UNIVERSIDADE FEDERAL FLUMINENSE

HENRIQUE BUENO RODRIGUES

Grid SA: Uma Sociedade Autônoma

NITERÓI

2009

UNIVERSIDADE FEDERAL FLUMINENSE

HENRIQUE BUENO RODRIGUES

Grid SA: Uma Sociedade Autônoma

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

Orientador:
Vinod Rebello

NITERÓI

2009

Grid SA: Uma Sociedade Autônoma

Henrique Bueno Rodrigues

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre.

Aprovada por:

Prof. Eugene Francis Vinod Rebello Ph.D. / IC-UFF
(Presidente)

Profa. Noemi de La Rocque Rodriguez D.Sc. / PUC-Rio

Profa. Maria Cristina Silva Boeres Ph.D. / IC-UFF

Niterói, 08 de dezembro de 2009.

Agradeço a Deus pela minha família.

Agradecimentos

O mestrado foi um dos maiores desafios que já enfrentei na minha vida. Mas como tudo, não conseguiria alcançar esse objetivo sem a ajuda de muitas pessoas. Gostaria de começar agradecendo à galera da UFF, em especial o pessoal do SGCLab. Essa galera que, apesar de muito bagunceira, rala muito para fazer aquele laboratório funcionar (funciona mesmo? Rs ...). Também gostaria de agradecer ao professor Vinod por todos os ensinamentos nesses 5 anos (a iniciação científica começou em 2004!). Através da busca incessante pelo resultado ótimo ele me ensinou o que é de fato a pesquisa. Também tenho que agradecer ao pessoal da Petrobras pelo incentivo de sempre.

Não posso esquecer da minha família que me apoiou durante todo o tempo. Obrigado pai e mãe por me ensinar que somente através do esforço, dedicação, trabalho e honestidade podemos conquistar nossos objetivos. Obrigado a minha irmã Aline e ao meu cunhado Igor (que é um irmão) por me apoiar e me acolher diversas vezes na sua casa. Também agradeço a minha irmã Helô por me incentivar e muitas vezes me mostrar coisas que a correria do dia a dia me fazia esquecer. Obrigado a Cíntia pelo incentivo que faltava e pelo apoio incondicional durante esses dois anos. Enfim, agradeço a Deus por mais essa vitória. Sem Ele nada disso seria possível.

Resumo

A popularidade de temas como computação nas nuvens e computação verde é consequência da crescente necessidade de maximizar a utilização de recursos computacionais. Há uma constante busca pelo aumento da capacidade de processamento sem a necessidade de incremento da quantidade de recursos e principalmente do consumo de energia.

Em grades computacionais a alocação das aplicações pelos recursos disponíveis é feito na maioria das vezes através de sistemas gerenciadores. A capacidade de um ambiente de grade maximizar o uso de seus recursos sem comprometer custos está diretamente ligado a eficiência de seu sistema gerenciador. Os sistemas gerenciadores de grade atuais funcionam como *brokers* que alocam aplicações em subconjuntos de recursos disponíveis. No processo de alocação, caso uma aplicação já esteja executando sobre um determinado recurso o gerenciador não irá alocar outra ao recurso até que a primeira tenha concluído sua execução. Assim, como não há compartilhamento simultâneo não haverá perda de eficiência causada pela troca de contexto das aplicações.

Entretanto, na maioria das vezes uma aplicação não utiliza ao mesmo tempo todos os serviços (*CPU*, *GPU*, entrada/saída, etc.) oferecidos por um recurso. Dessa forma, a alocação dedicada de um recurso a uma única aplicação subutiliza o mesmo. Além disso, como a alocação é feita por bloco, ou seja, um determinado conjunto de recursos é alocado a uma única aplicação até seu término, recursos poderão ficar ociosos uma vez que a aplicação não necessariamente utilizará todos os recursos durante toda sua execução.

Com o crescente número e variedade de aplicações e recursos tentando tirar proveito da computação em grades, o uso de um *broker* central torna-se mais complexo e não escalável. Uma alternativa para o uso de um *broker* central orientado ao sistema é um modelo de gerenciamento baseado nos conceitos de controle distribuído e aplicações autônomas. Nesta abordagem, aplicações se automonitoram, se ajustam dinamicamente aos recursos disponíveis e executam ações a fim de alcançar seus objetivos. O principal desafio desta idéia é definir qual deve ser o comportamento dessas entidades autônomas para que a sociedade (ou conjunto) de aplicações seja sustentável. O objetivo deste trabalho é avaliar a viabilidade desta idéia e propor uma solução de implementação para a mesma.

Palavras-chave: sistema gerenciador de grades, computação autônoma, sistema gerenciador de aplicações EasyGrid

Abstract

The popularity of cloud computing and green IT has been motivated by the growing demand for improved and cost effective resource utilization. System designers today must aim to increase processing capacity or output without simply aggregating ever larger numbers of resources at the expense of being faced with the corresponding explosion in power consumption.

In grid computing, the allocation of applications to available resources is mostly performed by Resource Management Systems (RMSs). Thus the capacity of the grid to maximize resource usage without compromising costs is directly linked to the efficiency of its RMS. Current grid RMSs typically operate as *brokers* to allocate jobs to the best subset of resources available. In the currently adopted allocation model, each job is scheduled exclusively to a given (set of) resource(s). Thus the RMS won't allocate another job to those resources until the previous job has ended, with the hypothesis being to avoid any loss in efficiency due to context switching between concurrently executing jobs.

However, most applications do not use all of the services (CPU, GPU, I/O, etc.) offered by a resource at the same time. Thus, the allocation of just one application to a resource may leave that resource severely underutilized. Moreover, resources are allocated to jobs in blocks of a size equivalent to application's maximum processor requirement. In other words, a fixed subset of resources is allocated exclusively to a unique application until it completes its execution, independent of whether or not individual resources are in fact being utilized for the entire duration.

With the growing variety of applications aiming to take advantage of grids with large numbers of heterogeneous resources, designing an efficient single central broker becomes extremely complex and unlikely to scale up. An alternative proposal is a management model based on autonomic computing concepts. In this approach, applications are self-managed, and thus are capable of discovering the best available resources dynamically and autonomously. Distributed Application Management Systems understand the specific requirements of their applications and can be designed to permit sharing of underutilized resources. The main challenge to turn this idea into a viable concept is to define the necessary behaviors required by a society of autonomous applications to maintain a sustainable and efficient utilization of the grid environment. The objective of this work is to evaluate the viability of this proposal and define some appropriate behaviors for autonomic MPI applications.

Keywords: grid middleware, autonomic computing, EasyGrid AMS

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Existem diversas áreas de conhecimento que demandam alto poder computacional para resolução de seus problemas. Negócios como exploração e produção de petróleo, análise climática e bioinformática [14, 19] são exemplos de atividades que requerem longos períodos de intenso processamento. Entretanto, essas necessidades não são atendidas através de computadores convencionais. Assim, para atender essa demanda surgiram os supercomputadores.

Supercomputadores [37] são sistemas de alto desempenho altamente especializados. Geralmente são formados por centenas de unidades de processamento idênticas dispostas em uma mesmo local, conectadas por redes de alta velocidade e sob um único domínio administrativo. *Clusters* de computadores [8] são exemplos de sistemas desse tipo. Entretanto, esses sistemas possuem um alto custo de aquisição e manutenção. Por exemplo, para expandir tal arquitetura é preciso que a infraestrutura de rede seja ampliada e recursos idênticos sejam adicionados ao sistema. Assim, conforme recursos mais modernos surgem, provavelmente o ambiente legado será abandonado já que há uma baixa possibilidade de se fazer uma integração. Dessa forma, o alto custo impossibilita que a maioria dos centros universitários e corporativos tenha um ambiente desse porte.

Com o objetivo de suprir a grande demanda por processamento a um baixo custo surgiram as grades computacionais [16, 17, 18]. Grades também podem ser consideradas supercomputadores, mas apresentam algumas características próprias quando comparadas aos sistemas de alto desempenho tradicionais (*clusters*). Tipicamente, grades são compostas por uma grande quantidade de recursos heterogêneos espalhados geograficamente, ou seja, não precisam estar fisicamente próximos nem em configuração uniforme. Em função desta distribuição, geralmente a rede responsável por interconectar esses recursos é a *Internet*. Além disso, em função da natureza distribuída, diversas falhas nos

nós e conexões dessa rede podem ocorrer. Entre os grandes desafios desta tecnologia destacam-se [39]:

- heterogeneidade do ambiente grade, resultado da grande quantidade de diferentes recursos que podem ser conectados e de diferentes tecnologias empregadas por esses;
- multiplicidade de domínios administrativos autônomos, cada um com diferentes regras de gerenciamento e controle de acesso;
- escalabilidade, que pode provocar redução do desempenho conforme o número de recursos na grade aumenta;
- natureza dinâmica desse tipo de ambiente, onde a taxa de ocorrência de falhas na rede de interconexão e nas máquinas agregadas é alta;
- compartilhamento de recursos, o que provoca uma variação no poder computacional oferecido por cada recurso.

Assim, para resolver esses problemas, uma solução adotada tem sido a implementação de um *middleware* [25], camada de *software* cujo objetivo é retirar a sobrecarga do programador na hora de projetar, programar e gerenciar aplicações distribuídas, fornecendo aos desenvolvedores a imagem de um ambiente de programação distribuído, integrado e consistente [24]. Esse *middleware*, também chamado de sistema gerenciador, funcionará como uma camada de abstração e oferecerá acesso simples, barato e eficiente a grade para usuários que desejam executar suas aplicações nela. A figura 1.1 apresenta as camadas dessa arquitetura.

Dessa forma, o desenvolvimento de um sistema gerenciador capaz de alocar e gerenciar aplicações nos recursos da grade eficientemente, e maximizar a utilização dos recursos desse ambiente é uma das principais áreas estudadas por pesquisadores.

1.1 Motivação

Uma grade computacional oferece aos usuários a possibilidade de executar suas aplicações paralelas em um ambiente extremamente potente em um tempo reduzido quando comparado a execução em computadores convencionais. Entretanto, essa oferta não deve introduzir complexidade demasiada no desenvolvimento de aplicações. Geralmente, os desenvolvedores dos *softwares* que irão rodar nesses ambientes são cientistas e engenheiros

Figura 1.1: Camadas da arquitetura grade

que possuem muito conhecimento do negócio em questão e não querem gastar tempo alterando suas aplicações para adequar estas às características da grade e na maioria das vezes desconhecem como. Assim, sistemas de gerenciamento que façam essas tarefas são necessários para adequar as aplicações ao ambiente grade.

Uma das funções de sistemas de gerenciamento é a alocação de *jobs* aos recursos disponíveis, chamado *Job Schedulers* (*JS*) [**?**]. *JSs* são tipicamente utilizados pelos sistemas de alto desempenho atuais (*clusters* de computadores). Eles funcionam como uma fila de acesso aos recursos do *cluster*. Quando um usuário deseja executar sua aplicação ele deve submetê-la a uma fila e definir a quantidade de recursos que ela precisa. Como geralmente a quantidade de recursos utilizados por uma aplicação varia durante sua execução, o usuário deverá definir a quantidade máxima de recursos que sua aplicação irá utilizar. Esses recursos ficarão alocados a esta aplicação até seu término. A aplicação irá aguardar até que seja a primeira da fila e a quantidade de recursos solicitada pelo usuário esteja disponível no ambiente. Todo esse controle (monitoramento e gerenciamento) é feito pelo *JS*, onde esse tipo de ambiente de execução é geralmente caracterizado pelo não compartilhamento simultâneo de recursos entre diferentes *jobs*.

Ao executar sua aplicação neste tipo de cenário, o usuário sabe que a chance de sua aplicação não encerrar sua execução no tempo esperado é pequena já que o ambiente é controlado e dedicado à aplicação. Entretanto, do ponto de vista do provedor de recursos e da fila de jobs, essa solução pode apresentar problemas como a subutilização de recursos e consequentemente, baixa escalabilidade.

A maioria das grades computacionais também adotou a utilização de *JSs*. Sendo

formado por uma grande quantidade de recursos de uma variedade de capacidades, é fácil ver que recursos também podem ser subutilizados neste cenário pelo fato das aplicações executarem de forma dedicada. Por exemplo, essa estratégia impede que uma aplicação utilize a CPU enquanto outra aplicação no mesmo recurso faz operações de entrada e saída. Como geralmente um recurso oferece vários serviços (*CPU*, *GPU*, E/S, etc.) é provável que a maioria fique ociosa enquanto a aplicação utiliza apenas um serviço em determinado período de tempo. Além disso, caso a primeira aplicação da fila necessite de $k+1$ recursos e haja apenas k recursos disponíveis, esses k recursos poderão ficar ociosos até que mais um seja liberado, ou dependendo da estratégia de escalonamento adotada pelo *JS*, a próxima aplicação da fila que precise de j recursos tal que $j \leq k$ poderá executar na frente das demais aumentando o tempo de espera na fila da primeira aplicação.

Conforme a demanda por processamento aumenta, maior será o número de aplicações disputando os recursos de processamento. Consequentemente, a fila do *JS* também irá crescer e o tempo total de espera desde a submissão da aplicação ao *JS* até o fim da execução também irá aumentar.

Além disso, um outro aspecto torna *JSs* não aplicáveis a grades computacionais. Este tipo de gerenciador supõe que os recursos são mantidos por uma única organização e que estes não estão sendo usados sem o seu controle. Assim, ele é capaz de escolher os melhores recursos para cada aplicação. Entretanto, em grades computacionais os recursos são distribuídos, o que implica que não há total controle sobre o estado dos mesmos. Assim, um recurso reservado a uma aplicação pelo *JS* poderá ser usado por um usuário local sem o conhecimento do gerenciador.

Uma alternativa para os problemas apresentados é o uso de gerenciadores capazes de monitorar cada aplicação independentemente. Assim, através da avaliação do estado da aplicação e do ambiente onde ela executa, o gerenciador poderá tomar as melhores ações, por exemplo reescalonamento de tarefas, para que a aplicação alcance seus objetivos. Dessa maneira, as aplicações se tornariam autônomas sendo cada uma responsável pelo seu próprio gerenciamento, não precisando de um coordenador central do ambiente. Esse gerenciamento centrado na aplicação traz benefícios como um maior conhecimento da aplicação e consequentemente maior precisão na tomada de ação. Além disso, o tempo de reação do gerenciador às mudanças do ambiente onde a aplicação executa será menor e a ação mais focada e específica a uma aplicação comparado com gerenciadores centralizados que olham para todas as aplicações ao mesmo tempo.

Uma outra vantagem da estratégia centrada na aplicação é que aplicações poderão

compartilhar recursos simultaneamente sem introduzir sobrecarga ao gerenciador. Quando *brokers* centralizados são utilizados como gerenciadores é complicado permitir esse compartilhamento simultâneo já que neste cenário o *broker* precisaria controlar todas as aplicações e possivelmente realizar balanceamentos de carga caso alguns recursos ficassem sobrecarregados. É nítido perceber que gerenciar o compartilhamento simultâneo de recursos por diversas aplicações na implementação centralizada do gerenciador seria muito custoso e provavelmente pouco eficiente.

Entretanto, apesar de permitir que a utilização de recursos seja maximizada, o compartilhamento simultâneo de recursos não controlado pode introduzir problemas no ambiente de execução como o aumento do *overhead* gerado pela troca de contexto das diversas aplicações que agora compartilham um mesmo recurso. Esse problema não aparecia com o uso de *brokers* centralizados porque estes tinham a visão do sistema como um todo e alocavam as aplicações a conjuntos não disjuntos de recursos. Já com a abordagem autônoma, a proliferação de aplicações, cada uma buscando minimizar seu tempo de execução de forma individualista, poderá tornar o sistema inviável.

Este problema ocorre com o *EasyGrid Application Management System* (*EasyGrid AMS*) [7]. O *EasyGrid AMS* é um sistema gerenciador de aplicações *MPI* [27] que é embutido na aplicação do usuário. Ele permite que aplicações que antes executavam em um ambiente cluster dedicado, executem no ambiente grade sem a necessidade de alterações significativas de seus códigos fontes. Esse *middleware* é baseado no conceito de computação autônoma. Assim, ele permite que as aplicações se auto monitorem e tomem decisões relativas a balanceamento de carga e tolerância a falhas automaticamente.

O objetivo do *EasyGrid AMS* é executar a aplicação do usuário no menor tempo possível considerando ambientes clássicos (dedicados ou oportunísticos) de grades com *JSs*. Resultados mostram que o *EasyGrid AMS* pode gerenciar eficientemente a execução de aplicações em ambientes dinâmicos e não previsíveis, conseguindo atingir *makespan* (tempo de execução da aplicação) perto do ótimo. Porém, no caso onde mais de uma aplicação *EasyGrid* estejam executando sobre conjuntos de recursos não disjuntos os resultados obtidos poderão ser ruins. Isso ocorre porque a estratégia atual do *EasyGrid* é gulosa e assim seu comportamento prejudica o ambiente. Cada aplicação pode ter políticas diferentes porém em geral aplicações que utilizam o *EasyGrid* objetivam sempre reduzir seu tempo de execução. Entretanto, será que usuários sempre desejam executar suas aplicações no menor tempo possível?

1.2 Objetivos

Quando usuários disparam aplicações no ambiente grade eles têm em mente diferentes objetivos. Para alguns o tempo de execução é fator crítico, outros admitem um tempo maior de execução, por exemplo, a fim de obter melhores resultados para sua heurística que resolve determinado problema de otimização, e outros até mesmo desconhecem o tempo ótimo de execução de suas aplicações. Além disso, por exemplo, projetos como [3] utilizam recursos somente quando estes estão ociosos.

O objetivo deste trabalho é investigar a viabilidade de criar um ambiente de execução sem *JS* central onde seria permitida compartilhamento simultâneo de recursos para maximizar a utilização dos recursos disponíveis da grade. Assim, busca-se modificar o comportamento guloso das aplicações *EasyGrid* e associar a cada aplicação, um comportamento próprio e mais sociável obtido a partir dos objetivos e expectativas do usuário que as disparou. Espera-se assim, reduzir conflitos (*overhead*) gerados pelo compartilhamento dos recursos. Neste trabalho tais objetivos e expectativas serão representados através de metas de execução e a grade vista como uma sociedade de aplicações autônomas.

1.3 Contribuições

Este trabalho apresenta uma proposta de gerenciamento descentralizado de aplicações para grades computacionais. Este gerenciamento centrado na aplicação permite que ela, de forma autônoma, escolha os melhores recursos para executar. Esta decisão será baseada em uma meta de execução (tempo de execução) definida pelo usuário ao disparar sua aplicação na grade. Essa meta introduzirá flexibilidade ao sistema e permitirá que aplicações tenham comportamentos diferentes. Assim, diversas aplicações que executam sobre os recursos da grade tomarão ações independentes baseadas em suas próprias metas, modificando efetivamente suas prioridades de acordo com a demanda e oferta de recursos. O objetivo dessa proposta é fazer com que uma sociedade de aplicações autônomas sustentável seja formada. É válido destacar que, como em qualquer sociedade, os elementos que a compõem podem ser altruístas, egoístas e até mesmo bipolares.

Este comportamento é formalizado através da heurística *Grid SA*. Essa heurística é baseada simplesmente em metas de execução (*soft deadlines*) que permitem que aplicações compartilhem recursos de uma forma mais eficiente. Uma outra contribuição é a implementação da heurística *Grid SA* no *EasyGrid* e a validação da mesma através de diversos

experimentos.

1.4 Organização

Esta dissertação está organizada da seguinte maneira: o capítulo 2 apresenta os trabalhos relacionados, o capítulo 3 avalia as estratégias de execução de aplicações em grade, o capítulo 4 apresenta uma proposta mais eficiente de compartilhamento de recursos e descreve a heurística *Grid SA*, o capítulo 5 faz uma análise experimental da heurística *Grid SA* e o capítulo 6 apresenta as conclusões e trabalhos futuros.

Capítulo 2

Trabalhos relacionados

O objetivo deste capítulo é apresentar os estudos que foram utilizados como fonte de pesquisa para o desenvolvimento da dissertação aqui apresentada. Estes foram fundamentais para o amadurecimento das idéias iniciais deste trabalho. Além disso, eles foram fonte de informações para a concepção de novas idéias, para a comparação de soluções propostas com soluções já existentes na literatura e para uma melhor compreensão de temas não ligados diretamente a este trabalho mas cujos conceitos puderam ser compreendidos, adaptados e aplicados. As próximas seções são brevemente descritas abaixo:

- Grades computacionais: são conjuntos de recursos distribuídos, heterogêneos e compartilhados conectados por redes de diferentes velocidades. Essa seção apresenta os objetivos desse ambiente e suas gerações;
- Computação autônoma: a complexidade dos sistemas computacionais atuais cresce tão rapidamente que o custo para instalá-los, configurá-los e mantê-los preocupa cientistas, pesquisadores e empresas. Computação autônoma é um novo paradigma que busca tornar sistemas autogerenciávies;
- Agentes: agentes são entidades computacionais autônomas que executam em um ambiente e possuem objetivos definidos. Essa seção apresenta o conceito e suas principais características;
- Sistemas gerenciadores de grades: em função de sua natureza instável, dinâmica e heterogênea, grades computacionais necessitam de sistemas gerenciadores que façam a interface entre os recursos do ambiente e as aplicações de usuários. Esta seção apresenta os tipos de sistemas gerenciadores e suas principais características;

- *EasyGrid Application Management System*: o *EasyGrid AMS* é o sistema gerenciador utilizado como base neste trabalho. Esta seção apresenta sua arquitetura e seu funcionamento.

2.1 Grades computacionais

Grades computacionais são formadas por conjuntos de recursos distribuídos geograficamente, tipicamente heterogêneos e compartilhados, conectados por redes de velocidades variadas. Este ambiente é 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 recursos suficientes localmente para suprir suas necessidades de poder computacional ou de armazenamento.

O termo grade computacional surgiu da analogia com as redes de interligação do sistema de energia elétrica (*power grids*), onde utiliza-se a eletricidade sem necessariamente saber a localização física de sua produção (hidroelétricas, termoelétricas, usinas nucleares, etc.), sendo sua utilização totalmente transparente aos seus usuários [17, 15]. A origem do nome retrata o objetivo de tornar o uso de recursos computacionais distribuídos tão simples quanto o uso da eletricidade.

Segundo [23] grades computacionais podem ser classificadas em 3 gerações. A primeira geração é caracterizada pelo compartilhamento dos recursos de um supercomputador, onde vários usuários se conectam por terminais e utilizam principalmente seu armazenamento. A segunda geração das grades é caracterizada pelo uso de *middlewares* que permitem a combinação de diferentes tecnologias para grades computacionais. A terceira geração é a combinação da tecnologia *Web* com a segunda geração e utiliza conceitos de *SOA (Service Oriented Architecture)* [12].

Contudo, a exploração eficiente do poder computacional nesse tipo de ambiente ainda é um desafio, 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, descoberta e agregação dinâmica de recursos e qualidade de serviço.

Diante de todos esses desafios, é necessário que a computação em grades 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,

sem que para isso o usuário tenha que se preocupar em tratar questões do ambiente. A seção 2.1.1 descreve as características principais da arquitetura grade, objetivando identificar os serviços mais importantes oferecidas por cada camada para garantir a execução da aplicação de forma eficiente e segura.

2.1.1 Arquitetura da grade

Uma característica marcante do ambiente grade é a sua diversidade de recursos. Além dos recursos de armazenamento e CPU, uma grade pode disponibilizar acesso a outros recursos e equipamentos especiais, tais como *softwares*, licenças e instrumentos científicos [11]. Com relação às características específicas de *hardware* e *software*, grades computacionais podem ser divididas em três níveis [18]: infraestrutura, *middleware* e aplicação. Esses níveis são apresentados na figura 2.1.

O primeiro nível corresponde à infraestrutura, 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 na grade. A padronização de protocolos facilita a implantação da transparência desejada na utilização de um ambiente complexo como esse. Para viabilizar a padronização das grades, *Foster et al.* [18] definiu 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 camada de *middleware* pode ser dividida nas camadas coletivos, recursos e conectividade [18]. 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 infraestrutura. 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

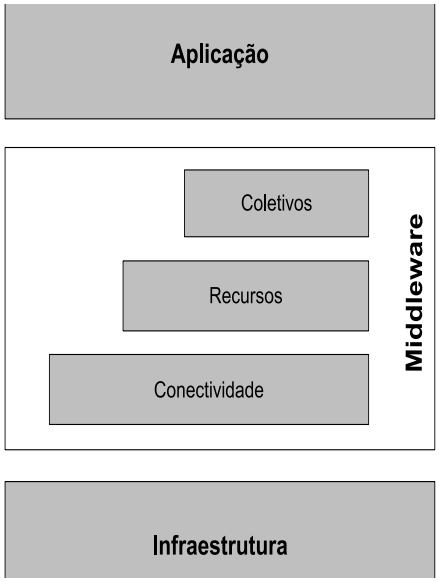

Figura 2.1: Detalhe da camada middleware da grade (adaptada de [18]).

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.

As maiores funcionalidades da arquitetura grade estão no nível de *middleware*. A fim de estruturar melhor os serviços oferecidos nesse nível, o mesmo foi subdividido em *middleware* básico e de serviços.

O *middleware* básico oferece os serviços fundamentais para a operação de uma grade, como o gerenciamento remoto de processos, a alocação de recursos, o registro e a coleta de informações, segurança e aspectos de qualidade de serviço. 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. 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.

Um problema relacionado à utilização de *middlewares* básicos e de serviço é o uso de diferentes tipos de infraestrutura na grade. Para que uma aplicação execute em um conjunto de recursos é preciso que *middlewares* básicos e de serviço estejam instalados em todos os recursos da camada de infraestrutura. Na maioria das vezes, os recursos da grade utilizam o mesmo *middleware básico*. Entretanto, diferentes *middlewares* de serviço são usados na grade. Assim, a aplicação do usuário precisará ser alterada para

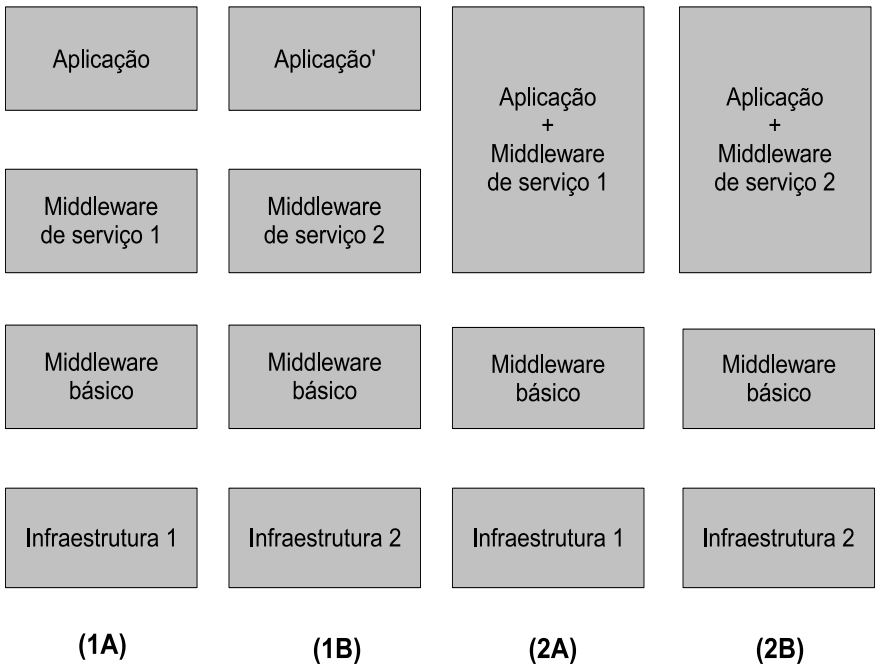

Figura 2.2: Visões *system centric* (1A e 1B) e *application centric* (2A e 2B)

executar em recursos com *middlewares* de serviço diferentes (visão *system centric*). Uma possível solução para este problema é a incorporação do *middleware* de serviço na própria aplicação (visão *application centric*). Assim, a aplicação poderá executar em diferentes tipos de infraestrutura sem a necessidade de adaptações.

Na figura 2.2 as colunas (1A) e (1B) mostram que quando o *middleware* de serviço está instalado na infraestrutura a aplicação precisará ser adaptada (aplicação alterada para aplicação'). Já as colunas (2A) e (2B) mostram que caso o *middleware* de serviço esteja acoplado à aplicação, esta não precisará ser alterada para executar em diferentes tipos de infraestrutura.

2.2 Computação autônoma

Atualmente é praticamente impossível encontrar alguma área de negócio ou ambiente onde não existem sistemas de computação. Pelo contrário, há um constante crescimento no uso de ambientes computacionais, impactando vários aspectos da vida humana. Entretanto, este crescimento desenfreado faz com que estratégias de planejamento, manutenção e gerenciamento antes concebidos não se apliquem mais.

Segundo [21] apesar da indústria de tecnologia da informação (TI) bater recordes com impressionante regularidade, ela agora se depara com um problema brotando a partir do núcleo de seu sucesso e apenas poucas pessoas estão focadas em solucioná-lo. Esse problema não está relacionado ao ritmo da lei de Moore [28], mas no tratamento das consequências do seu reinado de décadas. Ele também não está diretamente relacionado a quantos bits podem ser colocados em um quadrado de uma polegada e também não é a barreira da "máquina inteligente". O grande obstáculo é a complexidade. Lidar com a complexidade dos sistemas que estão sendo projetados é o desafio mais importante encarando a indústria de tecnologia da informação. Esse é o próximo grande desafio.

A indústria de TI continua a criar sistemas de computação altamente potentes. O objetivo desses sistemas é fazer indivíduos e negócios mais produtivos através da automação de tarefas e processos. Isso permite que pessoas foquem em problemas ainda não resolvidos. Entretanto, esse processo de automação também produz complexidade como um produto indesejado. A realidade é que o crescimento da complexidade da infraestrutura de tecnologia da informação ameaça enfraquecer os diversos benefícios que TI visa proporcionar. E até agora, somos dependentes principalmente da intervenção e administração humana para gerenciar essa complexidade.

A solução proposta por [21] é embutir a complexidade na própria infraestrutura do sistema (*hardware* e *software*) e automatizar seu gerenciamento. A inspiração desse modelo vem do sistema nervoso autônomo humano. Esse sistema controla a respiração, digestão, etc. sem a percepção e esforço do ser humano. Essa proposta foi chamada de Computação Autônoma [21, 32, 22, 34]. Assim, um Sistema de Computação Autônomo (SCA) deverá ter pelo menos 8 elementos chave. Seguem essas características:

1. *Self Awareness*: para ser autônomo um sistema de computação deve conhecer a si próprio, ou seja, deve conhecer seu estado e seu comportamento;

2. *Self Configuring*: um SCA deve se configurar e reconfigurar em condições variadas e imprevisíveis;

3. *Self Optimizing*: um SCA não deverá ficar parado ao alcançar um estado estável. Ele deverá sempre buscar maneiras de otimizar seus processos;

4. *Self-Healing*: um SCA deve ser capaz de "se curar". Ele deve ser capaz de se recuperar de eventos rotineiros ou extraordinários que podem causar mal funcionamento às suas partes;

5. *Self Protecting*: um mundo virtual não é menos perigoso que o real. Assim, um SCA deve ser capaz de se auto proteger de ataques internos e externos e manter a segurança e integridade do sistema;

6. *Context Aware*: um SCA conhece seu ambiente de execução e o contexto em que atua. Assim, ele deve agir de acordo com o cenário onde está;

7. *Open*: um SCA não pode existir em um ambiente hermético. Isso significa que ele deve operar em ambientes heterogêneos e implementar padrões abertos. Ele não pode ser uma solução proprietária;

8. *Anticipatory*: um SCA antecipará a necessidade de otimização de recursos e manterá essa complexidade escondida. Ele deve ser capaz de preencher o espaço que existe entre as necessidades dos usuários e as implementações de tecnologia da informação necessárias para alcançar seus objetivos.

Realisticamente, esses sistemas são bem difícies de construir e irão requisitar grande exploração de novas tecnologias e inovações. Esse é o motivo de Sistemas de Computação Autônomos serem vistos como um grande desafio por toda a indústria de tecnologia da informação.

2.3 Agentes

Agente é definido em [41] como um sistema de computador que se localiza em um ambiente e é capaz de executar ações autônomas a fim de alcançar objetivos definidos. A principal característica de um agente é seu comportamento "autônomo" que lhe permite tomar decisões com base em seus objetivos (metas) e informações de monitoramento do meio onde ele está executando.

Entretanto, o ambiente onde um determinado agente executa não está dedicado a ele. Assim, em um mesmo período de tempo, agentes irão compartilhar um meio e cada um irá buscar alcançar seus objetivos. Caso agentes tenham objetivos opostos eles poderão se comportar de uma maneira competitiva. O estudo da relação entre agentes é conhecido como Sistema multiagentes.

Sistema Multiagentes é definido por [40] como o problema de se construir um grupo de agentes. Um grupo de agentes é semelhante a uma sociedade humana onde pessoas são diferentes e possuem objetivos distintos. Assim como as pessoas, agentes podem se

agrupar para resolver problemas maiores. Entretanto, também existem alguns problemas em uma sociedade que devem ser resolvidos, por exemplo a falta de recursos para atender a todos e diversos interesses conflitantes.

De acordo com [38] o uso de agentes para a criação de sistemas é o próximo paradigma de desenvolvimento de *software*. Segundo os autores sistemas evoluem ao longo do tempo em função de três variáveis: código (o que o programa faz), dado (qual informação é processada) e invocação (quando a execução do código é acionada). Ao longo do tempo os paradigmas de desenvolvimento foram monolítico, estruturado, orientado a objetos e orientado a agentes. Em todos os paradigmas, exceto no monolítico, o código estava encapsulado, ou seja, acessado através de métodos. O dado só foi encapsulado a partir do paradigma orientado a objetos, onde os dados passaram a ser acessados através de métodos. Entretanto, apenas no paradigma orientado a agentes a invocação está encapsulada, ou seja, a chamada não é feita através de uma função principal (*main*).

Assim, um agente representa um "objeto ativo com iniciativa" , também chamado de "agente autônomo"para dar ênfase a não necessidade de acionamento externo e a presença de um conjunto de responsabilidades internas. Uma vez que um agente possui código, dado e controle, é esperado que o esforço de integração seja mínimo para a criação de aplicações. O sucesso da arquitetura de agentes dependerá da habilidade da população de agentes se organizar e se adaptar dinamicamente às mudanças sem a necessidade de um controle *top-down*.

Esse tipo de sistema aparece na natureza em populações de insetos e outros animais. Um estudo sobre esses sistemas é feito em [38] e a partir dele é derivado um conjunto de princípios de projeto que podem ser aplicados a sistemas baseados em agentes artificiais onde o comportamento da sociedade de agentes é muito mais complicado do que a complexidade de um agente individualmente. Abaixo é apresentada uma série de exemplos da natureza onde esse comportamento descentralizado funciona:

1. Formigas constroem caminhos entre o ninho e fontes de alimento. Apesar de não existir a aplicação de algoritmos de caminho mínimo neste processo, geralmente os caminhos criados são os menores possíveis.

2. Cupins constroem ninhos com mais de cinco metros de altura e dez toneladas sem um engenheiro chefe.

3. Vespas se dividem em grupos para buscar comida e grupos para cuidar dos filhotes. A quantidade de elementos de cada grupo é ajustada automaticamente de acordo

com a quantidade de comida disponível e o tamanho do ninho.

A partir da avaliação do comportamento desses sistemas naturais [38] derivou as principais características que um sistema multiagente deve ter:

1. Execução de funções através da iteração entre agentes: geralmente sistemas são divididos em funções. Esta abordagem é utilizada para sistemas onde existe uma função principal que invoca diferentes métodos. Em sistemas baseados em agentes, a invocação de métodos ocorrerá em função do cenário onde agentes executam e das iterações entre os mesmos;

2. Agentes devem ser pequenos: a ocorrência natural de sistemas adaptativos possui partes que são pequenas comparadas ao sistema inteiro. Essa motivação é derivada da experiência de engenheiros de software com a dificuldade de projetar, implementar e executar sistemas grandes. Pequenos agentes são mais fáceis de entender do que grandes sistemas monolíticos. Como eles são pequenos, caso um agente falhe ele poderá ser substituído por outro sem grande impacto.

3. Controle descentralizado: um agente central é o único ponto de falha que pode deixar o sistema vulnerável a um acidente. Em condições normais de operação ele pode se tornar um gargalo. Ele tende a atrair funcionalidades, destruindo o design descentralizado e as vantagens do uso de agentes.

4. Suportar a diversidade de agentes: a diversidade introduz diferentes comportamentos e evolução das soluções. Uma maneira de manter a diversidade é utilizar processos aleatórios.

5. Fornecer um canal para o vazamento da entropia: quanto maior a energia maior a entropia e consequentemente maior a desordem do sistema. Entretanto, em sistemas multiagentes a entropia ocorre em um nível micro que é a relação entre agentes e gera uma organização no nível macro. O sistema reduz a entropia no macro aumentando a entropia no nível micro. Por exemplo, quanto mais formigas se esbarram e se comunicam, mais rápido a melhor rota entre o ninho e o alimento será definida.

Por fim, [38] conclui que usuários ainda possuem resistência em descentralizar soluções e abandonar a abordagem *top-down*. Entretanto, os benefícios da abordagem de agentes são condicionais e não absolutos. Em um ambiente estável a abordagem centralizada pode ser otimizada e obter os melhores resultados. Entretanto, os ambientes atuais onde agentes

executam são dinâmicos e imprevisíveis (por exemplo, a *Web*). Assim, uma abordagem baseada em agentes provavelmente seria melhor.

2.4 Sistemas de gerenciamento de grades

Enquanto em ambientes de alto desempenho tradicionais usuários utilizam recursos exclusivamente, grades são baseadas no compartilhamento de recursos em larga escala [31]. Assim, a execução de uma aplicação em uma grade envolve uma série de serviços e camadas que podem introduzir uma sobrecarga considerável [13].

A heterogeneidade introduz complexidade na tarefa de alocar recursos a aplicações. Além disso, por serem dinâmicos e compartilhados, eles podem ter uma variação da capacidade de processamento durante a execução. Essas características tornam o ambiente grade complexo para a execução de programas. Em função de toda essa complexidade o acesso a grades computacionais vem sendo feito através de sistemas gerenciadores de grade (*middlewares*).

O objetivo de um sistema gerenciador de grade (SGG) é oferecer ao usuário uma visão simplificada deste ambiente. Dessa forma, busca-se simplificar o desenvolvimento da aplicação atribuindo ao SGG a responsabilidade de descobrir e acessar os recursos da grade, implementar mecanismos de tolerância a falhas e mecanismos de escalonamento, submeter processos, entre outras atividades.

Sistemas gerenciadores oferecem serviços como distribuição de arquivos [4], escalonamento de tarefas [5, 30], tolerância a falhas [10] e avaliação de desempenho [9]. Uma proposta de classificação desses sistemas é apresentada em [29].

- Sistemas Gerenciadores de Recursos (SGRs): o objetivo é maximizar a utilização dos recursos entre várias aplicações que compartilham a grade;
- Sistemas Gerenciadores de Usuários (SGUs): é responsável por gerenciar as aplicações de um único usuário por vez de acordo com seus requisitos;
- Sistemas Gerenciadores de Aplicações (SGAs): possui propriedades semelhantes a um SGU com a diferença de serem focados na otimização da execução de sua aplicação. Se o SGA está integrado a aplicação do usuário, a aplicação é transformada em uma versão *system aware*.

O principal problema do uso de um SGR é a escalabilidade, já que a definição de

um gerenciador centralizado possivelmente irá falhar quando a quantidade de recursos crescer criticamente. Além disso, por não estar próximo às aplicações ele poderá tomar decisões de gerenciamento (por exemplo, balanceamento de carga) genéricas e muita vezes independentes das características das aplicações. Assim, fica difícil e complexo executar ações certas dada a variedade das aplicações colocadas para executar na grade. Os SGUs apesar de serem menos centralizadores do que os SGRs também enfrentam problemas quando falamos de uma quantidade crescente de recursos dinâmicos e instáveis. Por exemplo, será extremamente difícil um SGU trabalhar com grades *ad hoc* [20, 36] onde recursos se aproximam ocasionalmente estabelecendo uma rede temporária. Já um SGA pode ser embutido em aplicações de usuários ao invés de instalado nos recursos da grade, permitindo melhor portabilidade e resolvendo o problema de escalabilidade já que torna a aplicação autogerenciável. Por ser otimizado para cada aplicação, um SGA pode executar ações mais eficientes. Este trabalho é baseado no SGA *EasyGrid*.

2.5 EasyGrid Application Management System (EasyGrid AMS)

Os temas estudados nas seções anteriores mostraram que um sistema gerenciador de aplicações é uma boa implementação de gerenciador para grade pois consegue obter características da aplicação onde ele executa e consequentemente executar as melhores ações (seção 2.4). Além disso, sendo embutido na própria aplicação esse gerenciador permite que a aplicação execute ações de forma autônoma a partir de informações coletadas do ambiente onde ela executa (seções 2.2 e 2.3).

O *EasyGrid AMS* [7] é um *middleware* para a execução de aplicações *MPI* em grades computacionais. Seu objetivo é encapsular aplicações *MPI* que originalmente eram projetadas e executadas em *clusters* de computadores e torná-las autônomas, isto é auto-adaptativas, eficientes e robustas para serem executas em grades computacionais sem sobrecarregar o desenvolvedor com o esforço de gridificação.

O *EasyGrid AMS* utiliza uma arquitetura hierárquica distribuída de processos gerenciadores (figura 2.3). No topo da hierarquia há o Gerenciador Global (*Global Manager - GM*) responsável por gerenciar os sites pertencentes a grade. Abaixo deste há os Gerenciadores de Site (*Site Manager - SM*) sendo cada um responsável pelo gerenciamento dos recursos dentro de um site ou conjunto de recursos específico. Por último, existem os Gerenciadores de Máquina (*Host Manager - HM*), um por recurso, que são respon-

sáveis pelo gerenciamento dos processos da aplicação do usuário que estão alocados em um recurso específico.

Cada processo gerenciador do *EasyGrid AMS* é estruturado em camadas como apresentado na figura 2.4. O comportamento de cada camada depende do nível do processo de gerenciamento na estrutura hierárquica (HM se comporta diferente do SM, e SM diferente do GM). Essa estrutura permite que a aplicação se adapte apropriadamente e eficientemente 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 de acordo com suas necessidades particulares.

A camada de gerenciamento de processos é responsável pela criação dinâmica de processos *MPI* (processos da aplicação e gerenciadores) e pelo redirecionamento de mensagens, sendo que a alocação de processos aos recursos pode mudar dinamicamente e também mensagens podem ser encaminhadas para um processo que ainda não foi criado. A coleta de informações relacionadas à execução de uma aplicação *MPI*, por exemplo tempo de execução da última tarefa da aplicação, é realizada pela camada de monitoramento da aplicação. Essas informações alimentam as camadas de escalonamento dinâmico, responsável pela redistribuição de tarefas da aplicação, e a camada de tolerância a falhas, responsável pela identificaçào e tratamento de falhas. O *middleware* é projetado tal que cada camada superior pode modificar o comportamento da camada imediatamente inferior. No topo da estrutura em camadas do *EasyGrid AMS* está a aplicação *MPI* do usuário e a camada de abstração *MPI* (wrapper) que permite que o *EasyGrid AMS* seja embutido na aplicação do usuário. Esta funcionalidade está escondida dos processos da aplicação do usuário através da camada de abstração na forma de funções *MPI* modificadas (*MPI wrappers*).

Como o *EasyGrid* é um *application management system* ele é capaz de escolher políticas de execução com o objetivo de otimizar o tempo de execução da aplicação. A versão atual do EasyGrid trabalha com aplicações *BoT* (*Bag of Tasks*) [29], que são aplicações paralelas cujas tarefas são independentes, aplicações *DAG* (*Direct Acyclic Graph*) [33], onde as tarefas possuem relações de precedência e aplicações não determinísticas [11], onde o número de tarefas da aplicação é desconhecido.

Resultados já mostram que a execução de aplicações *EasyGrid* obtem bons resultados inclusive na presença de aplicações concorrentes dinâmicas. Um dos objetivos deste trabalho é avaliar a execução de mais de uma aplicação *EasyGrid* sobre um mesmo conjunto ou subconjunto de recursos ao mesmo tempo. Durante a execução de aplicações parale-

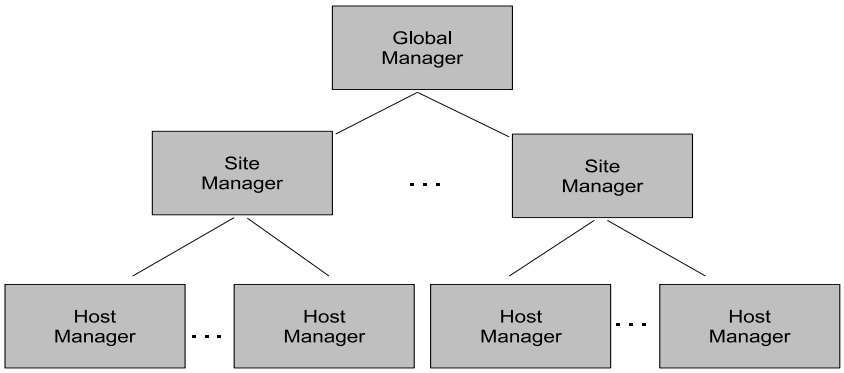

Figura 2.3: Hierarquia dos processos gerenciadores do *EasyGrid*

las, especialmente aplicações *DAG*, raramente recursos têm suas capacidades máximas utilizadas durante todo o tempo de execução. O objetivo desta dissertação é maximizar o uso dos recursos através do compartilhamento dos mesmos entre mais de uma aplicação. Entretanto, fazer esta demonstração com aplicações *BoT* pode se tornar uma tarefa complicada já que pelas suas características aplicações *BoT* sempre irão utilizar todos os recursos disponíveis de tal forma que dificilmente recursos serão subutilizados. O mesmo não ocorre com aplicações *DAG*, onde as relações de dependência poderão fazer com que um recurso não execute sua tarefa e fique ocioso aguardado a dependência de uma tarefa em outro recurso acabar. Dessa forma, uma vez que o objetivo deste trabalho é mostrar que é possível maximizar o uso de um recurso através de seu compartilhamento simultâneo por mais de uma aplicação, o uso de aplicações do tipo *DAG* cujas tarefas podem ser distribuídas independentemente pelos recursos caracteriza o pior caso para a realização desses experimentos. Assim, optou-se por aplicações *BoT* para a execução dos experimentos desse trabalho.

Em função da heterogeneidade e dinamismo do ambiente grade, o *EasyGrid AMS* implementa um mecanismo de escalonamento dinâmico de tarefas [29]. A estratégia adotada pelo *EasyGrid* é baseada em um modelo híbrido, ou seja, utiliza inicialmente políticas de escalonamento estático e depois, durante a execução, políticas de escalonamento dinâmico. O objetivo do escalonamento estático é definir uma alocação inicial das tarefas da aplicação do usuário no conjunto de recursos definidos. Esta distribuição é feita com base no estado das máquinas antes da submissão da aplicação e nas características da aplicação. Uma vez que o ambiente grade é instável, a visão do escalonamento estático pode não ser mais verdadeira logo após o início da execução. Para isso, o escalonamento dinâmico é

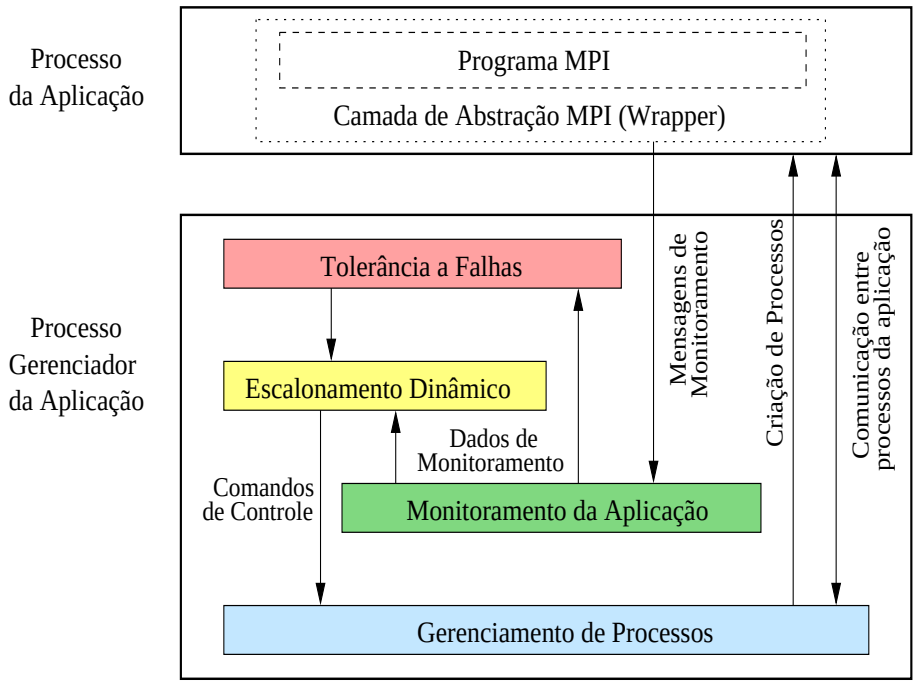

Figura 2.4: Estrutura de camadas do *EasyGrid*

usado, sendo responsável por realizar possíves redistribuições das tarefas pelos recursos em tempo de execução.

A estratégia de escalonamento dinâmico é implementada em 3 níveis: Escalonador Dinâmico Global (*Global Dynamic Scheduler - GDS*), Escalonador Dinâmico de Site (*Site Dynamic Scheduler - SDS*) e Escalonador Dinâmico de Máquina (*Host Dynamic Scheduler - HDS*). O objetivo geral dos escalonadores é estimar o tempo restante de execução da aplicação em cada recurso e tentar reduzir esse tempo final através de eventos de realocação de tarefas entre os recursos. A figura 2.5 apresenta um exemplo de grade computacional onde é possível identificar os escalonadores citados.

O *GDS* é o escalonador mais alto da hierarquia e tem a visão de todos os demais escalonadores. Ele é responsável por identificar possíveis desbalanceamentos entre os sites e redistribuir carga entre os mesmos. O *SDS* é responsável por realizar a redistribuição de tarefas dentro do site. O objetivo desta redistribuição é balancear a carga interna do site de forma a não sobrecarregar algum recurso.

O *HDS* é responsável por definir o número máximo de processos concorrentes que poderão executar em uma máquina e quando esses processos serão criados. Atualmente existem duas políticas de execução que definem quando uma tarefa poderá ser executada: Política de Execução Restrita (PER) e Política de Execução Irrestrita (PEI). A PER define que uma tarefa da aplicação só poderá ser executada caso o recurso esteja ocioso.

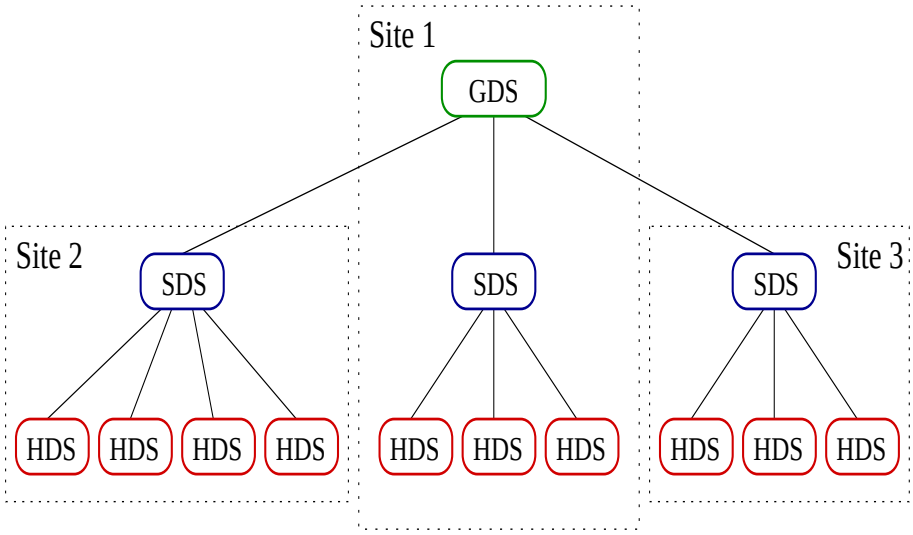

Figura 2.5: Exemplo de grid e processos gerenciadores do *EasyGrid AMS*

Já a PEI indica que tarefas poderão ser criadas independente da carga do recurso. Estas escolhas deverão ser feitas com base nas políticas de acesso e uso estabelecidas pelo dono do recurso onde o HM irá executar.

2.6 Resumo

Em função do modelo de execução adotado e do poder do *EasyGrid AMS*, aplicações *MPI*, tipicamente *moldable* (aplicações que podem rodar em vários números de processos, mas o número permanece fixo durante toda a execução), se tranformam em aplicações *evolving* ou *malleable* [26] (aplicações que podem variar o número de processos durante a execução). Isso permite que essas aplicações executem em um número variável de recursos e com poder computacional também variável. Entretanto, apesar do *EasyGrid AMS* (seções 2.4 e 2.5) permitir que aplicações *MPI* executem em grades computacionais (seção 2.1) de forma autônoma (seção 2.2), sua versão atual não implementa estratégias eficientes para que diversas aplicações compartilhem recursos (comportamento semelhante a sistemas multiagentes - seção 2.3). Isso ocorre porque o objetivo de cada *EasyGrid AMS* é fazer com que sua aplicação execute no menor tempo possível (comportamento egoísta). Assim, a partir do momento que diversas aplicações compartilham recursos simultaneamente e possuem esse mesmo comportamento, o ambiente onde elas executam poderá se tornar insustentável. O próximo capítulo apresentará as estratégias de compartilhamento de recursos para grades computacionais e fará uma avaliação das mesmas, apontando as vantagens e desvantagens de cada abordagem.

Capítulo 3

A execução de aplicações em grades computacionais

O modelo de execução adotado pelas maioria das grades computacionais atuais considera que para submeter aplicações usuários precisam definir a quantidade de recursos que suas aplicações irão utilizar. Esse conjunto de recursos ficará alocado exclusivamente a cada aplicação até que estas concluam a execução. Nesse cenário, o trabalho do *broker* responsável pelo escalonamento das aplicações nos recursos é verificar se a quantidade de recursos disponível no ambiente é maior ou igual a quantidade de recursos solicitada pelo usuário. Caso seja maior ou igual, esse conjunto de recursos será alocado a esta aplicação. Caso contrário, a aplicação deverá aguardar até que a quantidade mínima de recursos desejada esteja disponível. Apesar de simplificar o processo de escalonamento, esta estratégia poderá aumentar o tempo de espera de aplicações que necessitam de uma grande quantidade de recursos. Além disso, recursos poderão ficar alocados e ociosos já que geralmente a quantidade de recursos utilizados por aplicações pode variar durante a execução e usuários sempre irão requisitar a quantidade necessária para minimizar o tempo de execução.

O modelo de execução do *EasyGrid AMS* admite que aplicações compartilhem recursos simultaneamente. Assim, mais de uma aplicação poderá utilizar um mesmo recurso ao mesmo tempo. Nesta abordagem, aplicações não precisam aguardar que o conjunto de recursos desejado fique disponível. Sendo maleáveis [26], aplicações podem aproveitar as capacidades que estão disponíveis dos recursos enquanto esperam a quantidade ideal de recursos. Além disso, como o *EasyGrid* é um *AMS*, não há necessidade de um escalonador centralizado responsável por avaliar todos os recursos disponíveis e todas as aplicações da fila de espera. Cada aplicação autonomamente irá se ajustar aos recursos do ambiente

de acordo com sua própria necessidade. Entretanto, apesar dessas vantagens, o compartilhamento de recursos adotado pelo *EasyGrid* pode introduzir problemas, como por exemplo o aumento da perda de eficiência em função da troca de contexto.

O objetivo desse capítulo é avaliar as diferentes estratégias de alocação de recursos para aplicações que desejam executar na grade e definir vantagens e desvantagens de cada uma. A seção 3.1 apresenta uma avaliação conceitual das abordagens de alocação e a seção 3.2 faz uma avaliação experimental.

3.1 Estratégias de compartilhamento de recursos

O objetivo de uma grade computacional é oferecer um ambiente de alto desempenho a usuários, empresas e universidades que não tem recursos financeiros para adquirir e manter ambientes desse porte a um baixo custo [1, 2]. Isso é possível através da colaboração entre donos dos recursos com o objetivo de permitir que recursos sejam agregados e compartilhados pelos clientes de todos os demais.

Assim, conforme a popularidade e a quantidade de usuários da grade aumenta, maneiras mais eficientes de alocar recursos para as aplicações que irão executar nesse ambiente devem ser estudadas. Neste trabalho, as abordagens de distribuição das aplicações dos usuários pelos recursos da grade são chamadas de estratégias de compartilhamento de recursos. Basicamente, existem duas estratégias de compartilhamento de recursos: vertical e horizontal.

3.1.1 Compartilhamento de recursos vertical

No compartilhamento de recursos vertical o escalonador da grade irá alocar apenas recursos ociosos para as aplicações submetidas. Ao submeter sua aplicação um usuário deverá informar o número máximo de recursos que esta irá utilizar. Com essa informação, o *broker* gerenciador da grade buscará a quantidade solicitada de recursos ociosos e os alocará exclusivamente a este usuário até que sua aplicação conclua a execução. Caso a quantidade de recursos disponíveis (ociosos) na grade seja menor do que a desejada, o *broker* poderá adotar políticas de escalonamento, por exemplo, escolher a primeira aplicação da fila de submissão cuja quantidade de recursos solicitada seja menor ou igual a quantidade disponível no ambiente. A maioria dos ambientes grade em produção atualmente adota o compartilhamento de recursos vertical.

Tempo/Recurso	R_1	R_2	...	$R_{n/2}$	$R_{n/2+1}$	$R_{n/2+2}$	...	R_n
t_a	ap_1	ap_1	...	ap_1	ap_2	ap_2	...	ap_2
t_b			...		ap_2	ap_2	...	ap_2
t_c	ap_3	ap_3	...	ap_3	ap_3		...	

Tabela 3.1: Exemplo de compartilhamento de recursos vertical

A tabela 3.1 apresenta um exemplo de compartilhamento de recursos vertical. Seu objetivo é mostrar quais aplicações executaram em cada recurso ao longo do tempo. No exemplo, t_a, t_b e t_c representam períodos de tempo distintos, e R_i representa um determinado recurso da grade. No cenário ilustrado a grade possui n recursos (R_1 até R_n). As demais células da tabela são preenchidas por ap_1, ap_2 e ap_3, onde ap_i representa uma aplicação que foi submetida a grade e que aguardou na fila de submissão até que a quantidade de recursos desejada ficasse disponível e foi alocada a um determinado conjunto de recursos.

Neste exemplo (tabela 3.1), os conjuntos de recursos $\{R_1, ..., R_{n/2}\}$ e $\{R_{n/2+1}, ..., R_n\}$ foram alocados às aplicações ap_1 e ap_2 respectivamente no início da execução. No tempo t_b o conjunto de recursos $\{R_1, ..., R_{n/2}\}$ foi liberado porque a aplicação ap_1 concluiu sua execução e a aplicação ap_2 continuou sua execução no conjunto de recursos $\{R_{n/2+1}, ..., R_n\}$. Ainda no tempo t_b a aplicação ap_3 estava pronta na fila de execução aguardando que $n/2 + 1$ recursos estivessem disponíveis. Entretanto, esta aplicação não pode iniciar sua execução porque só havia disponíveis $n/2$ recursos ociosos. No tempo t_c a aplicação ap_2 concluiu sua execução e liberou seu conjunto de recursos. Nesse momento, a aplicação ap_3 teve seus $n/2 + 1$ recursos ociosos disponíveis e pode iniciar sua execução.

O exemplo mostrou que o processo de alocação de recursos do compartilhamento vertical é bastante simples. Além disso, como uma única aplicação utiliza um recurso em um período de tempo, a aplicação não sofre com atraso de execução por causa da concorrência de recursos. Entretanto, o exemplo também mostrou que recursos podem ser subutilizados quando no tempo t_b, ap_3 não iniciou sua execução porque precisava de $n/2+1$ recursos e só havia $n/2$ recursos disponíveis, caso não haja outra aplicação esperando por uma quantidade menor de recursos. Como é o usuário que define o número de recursos desejado, muitas vezes esse número pode ser maior do que o realmente necessário e recursos podem ficar alocados e não serem utilizados. Além disso, geralmente o número de recursos que uma aplicação utiliza varia ao longo de sua execução. Entretanto, esses recursos não utilizados durante todo o tempo de execução permanecerão alocados até o término da aplicação.

Um outro ponto importante é que este modelo não considera que recursos da grade podem ser utilizados por usuários locais. Assim, apesar do escalonador da grade acreditar que um determinado recurso está ocioso, uma vez que ele não alocou este para nenhuma outra aplicação da grade, um usuário local poderá utilizar este recurso no mesmo instante. Essa estratégia funciona bem em ambientes controlados e dedicados como *clusters* de computadores, mas apresenta problemas quando utilizada no ambiente grade.

Em função da necessidade de visualização de todas as aplicações da grade, o gerenciador para o compartilhamento vertical será tipicamente centralizado (*Resource Management System*). Isso é um problema já que uma falha neste ponto inviabilizará o uso da grade.

3.1.2 Compartilhamento de recursos horizontal

No compartilhamento de recursos horizontal, o escalonador da grade irá distribuir aplicações pelos recursos da grade independentemente se estes estão ociosos ou não. Assim, caso um *broker* gerenciador esteja sendo usado, quando um usuário submeter uma aplicação à grade, o *broker* escolherá os melhores recursos para a aplicação e iniciará sua execução.

Como nesta abordagem há o compartilhamento simultâneo de recursos, a variação das cargas das máquinas será maior do que na abordagem vertical (variação ocorre apenas em função de usuários locais). Entretanto, caso a implementação do gerenciador da grade utilize um *broker* (*Resource Management System - RMS*) a eficiência do processo de balanceamento de carga será comprometida já que o processo será centralizado. Assim, as soluções de gerenciadores para ambientes que adotam a abordagem horizonal se adequam melhor às implementações de *Application Management Systems - AMS*. No caso onde esses sistemas estão embutidos na aplicação, o processo de escolha dos melhores recursos será feita por cada aplicação autonomamente. Além disso, por estar acoplado à aplicação, o sistema gerenciador poderá obter informações das características desta e fazer uma melhor alocação. O *EasyGrid AMS* pode utilizar tanto o compartilhamento de recursos horizontal quanto o vertical. A tabela 3.2 apresenta um exemplo deste tipo de compartilhamento.

A estrutura da tabela 3.2 é semelhante a tabela 3.1 descrita na seção 3.1.1. Os conjuntos de recursos $\{R_1, ..., R_{n/2}\}$ e $\{R_{n/2+1}, ..., R_n\}$ foram alocados às aplicações ap_1 e ap_2 respectivamente no início da execução. No tempo t_d o conjunto de recursos $\{R_1, ..., R_{n/2}\}$ foi liberado porque a aplicação ap_1 concluiu sua execução e a aplicação ap_2

Tempo/Recurso	R_1	R_2	...	$R_{n/2}$	$R_{n/2+1}$	$R_{n/2+2}$	...	R_n
t_a	ap_1	ap_1	...	ap_1	ap_2	ap_2	...	ap_2
t_d	ap_3	ap_3	...	ap_3	ap_2/ap_3	ap_2	...	ap_2
t_e			...				...	

Tabela 3.2: Exemplo de compartilhamento de recursos horizontal

continuou sua execução no conjunto de recursos $\{R_{n/2+1}, ..., R_n\}$. Ainda no tempo t_d a aplicação ap_3 foi submetida e $n/2+1$ recursos foram alocados a ela. Repare que apesar de não existir no instante t_d $n/2+1$ recursos ociosos, ap_3 iniciou sua execução. No instante t_e as aplicações ap_2 e ap_3 acabaram e todos os recursos da grade ficaram disponíveis.

Diferentemente do exemplo da abordagem vertical (tabela 3.1), neste exemplo (tabela 3.2) o conjunto de recursos $\{R_1, ..., R_{n/2}\}$ não foi subutilizado durante o tempo t_b. Além disso, conforme a necessidade de recursos de uma aplicação e a carga das máquinas varia durante a execução, o gerenciador embutido na própria aplicação é capaz de fazer o ajuste da aplicação nos recursos do ambiente. Porém, um ponto de atenção desta abordagem é o aumento da sobrecarga por troca de contexto em função do compartilhamento simultâneo de recursos.

3.2 Avaliação experimental das estratégias de compartilhamento de recursos

O objetivo desta seção é apresentar uma análise experimental das estratégias de compartilhamento de recursos vertical e horizontal. Para a realização desses experimentos foram utilizadas duas aplicações *BoT* idênticas com 60 tarefas independentes. Com o objetivo de avaliar o efeito da granularidade das tarefas no resultado dos experimentos, o peso das mesmas variou entre 1, 5 e 10 segundos. Além disso, cada aplicação executou um processo por recurso. As aplicações foram executadas em 4 recursos monoprocessados idênticos conectados a um *Switch Gigabit*.

A tabela 3.3 apresenta a configuração dos experimentos. O objetivo desta tabela é descrever como as aplicações foram distribuídas pelos recursos em cada experimento. A coluna *Exp* identifica o experimento, a coluna *Estratégia de compartilhamento* indica se o experimento está utilizando a estratégia de compartilhamento de recursos vertical ou horizontal e as colunas R_i indicam quais aplicações estão executando em cada recurso R_i, $1 \leq i \leq 4$. Nesta tabela as aplicações são identificadas por ap_1 e ap_2.

Exp	Estratégia de compartilhamento	R_1	R_2	R_3	R_4
1	-	ap_1			
2	-	ap_1	ap_1	ap_1	ap_1
3	vertical	ap_1	ap_1	ap_2	ap_2
4	horizontal	ap_1/ap_2	ap_1/ap_2	ap_1/ap_2	ap_1/ap_2

Tabela 3.3: Configuracao dos experimentos

Nos experimentos 1 e 2 apenas a aplicação ap_1 foi executada. O objetivo destes experimentos foi avaliar o tempo de execução de uma única aplicação quando os recursos estão dedicados a ela. Já nos experimentos 3 e 4 o objetivo foi avaliar o tempo de execução de aplicações disputando os recursos disponíveis. Assim, nos experimentos 3 e 4, os 4 recursos disponíveis no ambiente foram compartilhados pelas aplicações ap_1 e ap_2 segundo as estratégias de compartilhamento vertical e horizontal respectivamente.

A tabela 3.4 apresenta os resultados dos experimentos. A coluna *Exp* identifica o experimento, a coluna *Peso tarefa* indica o peso das tarefas (em segundos) da aplicação do usuário em cada experimento (1, 5 e 10 segundos cada tarefa). As colunas $t(ap_i)$ indicam o tempo de execução (em segundos) da aplicação ap_i, onde $1 \leq i \leq 2$. As colunas $esp(ap_i)$ indicam o tempo de execução esperado (em segundos) de ap_i. As colunas $t(ap_i)/esp(ap_i)$ indicam a relação entre o tempo de execução e o tempo de execução esperado das aplicações ap_i.

Os valores de tempo esperado de execução foram calculados com base na quantidade de tarefas de cada aplicação (60), no peso das tarefas (1, 5 e 10) e na quantidade de recursos utilizados. Por exemplo, no experimento 2 quando o peso da tarefa foi 5 segundos e 4 recursos dedicados foram utilizados o tempo esperado foi calculado por (número de tarefas * peso da tarefa)/(número de recursos)= $(60 * 5)/(4) = 75$. Já no experimento 4, como os recursos foram compartilhados por 2 aplicações e cada aplicação tinha apenas 50% de cada recurso a fórmula utilizada para o cálculo do tempo esperado foi (número de tarefas * peso da tarefa)/(número de recursos * 0,5).

Os resultados dos experimentos 1 e 2 mostram que a sobrecarga introduzida pelo uso do *EasyGrid AMS* é de aproximadamente 6%. O resultado do experimento 2 utilizando tarefas de 1 segundo obteve uma sobrecarga alta de 12%. Provavelmente este resultado foi obtido em função de algum evento na rede ou nos recursos. Ainda comparando os experimentos 1 e 2 é possível perceber que os resultados de 1 foram um pouco melhores que 2 quando comparamos as relações entre o tempo de execução e o tempo esperado. Isso ocorreu porque em 1 a quantidade de recursos utilizada foi menor que em 2 e conse-

Exp	Peso tarefa	t(ap_1)	t(ap_2)	esp(ap_1)	esp(ap_2)	t(ap_1)/esp(ap_1)	t(ap_2)/esp(ap_2)
1	1	63,39	-	60	-	1,06	-
1	5	315,13	-	300	-	1,05	-
1	10	627,09	-	600	-	1,05	-
2	1	16,78	-	15	-	1,12	-
2	5	79,64	-	75	-	1,06	-
2	10	158,73	-	150	-	1,06	-
3	1	32,17	32,18	30	30	1,07	1,07
3	5	158,49	157,95	150	150	1,06	1,05
3	10	315,47	315,93	300	300	1,05	1,05
4	1	31,80	32,73	30	30	1,06	1,09
4	5	159,57	159,25	150	150	1,06	1,06
4	10	318,77	317,58	300	300	1,06	1,06

Tabela 3.4: Resultados dos experimentos

quentemente o custo pela inicialização e finalização dos processos também foi menor.

O objetivo dos experimentos 3 e 4 foi comparar as estratégias de compartilhamento de recursos vertical e horizontal. Os resultados mostraram que o compartilhamento vertical obteve em média uma sobrecarga de 5% enquanto que o horizontal foi 6%. A sobrecarga da estratégia horizontal foi um pouco maior porque nesta há perda em função da troca de contexto dos processos das aplicações. Entretanto, essa sobrecarga não comprometeu muito a qualidade dos resultados.

Com o objetivo de avaliar o impacto da perda introduzida pela troca de contexto na estratégia horizontal foram realizados experimentos com uma única aplicação BoT com 50 tarefas e um único recurso monoprocessado. O peso das tarefas também variou entre 1, 5 e 10 segundos. Em cada experimento o número de processos da aplicação que executavam no recurso era incrementado.

As figuras 3.1, 3.2 e 3.3 apresentam os resultados dos experimentos. As três figuras mostram que conforme o número de processos concorrentes em um mesmo recurso aumenta maior o tempo de execução da aplicação. Em especial, a figura 3.1 mostrou um resultado interessante quando o número de processos aumentou de 1 para 2. O resultado obtido com 2 processos executando foi melhor do que 1 processo porque, quando o peso da tarefa é 1 segundo, este é tão pequeno que o custo de preparação de um novo processo após a conclusão de uma tarefa é maior do que o custo de troca de contexto caso 2 processos sejam colocados em execução ao mesmo tempo.

Em função da comparação teórica e empírica realizada entre as estratégias de compartilhamento vertical e horizontal é possível perceber que ambas possuem vantagens e

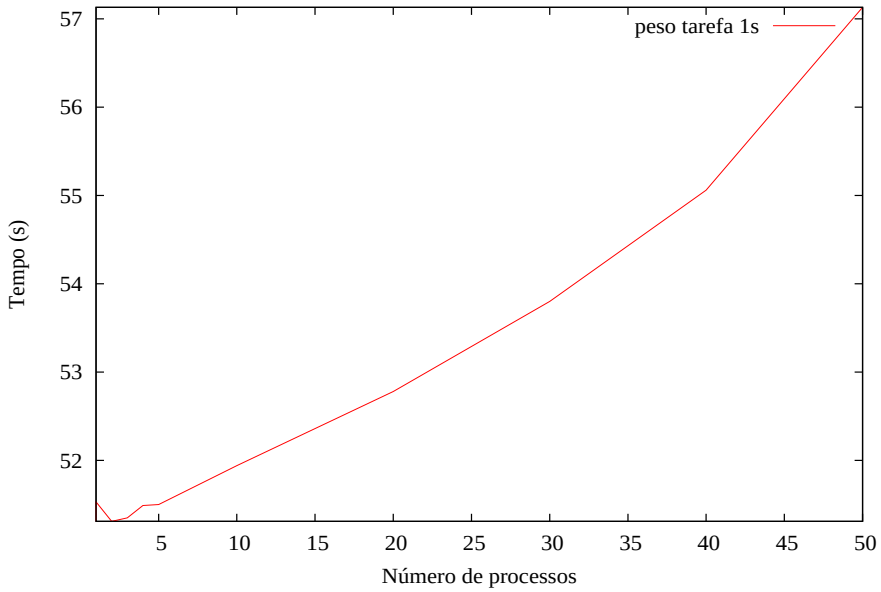

Figura 3.1: Experimentos com tarefas de 1 segundo. O gráfico mostra o número de processos sendo criados simultaneamente/concorrentemente pelo tempo de execução.

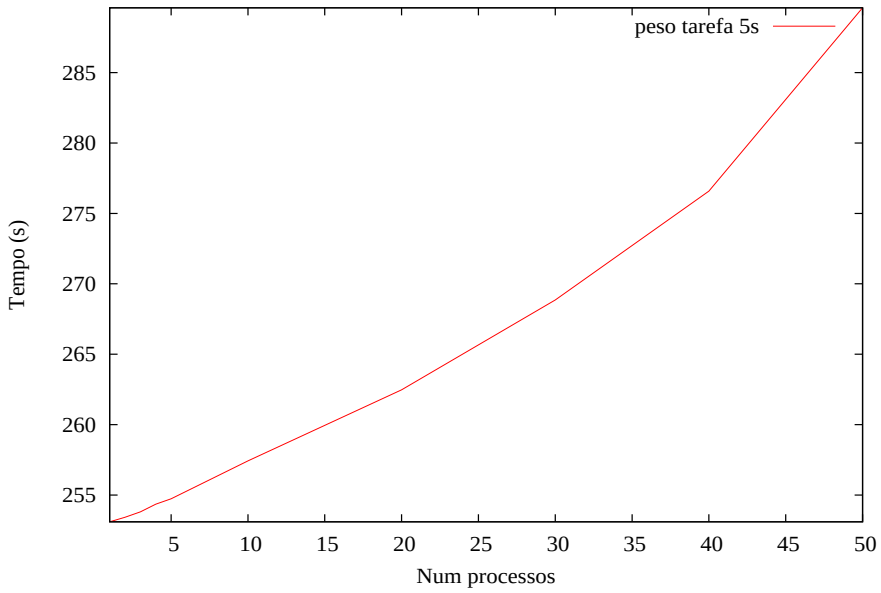

Figura 3.2: Experimentos com tarefas de 5 segundos. O gráfico mostra o número de processos sendo criados simultaneamente/concorrentemente pelo tempo de execução.

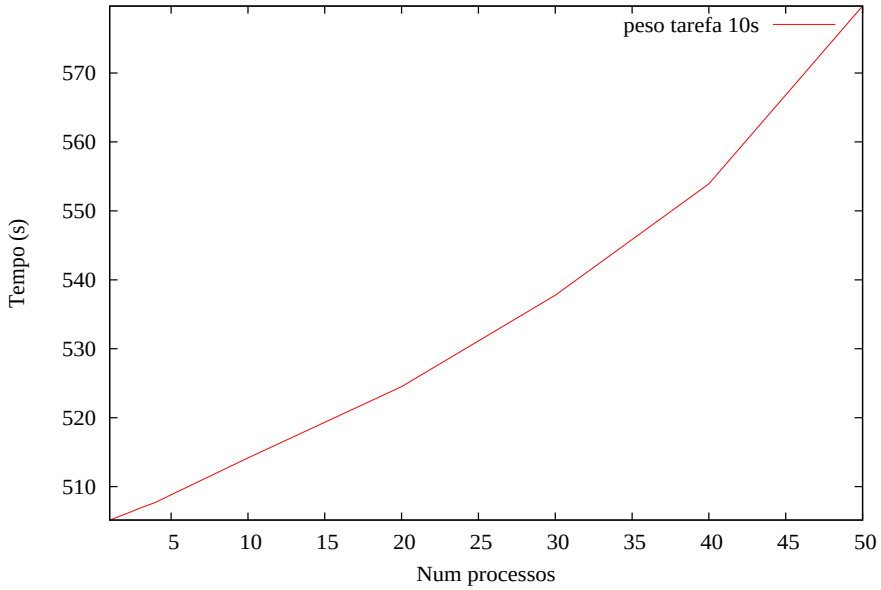

Figura 3.3: Experimentos com tarefas de 10 segundos. O gráfico mostra o número de processos sendo criados simultaneamente/concorrentemente pelo tempo de execução.

desvantagens. A estratégia vertical oferece um gerenciamento simplificado e elimina a perda de eficiência em função da troca de contexto já que aplicações não compartilham recursos simultaneamente. Entretanto, como aplicações podem não utilizar um recurso durante todo o período que este foi alocado e usuários podem superestimar a quantidade de recursos necessária, recursos podem ser subutilizados durante a execução de aplicações nesta abordagem. Por outro lado, na estratégia horizontal, aplicações tendem a maximizar o uso de recursos através do compartilhamento simultâneo dos mesmos. Além disso, não é preciso que um gerenciador central escolha em quais recursos cada aplicação irá executar já que elas podem se ajustar automaticamente aos recursos disponíveis de acordo com suas necessidades. Por outro lado, como nesta abordagem há o compartilhamento simultâneo de recursos, é possível que uma sobrecarga por troca de contexto dos processos das aplicações seja introduzida. Assim, uma estratégia de gerenciamento eficiente deverá se posicionar entre as duas abordagens citadas, possuindo as vantagens de cada uma e reduzindo suas desvantagens. Esta estratégia deverá implementar um gerenciamento cuidadoso entre as duas abordagens a fim de maximizar a utilização de recursos.

Através da observação dos usuários da grade [6] é possível perceber que muitos, ao submeterem suas aplicações, não estão muito interessados no tempo de execução ou até mesmo desconhecem o tempo ótimo de execução de suas aplicações. Entretanto, o principal objetivo dos principais sistemas gerenciadores de grade é minimizar o tempo de execução das aplicações. Por exemplo, o *EasyGrid AMS* é um sistema gerenciador de

aplicações que pode utilizar a estratégia de compartilhamento de recursos horizontal e tem como principal objetivo reduzir o tempo de execução das aplicações. Uma vez que a abordagem horizontal admite que vários usuários compartilhem um mesmo conjunto de recursos ao mesmo tempo é preciso que alguma política seja implementada a fim de impedir que aplicações gulosas (que buscam sempre executar no menor tempo possível) utilizem o ambiente sem controle e aumentem a sobrecarga de troca de contexto a tal ponto que comprometa os tempos de execução de todas as aplicações.

Este trabalho propõe uma heurística de execução que permite que um usuário, ao submeter sua aplicação, defina um tempo de execução que o atenda, ou seja, que para ele seja razoável. Esse tempo de execução é chamado de *meta da aplicação*. Dessa forma, cada aplicação irá executar ações que a auxiliem alcançar sua meta e, se possível, também executar ações que auxiliem outras aplicações a alcançar suas metas. Através da implementação de um comportamento próprio para cada aplicação, diferentemente da estratégia anterior onde todas as aplicações tinham o mesmo comportamento guloso, espera-se que o conjunto de aplicações se ajuste da melhor forma possível aos recursos da grade, formando assim uma sociedade de aplicações sustentável. Assim, a estratégia proposta visa unir as vantagens da abordagem horizontal (maximização da utilização dos recursos) e da abordagem vertical (eliminação/redução da sobrecarga por troca de contexto).

3.3 Resumo

Este capítulo apresentou as estratégias de compartilhamento de recursos vertical e horizonal. Através de avaliações conceitual e empírica das estratégias foi possível definir suas vantagens e desvantagens. A principal diferença entre as abordagens é que, enquanto na abordagem horizontal há o compartilhamento de recursos simultâneo, na vertical um recurso é utilizado por uma única aplicação em um mesmo período de tempo. Apesar de não introduzir sobrecarga em função da troca de contexto entre os processos da aplicação do usuário, a abordagem vertical pode subutilizar os recursos da grade já que recursos possuem diferentes tipos de serviços. Já a estratégia horizontal tende a maximizar o uso de recursos através do compartilhamento simultâneo. Dessa forma, é possível concluir que uma estratégia que unisse as vantagens das duas abordagens seria interessante. O próximo capítulo apresentará uma proposta de estratégia que possui características das duas abordagens apresentadas nesse capítulo.

Capítulo 4

Implementação do gerenciamento autônomo de múltiplas aplicações

A camada de escalonamento dinâmico do *EasyGrid AMS* implementa heurísticas cujo objetivo é executar a aplicação do usuário no menor tempo possível. Dessa forma, um conjunto de aplicações que esteja executando em um mesmo conjunto de recursos utilizando o *EasyGrid AMS* estará disputando estes recursos de forma individual e egoísta. Isso ocorre porque o objetivo de cada AMS é reduzir o tempo de execução de sua aplicação. Os experimentos do capítulo 3 mostraram que esta estratégia egoísta não é sustentável quando recursos são compartilhados e o número de processos concorrentes cresce (figuras 3.1, 3.2 e 3.3).

Muitos usuários quando disparam sua aplicação na grade não estão interessados em executar no menor tempo possível. Além disso, alguns desconhecem o tempo de execução ótimo de suas aplicações. Entretanto, como o objetivo do *EasyGrid AMS* é minimizar o tempo de execução da aplicação, todas as aplicações serão executadas com o mesmo comportamento guloso, na tentativa de obter o tempo ótimo. Porém, este comportamento poderá introduzir no ambiente um *overhead* que irá prejudicar a execução de todas as aplicações.

Este trabalho propõe uma estratégia autônoma de execução de múltiplas aplicações menos egoísta e não gulosa. O objetivo é permitir que usuários definam um tempo satisfatório de execução e com base neste tempo, as aplicações busquem mais recursos ou cedam recursos dinamicamente e autonomamente. Assim, quando um usuário for executar sua aplicação ele irá definir uma meta para esta e com base neste valor e na estimativa do tempo restante de execução, a aplicação autônoma irá executar ações que a ajudem a alcançar o tempo desejado e, se possível, ajudar outras aplicações a alcançar o mesmo

objetivo.

A seção 4.1 descreve qual deve ser o comportamento das aplicações para que o compartilhamento traga benefícios. Esse comportamento é definido em função da meta inicial de execução definida pelo usuário e do grau de compartilhamento do recurso. A seção 4.2 apresenta como esse comportamento foi implementado.

4.1 Comportamento das aplicações

O comportamento das aplicações que executam utilizando o *EasyGrid* atual é estático e guloso, ou seja, cada aplicação só enxerga e se preocupa com si própria (objetivo *single minded*). O único objetivo é executar sua aplicação no menor tempo possível. Entretanto, em um ambiente onde várias aplicações compartilham recursos, formando uma sociedade, esse comportamento não é razoável.

Uma estratégia para resolver esse problema é diferenciar as aplicações através do objetivo de seus usuários. Esse objetivo é definido como um prazo que o usuário deseja que sua aplicação conclua a execução. Neste trabalho este prazo será chamado de *meta*.

O comportamento de uma aplicação é definido por um conjunto de ações que serão executadas em função da meta definida pelo usuário e do percentual de *CPU* que a aplicação tem em cada recurso compartilhado onde que ela está executando.

A meta irá funcionar como uma referência para os processos gerenciadores do *EasyGrid* avaliarem como está o andamento da execução. O percentual de *CPU* irá permitir que os processos gerenciadores realizem projeções e avaliem se no cenário corrente a meta é alcançável ou não. Assim, de acordo com os resultados obtidos ações serão executadas.

Um dado extremamente importante nesse processo é o tempo restante de execução. Com base no percentual de *CPU* que uma aplicação possui em um determinado recurso e na quantidade de trabalho que ainda deve ser realizada, a estimativa de tempo restante é calculada (A seção 4.2 irá descrever a implementação).

Através da estimativa de tempo restante e da meta definida pelo usuário os processos gerenciadores são capazes de, em um determinado instante da execução, definir o estado da aplicação. Assim, uma aplicação poderá estar em um dos seguintes estados: adiantado, no tempo, atrasado com chance ou atrasado sem chance. A equação abaixo descreve cada um desses estados:

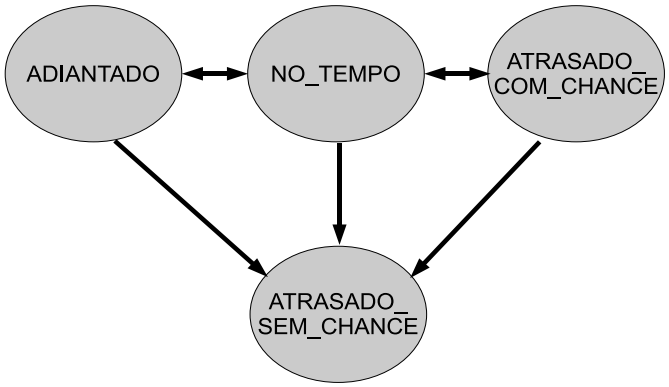

Figura 4.1: Diagrama de estados

$$agora + TR = \begin{cases} < meta & \text{então } adiantado \\ = meta & \text{então } noTempo \\ > meta & \text{então } atrasadoComChance \\ > (meta + TR) & \text{então } atrasadoSemChance; onde \end{cases}$$

onde:

- *agora*: tempo passado do início da execução até o momento em que o gerenciador realiza a operação;
- *TR*: tempo restante estimado de execução;
- *meta*: tempo que o usuário deseja executar sua aplicação;

Durante a execução uma aplicação poderá alternar seu estado. O único estado que se alcançado será o final é *atrasado sem chance* uma vez que neste a meta do usuário já foi ultrapassada pelo instante corrente. A figura 4.1 apresenta as possíveis trocas de estados que uma aplicação pode ter durante sua execução.

Quando uma aplicação está no estado *atrasado com chance* ela irá executar uma ação que a permita ganhar mais *CPU* na tentativa de sair deste estado. Essa ação irá trazer resultados caso os recursos onde esta aplicação esteja executando também estejam sendo compartilhados com outras aplicações. Caso contrário, ela já terá 100% da *CPU* e qualquer ação poderá gerar sobrecargas desnecessárias. Assim, uma outra variável importante neste processo indica se uma aplicação está sozinha ou acompanhada em um determinado recurso (a seção 4.2.1 explicará a implementação dessa variável).

Através do estado da aplicação, da variável "sozinho" e da política de execução do recurso (PER ou PEI) é possível definir um comportamento para uma aplicação. Esse comportamento é descrito nas tabelas 4.1 e 4.2.

Estado	Política de execução do host	Sozinho	Ação	Cenário
Adiantado	PER	SIM	1	1
Adiantado	PER	NÃO	2	2
Adiantado	PEI	SIM	1	3
Adiantado	PEI	NÃO	4	4
No tempo	PER	SIM	1	5
No tempo	PER	NÃO	2	6
No tempo	PEI	SIM	1	7
No tempo	PEI	NÃO	3	8
Atrasado com chance	PER	SIM	1	9
Atrasado com chance	PER	NÃO	2	10
Atrasado com chance	PEI	SIM	1	11
Atrasado com chance	PEI	NÃO	1	12
Atrasado sem chance	PER	SIM	1	13
Atrasado sem chance	PER	NÃO	2	14
Atrasado sem chance	PEI	SIM	1	15
Atrasado sem chance	PEI	NÃO	1	16

Tabela 4.1: Tabela de decisões

Ação	**Descrição**
ação 1	Aumentar percentual de *CPU*
ação 2	Interromper execução
ação 3	Manter estado
ação 4	Reduzir percentual de *CPU*

Tabela 4.2: Descrição das ações da tabela de decisões

Quando uma aplicação está executando sozinha, ou seja, ela é a única aplicação em um recurso em um determinado momento, seu objetivo principal é acabar da forma mais rápida possível. Assim, quando uma aplicação estiver sozinha, independente do seu estado de execução (atrasado, normal ou adiantado), ela irá busca maximizar o percentual de uso de *CPU*.

Por outro lado, quando uma aplicação está executando concorrentemente com outra em um mesmo recurso, outros itens devem ser avaliados. Se a política de execução definida na máquina compartilhada for PER isso indicará que o recurso só poderá ser usado se este estiver ocioso. Assim, a ação correta neste momento é interromper a execução da aplicação naquele recurso. Caso a PEI esteja definida, será preciso avaliar o estado da aplicação em relação a seu tempo de execução. Caso a aplicação esteja atrasada ela irá

tentar maximizar o uso de *CPU* para alcançar a meta solicitada. Quando ela estiver adiantada, ela poderá reduzir seu percentual de *CPU* a fim de ajudar outras aplicações que estejam atrasadas. Já, quando ela estiver executando no tempo esperado, definido como normal, manterá seu estado. A seção 4.2 detalha características de implementação do comportamento descrito nesta seção.

4.2 Implementação do comportamento das aplicações

O objetivo desta seção é explicar como o comportamento de aplicações descrito na seção 4.1 foi implementado no *EasyGrid AMS*. Para isso, foi desenvolvido a heurística *Grid SA*. Essa seção explica como são calculados os parâmetros de entrada da heurística e posteriormente a heurística é apresentada e discutida.

4.2.1 Cáculo de parâmetros

O *EasyGrid AMS* realiza o monitoramento da aplicação do usuário durante toda sua execução. Periodicamente, os processos gerenciadores fazem a análise de alguns parâmetros do ambiente onde se está executando para avaliar se a meta definida pelo usuário está alcançável. Esses parâmetros são descritos abaixo.

- Percentual de *CPU* (*percCPU*)
  Para que a camada de escalonamento dinâmico consiga fazer a realocação de tarefas entre as máquinas onde uma aplicação está executando, ela precisa saber como está a carga do ambiente, ou seja, qual o percentual de *CPU* está sendo usado pela aplicação do usuário em cada recurso.

  O percentual de *CPU* é calculado periodicamente por cada *HM* e enviado ao *SM* para possíveis escalonamentos. O percentual pode ser calculado através de *cpu time* ou *uptime*. Caso uma tarefa u da aplicação tenha acabado de executar, *percCPU* é calculado através da equação 4.1. Caso contrário, *percCPU* é calculado por 4.2.

$$percCPU = (\frac{cpuTime(u)}{wallTime(u)}) \times n \tag{4.1}$$

$$percCPU = (\frac{n}{calculaNumProcsUptime()}) \tag{4.2}$$

  onde:

  - *cpuTime(u)*: tempo de *CPU* da última tarefa executada;
  - *wallTime(u)*: tempo de parede da última tarefa executada;
  - n: número de tarefas da aplicação que estão sendo executas no recurso;
  - *calculaNumProcsUptime()*: método que estima o número de processos que estão executando no recurso. A função *uptime* do sistema operacional é utilizada para este cálculo. Esta oferece as médias de processos em execução dos últimos 1, 5 e 15 minutos.

- Poder computacional (PC)
  O poder computacional representa o trabalho que um recurso é capaz de executar por segundo. A equação 4.3 ilustra o cálculo de PC.

$$PC = \frac{percCPU}{csi} \tag{4.3}$$

  onde:

  - *csi (computational slowdown index)*: está relacionado à velocidade de processamento de um recurso considerando todos os recursos da arquitetura que estão dedicados a aplicação do usuário. Ele indica quanto tempo é necessário para executar uma tarefa de peso 1 da aplicação do usuário. O recurso mais rápido possui o menor *csi*.

- Sozinho (*sozinho*)
  Esta variável é utilizada pelo *HM* para verificar se existem processos concorrendo pelo recurso. Quando a *PER* está definida para um recurso o *HM* precisa saber se ele está sozinho no recurso para criar processos. Ela é derivada do cálculo de *percCPU*. Caso *percCPU* seja maior ou igual a um valor mínimo a aplicação acreditará estar sozinha. Nos experimentos realizados neste trabalho esse valor máximo foi 80% [29].

- Tempo restante (TR)
  É a estimativa de quanto tempo falta para o *HM* executar todas as tarefas a ele alocadas. Esse valor é calculado pela equação 4.4.

$$TR = (\sum_{u \in V_{HM}} peso(u)) \times \frac{1}{PC} \tag{4.4}$$

  onde:

- u: tarefa da aplicação.
- *peso(u)*: peso (quantidade de trabalho) da tarefa u.
- V_{HM}: conjunto de tarefas da aplicacao alocadas a um *HM*.

4.2.2 Heurística *Grid SA*

Os parâmetros descritos na seção 4.2.1 são utilizados pelos gerenciadores para a escolha da ação a ser executada. O Algoritmo 1 descreve a heurística que implementa as ações da tabela 4.2. Seu retorno é o número de processos da aplicação do usuário que devem ser executados. Essa heurística é chamada *Grid SA*. O símbolo $*$ após alguns parâmetros indica que estes são passados por referência.

Algoritmo 1 : *GridSA*(tAgora, meta, tRestante, numeroDeProcessosAtual, *estadoAnterior, *dormindo)

```
1  tPrevisto = tAgora + tRestante;
2  novoEstado = calculaEstado(meta, tPrevisto, tAgora);
3  se (sozinho) então
4     *dormindo = falso;
5     *estadoAnterior = novoEstado;
6     retorne numeroDeNucleosDoProcessador;
   fim se
7  se (politica=PER) então
8     *estadoAnterior = novoEstado;
9     retorne 0;
   fim se
10 se (estado=ADIANTADO) então
11    *estadoAnterior = novoEstado;
12    *dormindo = verdadeiro;
13    retorne 0;
   fim se
14 se (estado ∈ {NO_TEMPO, ATRASADO_COM_CHANCE,
   ATRASADO_SEM_CHANCE}) então
15    se (deseja incrementar estado pela segunda vez seguida) então
16       *estadoAnterior = novoEstado;
17       *dormindo = falso;
18       caso (estado)
19          NO_TEMPO: retorne 1;
20          ATRASADO_COM_CHANCE: retorne 2;
21          ATRASADO_SEM_CHANCE: retorne 3;
         fim caso
      fim se
22    senao
23       retorne numeroDeProcessosAtual;
      fim senao
   fim se
```

Basicamente, o objetivo da heurística GridSA é permitir que aplicações que estão adiantadas reduzam o percentual de *CPU* alocada a elas a fim de beneficiar aplicações que estejam executando sobre o mesmo recurso e atrasadas.

Alguns pontos do Algoritmo 1 devem ser destacados.

- Tolerância

  Indica qual o percentual de tolerância do *HM* para dizer que seu estado é NO_TEMPO. Dessa forma, o *HM* estará NO_TEMPO caso a equação 4.5 seja obedecida.

$$meta - tol \times meta \leq agora + TR \leq meta + tol \times meta \quad (4.5)$$

onde:

- *meta*: tempo que o usuário deseja executar sua aplicação.
- *tol*: percentual de tolerância. O valor utilizado nos experimentos deste trabalho foi 10%. Durante os experimentos valores menores para essa variável fizeram a heurística variar com uma frequência muito alta entre os estados. Por outro lado, valores maiores retardaram a reação da heurística diante de mudanças do ambiente.

- Frequência de verificação
  É o intervalo de tempo em que o *EasyGrid AMS* executa a heurística *Grid SA* para verificar em qual estado a aplicação está e executar as ações necessárias para atingir a meta definida pelo usuário. Caso esse valor seja muito pequeno um *overhead* poderá ser introduzido já que a frequência de execução da heurística *Grid SA* será alta. Entretanto, se esse intervalo for muito grande o *EasyGrid* poderá demorar para perceber possíveis mudanças no ambiente e não executar ações a tempo. Durante os experimentos o intervalo de teste foi igual a 4 vezes o peso das tarefas da aplicação. Experimentos foram realizados para a definição deste valor.

- Número de processos
  Dependendo do estado da aplicação e da configuração do ambiente de execução uma aplicação tentará aumentar ou diminuir ser percentual de *CPU* na máquina. A estratégia deste trabalho para implementar essas ações é aumentar ou diminuir o número de processos da aplicação em execução na máquina.

  A estratégia inicial foi definir apenas os estados ATRASADO E ADIANTADO. Entretanto, através de experimentos observou-se que a frequência de troca de estados era muito alta. Além disso, a definição do estado atrasado não diferenciava os *HMs* que já tinham "perdido"sua meta (tempo agora é maior que meta) daqueles que estavam atrasados e ainda tinham chance de se recuperar (atrasado mas tempo agora é menor que meta). Para resolver esses problemas foram introduzidos os estados NO_TEMPO e ATRASADO_SEM_CHANCE, e o antigo estado ATRASADO passou a se chamar ATRASADO_COM_CHANCE.

  Nesse novo conjunto de estados, as ações eram:

  - ADIANTADO: número de processos é alterado para 0.
  - NO_TEMPO: número de processos é mantido.
  - ATRASADO_COM_CHANCE: número de processos é alterado para 1.
  - ATRASADO_SEM_CHANCE: número de processos é alterado para 2.

  Essa nova abordagem não resolveu os problemas completamente. No cenário onde um *HM* está ADIANTADO e após um tempo muda para o estado NO_TEMPO seu número de processos continuar 0 e ele continua dormindo. Após mais um tempo este *HM* passa para o estado ATRASADO_COM_CHANCE e só agora passará a executar 1 processo e irá acordar. Este exemplo mostra que o que ocorreu de fato foi um salto do *HM* do estado ADIANTADO para o estado ATRASADO. Dessa forma, uma nova proposta foi criada:

  - ADIANTADO: número de processos é alterado para 0.
  - NO_TEMPO: número de processos é alterado para 1.
  - ATRASADO_COM_CHANCE: número de processos é alterado para 2.
  - ATRASADO_SEM_CHANCE: número de processos é alterado para 3.

  Apesar de aumentar o número de processos em execução e consequentemente a perda por troca de contexto essa abordagem permitiu melhor diferenciar os estados do *HM* e consequentemente refinar o comportamento da heurística *GridSA*. Além disso, os experimentos do capítulo 5 irão mostrar que o *overhead* introduzido é pequeno.

- Cálculo de variável *PC* durante estado ADIANTADO
  Um problema durante o desenvolvimento da heurística *Grid SA* foi a implementação do estado ADIANTADO no *EasyGrid AMS*. Como toda a arquitetura do *middleware* não considerava a possibilidade do número de processos em execução ser 0, houve a necessudade de algumas mudanças.

  Quando um *HM* ficava adiantado e atribuía 0 a seu número de processos em execução o valor do *PC* também mudava para 0. Nesse caso, esperava-se que o *HM* ficasse sem executar nenhum trabalho até ficar atrasado e voltar a executar. Entretanto, nesse momento entrou em execução a camada de escalonamento dinâmico. Quando o *PC* do *HM* foi para 0 o escalonamento dinâmico, acreditando que o recurso estava sobrecarregado, retirou suas tarefas e distribuiu para outros *HMs*. Assim, outros *HMs* ficaram sobrecarregados e esse ficou dormindo sem executar nada até o final da execução.

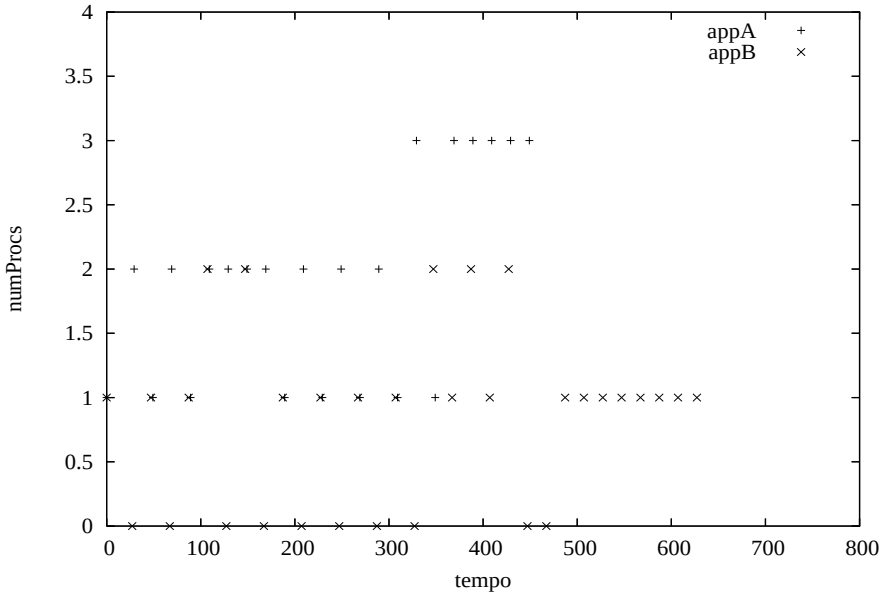

Figura 4.2: Execução de duas aplicações com erro

A primeira proposta para resolver este problema foi tirar uma foto do ambiente no momento antes do *HM* dormir e utilizar esta para calcular o PC enquanto o *HM* dormia. Entretanto, como o ambiente grade é dinâmico, uma foto do ambiente quando o *HM* dormiu não representa o estado do mesmo enquanto este dorme.

A segunda e final proposta para resolver este problema foi permitir que durante sua "soneca" o *HM* imaginasse o quanto de PC ele teria caso tivesse um único processo em execução. Essa abordagem mostrou-se mais eficiente e assim o *HM* passou a ter um valor mais preciso do PC.

- Impacto da estratégia gulosa no ambiente
  Como o estado do ambiente que o *HM* enxerga é sempre uma foto de um passado recente, suas ações são tomadas acreditando-se que aquelas configurações foram mantidas. Entretanto, isso não é verdade sempre. Alguns experimentos mostraram que a ação gulosa de um *HM* que "incrementa seu estado"muito cedo pode impactar o sistema e "assustar"outras aplicações que estão "dormindo"(adiantadas) e as fazer acordar desnecessariamente. A figura 4.2 mostra um caso onde esse problema ocorreu.

  Como as duas aplicações iniciam a execução com 1 processo ambas acreditam ter 50% da *CPU*. Para a aplicação B, que tem meta igual a 900, esse cenário é excelente e ela muda para o estado ADIANTADO. Entretanto, para a aplicação A, que tem meta 300, o cenário é ruim e esta muda para o estado ATRASADO. Essa ação de A irá assustar a aplicação B que por sua vez irá reagir e desbalancear todo o sistema.

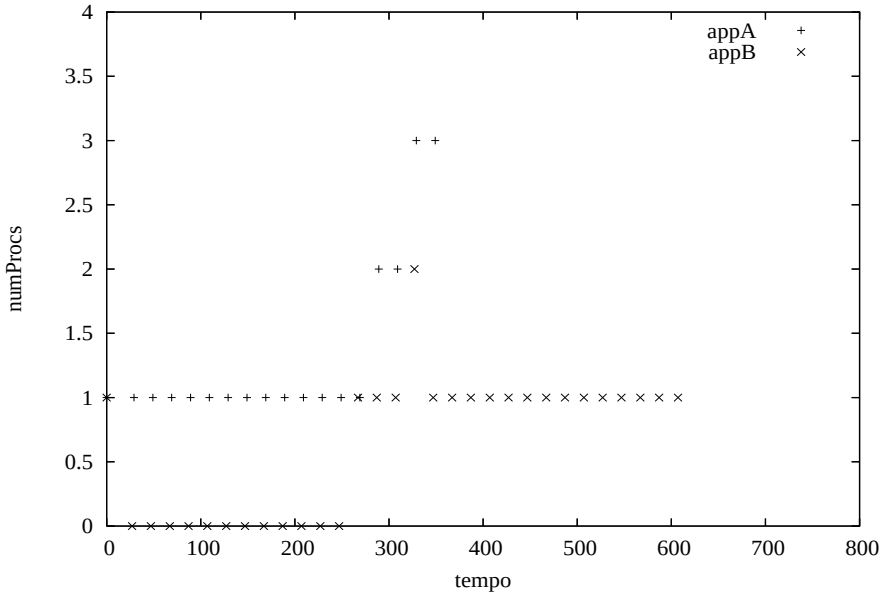

Figura 4.3: Execução de duas aplicações sem erro

Para resolver este problema foi proposta uma estratégia de incremento de estado retardado (inspirado na idéia de crescimento aditivo/decrescimento multiplicativo TCP/IP [35]). Nessa estratégia, um *HM* que deseja "incrementar" seu estado só irá fazer isso se desejar por duas vezes seguidas. Dessa forma, a aplicação *A* não mudará para o estado ATRASADO no primeiro teste em função da nova estratégia. Já na segunda, a verificação perceberá que a aplicação *A* está sozinha e irá desistir da mudança. A figura 4.3 mostra o resultado com a nova estratégia.

4.3 Resumo

Este capítulo apresentou a proposta deste trabalho para a implementação do gerenciamento autônomo de múltiplas aplicações. Essa proposta foi chamada de heurística *Grid SA*. Para avaliar essa heurística diversos experimentos foram realizados. O capítulo 5 apresenta o resultado desses experimentos e faz uma análise dos mesmos.

Capítulo 5

Análise Experimental

O objetivo deste capítulo é avaliar experimentalmente o conceito de autogerenciamento distribuído e a heurística *Grid SA*. A seção 5.1 apresenta a configuração dos experimentos, a seção 5.2 avalia o comportamento da heurística durante os experimentos e a seção 5.3 apresenta os resultados. Por fim, a seção 5.4 apresenta um experimento com o objetivo de mostrar que a execução simultânea de aplicações em um mesmo recurso pode maximizar o uso do recurso e produzir melhores resultados do que a execução sequencial.

5.1 Configuração

Os experimentos foram organizados em três grupos. O objetivo do primeiro grupo é determinar qual o tempo de execução de uma única aplicação considerando os recursos dedicados, sendo que a aplicação utiliza o *EasyGrid AMS* sem a política *Grid SA*. No segundo grupo, aplicações executam simultaneamente sobre conjuntos de recursos não disjuntos, com *EasyGrid AMS* e sem a heurística *Grid SA*. Por fim, o terceiro grupo tem a mesma configuração do grupo dois mas desta vez utilizando a heurística *Grid SA*.

Todos os experimentos utilizaram aplicações do tipo *BoT* com 60 tarefas. O peso de cada tarefa foi 5 segundos. Aplicações *BoT* foram escolhidas porque são massivamente paralelas e puramente *CPU intensive*. Dessa forma, uma aplicação *BoT* não oferece oportunidade de outras aplicações aproveitarem ciclos de *CPU* durante sua execução. Este cenário representa o pior caso para a heurística *Grid SA*, uma vez que esta busca utilizar ciclos de *CPU* que são perdidos com a execução sequencial. Foram utilizados 9 recursos *Pentium 4 - 512 MB de RAM*. A figura 5.1 apresenta a topologia de rede utilizada.

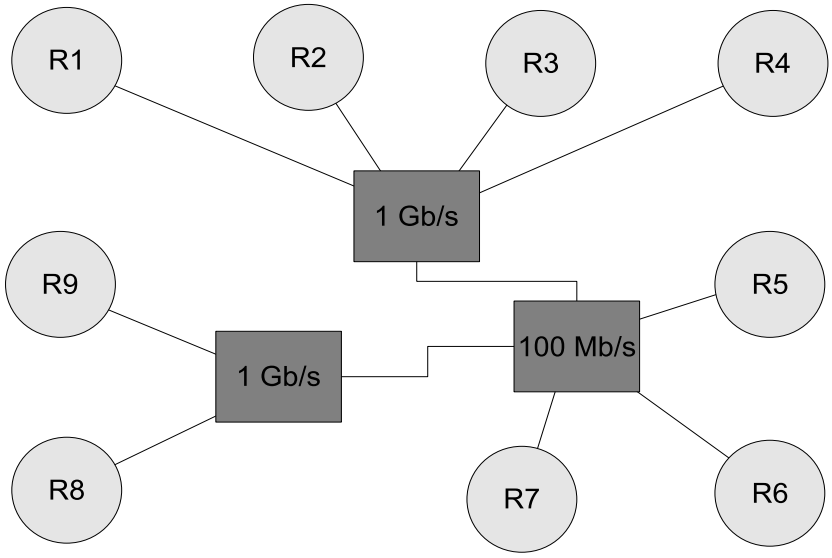

Figura 5.1: Topologia da rede

A tabela 5.1 apresenta a configuração dos experimentos. A coluna *Grupo* indica a que grupo o experimento pertence, *Exp* identifica o experimento, *Grid SA* indica se a heurística *Grid SA* é usada ou não, *Número de aplicações* e *Número de recursos* indicam a quantidade de aplicações e recursos utilizados nos experimento respectivamente. Os valores da coluna *Número de recursos* estão dispostos no formato $x + y$, onde x indica o número de recursos utilizados para executar a aplicação do usuário e y a quantidade de recursos que executam apenas processos gerenciadores do *EasyGrid*.

Grupo	Exp	Grid SA	Número de aplicações	Número de recursos
1	1.1	não	1	1+1
1	1.2	não	1	5+1
2	2.1	não	2	1+1
2	2.2	não	2	8+1
3	3.1	sim	2	1+1
3	3.2	sim	2	6+1

Tabela 5.1: Configurações dos experimentos

A tabela 5.2 descreve como os processos gerenciadores do EasyGrid de cada aplicação estão distribuídos pelos recursos disponíveis. Os processos gerenciadores *host manager*, *site manager* e *global manager* da aplicação i são identificados nesta tabela por *Hi*, *Si* e *Gi* respectivamente. A primeira coluna identifica o grupo de experimentos, a segunda coluna identifica os experimentos, a terceira coluna identifica a aplicação e as demais indicam quais processos gerenciadores estão executando no recurso.

Grupo	Exp	Aplicação	R1	R2	R3	R4	R5	R6	R7	R8	R9
1	1.1	1	G1/S1	H1							
1	1.2	1	G1/S1	H1	H1	H1	H1	H1			
2	2.1	1	G1/S1	H1							
2	2.1	2	G2/S2	H2							
2	2.2	1	G1/S1	H1	H1	H1	H1	H1	H1	H1	H1
2	2.2	2	G2/S2	H2	H2	H2	H2	H2	H2	H2	H2
3	3.1	1	G1/S1	H1							
3	3.1	2	G2/S2	H2							
3	3.2	1	G1/S1	H1	H1	H1	H1	H1	H1		
3	3.2	2	G2/S2	H2	H2	H2	H2	H2	H2		

Tabela 5.2: Distribuição dos processos gerenciadores do EasyGrid pelos recursos

De acordo com a quantidade de tarefas da aplicação e a quantidade de recursos do experimento é possível calcular o tempo ótimo de execução. A tabela 5.3 descreve o tempo ótimo de execução dos experimentos. A primeira coluna indica o grupo do experimento, a segunda indica o experimento e as demais indicam o tempo ótimo de execução das aplicações.

Grupo	Exp	Aplicação 1	Aplicação 2
1	1.1	300	-
1	1.2	60	-
2	2.1	600	600
2	2.2	75	75
3	3.1	600	600
3	3.2	100	100

Tabela 5.3: Tempo ótimo de execução dos experimentos

Nos experimentos do grupo 3 foi preciso definir metas de execução para cada aplicação uma vez que a heurística GridSA foi utilizada. As tabelas 5.4 e 5.5 descrevem as metas de execução dos experimentos do grupo 3. Três valores de meta foram definidos. O primeiro foi o tempo ótimo de execução, o segundo o dobro do tempo ótimo e o terceiro o triplo do tempo ótimo.

5.2 Avaliação qualitativa

O objetivo desta seção é fazer uma análise qualitativa dos resultados dos experimentos. Para isso, será feita uma análise do comportamento da heurística *Grid SA*. Como apenas os experimentos do grupo 3 utilizam a heurística, os experimentos dos grupos 1 e 2

Sub exp	Aplic. 1	Aplic. 2
3.1.1	300	300
3.1.2	300	600
3.1.3	300	900
3.1.4	600	600
3.1.5	600	900
3.1.6	900	900

Tabela 5.4: Metas do experimento 3.1

Sub exp	Aplic. 1	Aplic. 2
3.2.1	50	50
3.2.2	50	100
3.2.3	50	150
3.2.4	100	100
3.2.5	100	150
3.2.6	150	150

Tabela 5.5: Metas do experimento 3.2

não serão contemplados, sendo analisados apenas quantitativamente na seção 5.3. Os experimentos 3.1 detalham o comportamento da heurística. Como os experimentos 3.1 utilizaram apenas 1 recurso, o experimento 3.2 mostra que o comportamento é o mesmo quando vários recursos são utilizados.

5.2.1 Experimentos 3.1

As figuras 5.2 a 5.7 apresentam a variação do número de processos de cada aplicação que está executando em um determinado momento durante o experimento 3.1.

A figura 5.2 apresenta um cenário onde duas aplicações desejam acabar no tempo ótimo. Como essas aplicações compartilham os mesmos recursos isso não é possível. Assim, ao longo do tempo o estado das aplicações muda de NO_TEMPO para ATRASADO_COM_CHANCE e no fim para ATRASADO_SEM_CHANCE (a explicação dos estados é feita na seção 4.1). Esse resultado foi obtido porque neste cenário as metas de ambas aplicações eram inviáveis. Assim, a heurística não pode atuar no escalonamento das aplicações e o resultado obtido foi semelhante ao resultado sem o uso da heurística *Grid SA*.

No cenário ilustrado na figura 5.3 a aplicação 1 possui o tempo ótimo de execução como meta e a aplicação 2 possui uma meta um pouco mais relaxada. A meta da aplicação

2 seria alcançada se ela tivesse 50% dos recursos disponíveis durante toda a execução.

Um pouco depois do início da execução a aplicação 1 percebe que possui 50% dos recursos e que sua meta não será atingida. Assim, ela muda para o estado ATRASADO_COM_CHANCE. Enquanto isso a aplicação 2 acreditando ter ainda 50% dos recursos mantem seu estado inalterado uma vez que dessa forma irá atingir sua meta. A partir desse ponto a aplicação 2 percebe que possui apenas 33% dos recursos e não conseguirá mais atingir sua meta. Dessa forma ela muda para o estado ATRASADO_COM_CHANCE e passa a ter 50% dos recursos.

Entretanto, com 50% dos recursos sua meta é alcançável e ela volta ao estado NO_TEMPO. Essa oscilação permanece até a aplicação 1 mudar para o estado ATRASADO_SEM_CHANCE e a aplicação 2 reagir mudando para o estado ATRASADO_COM_CHANCE. Por fim, aproximadamente no tempo 500, a aplicação 2 volta a executar 1 único processo ao perceber que está sozinha. O resultado ideal para este cenário ocorreria caso a aplicação 2 aguardasse a aplicação 1 concluir sozinha sua execução e somente depois iniciar sua execução no instante 300 e concluir no tempo 600. Entretanto, não foi possível implementar esse escalonamento porque, como o comportamento é autônomo e as aplicações não se comunicam, a aplicação 2 não sabe que a aplicação 1 irá acabar sua execução em 300 segundos. Caso a heurística *Grid SA* não fosse utilizada nesse experimento ambas aplicações acabariam aproximadamento no tempo 600. Entrentanto, utilizando a heurística, a aplicação 1 acabou aproximadamente no tempo 500 graças a aplicação 2 que, ao perceber que estava um pouco adiantada e acompanhada, cedeu *CPU* para a aplicação 1.

O cenário da figura 5.4 apresenta a eficiência da heurística *Grid SA*. Neste, duas aplicações estão executando. A aplicação 1 está atrasada (meta igual a 300 segundos e tempo final esperado, com duas aplicações executando simultaneamente, de 600 segundos) enquanto a aplicação 2 está adiantada (meta igual a 900 segundos e tempo final esperado, com duas aplicações executando simultaneamente, de 600 segundos). Dessa forma, a aplicação 2, aproximadamente no instante 20, muda para o estado ADIANTADO e cede *CPU* enquanto a aplicação 1 executa nos recursos de forma dedicada. Aproximadamente no instante 250, a aplicação 2 que está dormindo percebe que pode não atingir sua meta se continuar dormindo e passa a executar um único processo. Apesar da aplicação 1 estar acabando ela se assusta com a aplicação 2 e por um período pequeno (aproximadamente 300 a 320) passa para o estado ATRASADO_SEM_CHANCE. Quanto a aplicação 2 percebe que está sozinha, no instante 320, ela passa a executar um único processo até o

fim aproximadamente no tempo 600. Observe que mesmo cedendo espaço à aplicação 1 o resultado da aplicação 2 não é comprometido. Caso a estratégia gulosa implementada na versão anterior do *EasyGrid* fosse utilizada, ambas aplicações terminariam no instante 600. Entretanto, uma vez que a aplicação 2 desejava apenas acabar antes do instante 900, ela pode liberar os recursos para outras aplicações com metas mais críticas, como a aplicação 1 e assim emular o comportamento de um *Job Scheduler* central.

Na figura 5.5 é possível perceber que quando as metas de execução são viáveis a heurística *Grid SA* é sensata. Neste caso as metas das aplicações são iguais ao tempo de execução quando os recursos são compartilhados. Nesse caso o comportamento é semelhante ao da execução sem a heurística *Grid SA*. Assim, a heurística *Grid SA* não comprometeu o resultado e obteve os resultados desejados pelo usuário de cada aplicação.

O cenário apresentado na figura 5.6 é semelhante ao cenário da figura 5.4. Porém em 5.6 as metas das aplicações são maiores e consequentemente o grau de liberdade também é maior. Inicialmente, por estar adiantada (meta igual a 900 segundos e tempo final esperado com duas aplicações executando simultaneamente de 600 segundos.), a aplicação 2 cede espaço para a aplicação 1. Aproximadamente no instante 250, a aplicação 2 percebe que poderá ficar atrasada e muda para o estado NO_TEMPO. Imediatamente, por estar adiantada e agora não mais sozinha, a aplicação 1 começa a "dormir". Esse experimento demonstra um comportamento muito interessante, pois mostra que é possível compartilhar recursos e atingir metas sem uma estratégia gulosa que prejudique todo o ambiente. Neste caso, apesar de ambas aplicações estarem adiantadas no início da execução, os comportamentos foram distintos em função da diferença das metas. Esse resultado mostra que aplicações podem se ajustar aos recursos de forma autônoma na presença de outras aplicações. Assim, é possível dizer que as aplicações foram "cordiais", mostrando boa vontade e cedendo espaço para outra por determinado período de tempo.

Como no cenário da figura 5.7 ambas aplicações estão muito adiantadas e não existem muitos problemas de compartilhamento. É interessante perceber que enquanto a aplicação 2 está "dormindo"a aplicação 1 está executando mesmo estando adiantada, tirando proveito da *CPU* disponível.

5.2.2 Experimentos 3.2

Neste experimento são utilizados 7 recursos. Além disso, todos os recursos são compartilhados pelas aplicações. O comportamento das aplicações é semelhante ao do experimento 3.1. A principal diferença é que nesta configuração, como existe mais de um recurso de

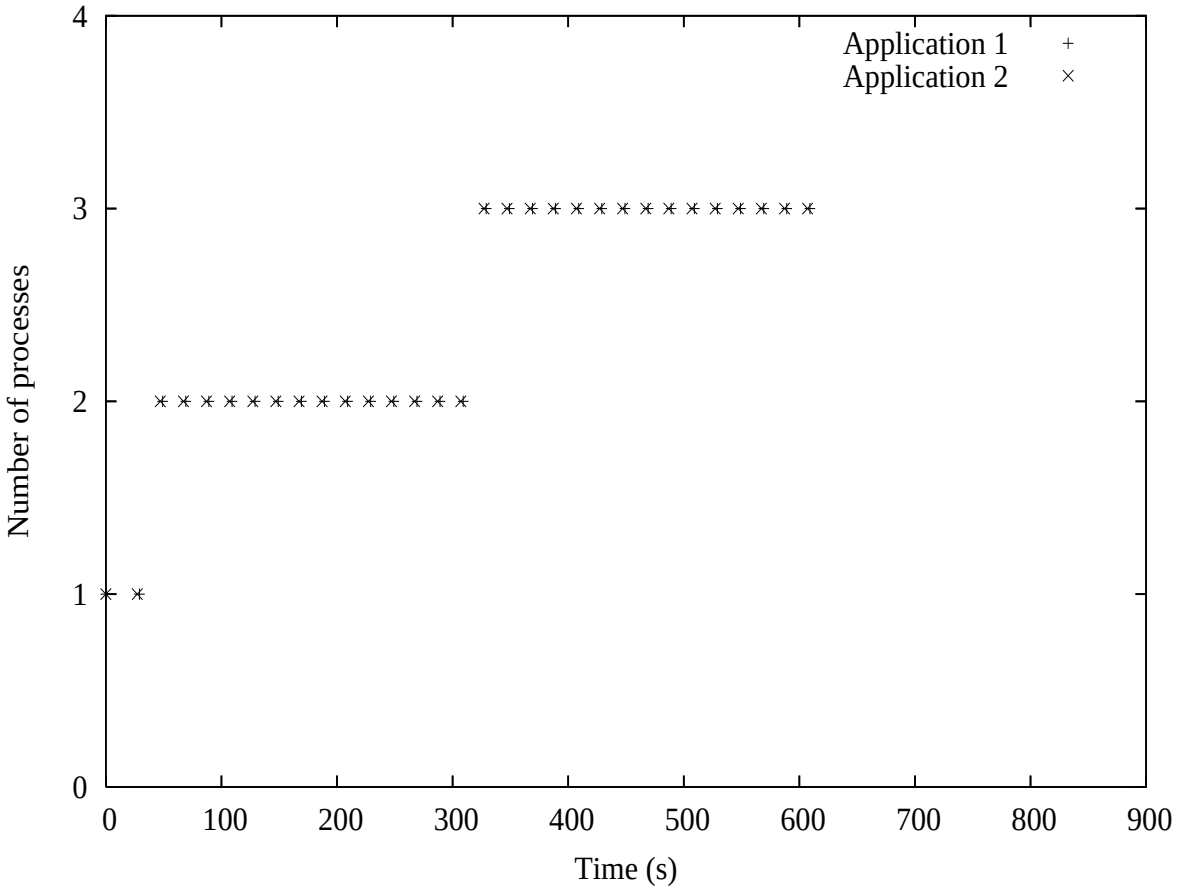

Figura 5.2: Variação do número de processos em execução das aplicações no experimento 3.1.1. As metas das aplicações 1 e 2 são 300 e 300 respectivamente (tabela 5.4).

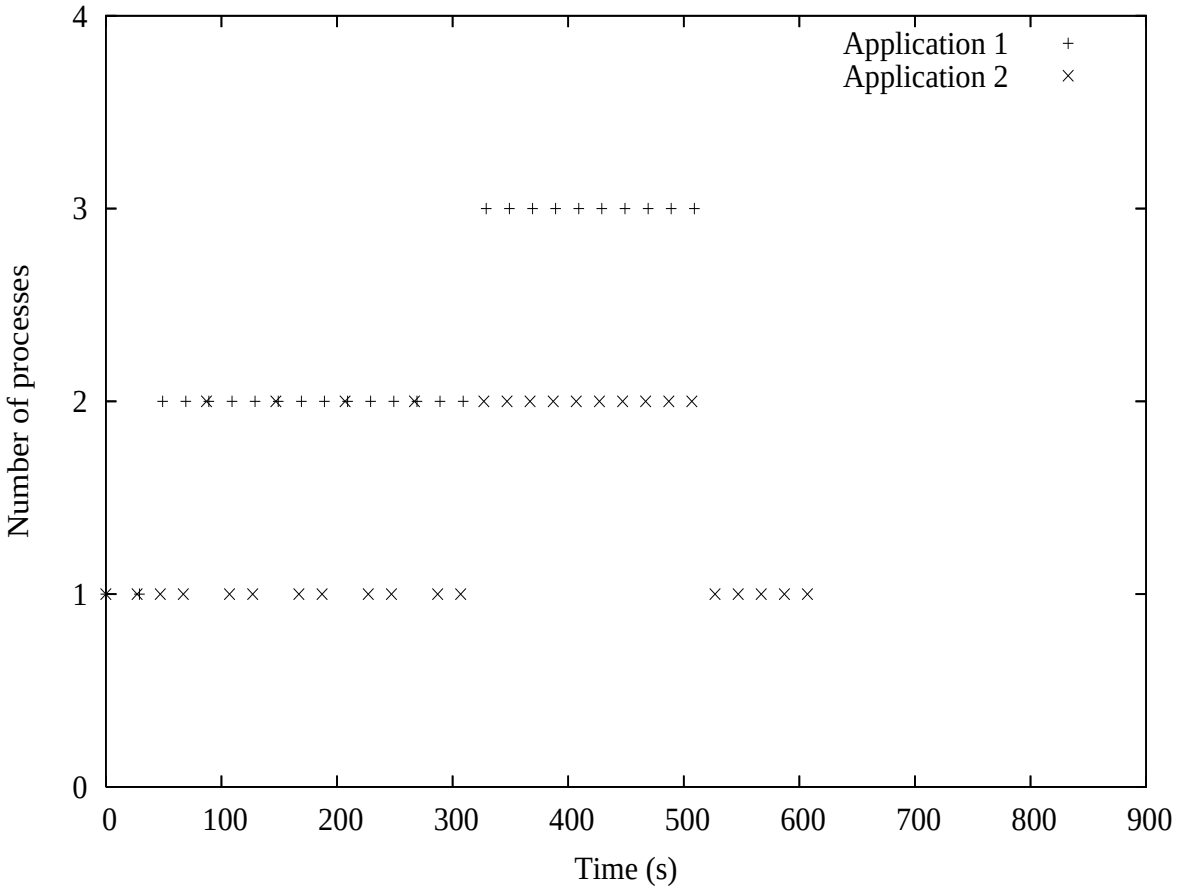

Figura 5.3: Variação do número de processos em execução das aplicações no experimento 3.1.2. As metas das aplicações 1 e 2 são 300 e 600 respectivamente (tabela 5.4).

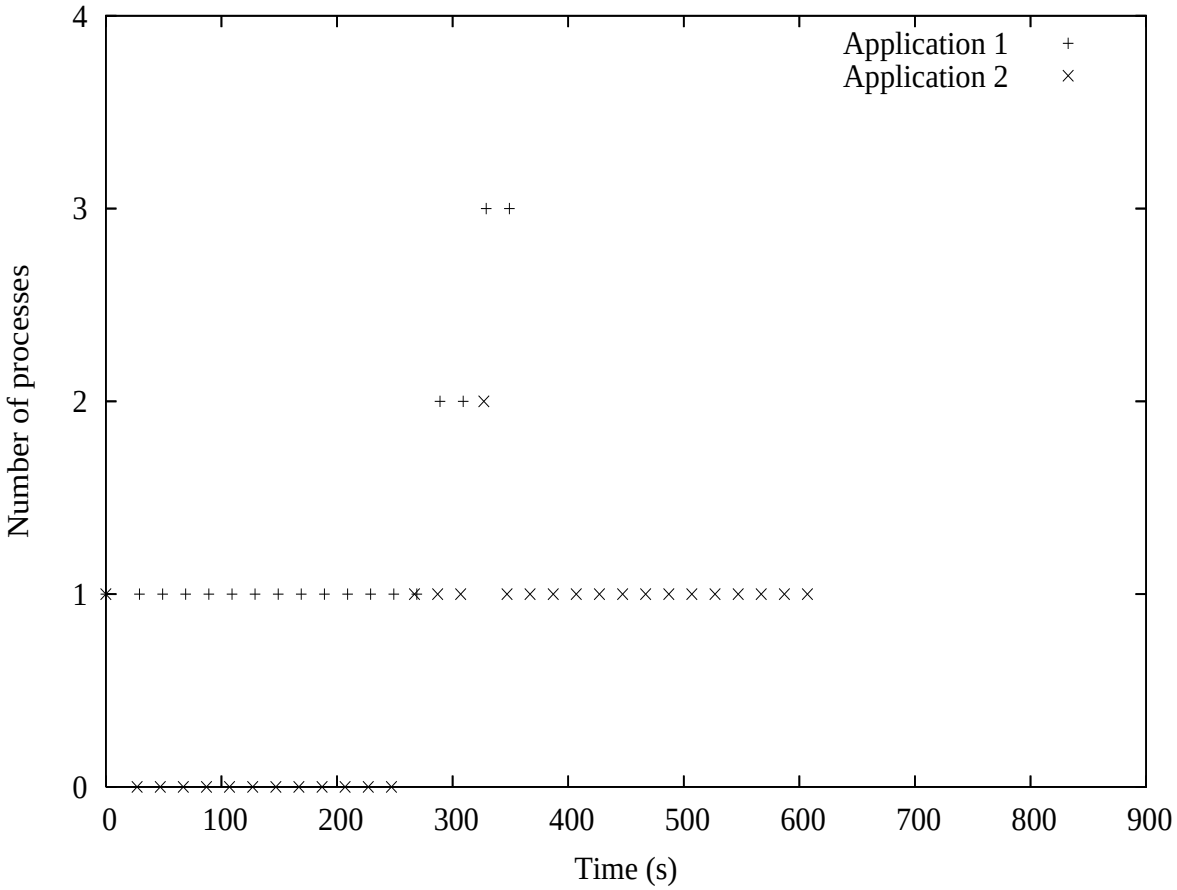

Figura 5.4: Variação do número de processos em execução das aplicações no experimento 3.1.3. As metas das aplicações 1 e 2 são 300 e 900 respectivamente (tabela 5.4).

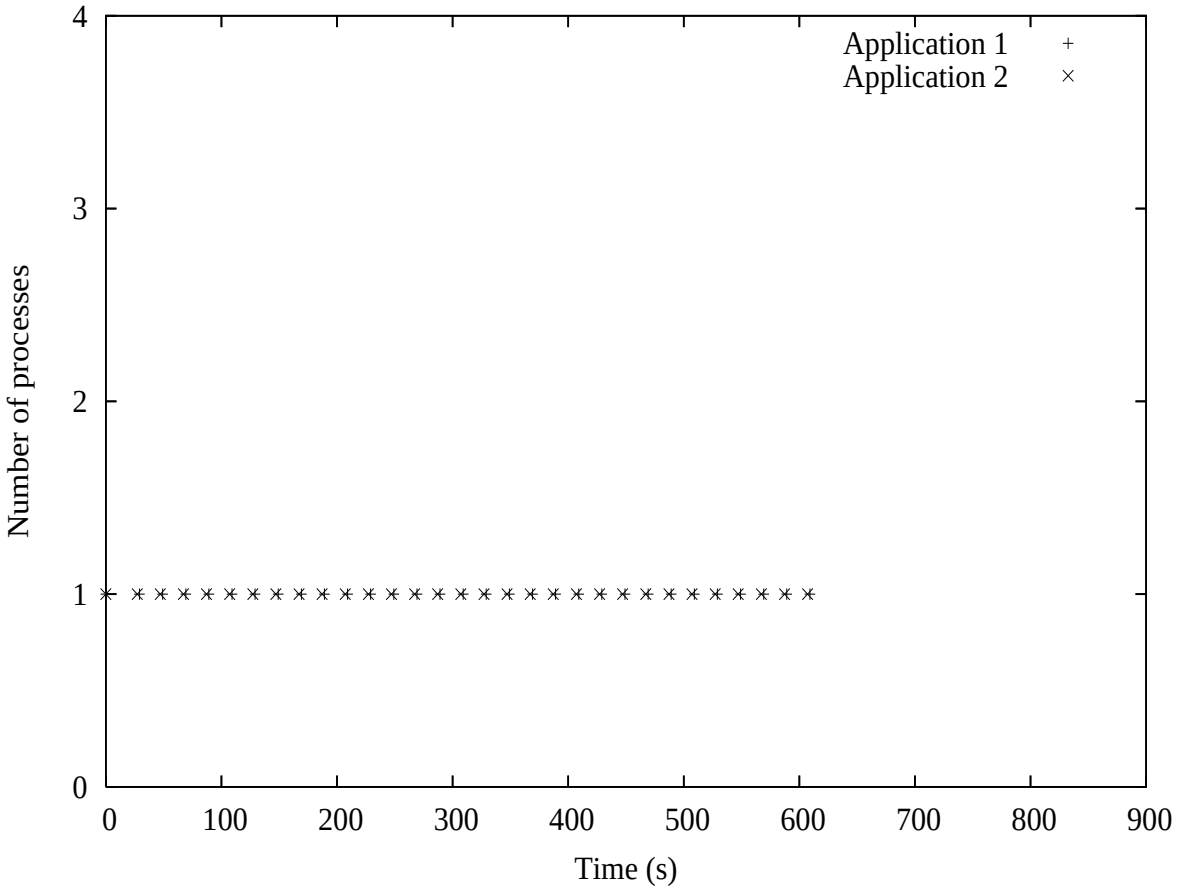

Figura 5.5: Variação do número de processos em execução das aplicações no experimento 3.1.4. As metas das aplicações 1 e 2 são 600 e 600 respectivamente (tabela 5.4).

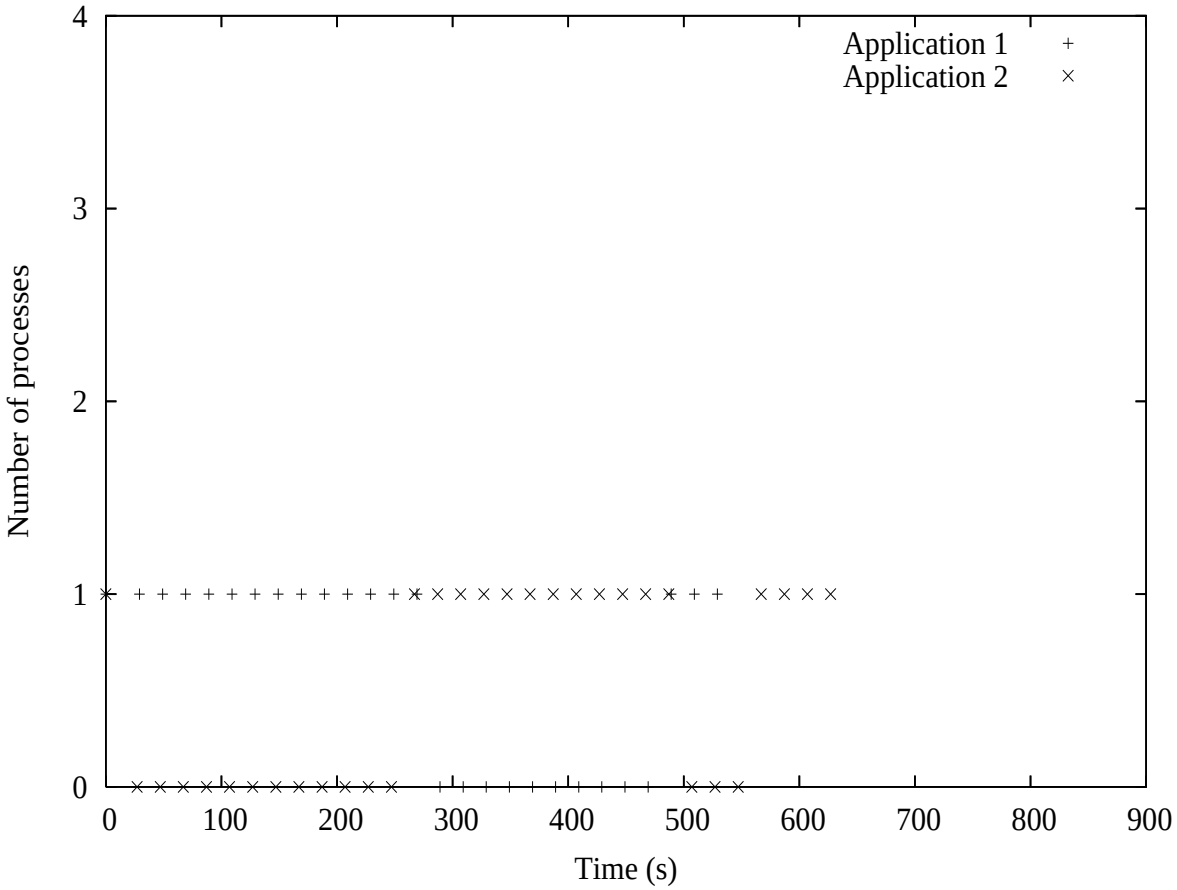

Figura 5.6: Variação do número de processos em execução das aplicações no experimento 3.1.5. As metas das aplicações 1 e 2 são 600 e 900 respectivamente (tabela 5.4).

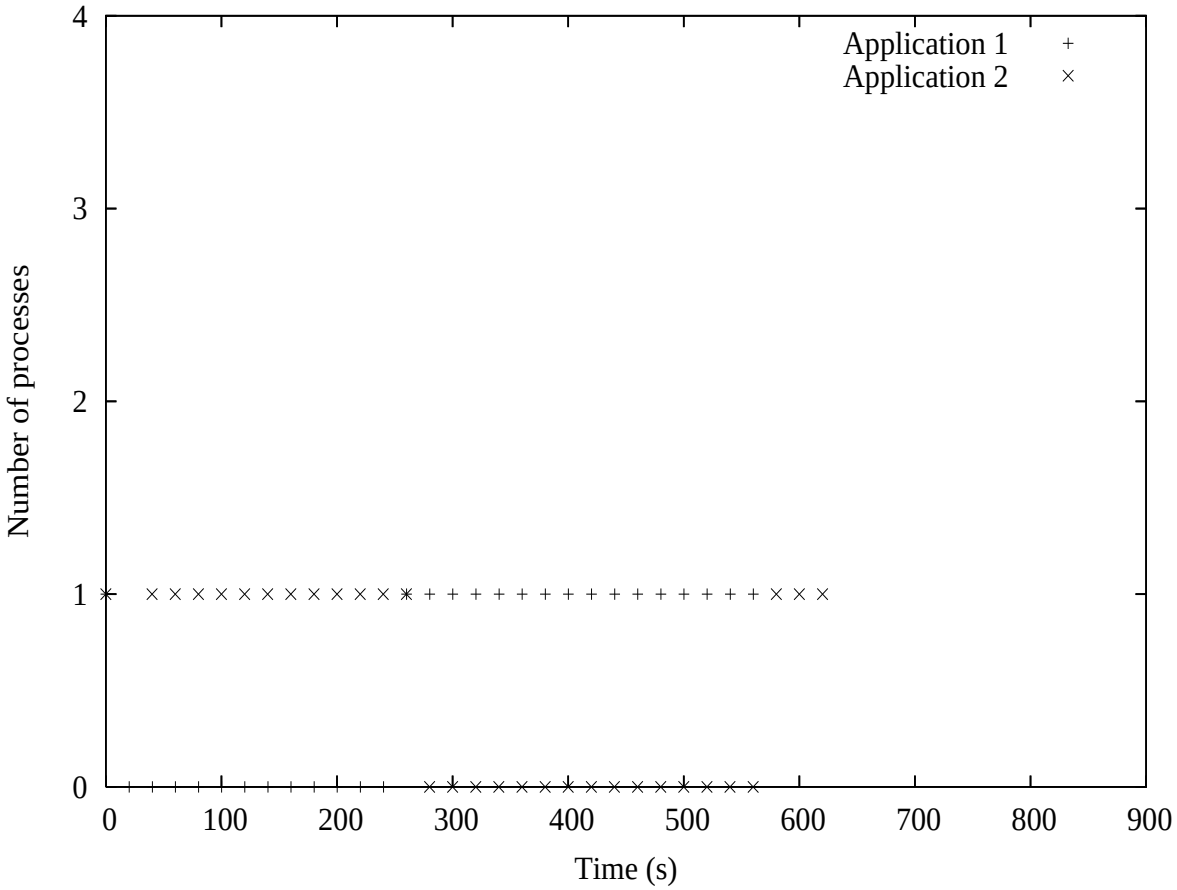

Figura 5.7: Variação do número de processos em execução das aplicações no experimento 3.1.6. As metas das aplicações 1 e 2 são 900 e 900 respectivamente (tabela 5.4).

execução, a camada de escalonamento dinâmico do *EasyGrid* pode atuar e transferir tarefas. Um problema descrito no capítulo 3 foi detectado nesses experimentos. Quando uma aplicação passava para o estado ADIANTADO e não executava mais processos da aplicação, o valor de seu poder computacional chegava a 0 e a camada de escalonamento dinâmico, acreditando que o recurso estava sobrecarregado, transferia toda a carga para outras máquinas. Esse problema não ocorreu no experimento 3.1 porque neste só existia 1 recurso. Através desses experimentos foi possível avaliar a eficiência da heurística *Grid SA* em cenários com vários recursos, diferentemente dos experimentos 3.1.

5.3 Análise quantitativa

As tabelas 5.6, 5.7 e 5.8 apresentam o tempo de execução dos experimentos. As colunas *Exp* e *Sub exp* identificam o experimento, as colunas *t1* e *t2* indicam o tempo de execução das aplicações 1 e 2 respectivamente e as últimas colunas (t_i/esp_i) indicam a relação entre o tempo de execução e o tempo esperado de execução das aplicações. O tempo estimado é o tempo de execução das aplicações no cenário utilizado sem a heurística *Grid SA*.

Exp	t_1	t_1/esp_1
1.1	304,66	1,01
1.2	64,15	1,07

Tabela 5.6: Tempo de execução dos experimentos do grupo 1

Exp	t_1	t_2	t_1/esp_1	t_2/esp_2
2.1	615,22	615,84	1,02	1,03
2.2	80,11	80,5	1,07	1,07

Tabela 5.7: Tempo de execução dos experimentos do grupo 2

A tabela 5.6 mostra que o *EasyGrid* introduz um *overhead* muito pequeno. O *overhead* do experimento 1.2 foi maior em relação a 1.1 em função da maior quantidade de recursos utilizados e consequentemente em função da maior troca de mensagens entre os processos gerenciadores, inicialização e finalização dos recursos.

Assim como nos experimentos do grupo 1, os experimentos do grupo 2 (tabela 5.7) também apresentaram resultados muito bons. O *overhead* do experimento 2.1 foi menor em relação a 2.2 porque 2.1 utilizou menos recursos. Além disso, o *overhead* do experimento 2.1 é um pouco maior que 1.1 em função da troca de contexto dos processos das aplicações 1 e 2.

Exp	Sub exp	t_1	t_2	t_1/esp_1	t_2/esp_2	$t_1/meta_1$	$t_2/meta_2$
3.1	3.1.1	620,67	617,55	1,03	1,03	2,07	2,06
3.1	3.1.2	513,21	617,14	0,85	1,03	1,71	1,03
3.1	3.1.3	359,40	613,04	0,60	1,02	1,20	0,68
3.1	3.1.4	616,48	616,20	1,03	1,03	1,03	1,03
3.1	3.1.5	543,63	638,23	0,91	1,06	0,91	0,71
3.1	3.1.6	569,44	625,90	0,95	1,04	0,63	0,69
3.2	3.2.1	108,96	109,67	1,09	1,10	2,18	2,19
3.2	3.2.2	95,74	109,30	0,96	1,09	1,91	1,09
3.2	3.2.3	70,13	139,47	0,70	1,39	1,40	0,93
3.2	3.2.4	111,75	111,38	1,12	1,11	1,12	1,11
3.2	3.2.5	70,30	136,78	0,70	1,37	0,70	0,91
3.2	3.2.6	127,24	64,51	1,27	0,64	0,85	0,43

Tabela 5.8: Tempo de execução dos experimentos do grupo 3

A tabela 5.8 apresenta os resultados dos experimentos executados utilizando a heurística GridSA. Os experimentos (3.1.1, 3.1.2, 3,1,3) e (3.2.1, 3.2.2, 3.2.3) mostram que o tempo de execução da aplicação 1 melhora conforme a meta da aplicação 2 aumenta. Isso ocorre porque quanto maior é a diferença entre a meta e o tempo final previsto de uma aplicação (se meta-esperado > 0 então adiantado) maior é a flexibilidade que a heurística *Grid SA* tem para introduzir no sistema ajustes a fim de beneficiar aplicações que estejam com metas mais críticas. Assim, a heurística proposta mostra-se eficiente ao permitir que aplicações compartilhem recursos e obtenham bons resultados através da diferenciação de seus objetivos e consequentemente seus comportamentos, e maximizem a utilização dos recursos do ambiente.

É válido destacar que a heurística *Grid SA* trabalha em conjunto com a camada de escalonamento dinâmico. Enquanto a camada de escalonamento dinâmico tem como objetivo fazer o balanceamento de carga dentro da aplicação a fim de minimizar o tempo de execução da mesma, a heurística *Grid SA* tem como objetivo fazer um escalonamento distribuído das diferentes aplicações que executam em recursos concorrentemente a fim de permitir que cada uma alcance sua meta sem comprometer o rendimento das demais aplicações.

Caso as filas dos *HMs* de um determinado *site* não estejam balanceadas, provavelmente os *HMs* estarão em estados diferentes.

A tabela 5.8 também apresenta a relação dos tempos e das metas de execução dos experimentos do grupo 3. As 2 últimas colunas indicam a relação do tempo de execução e a meta de execução das aplicações em cada experimento (t_i/m_i) Os experimentos mostram

que quando as metas desejadas são viáveis ($meta \geq timo$) a heurística GridSA se comporta muito bem.

5.4 Execução com compartilhamento simultâneo de recursos

O objetivo desta seção é apresentar experimentos onde o tempo de execução de duas aplicações que compartilham um mesmo recurso simultaneamente é menor do que a execução das aplicações de forma sequencial, como é feito tipicamente por *brokers*. Os experimentos foram realizados em um único recurso monoprocessado e as características das aplicações utilizadas são descritas na tabela 5.9.

Características	Aplicação 1	Aplicação 2
Número de tarefas	30	200
Tempo de *CPU* tarefa	5	2
Tempo de *IO* tarefa	15	0
Tempo total tarefa	20	2
Tempo aplicação	600	400
Meta	900	1200
Tempo de execução	722,84	868,90

Tabela 5.9: Configurações dos experimentos

Como a aplicação 1 possui tarefas que utilizam recursos de entrada/saída (*IO*) espera-se que, através do compartilhamento simultâneo, a aplicação 2 utilize esses períodos onde a *CPU* está ociosa para executar suas tarefas. Assim, o tempo final de execução das duas aplicações deverá ser menor que a soma dos tempos das duas aplicações (1000 = 600+400).

A figura 5.8 apresenta a variação do número de processos em execução das aplicações 1 e 2 ao longo do tempo. Nesta figura é possível perceber que enquanto o número de processos em execução da aplicação 1 permanece constante com o valor 1, o número de processos em execução da aplicação 2 varia entre 0 e 1. Isso ocorre porque, mesmo estando adiantada, a aplicação 2 percebe que a aplicação 1 está fazendo operações de entrada e saída e utiliza a *CPU* nestes períodos. Dessa forma, o tempo final de execução das duas aplicações foi 868,90 segundos (13% menor que a execução sequencial). Apesar do resultado ter sido melhor que a execução sequencial, o tempo final de execução das aplicações não foi o ideal. Como a aplicação 1 realiza operações de entrada e saída durante 450 segundos (15 segundos de entrada e saída em cada tarefa), era esperado que a aplicação 2 acabasse sua execução antes da aplicação 1. Isso não ocorreu porque, em alguns

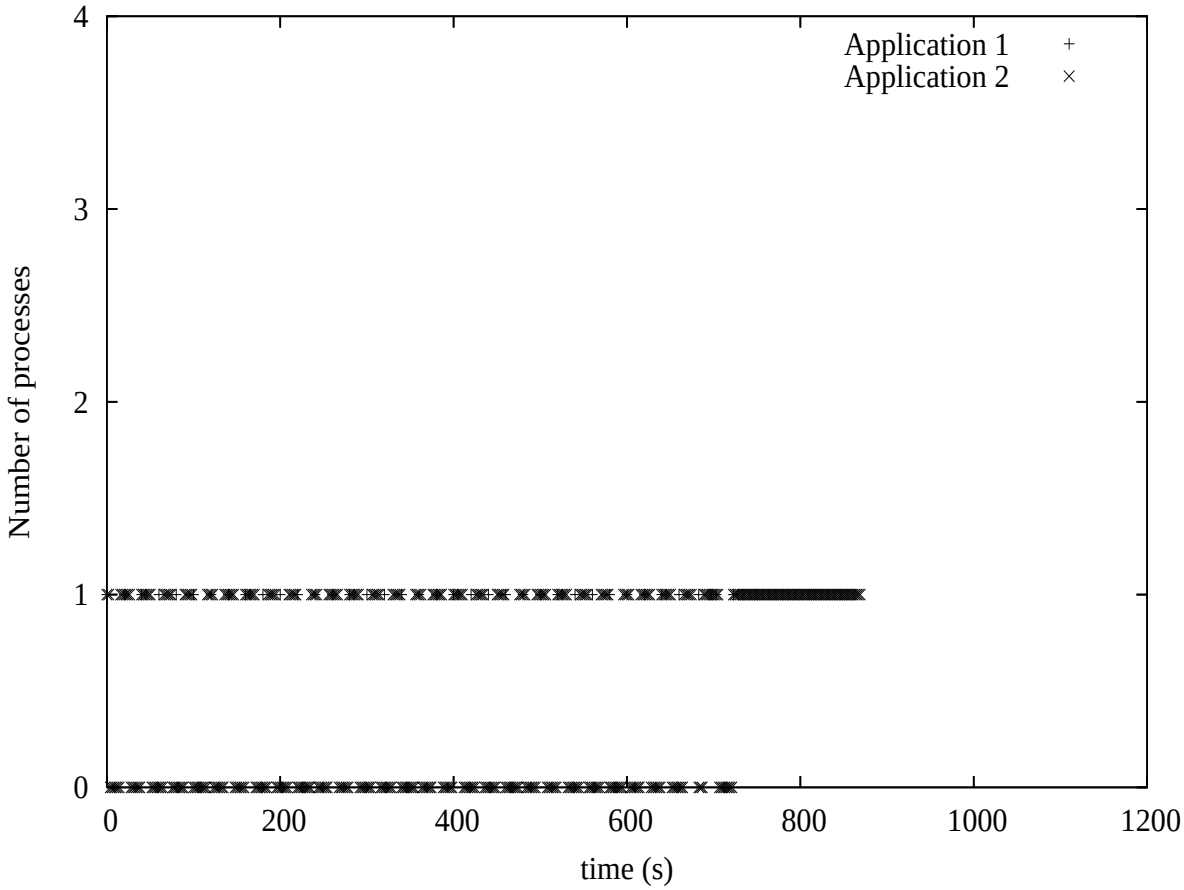

Figura 5.8: Variação do número de processos em execução das aplicações 1 e 2. As metas das aplicações 1 e 2 são 900 e 1200 respectivamente. Cada tarefa da aplicação 1 realiza 5 segundos de CPU e 15 de operações de E/S

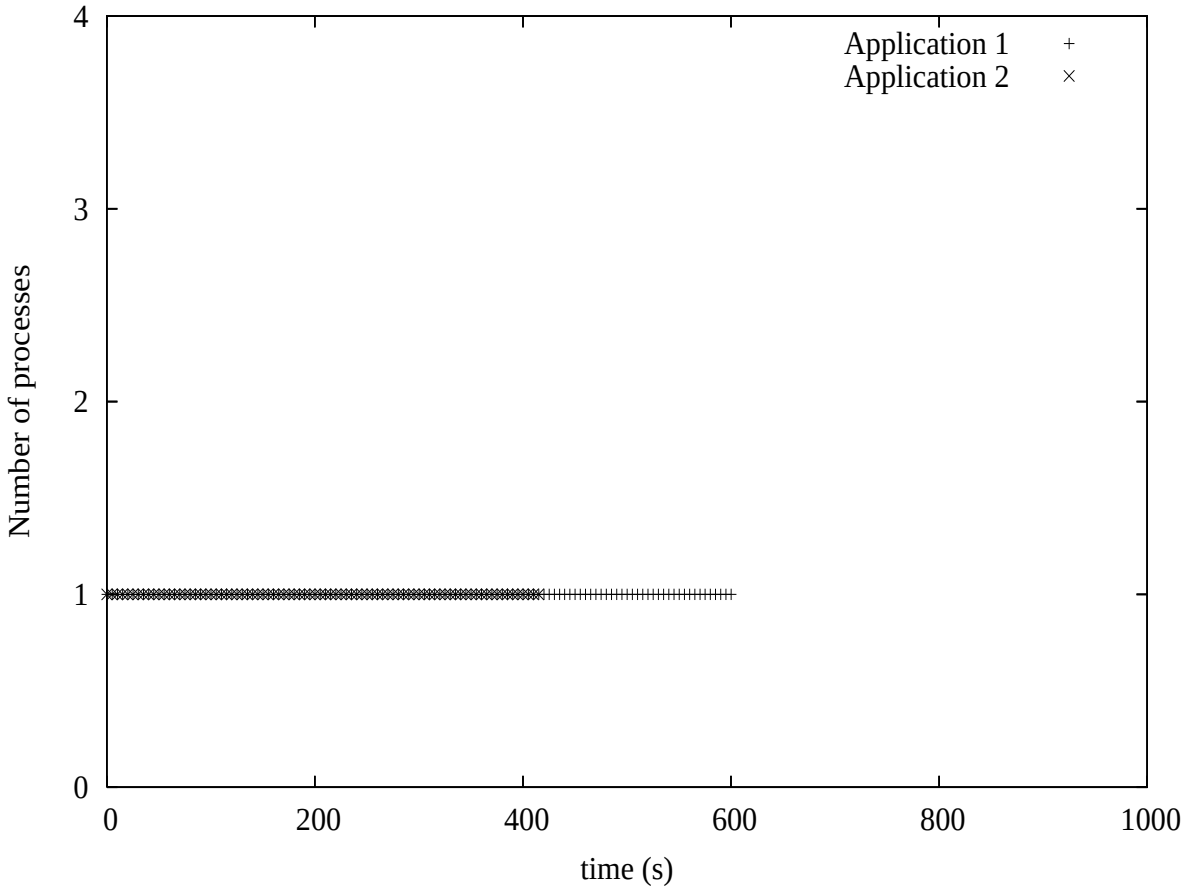

Figura 5.9: Variação do número de processos em execução das aplicações 1 e 2. As metas das aplicações 1 e 2 são 900 e 1200 respectivamente. Cada tarefa da aplicação 1 realiza 0 segundos de CPU e 20 de operações de E/S

momentos, a aplicação 2 demorava algumas iterações para perceber que estava sozinha e iniciar a execução de 1 processo. O mesmo ocorria quando a aplicação 2 estava executando 1 processo e a aplicação 1 passava a utilizar *CPU*. Como a aplicação 2 demorava a perceber que estava acompanhada, ela atrapalhava o resultado da aplicação 1.

A figura 5.9 ilustra o mesmo experimento considerando as tarefas da aplicação 1 com peso igual a 20 segundos mas realizando 20 segundos de *IO* e 0 segundos de *CPU*. Nesse caso, a aplicação 1 acabou no tempo 601,27 e a aplicação 2 no tempo 416,16. O experimento mostra que o compartilhamento é viável e que aplicações podem aproveitar ciclos de *CPU* não utilizados por outras aplicações em função da realização de operações de entrada e saída. Já no experimento da figura 5.10, as tarefas da figura 1 são *CPU intensive*, ou seja, utilizam a *CPU* durante todo o tempo. Nesse caso, a aplicação 2 fica dormindo até a aplicação 1 concluir sua execução. Neste experimento os tempos de execução das aplicações 1 e 2 foram 602,20 e 1029,25 respectivamente.

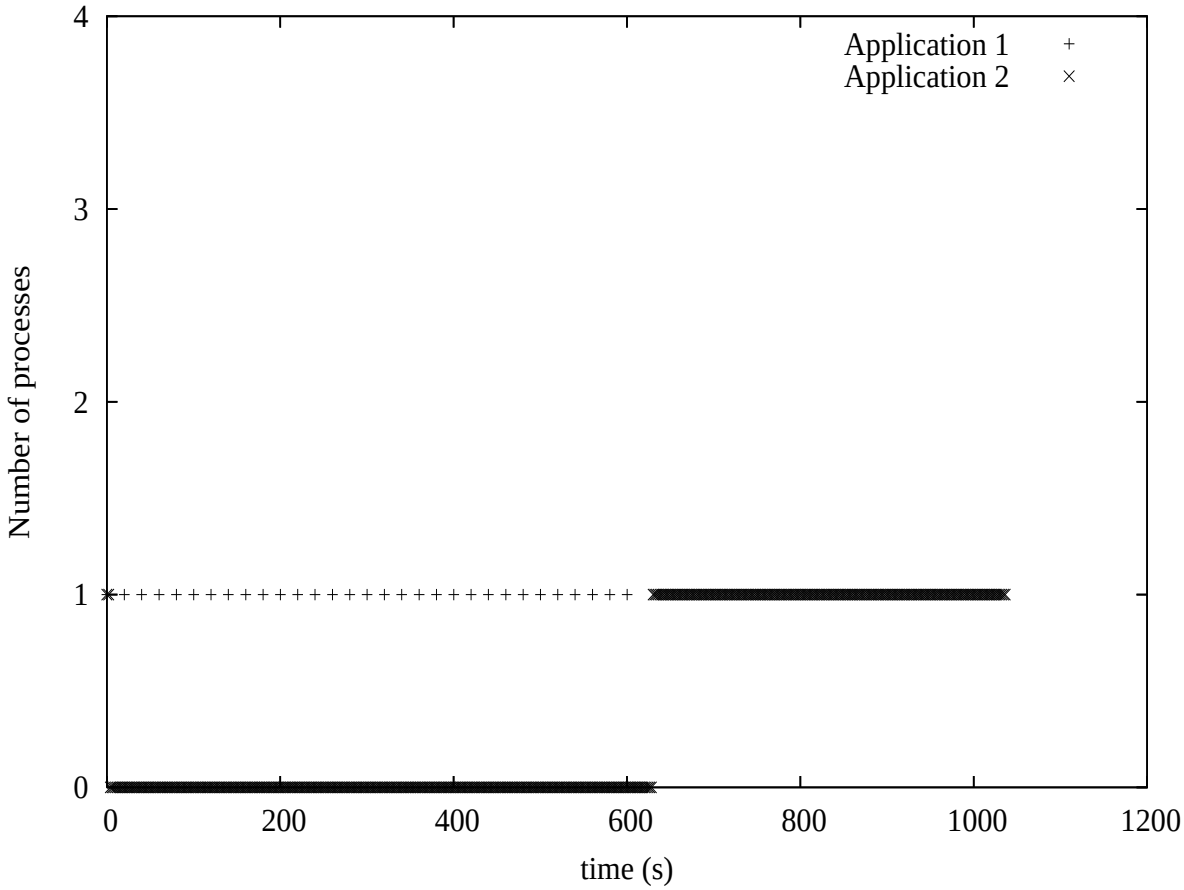

Figura 5.10: Variação do número de processos em execução das aplicações 1 e 2. As metas das aplicações 1 e 2 são 900 e 1200 respectivamente. Cada tarefa da aplicação 1 realiza 20 segundos de CPU e 0 de operações de E/S

5.5 Resumo

Esta seção apresentou uma avaliação teórica e empírica da implementação da heurística *Grid SA*. Através da análise teórica foi possível perceber que o comportamento proposto para esta heurística no capítulo 4 funcionou. As figuras 5.2 a 5.7 mostraram que através de metas aplicações podem compartilhar recursos simultaneamente sem introduzir grande sobrecarga ao sistema através de constantes trocas de contexto. Além disso, o comportamento consciente das aplicações que estão adiantadas permite que aplicações atrasadas se recuperem e alcancem suas metas, o que não seria possível caso todas adotassem a estratégia gulosa. A seção 5.3 fez uma análise quantitativa dos experimentos e mostrou que a sobrecarga introduzida pela heurística é pequena. Além disso, em todos os cenários onde uma aplicação poderia auxiliar uma outra aplicação a alcançar sua meta a heurística *Grid SA* mostrou-se eficaz. Assim, apesar de possuir pontos a evoluir, os resultados mostraram que a heurística *Grid SA* é viável e que o compartilhamento simultâneo de recursos com o objetivo de maximizar o uso dos mesmos pode produzir bons resultados quando comparados com a execução sequencial.

Capítulo 6

Conclusões e trabalhos futuros

Os sistemas gerenciadores oferecem ao usuário uma visão unificada dos recursos da grade. Eles são responsáveis por distribuir a aplicação do usuário entre os recursos, oferecendo uma execução eficiente. Os sistemas gerenciadores dos ambientes grade atuais funcionam como *brokers* que distribuem as aplicações dos usuários pelos recursos disponíveis mais apropriados. Nesta abordagem, adotando o modelo de gerenciamento de *clusters*, um recurso não será compartilhado por aplicações em um mesmo período de tempo, ou seja, um recurso será dedicado a uma única aplicação até que esta termine. O escalonamento do tipo *batch*, apesar de reduzir a sobrecarga por troca de contexto entre aplicações, pode subutilizar os recursos uma vez que estes oferecem diversos serviços que não serão usados o tempo todo pelas aplicações (múltiplas *CPUs*, *GPU*, *I/O*, etc.).

Este trabalho propôs uma nova estratégia para o gerenciamento de aplicações em grades computacionais sem a utilização de *brokers*. Nesta nova abordagem aplicações são autônomas e buscam o conjunto de recursos que irá oferecer a elas uma execução mais eficiente. A partir do momento que o número de aplicações que utilizam este ambiente cresce, maior será o compartilhamento dos recursos e consequentemente a perda introduzida pela troca de contexto destas aplicações. Dessa forma, uma outra questão avaliada neste trabalho foi como esse conjunto de aplicações autônomas podem alcançar seus objetivos sem prejudicar outras aplicações que utilizam os mesmos recursos ou subconjuntos de recursos. Assim, este trabalho propôs uma heurística que define o comportamento dessas aplicações para criar uma sociedade de aplicações autônomas capaz de executar em uma grade computacional de forma sustentável.

Inicialmente, com o objetivo de aprofundar os conhecimentos dos assuntos relacionados ao tema central deste trabalho uma pesquisa da literatura atual foi feita. Nesta pesquisa os temas grades computacionais, computação autônoma, agentes, sistemas geren-

ciadores de grades e *EasyGrid AMS* foram aprofundados. Dentre os temas estudados destaca-se a seção sobre o *EasyGrid AMS*. Este sistema gerenciador foi utilizado como base para este trabalho e a heurística proposta foi testada e validada através da implementação de um módulo para o mesmo. O capítulo 2 apresentou uma síntese dos estudos realizados sobre os temas citados.

O principal objetivo do capítulo 3 foi comparar o modo de execução dos sistemas gerenciadores de grades atuais e a estratégia autônoma do *EasyGrid*. Para isso diferentes cenários de execução foram testados. Os resultados mostraram que a sobrecarga introduzida pela estratégia do *EasyGrid AMS* é um pouco maior do que a abordagem tradicional. Esse resultado foi obtido porque o objetivo do *EasyGrid AMS* é minimizar o tempo de execução da aplicação. Logo, como todas as aplicações que compartilham o ambiente terão essa visão gulosa, a sociedade formada por esse conjunto de aplicações ficará insustentável. Assim, a fim de viabilizar essa sociedade foi proposta uma heurística para diferenciar o comportamento das aplicações. É importante destacar que a implementação da heurística *Grid SA* é totalmente descentralizada, ou seja, não há troca de informações diretas entre as aplicações. As aplicações se comunicam indiretamente através do monitoramento do meio onde elas executam. Assim, quando uma aplicação percebe que a carga de uma máquina onde ela está executando aumentou consideravelmente, ela irá deduzir que não está mais sozinha naquele recurso. Assim, fazendo uma analogia da sociedade de aplicações autônomas com a sociedade de seres humanos é possível dizer que as aplicações possuem o sentido da visão, já que conseguem perceber que uma outra aplicação está no mesmo recurso que ela está, mas não possuem a audição, já que não conseguem falar diretamente seu estado para as outras aplicações. Essa heurística foi chamada de *Grid SA* e apresentada no capítulo 4.

A heurística *Grid SA* indica quais ações uma aplicação deverá executar a partir das informações do ambiente em que ela executa e seu estado corrente. As informações do meio são capturadas pela camada de monitoramento do *EasyGrid* e o estado da aplicação é calculado a partir de sua meta de execução e do tempo passado do início da execução ao momento. Um ponto importante desta heurística é que, diferentemente da estratégia gulosa da versão original do *EasyGrid*, uma aplicação que esteja em um estágio adiantado de execução irá reduzir seu consumo de *CPU* e ceder espaço para outras aplicações que estejam compartilhando o mesmo recurso. Este capítulo também propôs uma implementação da heurística *Grid SA*.

O capítulo 5 apresentou uma avaliação experimental da heurística *Grid SA*. Esta

avaliação foi realizada sob as perspectivas qualitativas e quantitativas. O objetivo da avaliação qualitativa foi observar o comportamento da heurística em cenários controlados onde é possível deduzir os resultados finais. Nestes experimentos o comportamento da heurística mostrou-se correto conforme as metas das aplicações em questão variavam. Já a avaliação quantitativa comparou os resultados numéricos obtidos em relação ao tempo ótimo de execução e as metas de execução. Os resultados obtivos nesses experimentos mostraram que o *overhead* gerado pela heurística foi pequeno e que aplicações com metas críticas (atrasadas) podem se beneficiar quando executam junto com aplicações com metas mais relaxadas (adiantadas).

Em relação a trabalhos futuros, diversos estudos podem ser realizados a fim de melhorar a heurística *Grid SA* e permitir comportamentos mais sofisticados, por exemplo, uma aplicação perceber que outra aplicação que concorre por um mesmo recurso mudou para o estado atrasado quando há um *burst* no consumo de *CPU*. Uma possível idéia é introduzir o conceito de leilões a fim de classificar e priorizar aplicações. Dessa forma, quando um usuário submeter sua aplicação a grade, além de informar o tempo máximo de execução para sua aplicação ele irá informar um lance (valor monetário). Assim, de acordo com o lance do usuário um portal de submissão poderá informar o tempo de execução estimado. Caso o usuário não fique satisfeito com o tempo oferecido pelo portal, ele poderá fazer um lance maior. Uma outra idéia é definir estratégias de compartilhamento simultâneo para os gerenciadores *global* (GM) e *site* (SM) uma vez que este trabalho atuou apenas no gerenciador de *host* (HM). Assim, o SM poderá implementar heurísticas que habilite ou desabilite completamente um recurso de seu site para determinadas aplicações, a fim de reduzir a perda pela sobrecarga do compartilhamento.

Referências

[1] Amazon. Amazon elastic compute cloud (amazon ec2), 2009. Último acesso 21/11/2009.

[2] Amazon. Amazon simple storage service (amazon s3), 2009. Último acesso 21/11/2009.

[3] Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M., Werthimer, D. Seti@home: an experiment in public-resource computing. *Communications of the ACM 45*, 11 (2002), 56–61.

[4] Berman, F., Chien, A., Cooper, K., Dongarra, J., Foster, I., Gannon, D., Johnsson, L., Kennedy, K., Kesselman, C., Mellor-Crummey, J., Reed, D., Torczon, L., Wolski, R. The GrADS Project: Software support for high-level Grid application development. *The International Journal of High Performance Computing Applications 15*, 4 (2001), 327–344.

[5] Berman, F., Wolski, R. The AppLeS project: A status report. In *Proceedings of the 8th NEC Research Symposium* (Berlin, Germany, Maio 1997), Springer-Verlag.

[6] Boeres, C., Fonseca, A. A., Mendes, H. A., Menezes, L. T., Moura, N. T., Silva, J. A., Vianna, B. A., Rebello, V. E. F. An EasyGrid Portal for scheduling system-aware applications on computational grids. *Concurrency and Computation: Practice and Experience 18*, 6 (2005), 553–566.

[7] Boeres, C., Rebello, V. E. F. EasyGrid: Towards a framework for the automatic grid enabling of legacy MPI applications. *Concurrency and Computation: Practice and Experience 16*, 5 (Abril 2004), 425–432.

[8] Buyya, R. *High Performance Cluster Computing: Architectures and Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[9] Cao, J., Jarvis, S., Saini, S., Nudd, G. Gridflow: Workflow management for grid computing. In *Proceedings of 3rd International Symposium on Cluster Computing and the Grid (CCGrid 2003)* (Tóquio, Japão, Maio 2003), IEEE Computer Society, p. 198–205.

[10] da Silva, J. A., Rebello, V. E. F. Low cost self-healing in MPI applications. In *Proceedings of the 14th European PVM/MPI User's Group Meeting, Paris, France* (Outubro 2007), Springer, p. 144–152.

[11] de Araújo, A. P. F. *Paralelização Autonômica de Metaheurísticas em Ambientes de Grid*. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, RJ, Brasil, Março 2008.

[12] Erl, T. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[13] Ernemann, C., Hamscher, V., Schwiegelshohn, U., Yahyapour, R., Streit, A. On advantages of grid computing for parallel job scheduling. *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid* (Maio 2002), 39–39.

[14] Fosdick, L. D., Jessup, E. R., Schauble, C. J. C., Domik, G. *An introduction to high-performance scientific computing*. MIT Press, Cambridge, EUA, 1996.

[15] Foster, I., Kesselman, C. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing 11*, 2 (Summer 1997), 115–128.

[16] Foster, I., Kesselman, C., Eds. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[17] Foster, I., Kesselman, C., Eds. *The GRID 2: Blueprint for a New Computing Infrastructure*. 2^a edição. Morgan Kaufmann, 2004.

[18] Foster, I., Kesselman, C., Tuecke, S. The anatomy of the Grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications 15*, 3 (Agosto 2001), 200–222.

[19] Grand Challenges for Engineering. `http://www.engineeringchallenges.org/`, Último acesso 10/10/2008.

[20] Gregor, K. A., Laszewski, G., Mikler, A. R. Toward an architecture for ad hoc grids. In *in 12th International Conference on Advanced Computing and Communications (ADCOM 2004), Ahmedabad* (2004), p. 15–18.

[21] Horn, P. Autonomic computing: IBM's perspective on the state of information technology. `http://www.research.ibm.com/autonomic`, Outubro 2001.

[22] Kephart, J., Chess, D. The vision of autonomic computing. *IEEE Computer Magazine 36*, 1 (Janeiro 2003), 41–50.

[23] Kurdi, H., Li, M., Al-Raweshidy, H. A classification of emerging and traditional grid systems. *Distributed Systems Online, IEEE 9*, 3 (2008), 1.

[24] Laforenza, D. Grid programming: some indications where we are headed. *Parallel Comput. 28*, 12 (2002), 1733–1752.

[25] Laure, E., Hemmer, F., Prelz, F., Beco, S., Fisher, S., Livny, M., Guy, L., Barroso, M., Buncic, P., Kunszt, P. Z., Di Meglio, A., Aimar, A., Edlund, A., Groep, D., Pacini, F., Sgaravatto, M., Mulmo, O. Middleware for the next generation grid infrastructure. 4 p.

[26] Maghraoui, K. E., Desell, T. J., Szymanski, B. K., Varela, C. A. Dynamic malleability in iterative mpi applications. *Cluster Computing and the Grid, IEEE International Symposium on 0* (2007), 591–598.

[27] Message Passing Forum. MPI: A Message Passing Interface. Technical report, University of Tennessee, 1995.

[28] Mollick, E. Establishing moore's law. *IEEE Annals of the History of Computing 28*, 3 (2006), 62–75.

[29] Nascimento, A. P. *Escalonamento Dinâmico para Aplicações Autonômicas MPI em Grades Computacionais*. PhD thesis, Instituto de Computação, Universidade Federal Fluminense, Niterói, Brasil, Maio 2008.

[30] Nascimento, A. P., Sena, A. C., da Silva, J. A., Vianna, D. Q. C., Boeres, C., Rebello, V. Managing the execution of large scale MPI applications on computational grids. In *Proceedings of the 17th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2005)* (Rio de Janeiro, Brasil, Outubro 2005), IEEE Computer Society.

[31] Németh, Z., Sunderam, V. S. A comparison of conventional distributed computing environments and computational grids. In *ICCS '02: Proceedings of the International Conference on Computational Science-Part II* (London, UK, 2002), Springer-Verlag, p. 729–738.

[32] Parashar, M., Salim, H. Autonomic computing: An overview. In *In Proceedings of the Unconventional Programming Paradigms: International Workshop (UPP 2004)* (Le Mont St-Michel, França, 2005), Springer, p. 247–259.

[33] Sena, A. *Um Modelo Alternativo para Execução Eficiente de Aplicações Paralelas MPI nas Grades Computacionais*. PhD thesis, Instituto de Computação, Universidade Federal Fluminense, Niterói, Brasil, Novembro 2008.

[34] Sterritt, R., Parashar, M., Tianfield, H., Unland, R. A concise introduction to autonomic computing. *Advanced Engineering Informatics 19*, 3 (Julho 2005), 181–187.

[35] Stevens, W. R. *TCP/IP illustrated (vol. 1): the protocols*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[36] Thomas, M. S., Friese, T., Freisleben, B. Towards a service-oriented ad hoc grid. In *In Proceeedings of the 3rd International Symposium on Parallel and Distributed Computing* (2004), IEEE Press, p. 201–208.

[37] Top 500 Supercomputer Sites. `http://top500.org`, Último acesso 29/10/2009.

[38] Van Parunak, H. "go to the ant": Engineering principles from natural multi-agent systems. *Annals of Operations Research 75*, 0 (January 1997), 69–101.

[39] Vianna, D. Q. C. Um sistema de gerenciamento de aplicações MPI para ambientes grid. Tese de Mestrado, Instituto de Computação, Universidade Federal Fluminense, 2005.

[40] Wooldridge, M. *Introduction to MultiAgent Systems*. John Wiley & Sons, June 2002.

[41] Wooldridge, M., Jennings, N. R. Intelligent agents: Theory and practice. HTTP://www.doc.mmu.ac.uk/STAFF/mike/ker95/ker95-html.h (Hypertext version of Knowledge Engineering Review paper), 1995.