twenty Third National Radio Science Conference, 2006. NRSC 2006.*, pages 1–9.

Shan, H.; Oliker, L.; Smith, W.; Biswas, R. (2004). Scheduling in heterogenous grid environments: The effects of data migration. Gujarat, India, 2004.

Sih, G. C.; Lee, E. A. (1993). A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Trans. Parallel Distrib. Syst.*, 4(2):175–187.

Song, H. J.; Liu, X.; Jakobsen, D.; Bhagwan, R.; Zhang, X.; Taura, K.; Chien, A. (2000). The microgrid: A scientific tool for modeling computational grids. *Scientific Programming*, 8(3):127–141.

Topcuouglu, H.; Hariri, S.; Wu, M. (2002). Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274.

Vallée, G.; Morin, C.; Berthou, J.-Y.; Rilling, L. (2003). A new approach to configurable dynamic scheduling in clusters based on single system image technologies. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 91, Washington, DC, USA. IEEE Computer Society.

Varga, A. (2001). The omnet++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference*, pages 319–324, Prague, Czech Republic. SCS – European Publishing House.

Vianna, B. A.; Fonseca, A. A.; Moura, N. T.; Menezes, L. T.; Mendes, H. A.; Silva, J. A.; Boeres, C.; Rebello, V. E. F. (2004). A tool for the design and evaluation of hybrid scheduling algorithms for computational grids. In *MGC '04: Proceedings of the 2nd workshop on Middleware for grid computing*, pages 41–46, New York, NY, USA. ACM.

Vianna, D. Q. D. C. (2005). Um sistema de gerenciamento de aplicações MPI para ambientes de grid. Master's thesis, Universidade Federal Fluminense.

Wolski, R.; Spring, N.; Peterson, C. (1997). Implementing a performance forecasting system for metacomputing: the network weather service. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–19, New York, NY, USA. ACM Press.

Yang, T.; Gerasoulis, A. (1994). Dsc: Scheduling parallel tasks on an unbounded number of processors. Technical report, Santa Barbara, CA, USA.

Yu, J.; Buyya, R. (2005a). A taxonomy of scientific workflow systems for grid computing. Technical Report 3, New York, NY, USA.

Yu, J.; Buyya, R. (2005b). A taxonomy of workflow management systems for grid computing. Technical report, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia.