4
Grids Computacionais

Ambientes de *grids* computacionais surgiram como uma solução prática para executar aplicações paralelas que necessitam de uma capacidade de processamento e armazenamento não disponível em um único conjunto de máquinas locais. Inicialmente, este capítulo dá uma visão geral da arquitetura do ambiente *grid*, destacando as funcionalidades típicas de cada uma de suas camadas. Posteriormente, são destacados os principais *middlewares* básicos e de serviços existentes atualmente na literatura para *grids*. A Seção 4.3 apresenta o *middleware* de serviço SGA EasyGrid que faz parte da própria aplicação, tornando-a autonômica.

4.1
Considerações Iniciais

Os ambientes *grids* são formados por uma coleção de recursos distribuídos geograficamente, tipicamente heterogêneos e compartilhados, conectados por uma rede de alta velocidade, capaz de oferecer um poder computacional maior do que ambientes como *cluster* de computadores para aplicações de alto desempenho. Assim, esse novo ambiente emerge com o objetivo de fazer com que a computação de alto desempenho também seja acessível aos usuários que não possuem, necessariamente, recursos suficientes localmente para suprir suas necessidades de poder computacional ou de armazenamento.

O termo *grid* computacional surgiu da analogia com as redes de interligação do sistema de energia elétrica (do inglês *power grids*), nas quais utiliza-se a eletricidade sem necessariamente saber a localização física da usina que a gerou, sendo sua utilização totalmente transparente aos seus usuários [65, 66]. A origem do nome retrata o objetivo de tornar o uso de recursos computacionais distribuídos tão simples quanto o uso da eletricidade.

Contudo, a exploração eficiente do poder computacional nesse tipo de ambiente ainda apresenta uma elevada complexidade, principalmente devido ao seu comportamento dinâmico e instável. Ao contrário dos sistemas distribuídos tradicionais, esse novo ambiente precisa considerar questões como segurança, acesso uniforme aos recursos geograficamente distribuídos (independentemente

da localização física dos mesmos), descoberta e agregação dinâmica de recursos e qualidade de serviço. Dessa forma, entre os grandes desafios apresentados por esse ambiente, são relevantes [20]:

- heterogeneidade: os recursos de hardware e software, que constituem o ambiente podem ser os mais diversos possíveis e agregam uma grande quantidade de tecnologias diferentes;
- escalabilidade: o aumento do número de recursos pode acarretar em perda de desempenho do ambiente;
- dinamismo: a natureza dinâmica desse tipo de ambiente deve ser cuidadosamente tratada porque a ocorrência de falhas na rede de comunicação e nas máquinas agregadas pode alterar a disponibilidade dos recursos;
- compartilhamento de recursos: como os recursos não são dedicados, pode haver uma elevada variação na quantidade de poder computacional disponível no ambiente; e
- multiplicidade de domínios administrativos: os recursos podem pertencer a diferentes proprietários, cada um com regras distintas para a gerência e utilização dos recursos.

Diante de todos esses desafios, é necessário que a computação em *grid* ofereça meios para que o desenvolvedor da aplicação possa escrever programas em linguagens de alto nível, capazes de acessar o *grid* e utilizar seus recursos de forma eficiente e transparente (importante ressaltar que durante toda esta tese, o conceito de transparente será usado como sinônimo para o não conhecimento dos detalhes envolvidos), sem que para isso o usuário tenha que se preocupar em tratar questões do ambiente. A próxima seção descreve as características principais da arquitetura *grid*, objetivando identificar os serviços mais importantes oferecidos por cada camada para garantir a execução da aplicação de forma eficiente e segura.

4.2 Arquitetura Grid

Uma característica marcante do ambiente *grid* é a sua diversidade de recursos. Além dos recursos de armazenamento e processamento, um *grid* pode disponibilizar acesso a outros recursos e equipamentos especiais, tais como *softwares*, licenças e instrumentos científicos. Com relação às características específicas de *hardware* e *software*, os *grids* podem ser divididos em três níveis [68]: infra-estrutura, *middleware* e aplicação, apresentados na Figura 4.1.

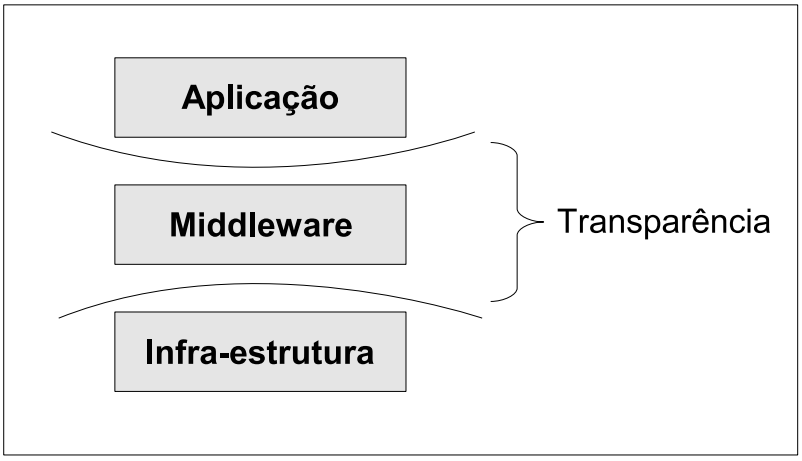

Figura 4.1: Níveis da plataforma *grid*.

O primeiro nível corresponde à infra-estrutura, formada por componentes de *hardware* e *software* integrados em uma única rede de recursos. No segundo nível, encontra-se o *middleware*, composto por ferramentas e serviços responsáveis por disponibilizar os recursos à aplicação. O último nível refere-se à aplicação, que tem como objetivo explorar os recursos disponíveis no *grid*.

A padronização de protocolos facilita a implantação da transparência desejada na utilização de um ambiente complexo como esse. Para viabilizar a padronização nos *grids*, Foster et al. [68] definiram uma arquitetura organizada em camadas. Assim, componentes de uma mesma camada compartilham funcionalidades comuns, e podem ser construídos com base nas capacidades e serviços prestados pelas camadas de níveis inferiores. A Figura 4.2 apresenta as cinco camadas que constituem a arquitetura *grid* de forma geral [68].

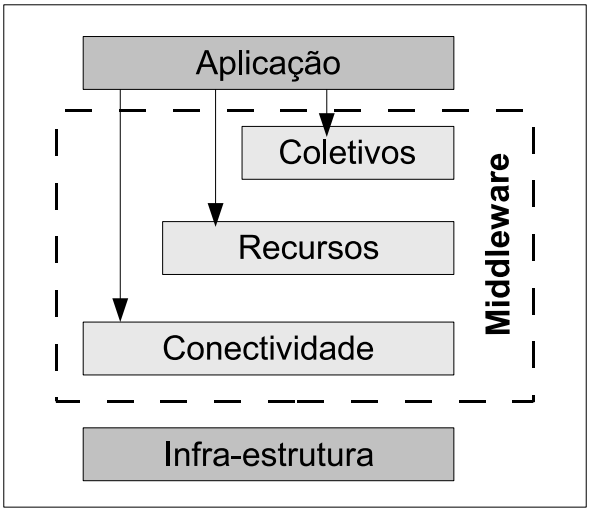

Figura 4.2: Camadas da arquitetura de protocolos e serviços da plataforma grid [68].

A camada de infra-estrutura é responsável por fornecer os recursos de processamento, de armazenamento e de comunicação, para os quais o acesso compartilhado pode ser mediado pelos protocolos do *grid*. A camada de conectividade representa o núcleo de comunicação e protocolos de autenticação para transações específicas em redes. Nessa camada há os protocolos de comunicação e os de autenticação. Os primeiros são responsáveis por viabilizar a troca de dados entre os recursos heterogêneos disponíveis na camada de infra-estrutura. Os protocolos de autenticação fornecem mecanismos seguros para a verificação da identidade dos recursos e dos usuários.

Com a comunicação garantida pela camada de conectividade, a camada de recursos tem a responsabilidade de inicializar e controlar o compartilhamento dos recursos individuais. Para isso, os protocolos de informação pertencentes a essa camada são usados para obter informações sobre a estrutura e o estado dos recursos. Os protocolos de gerenciamento são usados para negociar o acesso a recursos compartilhados, especificando, por exemplo, os recursos requeridos e as operações a serem desempenhadas.

A camada coletivos adota protocolos e serviços que não são associados a recursos específicos, tais como serviços de diretório para descoberta de recursos, serviços de co-alocação e escalonamento, monitoramento, replicação de dados e serviços de colaboração. A última camada é a de aplicação, cuja responsabilidade é viabilizar a execução das aplicações. O objetivo é garantir que as aplicações aproveitem os benefícios do ambiente sem o conhecimento das características específicas dos recursos distribuídos.

As maiores funcionalidades da arquitetura *grid* estão no nível de *middleware*. A fim de estruturar melhor os serviços oferecidos nesse nível, o mesmo foi subdividido em *middleware* básico e *middleware* de serviços. A Figura 4.3 apresenta os principais serviços oferecidos por cada um deles.

O *middleware* básico oferece os serviços fundamentais para a operação de um *grid*, como o gerenciamento remoto de processos, a alocação de recursos, o registro e a coleta de informações, segurança e aspectos de qualidade de serviço. Por outro lado, o *middleware* de serviço fornece um alto nível de abstração, incluindo ambientes de desenvolvimento de aplicações, ferramentas de programação, escalonamento de tarefas e tolerância a falhas, entre outros. Assim, o *middleware* básico garante que os processos possam executar em recursos remotos, enquanto que o *middleware* de serviço fornece uma imagem única do sistema. As Subseções 4.2.1 e 4.2.2 descrevem detalhes de cada tipo de *middleware*.

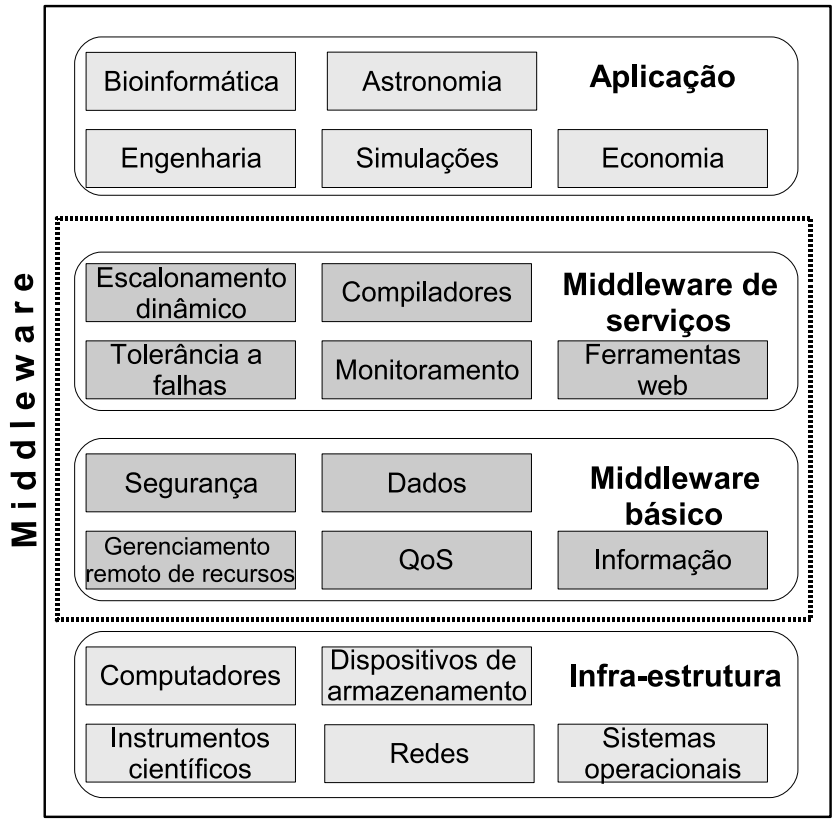

Figura 4.3: Subdivisão em camadas do nível de *middleware* [150].

4.2.1 Middleware Básico

Em ambientes *grid*, particularmente, o *middleware* é definido como uma camada de *software* que permite aos usuários compartilharem recursos heterogêneos em uma plataforma distribuída. A intenção é amenizar a carga do programador na hora de projetar, programar e gerenciar aplicações distribuídas, fornecendo aos desenvolvedores um ambiente de programação distribuída integrado e consistente [100].

Há na literatura várias implementações de *middlewares* básicos. O Globus Toolkit [75] é considerado pela comunidade científica um padrão de fato para esse novo ambiente, pois fornece serviços tais como autenticação, autorização, descoberta e acesso a recursos, segurança, execução das tarefas remotas e transferência de dados [61].

O Globus é construído com uma arquitetura em camadas, na qual serviços de alto nível podem ser desenvolvidos usando serviços de mais baixo nível. Atualmente, três versões do Globus *Toolkit* encontram-se disponíveis (versões 2, 3 e 4). A versão 2 foi toda desenvolvida em linguagem estruturada e está integrada tanto com a implementação LAM, como com a implementação MPICH do padrão MPI. A fim de facilitar a integração transparente de recursos

e serviços distribuídos, o Globus *Toolkit* passou a adotar, a partir da versão 3, uma abordagem orientada a serviços, resultado da integração das tecnologias e conceitos associados aos *grids* com as tecnologias de *web services* [66].

Nesta tese foi adotada a versão 2 do Globus *Toolkit*, pois as aplicações paralelas foram todas desenvolvidas com o LAM/MPI, e as versões 3 e 4 não tinham suporte ao LAM/MPI no momento das implementações. As principais funcionalidades disponíveis na versão 2 são apresentadas a seguir.

Segurança

No ambiente *grid* é necessário haver uma integração entre todas as soluções de segurança implementadas em cada *sítio* local (ou provedor de recursos). Dessa forma, o Globus *Toolkit* implementa seu serviço de segurança baseado no protocolo GSI (*Grid Security Infrastructure*), que usa uma infra-estrutura de chaves públicas para fornecer autenticação, comunicação segura e autorização.

O compartilhamento seguro dos recursos é garantido através da autenticação e autorização dos usuários. Os protocolos de autenticação fornecem mecanismos seguros de criptografia para verificação da identidade do usuário e recurso. Um *proxy* temporário é criado com base na chave privada do usuário, permitindo que esse faça solicitações remotas. Com o *proxy* não é necessária a intervenção do usuário para autenticar cada acesso a um novo recurso do *grid*.

Acesso aos Recursos Remotos

Outro importante serviço fornecido pelo Globus *Toolkit* é a descoberta dos recursos pertencentes ao ambiente *grid*. O controle de um recurso local no Globus *Toolkit* é realizado pelo GRAM (*Globus Resource Allocation Manager*), que atua como uma interface abstrata de recursos heterogêneos do *grid*. Assim, cada recurso é controlado por uma instância do GRAM, sendo essa responsável por instanciar, monitorar e relatar o estado das tarefas alocadas a tal recurso.

Gerenciamento de Dados

O Globus *Toolkit* trabalha com um módulo chamado GASS (*Global Access to Secondary Storage*), cuja função é fornecer serviços de acesso a armazenamento secundário. Em conjunto com outros módulos do Globus, é possível utilizar o GASS para a manipulação de arquivos remotos, ou seja, disparar executáveis, redirecionar as saídas de um processo, ler e escrever em arquivos remotos.

Serviço de Informação

Este talvez seja o serviço mais importante fornecido pelo Globus *Toolkit*, pois dá suporte para todos os demais. O MDS (*Monitoring and Discovery System*), também chamado de GIS (*Grid Information Service*), fornece o serviço de informação. O MDS usa o protocolo LDAP (*Lightweight Directory Access Protocol*) como uma interface para obter a informação sobre o recurso.

O MDS atua com o auxílio de vários componentes, mas dois desses componentes merecem destaque: *Grid Resource Information Service* (GRIS) e *Grid Index Information Service* (GIIS). O primeiro atua como um repositório de informações de recursos locais derivados dos provedores de informação. O segundo é um repositório que contém índices de informações de recursos registrados pelos GRISs e outros GIISs.

4.2.2 Middleware de Serviço

Os *middlewares* de serviço objetivam fornecer ferramentas que possam criar abstrações sobre a complexidade dos ambientes de *grid*. Grande parte dos serviços oferecidos pelos *middlewares* pode variar de sítio para sítio. Assim, em geral, para executar em uma plataforma, novas aplicações têm que ser escritas, ou aplicações existentes precisam ser adaptadas, com o objetivo de interagir com os componentes de *middleware* de serviços apropriados.

Atualmente, para aproveitar as vantagens dos serviços fornecidos pelo *middleware*, o programador deve ter um conhecimento profundo não só do problema a ser solucionado, mas também das funcionalidades disponíveis no *middleware* de serviço instalado sobre os recursos nos quais a aplicação será executada. Além disso, como existe uma grande variedade de *middlewares* e as aplicações são desenvolvidas com o objetivo de serem executadas em ambientes distintos, o programador da aplicação é obrigado a implementar diferentes versões da aplicação para cada ambiente de execução.

O objetivo é obter aplicações eficientes e robustas que sejam capazes de executar em *grid*, sem que isso gere uma carga ainda maior para o programador ou usuário, ou seja, que a mesma aplicação possa rodar nas mais diversas arquiteturas sem a necessidade de modificações. Para resolver esse problema há três classes de *middlewares* de serviço, que adotam diferentes decisões de projeto para resolver o mesmo problema.

A Figura 4.4 apresenta uma classificação proposta em [116], baseada no trabalho [98]. Essa classificação subdivide o *middleware* de serviço em três grupos diferentes: Sistema de Gerenciamento do Usuário (SGU), Sistema de

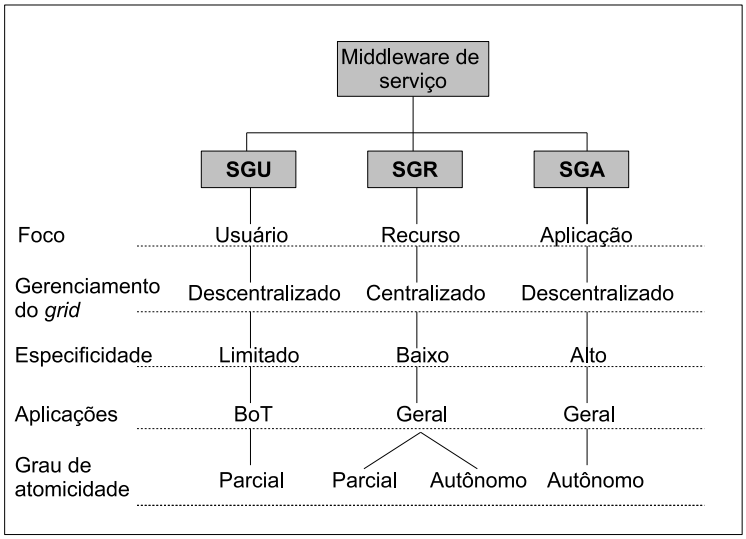

Figura 4.4: Classificação dos *middlewares* de serviço [116].

Gerenciamento de Recurso (SGR) e Sistema de Gerenciamento de Aplicação (SGA).

A primeira classe, SGU, trabalha com *middlewares* que adotam a gerência centrada no usuário (*user-centric*), no sentido em que a aplicação é executada de acordo com o objetivo definido pelo usuário (por exemplo, executar a aplicação por um tempo pré-determinado). Assim, a habilidade de um SGU atingir os objetivos definidos depende da qualidade da informação sobre a aplicação passada pelo usuário. Além disso, outro ponto negativo dos SGUs é que eles devem ser instalados em todos os recursos *grid* que a aplicação for disparada [116]. O Nimrod-G [35] e o MyGrid [39, 40] são representantes desta classe.

A segunda classe trabalha com *middlewares* que adotam uma visão centrada no sistema (*system-centric*), conhecidos como SGRs. O Condor-G [69, 143] é um exemplo desse tipo de *middleware* de serviço. Os SGRs são responsáveis por manter uma utilização eficiente de todos os recursos do *grid*. Para isso, a maioria dos SGRs depende de outros *middlewares* básicos para monitorar e analisar informações capturadas do sistema (por exemplo, carga dos recursos e vazão da rede), com o objetivo de monitorar a execução de várias aplicações nos recursos do ambiente *grid* simultaneamente. Para que as aplicações utilizem *middlewares* classificados como SGRs, é necessário que os mesmos estejam instalados em todos os recursos do *grid* onde serão executados os processos da aplicação. Além disso, outro ponto negativo desse grupo é que todas as decisões são tomadas com base apenas no estado dos recursos, sem considerar nenhuma informação sobre a aplicação [98, 116].

Como alternativa aos problemas apresentados pelos *middlewares* acima,

o terceiro grupo baseia-se na utilização de um Sistema de Gerenciamento de Aplicação (*application-centric*), que torna a própria aplicação responsável pela sua gerência. O SGA utiliza o *grid* de acordo com a disponibilidade dos recursos e com características específicas de cada aplicação. Dessa forma, uma aplicação adquire a habilidade de se auto-ajustar dinamicamente de acordo com as mudanças ocorridas no sistema, por exemplo ajustar o número de tarefas disparadas nos recursos do *grid* de acordo com as mudanças na política de administração. Assim sendo, um SGA gerencia recursos *grid* considerando não somente as requisições da aplicação, mas também suas características internas, resultando em execuções mais eficientes. Quando o SGA é parte de uma aplicação (chamado *system-aware*), diz-se que este está integrado à aplicação e a portabilidade entre *grids* ou *clusters* é garantida através da eliminação das dependências em relação ao *middleware* de serviço específico. Dessa maneira, aumenta-se o número de recursos disponíveis para a aplicação e a possibilidade de execuções rápidas. Exemplos dessa classe de *middleware* são o GrADS [148], o MyGrid [39, 40] e o SGA EasyGrid [32, 115].

4.3 Middleware SGA EasyGrid

O *framework* SGA EasyGrid é um *middleware* de serviço, classificado como SGA, para a execução automática e eficiente de aplicações MPI (do padrão MPI-2 [84]) em *grids* computacionais.

O projeto SGA EasyGrid [32] começou a ser desenvolvido em 2004 na Universidade Federal Fluminense, com o objetivo de construir um *framework* para a transformação automática dos programas paralelos escritos para *clusters* que utilizam a biblioteca MPI, em aplicações autonômicas e *system-aware* [119, 141] prontas para executarem em ambiente *grid*, que ofereça serviços de gerenciamento básicos (como o Globus [65]) e uma biblioteca de comunicação MPI (versões que tenham suporte a criação dinâmica de processos) [114]. As aplicações autonômicas são capazes de se auto-adaptarem, são robustas a falhas e capazes de reagirem às mudanças do sistema, que podem ocorrer em ambientes dinâmicos como os *grids*.

Para conseguir transformar as aplicações paralelas automaticamente em aplicações *system-aware*, o SGA EasyGrid incorpora, em tempo de compilação, um *middleware* específico da aplicação na forma de um SGA. Dessa forma, a utilização eficiente dos recursos é obtida pelo SGA distribuído junto com cada aplicação. Assim, cada instância do SGA EasyGrid gerencia a execução apenas de uma aplicação, de modo que as políticas de balanceamento e tolerância a falhas usadas são específicas para essa aplicação. Para gerenciar duas aplicações

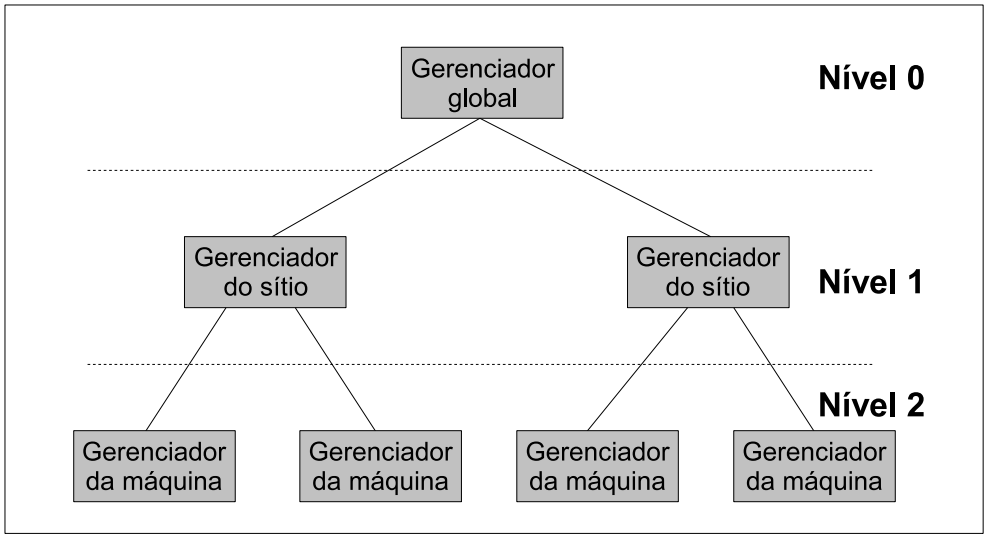

Figura 4.5: Hierarquia de gerenciadores baseada na organização da plataforma *grid*.

com o SGA EasyGrid, por exemplo, são necessárias duas instâncias dele, em que cada uma das instâncias pode adotar políticas de balanceamento e tolerância a falhas completamente diferentes.

O SGA EasyGrid emprega uma arquitetura hierárquica distribuída, dividida em três níveis de gerência [150], mostrada na Figura 4.5. Essa arquitetura foi adotada com o intuito de adequar os processos gerenciadores à organização dos recursos pelo *grid*, minimizando o custo embutido pelo gerenciamento da aplicação. No topo da hierarquia (nível 0), surge o Gerenciador Global (GG), encarregado de gerenciar todos os sítios envolvidos na execução da aplicação gerenciada.

O nível 1 está relacionado aos processos Gerenciadores dos Sítios (GS), que respondem pelo gerenciamento dos processos da aplicação atribuídos a cada sítio. Para finalizar, o nível 2 é composto pelos Gerenciadores Locais das Máquinas (GM), responsáveis pelo escalonamento, criação e execução de processos da aplicação atribuídos à máquina local [32].

Cada um dos três processos gerenciadores citados acima são estruturados em camadas, como apresentado na Figura 4.6. Cada camada é responsável por um serviço específico e essencial para a execução eficiente de uma aplicação MPI sob um *grid* computacional. Contudo, a funcionalidade de cada camada depende do nível do processo de gerenciamento na estrutura hierárquica.

A adoção de uma arquitetura em camadas possibilita que a aplicação se adapte às mudanças sofridas pelo ambiente de execução, uma vez que cada processo gerenciador é capaz de adotar diferentes políticas de escalonamento dinâmico e tolerância a falhas específicas às suas necessidades.

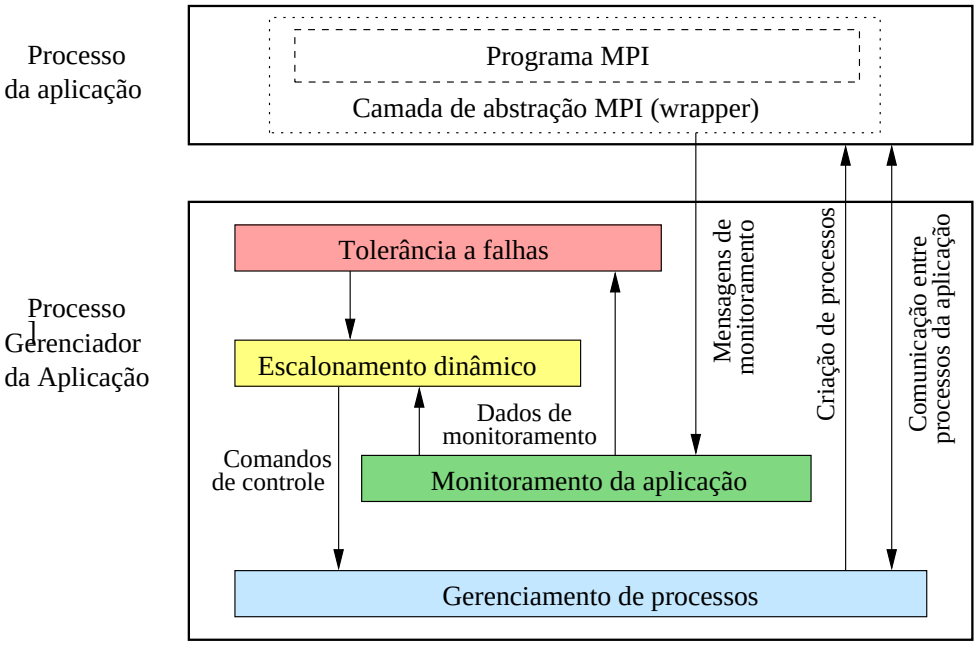

Figura 4.6: Estrutura em camadas do SGA EasyGrid [115].

Dentre todas as camadas, a que dá suporte às demais é a camada de *gerenciamento de processos*. Ela é responsável pela criação dinâmica de processos MPI, tanto da aplicação como dos gerenciadores, e pelo roteamento de mensagens trocadas entre eles. A camada de *monitoramento* tem a responsabilidade de coletar e disponibilizar as informações relevantes à execução de uma aplicação MPI. A camada de *escalonamento dinâmico* objetiva manter o balanceamento de carga no ambiente. Para isso, suas decisões são baseadas nas informações fornecidas pela camada de monitoramento. As informações coletadas também alimentam a camada de *tolerância a falhas*, cuja função é identificar e tratar as falhas [150].

Do lado cliente, para que a aplicação possa ser transformada em autonômica, é acrescentado uma camada de *abstração MPI* (*wrapper*). A funcionalidade da camada de abstração é embutir as funções do SGA EasyGrid nos processos MPI através do código fonte do usuário, de maneira totalmente transparente. A transparência é alcançada porque a implementação LAM/MPI não foi alterada, garantindo a portabilidade da aplicação e facilidade de implementação. Com essa camada de abstração, as funções MPI da aplicação do usuário são interceptadas e controladas pelos processos gerenciadores do SGA EasyGrid. As três próximas seções descrevem as funcionalidades de cada processo gerenciador do SGA EasyGrid.

4.3.1 Gerenciador Global - GG

A aplicação *system-aware* gerada pelo SGA EasyGrid inicia sua execução criando um único processo, o Gerenciador Global. O gerenciador global é o único processo da hierarquia que tem conhecimento sobre todas as máquinas pertencentes ao *grid* e escolhidas pelo usuário para execução da aplicação [150].

Dessa forma, esse processo é o responsável por criar todos os demais processos da arquitetura hierárquica, identificar falhas nos sítios pertencentes a execução, realizar trocas de mensagens entre sítios, controlar o processo de escalonamento e re-escalonamento das tarefas da aplicação entre sítios (chamado de re-escalonamento global).

Para o usuário, a criação do GG é transparente. O usuário precisa definir apenas em qual máquina ele será ativado, em quais máquinas o usuário tem acesso liberado para execução e o número total de tarefas que devem ser criadas durante a execução da aplicação.

Todavia, há aplicações dinâmicas, como as metaheurísticas, em que o número total de processos necessários para executar uma aplicação não é conhecido até o término da execução. Nesse caso, tanto o SGA EasyGrid como nenhum outro *middleware* já publicado na literatura são capazes de gerenciar a execução de metaheurísticas com criação dinâmica de processos no ambiente de *grid*.

Para resolver esse problema, durante esta tese, implementou-se uma nova versão do SGA EasyGrid em que a quantidade total de processos a ser criada não precisa ser conhecida. Nessa nova versão, os processos gerenciadores foram alterados, mas a compatibilidade com a versão anterior foi mantida. O Capítulo 6 descreve as características específicas dessa nova versão.

4.3.2 Gerenciador do Sítio - GS

Estes processos são criados dinamicamente pelo GG, a partir de arquivos de esquema (*appschema*) definidos diretamente pelo usuário (ou pelo portal EasyGrid) [30]. O arquivo de esquema de uma aplicação LAM/MPI é um arquivo ASCII que permite ao programador especificar um conjunto arbitrário de máquinas onde serão disparados os processos MPI. A versão LAM/MPI permite a utilização de arquivos de esquema nas chamadas à função $MPI\_Comm\_spawn()$. Dessa forma, esse arquivo define em que máquina de cada sítio deve ser disparado o processo GS e os parâmetros que devem ser repassados para cada um após sua criação.

Através das informações recebidas pelo arquivo de esquema, cada GS identifica as máquinas do sítio local que fazem parte da hierarquia, sendo capaz de disparar processos gerenciadores, chamados de Gerenciadores Locais das Máquinas, em cada uma das máquinas pertencentes ao seu sítio. Além de criar os GMs, o GS tem também a responsabilidade de gerenciar todas as máquinas pertencentes ao seu sítio. Para isso, ele deve checar se há máquinas com falha e a carga de cada uma.

4.3.3 Gerenciador Local da Máquina - GM

Os GMs são os últimos processos da hierarquia de gerência a serem criados. Eles são responsáveis apenas pelo gerenciamento de tarefas da aplicação atribuídas à máquina local, não exercendo nenhuma influência direta em relação às demais máquinas envolvidas na execução.

Uma importante característica do SGA EasyGrid é a sua capacidade de executar um ou mais processos por vez, concorrentemente no mesmo processador. Essa característica é garantida pelos GMs que controlam o número de tarefas que devem ser criado em cada recurso. Esse valor é definido pelo usuário, de acordo com as características específicas da aplicação.

Experimentos realizados com o SGA EasyGrid [115], revelam que tarefas de aplicações *bag-of-tasks* com tempo de execução menor do que um segundo, devem ser executadas concorrentemente com mais uma ou duas tarefas, para sobrepor o custo da criação dinâmica de um processo MPI (em torno de 10 milesegundos). Todavia, tarefas com tempo de execução maior do que dez segundos, devem ser executadas apenas uma em cada recurso, pois nesse caso o custo da criação dinâmica torna-se irrisório e a concorrência passa a acarretar um número maior de troca de contexto. Processos com tempo de execução entre um e dez segundos se comportam praticamente da mesma maneira quando um ou dois processos executam simultaneamente, com um ganho de desempenho desprezível caso sejam criadas duas tarefas por vez [115, 116].

A Figura 4.7 apresenta uma visão da construção da hierarquia de gerenciadores em um *grid* computacional formado por quatro sítios. Como pode ser observado, há apenas um processo GG em todo o ambiente, quatro processos GSs (um por sítio) e um GM em cada máquina do ambiente em que processos da aplicação estão rodando ou podem vir a rodar.

É importante manter um GM em cada máquina do ambiente *grid*, mesmo nas que não vão receber carga inicial da aplicação, pois caso haja necessidade de usá-las mais tarde (por exemplo, devido a uma sobrecarga no ambiente), as mesmas já estarão prontas para receber tarefas da aplicação.

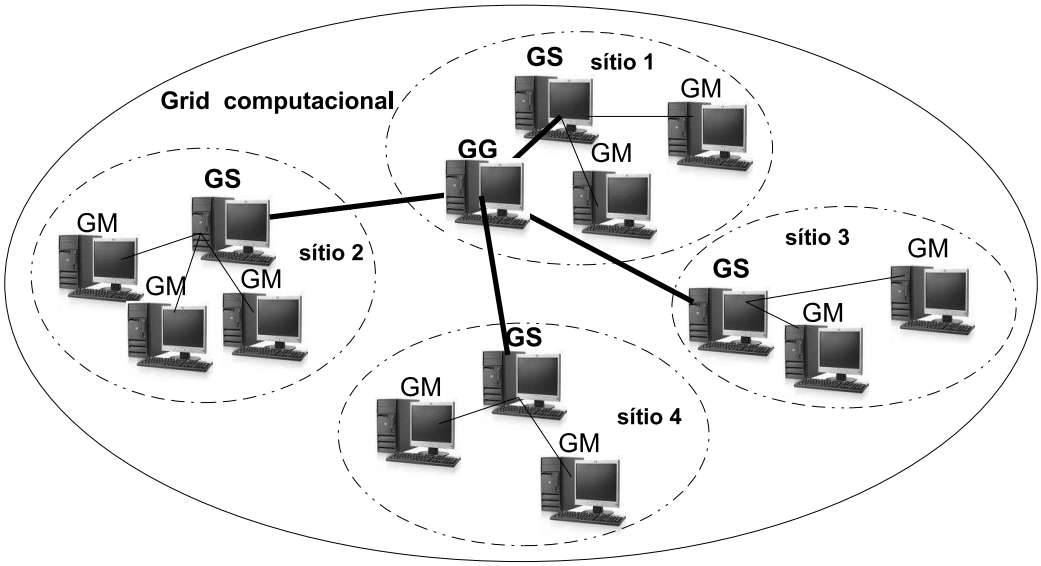

Figura 4.7: Hierarquia de gerenciadores no *middleware* SGA EasyGrid.

Apesar da especificação do comando *lamgrow* na biblioteca LAM/MPI, responsável pela inclusão de novos processadores a um ambiente LAM já criado, este ainda não é suportado em conjunto com o Globus Toolkit. Como conseqüência, na versão LAM/MPI para *grid*, novos processadores não podem ser incluídos ao ambiente após a aplicação ter sido iniciada.

4.3.4 Processos da Aplicação

Como visto na Seção 4.3.1, a ativação dos processos gerenciadores no SGA EasyGrid inicia na direção *top-down*, ou seja, o gerenciador global é o primeiro a ser criado e os GMs são os últimos. O GG ao ser criado recebe um arquivo contendo um escalonamento estático, tipicamente gerado por um algoritmo de escalonamento, e o repassa para os gerenciadores dos sítios, que em seguida, encaminham para todos os gerenciadores das máquinas pertencentes a ele. Cada processo gerenciador da máquina é capaz de identificar quais e em qual ordem tarefas da aplicação devem ser executadas na máquina gerenciada por ele.

O escalonamento estático não é obrigatório, mas é aconselhável para que seja feita uma distribuição inicial adequada de tarefas entre os recursos do *grid* selecionados pelo usuário para execução da aplicação.

Após o SGA EasyGrid preparar o ambiente de execução, os processos que não possuem dependência em relação a nenhuma outra tarefa da aplicação, e que foram atribuídos a alguma máquina, são inseridos em uma lista de tarefas prontas para execução. Contudo, as tarefas que possuem algum tipo de

dependência são inseridas na lista de tarefas pendentes até que essas pendências sejam atendidas.

Além de verificar a dependência entre as tarefas, faz-se necessária a adoção de políticas que possam controlar o número de processos concorrentemente em execução em uma máquina do *grid*. A falta de controle sobre o número de processos prontos disparados pode levar a máquina local a uma sobrecarga, pois todos os processos prontos terão sido imediatamente criados, resultando em uma queda significativa no desempenho da aplicação [33, 115].

A criação dinâmica é uma alternativa para os problemas citados acima, pois permite-se controlar o número de processos concorrentes em execução em uma máquina; adaptar a aplicação às mudanças sofridas no ambiente; oferecer maior facilidade na recriação de um processo em caso de falha e, principalmente, controlar a escalabilidade e eficiência na execução de aplicações em grande escala.

Esse aumento na escalabilidade das aplicações paralelas via MPI é possível porque o número de processos criado em cada máquina é controlado de acordo com a carga e capacidade do recurso, não correndo o risco de estourar a capacidade de algum recurso requerido, como o número de *sockets* abertos, por exemplo. Isso é uma característica importante do SGA EasyGrid, pois diferentemente da maioria do sistemas gerenciadores [28, 40, 69], com o SGA EasyGrid não há limite de processos que uma aplicação pode ter. Com o objetivo de executar em ambiente *grid*, estima-se que as aplicações sejam constituídas de milhares de tarefas (aplicações de larga escala). Experimentos com até 100.000 tarefas já foram realizados com o SGA EasyGrid [115, 150].

4.3.5 Escalonamento Híbrido

A metodologia de escalonamento adotada pelo SGA EasyGrid é baseada no modelo de escalonamento híbrido [31], no qual são empregadas heurísticas de escalonamento estática e dinâmica [37]. O escalonador estático é baseado em estimativas do tempo de execução de cada tarefa, e tem a responsabilidade de gerar um escalonamento inicial das tarefas da aplicação paralela. Por outro lado, o escalonamento dinâmico tem a função de ajustar a aplicação as mudanças sofridas no ambiente *grid*, durante a execução da aplicação.

Assim, o escalonamento de tarefas no SGA EasyGrid inicia com a distribuição das tarefas da aplicação nos recursos indicados pelo escalonador estático. Porém, como dito anteriormente, essas tarefas não são criadas todas de um só vez. Somente as tarefas prontas (que receberam todas as mensagens necessárias) podem executar, sendo que o número de tarefas executando

simultaneamente em um mesmo recurso é controlado pelos GMs com o objetivo de otimizar o desempenho da aplicação.

Todavia, em conseqüência das mudanças dinâmicas que ocorrem em um ambiente *grid*, é necessário modificar a alocação estática inicial, objetivando atingir um bom desempenho da aplicação. No *middleware* SGA EasyGrid, a realocação de processos é feita apenas com os processos que ainda não foram criados, ou seja, que não estão em execução. Isso é uma decisão de projeto do SGA EasyGrid, a fim de simplificar o processo de escalonamento, pois a migração de processos (em execução) é muito cara, principalmente quando há necessidade de *checkpoint*.

A política de escalonamento dinâmica do SGA EasyGrid é implementada nos três níveis de gerência do *middleware*, distribuindo a sobrecarga de manter o sistema em equilíbrio entre todas as camadas da arquitetura. Dessa forma, há um Escalonador Dinâmico Global (EDG) executando no GG; um Escalonador Dinâmico de Sítio (EDS), em cada GS; e um Escalonador Dinâmico de Máquina (EDM), em cada GM [115].

O EDM é o responsável por definir, no momento da criação, o número máximo de processos concorrentes e a ordem com que esses processos serão criados na sua máquina. Essas ações estão relacionadas à política local de escalonamento adotada por cada processo gerenciador das máquinas em que foram atribuídas tarefas da aplicação.

O mecanismo de escalonamento que considera apenas as máquinas pertencentes a um único sítio é denominado *redistribuição local de tarefas* e é implementado pelo EDS.

No último nível da arquitetura em camadas do EasyGrid, há um mecanismo que leva em consideração todos os sítios do *grid*, chamado de *redistribuição global de tarefas*, realizado pelo EDG.

A redistribuição local do sítio, considera apenas as máquinas de um único sítio durante a re-alocação de tarefas. Através das informações obtidas durante o monitoramento da aplicação, o processo GS é capaz de acompanhar a execução de tarefas da aplicação atribuídas às máquinas gerenciadas pelos GMs, identificando e tratando possíveis situações que possam acarretar em uma queda no desempenho da aplicação. O GS mantém uma lista com dados referentes a todas as tarefas atribuídas às máquinas do sítio.

Dessa forma, o EDG tem uma visão geral da aplicação e dos recursos envolvidos na execução da aplicação, sendo responsável por decidir se há ou não necessidade de reescalonar tarefas entre os sítios do ambiente. A decisão de reescalonar ou não tarefas entre máquinas é atividade do EDS. Para isso, ele analisa informações recebidas a partir de cada GM que esteja sob sua

gerência. Por último, os EDMs decidem o momento em que cada tarefa deverá ser executada no recurso, e essa decisão depende da política de escalonamento adotada por ele.

4.3.6 Tolerância a Falhas

Em ambientes heterogêneos e dinâmicos como os *grid* computacionais, é muito comum a ocorrência de falhas nos recursos disponíveis. A versão atual do SGA EasyGrid implementa tolerância a falhas, permitindo a recuperação sem interromper a execução da aplicação. O SGA EasyGrid usa intercomunicadores distintos entre cada par de processos (tanto para processos gerenciadores quanto para processos da aplicação) para que falhas possam ser identificadas e isoladas [139].

Além disso, o SGA EasyGrid usa as técnicas de retentativa (*retry*) e de recurso alternativo (*alternative resource*), juntamente com a técnica de mensagens de *log* para detectar as falhas das tarefas [115]. A técnica de retentativa consiste em executar a mesma tarefa nos mesmos recursos em que a falha ocorreu. A técnica de recurso alternativo submete as tarefas que falharam a um outro recurso (assim, manipulando falhas de recursos).

Dessa forma, o SGA EasyGrid implementa um esquema para o gerenciamento de mensagens de *log* para manipular as mensagens de entrada de uma tarefa da aplicação até que as tarefas executem com sucesso. Se o tempo estimado para uma determinada tarefa exceder um limite pré-determinado, a tarefa será investigada pelo seu GM. Se essa tarefa tiver falhado, isso será detectado e o subsistema de tolerância a falhas ativará o módulo de escalonamento para criar um novo processo para substituir a tarefa que falhou. Todas as mensagens para ela serão recuperadas pelo GM a partir do *log* de mensagem e a re-execução de tarefas poderá continuar sem a necessidade de sincronização entre os processos espalhados pelas máquinas do *grid* [139].

Apesar do módulo de tolerância a falhas do SGA EasyGrid ser capaz de recuperar falhas em todos os processos, ele não recupera falhas no GG. Na versão atual do SGA EasyGrid, uma falha no processo GG, do ponto de vista da aplicação, é considerada fatal e tem como conseqüência falhas em todos os processos da hierarquia. Para evitar que uma falha no GG comprometa a execução da aplicação, o GG deve ser disparado em uma máquina persistente, isto é, uma máquina confiável, estável e segura. Atualmente, a maioria dos sistemas de tolerância a falhas necessitam ao menos de uma máquina persistente capaz de armazenar informações sobre a execução da aplicação, para possíveis recuperações em caso de falhas nos recursos ou processos [139].

4.4 Considerações Finais

Como visto neste capítulo, *grids* objetivam agregar um grande número de recursos geograficamente distribuídos para fornecer poder computacional suficiente para executar aplicações que necessitam de alto poder de processamento e armazenamento.

Todavia, escrever aplicações paralelas que possam executar eficientemente nesse ambiente não é uma tarefa fácil, pois os recursos envolvidos são heterogêneos, não dedicados e geralmente não oferecem garantia alguma de desempenho ou disponibilidade. Além disso, a maioria dos programadores não está familiarizada com esse tipo de ambiente. Como conseqüência, as aplicações são implementadas seguindo as estratégias tradicionais de paralelização, sem preocupação com a gerência do ambiente.

Para minimizar essas dificuldades, alguns projetos têm sido dedicados ao desenvolvimento de *middlewares* que objetivam gerenciar o ambiente e tornar transparente esses detalhes para o nível de aplicação, deixando os programadores se concentrarem em como explorar o paralelismo da aplicação para resolver o problema. O *middleware* usado nesta tese para gerenciar a aplicação paralela foi o SGA EasyGrid.

Inicialmente o SGA EasyGrid foi projetado para operar com aplicações *bag-of-tasks*. Essas aplicações caracterizam-se por terem um número fixo de tarefas com pouca comunicação entre elas. Todavia, ainda que o *middleware* EasyGrid permitisse a criação dinâmica de tarefas, suas estruturas de dados necessárias para a gerência da aplicação e do ambiente eram todas alocadas estaticamente e grande parte de suas estruturas precisavam conhecer o número de tarefas da aplicação, para que pudessem ser alocadas.

Em computação paralela, o número de tarefas de uma aplicação depende do trabalho a ser realizado e da estratégia de paralelização adotada. Todavia, em aplicações como metaheurísticas, o trabalho a ser executado por cada aplicação não é necessariamente determinístico. Assim, fatores como o critério de parada e as sementes usadas podem influenciar no número de tarefas necessárias para completar a aplicação.

Conseqüentemente, usar o *middleware* EasyGrid para gerenciar metaheurísticas paralelas no ambiente *grid* só seria possível quando o número de tarefas fosse conhecido. A fim de tratar essa limitação do EasyGrid, foi desenvolvida, durante esta tese, uma versão dinâmica do SGA EasyGrid que não precisar ter nenhum conhecimento do número de tarefas a serem criadas. Nessa versão, todas as estruturas de gerência do EasyGrid são alocadas dinamicamente, de acordo com a demanda de tarefas necessárias para a execução. Todos

os detalhes dessa nova versão são discutidos no Capítulo 6.

A fim de analisar o desempenho obtido através do uso do paradigma de programação tradicional mestre-trabalhador, o Capítulo 5 apresenta quatro versões paralelas da heurística `GRILS-mTTP` desenvolvidas através dessa estratégia. A idéia é observar o comportamento das implementações paralelas em um ambiente *grid*.