Trabalho de Formatura Supervisionado

# Escalonamento de aplicações utilizando análise de padrões de uso no *InteGrade*

**Thiago Henrique Coraini**

**Orientador:** Marcelo Finger

**Departamento de Ciência da Computação**

**Instituto de Matemática e Estatística**

**Universidade de São Paulo**

**Dezembro de 2008**

# Sumário

# Parte I

# Parte técnica

# Capítulo 1

# Introdução

Neste trabalho, descreveremos as atividades realizadas durante uma iniciação científica dentro do projeto *InteGrade*, sob a orientação do professor Marcelo Finger. O objetivo principal era desenvolver um escalonador para uma grade computacional oportunista, que utilizasse de informações sobre a utilização futura dos computadores, obtidas através de análises de padrões de uso, para tomar decisões que preservassem os donos de máquinas compartilhadas com a grade.

Primeiramente daremos uma introdução dos temas relacionados ao trabalho, como o projeto *InteGrade* em si e alguns conceitos importantes em grades computacionais. Depois, descreveremos a arquitetura da grade e, especificamente, o funcionamento do módulo responsável pela análise dos padrões de uso e pela realização das previsões. A seguir, discutiremos a implementação do escalonador e sua integração com o resto do sistema. Por fim, apresentaremos os resultados do trabalho desenvolvido, além de idéias para possíveis trabalhos futuros.

## 1.1 O *InteGrade*

O projeto *InteGrade* é uma iniciativa entre várias universidades brasileiras que procura desenvolver um *middleware* para a implantação de grades computacionais oportunistas [1] [2] [3]. A idéia é que, usando o *InteGrade*, empresas ou organizações possam obter um alto poder de processamento, talvez próximo àquele conseguido com o uso de supercomputadores, porém sem a necessidade de aquisição de novos equipamentos. Isso permitiria a execução de aplicações de grande custo computacional, como aplicações científicas, porém sem demandar muitos gastos para as instituições.

Ao invés disso, esse grande poder computacional seria alcançado utilizando-se toda a infra-estrutura de computadores já existente nessas organizações. Isso significa combinar o potencial de diversos computadores comuns, como *desktop's*, para obter-se uma grande capacidade de computação. A essa malha de computadores, ligados em rede e capazes de executarem coordenadamente aplicações (possivelmente distribuídas) submetidas a partir de algum computador, chamamos de grade computacional.

Como dito, o *InteGrade* propõe formar uma grade computacional com computadores comuns, já existentes e sendo utilizados para outros fins. Portanto, é importante que a execução de aplicações submetidas à grade não comprometa a utilização desses computadores para a sua finalidade original. É aqui que entra o conceito de oportunismo na grade proposta pelo *InteGrade*. Todos os computadores compartilhando recursos com a grade somente devem ser usados por esta nos momentos em que não estiverem sendo usados por

seus usuários locais, ou seja, apenas em momentos de ociosidade.

É importantíssimo que esse compromisso seja atendido na maior parte do tempo. Como a grade será formada, basicamente, por computadores comuns sendo compartilhados por seus usuários, é evidente que esses devem ter completa prioridade sobre o uso das máquinas. Caso uma intensa degradação na performance começasse a ser sentida, rapidamente usuários começariam a retirar seus computadores da grade, e essa não teria como continuar executando suas aplicações. Portanto, o *InteGrade* baseia fortemente seu desenvolvimento na idéia de que os usuários locais dos computadores compartilhados têm prioridade máxima sobre os mesmos.

## 1.2 Escalonamento em grades oportunistas

Como discutido na seção anterior, uma grade computacional oportunista deve manter uma garantia vital aos usuários que compartilham seus computadores com a grade: eles não irão perceber nenhuma degradação na performance de sua máquina, devido à execução de aplicações da grade ali enquanto eles próprios a estivessem utilizando.

Isso certamente traz um problema sério ao escalonador de aplicações do *InteGrade*. Garantir simplesmente que as aplicações sejam submetidas para máquinas ociosas claramente não é o bastante, já que em geral essas aplicações demoram algum tempo para executar e, nesse tempo, possivelmente a máquina venha a ser utilizada localmente. Deve-se, portanto, garantir que a aplicação não atrapalhe os processos do usuário caso ele retorne ao computador durante sua execução. Várias abordagens podem ser implementadas, como fazer com que aplicações da grade executem com prioridade mínima, ou pausá-las no momento do retorno do usuário, para serem reinicidas posteriormente ou mesmo migradas para algum computador que esteja ocioso no momento. No *InteGrade*, atualmente, apenas a execução da aplicação com prioridade mínima é dada como garantia de que esta não irá atrapalhar o usuário local.

Todavia, apenas a adoção de uma ou mais das abordagens acima não parece o suficiente. Caso fosse comum uma aplicação ter que ser interrompida pelo aparecimento repentino do usuário no computador onde esta executa, certamente a eficiência da grade em geral seria muito baixa, e a maior parte das execuções levaria muito mais tempo que o necessário para encerrar.

Dado esse cenário, não fica difícil perceber que o ideal, para o escalonamento, seria poder prever, com uma boa taxa de acerto, como seria a taxa de utilização de cada máquina num futuro próximo. Isso certamente permitiria ao escalonador tomar decisões mais "conscientes", evitando que aqueles computadores que não permaneceriam ociosos por muito tempo fossem utilizados na execução de uma aplicação submetida.

Dentro do projeto *InteGrade*, foi proposto por Germano Bezerra a utilização de análise de padrões de uso de um computador para prever seu uso futuro [4]. Basicamente, a idéia consiste em monitorar o uso de uma máquina e detectar padrões de utilização que se repetem constantemente. Assim, dado um certo instante do dia, poderíamos verificar, pela utilização recente, qual padrão de utilização está ocorrendo no momento, e então, assumindo que esse padrão irá manter-se, decidir como estaria a utilização da máquina pelas próximas horas.

Com essa informação, o escalonador poderia tomar decisões que preservassem os usuários locais das máquinas compartilhadas com a grade ao mesmo tempo em que aumentassem a eficiência geral da grade, permitindo que as aplicações terminassem mais rapidamente.

## 1.3 Objetivos do trabalho

Além do trabalho desenvolvido por Bezerra, como citado anteriormente, o *InteGrade* ganhou a implementação de um módulo responsável pela realização das atividades de monitoramente e análise da utilização dos computadores, capaz de realizar previsões sobre o uso futuro. Esse módulo, chamado *LUPA – Local Usage Pattern Analyzer*, já estava previsto desde o começo do desenvolvimento do *InteGrade*, e foi finalmente implementado por Danilo Conde como seu trabalho de mestrado [5].

Todavia, essa implementação não foi integrada de maneira satisfatória ao resto do sistema, e na prática o *LUPA* não estava sendo utilizado, apesar de todo seu potencial. Esse trabalho tinha como objetivo inicial, portanto, realizar essa integração, mesmo que a princípio de uma maneira um tanto quanto rudimentar.

Além disso, o trabalho visava também realizar alguns aprimoramentos no *LUPA*, como serão discutidos posteriormente. Esses aprimoramentos incluiam tanto a extensão de algumas funcionalidades como algumas mudanças na implementação do que já existia de maneira a propiciar um melhor funcionamento ao módulo.

Por fim, com a finalização dessas melhorias no *LUPA*, o objetivo era a implementação de um escalonador para a grade que utilizasse as previsões de uso futuro realizadas pelo módulo para tomar suas decisões. Isso era o que a grade de fato necessitava, já que até então, mesmo com a existência do *LUPA*, ele não era considerado no momento do escalonamento. Portanto, essa era uma tarefa de grande relevância dentro do projeto *InteGrade*.

# Capítulo 2

# Arquitetura do *InteGrade*

Não é o objetivo deste trabalho fazer uma descrição detalhada da arquitetura do *InteGrade*, nem dar detalhes de seu funcionamento. Todavia, dado que o trabalho se inseriu totalmente neste sistema, achamos necessário dar uma visão geral sobre como ele foi construído e descrever brevemente seus principais módulos e sua integração.

Este capítulo dará, portanto, apenas uma visão superficial do funcionamento da grade. Maiores detalhes podem ser encontrados em trabalhos contidos na bibliografia, especialmente em [2] e [3], e também no site do projeto [1].

## 2.1 Arquitetura de aglomerados

A arquitetura do *InteGrade* baseia-se no conceito de aglomerados (*clusters*), o que significa que os computadores que compõe a grade são agrupados em conjuntos de acordo com algum critério. Normalmente, esse critério é a localização física das máquinas, ou seja, aquelas que estão num mesmo laboratório em geral farão parte de um mesmo aglomerado. Isso, todavia, não é obrigatório, e aglomerados podem ser formados seguindo qualquer regra que desejar-se.

Dentro de cada aglomerado, cada máquina assume um papel específico, de acordo com sua utilização dentro da grade. Portanto, um computador na grade pode ser classificado como:

- **Gerenciador do Aglomerado:** É o nó que coordena o funcionamento daquele aglomerado. Ele mantém o controle sobre todos os outros nós, é quem recebe as aplicações submetidas e realiza seu escalonamento para os nós que compartilham recursos com a grade, devolvendo posteriormente os resultados.

- **Nó Dedicado:** É uma máquina que existe exclusivamente para executar aplicações para a grade. Nesse caso, não se enquadra no conceito de grade oportunista, porém nada impede que nós dedicados sejam usados para aumentar a eficiência da grade.

- **Nó Compartilhado:** Esse é o computador no qual se baseia toda a idéia do oportunismo do *InteGrade*. Basicamente, é um computador comum, que é utilizado localmente para outros fins, porém que, nos momentos de ociosidade, será aproveitado pela grade para executar suas aplicações.

- **Nó de Usuário:** Esse nó representa o meio de acesso de algum usuário à grade. É através dele que as aplicações são submetidas à grade, para então serem distribuídas

aos nós compartilhadores de recursos. Será também através desse nó que o usuário da grade recuperará posteriormente os resultados de sua aplicação.

Dada essa classificação dos nós, é muito importante deixar claro que um mesmo computador pode assumir mais de um papel dentro dela. Isso significa, por exemplo, que um **Nó Compartilhado** pode, simultaneamente, ser também um **Nó de Usuário**. Ou seja, ao mesmo tempo em que o computador compartilha seus recursos com a grade, ele também permite que aplicações sejam submetidas por ele. Até mesmo o nó **Gerenciador de Aglomerado** pode, sem problemas, assumir o papel de compartilhador de recursos e/ou de **Nó de Usuário**.

## 2.2 Os módulos do *InteGrade*

Dependendo de quais papéis um determinado computador irá assumir, segundo a classificação dada na seção anterior, ele deverá rodar um ou mais módulos que executem as tarefas adequadas à sua finalidade.

A Figura 2.1 mostra uma estrutura básica dos tipos de nós que compõe o *InteGrade*, e os módulos que cada um deve executar. A seguir, daremos uma descrição resumida de cada um desses módulos.

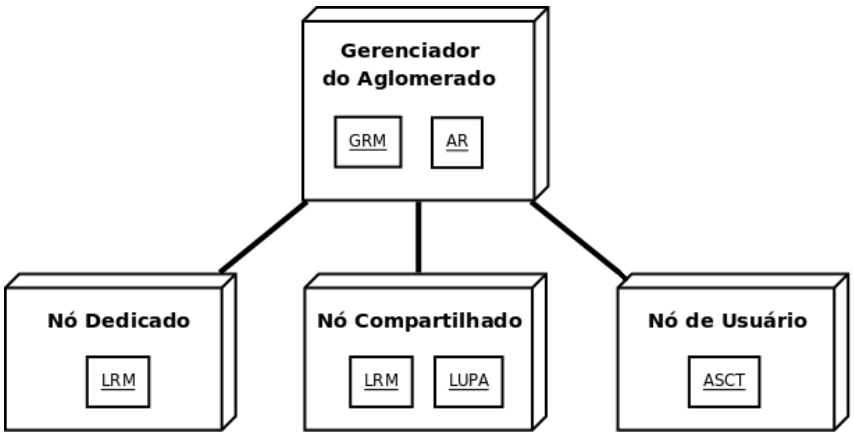


Figura 2.1: Tipos de nós do *InteGrade* e seus respectivos módulos

**GRM – Global Resource Manager:** Esse módulo é o responsável por guardar informações sobre todos os nós que compartilham recursos com a grade. Sempre que um novo nó desse tipo deseja participar da grade, ele realiza um registro junto ao **GRM**, passando suas características. Além disso, periodicamente o **GRM** verifica como está a disponibilidade desses nós, para decidir se eles seriam candidatos a executar alguma aplicação. No momento da submissão de uma aplicação, é o **GRM** quem terá informações sobre todos os nós disponíveis para executá-la, e portanto é aqui onde irá inserir-se o escalonador desenvolvido nesse projeto.

**AR – Application Repository:** Esse módulo, que também executa no nó gerenciador, é quem armazena as aplicações a serem executadas na grade. Quando um usuário submete uma aplicação, ela é copiada para o **AR**, e, depois de escolhido o nó que irá executá-la, este a requisita ao **AR**.

**LRM – Local Resource Manager:** É esse o nó que controla a execução das aplicações da grade localmente, nas máquinas compartilhadas. Ele é responsável, primeiramente, por registrar o computador junto ao **GRM**, e posteriormente realizar atualizações periódicas de suas informações junto ao nó gerenciador. Além disso, é também o **LRM** quem executa as aplicações localmente: ao ser escalonado pelo **GRM**, ele busca a aplicação no **AR** e dispara um novo processo na máquina para executar tal aplicação, informando posteriormente o fim de sua execução.

**LUPA – Local Usage Pattern Analyzer:** Como já mencionado anteriormente, foi aqui que se concentraram as atividades realizadas neste trabalho de conclusão de curso. Portanto, uma descrição mais detalhada do **LUPA** será dada no próximo capítulo.

**ASCT – Application Submission and Control Tool:** Esse módulo serve de "porta de entrada" da grade para um usuário. Ele oferece uma interface gráfica na qual o usuário pode submeter aplicações e requisitar sua execução, definindo inclusive restrições ou preferências que julgar necessárias. Essa interface manterá o usuário informado sobre o estado da execução e, ao seu término, permitirá que este visualize os resultados obtidos.

## 2.3 Execução de Aplicações

Descreveremos aqui, brevemente, como era realizada a execução de uma aplicação no *InteGrade*, antes da implementação do escalonador que utiliza as informações providas pelo **LUPA**. A Figura 2.2[1] exemplifica o protocolo utilizado durante a execução.

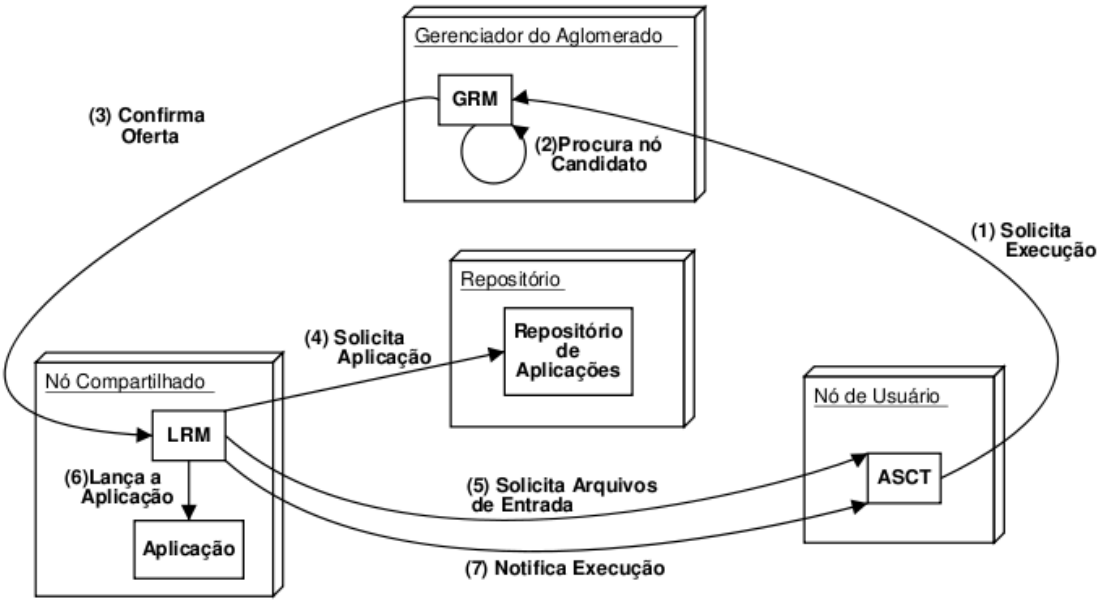


Figura 2.2: Passo-a-passo da execução de uma aplicação no *InteGrade*

Primeiramente, o usuário solicita a execução de uma aplicação através do **ASCT** (1). O **GRM** então procura, entre os **LRM's** registrados, candidatos que atendam os

[1]Imagem originalmente em [2], gentilmente cedida por Andrei Goldchleger.

requisitos da aplicação (2). Após escolhido, verifica se esse nó está disponível (3). Caso esteja, o **LRM** aceita a requisição, busca a aplicação no **AR** (4), solicita possíveis arquivos de entrada ao **ASCT**, e executa a aplicação localmente (6), avisando ao **ASCT** que requisitou a execução que a aplicação está rodando ali (7).

As modificações implementadas nesse trabalho, envolvendo o escalonamento, dizem respeito principalmente ao passo (2), ou seja, a escolha do melhor nó candidato a executar a aplicação. Isso, implicitamente, acaba envolvendo o passo (3), já que agora a confirmação de ociosidade será feita levando-se em conta as informações sobre utilização futura disponibilizadas pelo **LUPA**.

# Capítulo 3

# O módulo *LUPA*

Como discutido anteriormente, as máquinas que compartilham recursos com a grade (chamadas de Nós Compartilhados) rodam o módulo *LUPA*, que é encarregado de realizar um monitoramento e análise da utilização daquele nó para, posteriormente, ser capaz de dar informações sobre como estará a taxa de utilização daquele computador num futuro próximo.

A implementação atual do *LUPA* foi desenvolvida pelo aluno Danilo Conde [5]. A seguir, descreveremos o funcionamento do *LUPA*, além de dar alguns detalhes sobre sua implementação. Mostraremos também quais as principais alterações realizadas no trabalho inicial de Conde, e também quais extensões de funcionalidades foram implementadas de maneira a permitir o desenvolvimento do escalonador.

## 3.1 Funcionamento do *LUPA*

O *LUPA* busca prever a utilização futura de computadores através de técnicas de *clustering* (análise de conglomerados) [4]. A idéia é identificar alguns comportamentos recorrentes na utilização do computador e agrupá-los, por semelhança, em aglomerados, ou *clusters*. Ou seja, a técnica baseia-se na idéia de que uma mesma máquina possui um padrão de uso razoavelmente bem definido como, por exemplo, ser utilizada em dias de semana pela manhã e tarde, e ficar livre durante as noites e nos finais de semana. No momento de realizar uma previsão, então, o primeiro passo seria identificar, através da utilização recente daquela máquina, em qual desses comportamentos típicos ela encontra-se nesse momento, e então verificar, através das observações anteriores, como irá variar a utilização nas próximas horas.

A primeira atribuição do *LUPA* é monitorar o uso do computador. Em intervalos de tempo pré-definidos (por padrão, 5 minutos), realiza-se uma verificação de como está a utilização da máquina naquele instante. Atualmente, o *LUPA* monitora apenas os usos de CPU e memória RAM, sendo o primeiro medido em porcentagem livre de acordo com a média de utilização dos últimos 5 minutos, e o segundo medido em valores absolutos, em KB. Seria interessante, no futuro, que mais recursos tivessem seu uso monitorado, mas claramente esses dois são os mais importantes na definição da ociosidade de uma máquina.

Os dados sobre a utilização que estão sendo monitorados vão sendo gravados em arquivos de *log*, para que essas medições não sejam perdidas mesmo que a máquina seja desligada. Uma das primeiras alterações na implementação de Conde [5] diz respeito justamente a esse arquivo de *log*. Anteriormente, ele chamava-se apenas `lupa.log`. Porém

caso o *InteGrade* estivesse sendo executado num sistema de arquivos NFS, com todos os *LUPA's* rodando a partir do mesmo diretório, todos tentariam escrever no mesmo arquivo. Isso, além de produzir resultados incorretos, poderia causar problemas de concorrência no acesso ao arquivo de *log*. A solução encontrada foi incluir, no nome do arquivo, o nome da máquina a qual diz respeito aquele *log*. Assim, o arquivo de *log* da máquina `lisa`, por exemplo, deveria chamar-se `lupa.lisa.log`.

Uma vez ao dia, então, o *LUPA* executa um algoritmo sobre toda essa massa de dados coletados desde a primeira vez em que foi executado. É esse algoritmo que irá agrupar os comportamentos semelhantes em *clusters*. Como essa execução é diária, o que acontece é que os *clusters* possivelmente irão mudando, acompanhando mudanças que porventura ocorram no padrão de utilização da máquina.

No momento de comparar-se a utilização recente com os *clusters*, a fim de verificar em qual padrão encontra-se o uso atual, devemos ter uma maneira de comparar esses dados, a princípio incompatíveis. Para isso, cada *cluster* possui um elemento representativo, que não passa de um elemento que é a média dos comportamentos pertencentes àquele *cluster*. É com esse elemento, portanto, que a utilização das últimas horas será comparado.

Um detalhe importante a ser ressaltado é que os dados monitorados pelo *LUPA* nunca deixam a máquina local. Isso envolve uma preocupação com a privacidade dos usuários que compartilham seus computadores com a grade. Esses usuários provavelmente não veriam com bons olhos o fato de toda a utilização do computador estar sendo constantemente monitorada, e que esses dados estariam trafegando pela rede para um outro computador.

Porém, já que esses dados são mantidos junto ao *LUPA*, é preciso definir-se uma interface que permita fazer consultas ao módulo sobre a utilização futura daquele computador. É através dessa interface que o escalonador irá decidir quais os computadores mais indicados a executar uma determinada aplicação. Portanto, a interface deve disponibilizar métodos variados de questionar o *LUPA* sobre o estado futuro da máquina. Os métodos que compõe a interface do *LUPA* serão discutidos em detalhes na Seção 3.3.

## 3.2 Detalhes de implementação

O *LUPA*, assim como todos os módulos do *InteGrade* que rodam nas máquinas compartilhadas, foi implementado em C++, de maneira a causar a menor sobrecarga possível na máquina. O módulo não possui um método `main` em nenhuma classe. Ao invés disso, o módulo *LRM*, ao ser iniciado, instancia um objeto da classe `Lupa`, e a partir desse momento o *LUPA* começa a executar suas obrigações. O *LRM* realiza então as consultas através dessa instância.

Uma das principais classes do módulo é a chamada `UsageData`. Essa classe representa a utilização do computador num período de 48 horas. Internamente, ela possui um vetor que guardará a medida de utilização de CPU e de memória para cada intervalo de 5 minutos nesse período de 2 dias consecutivos. Enquanto o uso está sendo monitorado, um `UsageData` vai sendo preenchido, sempre na segunda metade (últimas 24 horas). Ao término de um dia, essa segunda metade passa a ser a primeira metade de um novo `UsageData`, e começamos a preenchê-lo a partir do começo da segunda metade. Essa abordagem de guardar os dados em estruturas de 48 horas serve para garantir que, ao estarmos preenchendo um `UsageData`, em qualquer momento do dia, teremos sempre as medições das últimas 24 horas para serem consultadas.

São instâncias da classe `UsageData` que representam os comportamentos recorrentes do computador. São elas, portanto, que serão agrupadas por semelhança em *clusters*, cada

um representando um uso típico. A maneira utilizada para definir-se a semelhança entre 2 `UsageData's` é o cálculo de uma distância euclidiana entre eles, sendo as coordenadas cada uma das medições realizadas. O elemento representativo de cada *cluster*, como discutido anteriormente, não passa de um `UsageData` também, que originalmente não se encontrava no *cluster*, mas que foi criado artificialmente com a média dos `UsageData's` daquele aglomerado.

No momento de uma previsão, compara-se, no `UsageData` atual, as últimas 24 horas de monitoramento, com os elementos representativos de cada *cluster*. Ao encontrar-se aquele mais próximo, assume-se que o uso futuro será aquele descrito pelos valores contidos no elemento representativo selecionado, a partir da hora atual. Caso o *LUPA* esteja executando há pouco tempo, e não tenha no `UsageData` atual as últimas 24 horas monitoradas, o algoritmo toma uma abordagem diferente, fazendo uma média do uso das últimas horas e assumindo que aqueles valores irão se manter.

O procedimento realizado para agrupar os `UsageData's` em *clusters* é realizado uma vez ao dia, como mencionado anteriormente. Para isso é utilizado o chamado algoritmo das $k$-Médias. O que o algoritmo faz é escolher arbitrariamente $k$ `UsageData's` e definir $k$ *clusters*, cada um contendo um dos `UsageData's`. Então, ele coloca os elementos restantes no *cluster* mais próximo de cada um, e repetidas vezes percorre os elementos realocando-os para o *cluster* mais próximo, até que o algoritmo convirja para um estado onde todo `UsageData` já está dentro do *cluster* mais apropriado. No caso da implementação atual do *LUPA*, o valor de $k$ é 4.

A Figura 3.1 demonstra simplificadamente o funcionamento do *LUPA*.

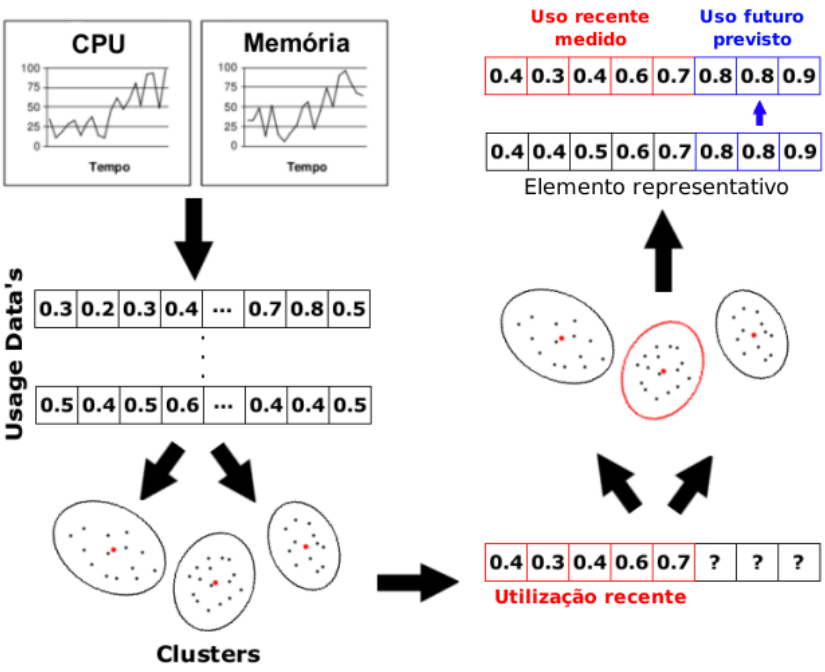


Figura 3.1: Funcionamento do *LUPA*: O uso é monitorado constantemente, e os dados são guardados em `UsageData's`. Estes são agrupados, por semelhança, em *clusters*. No momento de realizar uma previsão, procura-se o *cluster* mais próximo da utilização nas últimas 24 horas. A partir do elemento representativo desse *cluster*, infere-se o estado da máquina nas próximas horas.

### 3.2.1 O problema das previsões no fim do dia

Apesar da versão simplificada, a Figura 3.1 demonstra um problema grave que ocorria com o método de realizar previsões do *LUPA*. Nos últimos passos, para definir como seria a utilização futura, primeiro escolhíamos o *cluster* apropriado, e então, a partir de seu elemento representativo, completávamos o `UsageData` atual com as informações sobre as próximas horas. Contudo, se lembrarmos que cada `UsageData` representa um período de 48 horas consecutivas, veremos que a quantidade de informação que temos para realizar a previsão é extremamente limitada, como mostra a Figura 3.2.

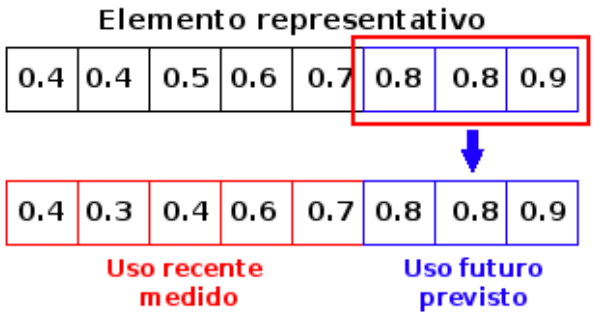


Figura 3.2: Limitação no número de horas a serem previstas no *LUPA*

De fato, caso a previsão fosse realizada às 20 horas, por exemplo, somente teríamos a disposição uma previsão pelas próximas 4 horas. Isso porque, ao recuperar o `UsageData` que conterá os dados a serem usados como previsão, esse terá, a partir do índice correspondente à hora atual, apenas essa quantidade de informação disponível. E os `UsageData's` não se relacionam temporalmente, assim fica impossível saber qual seria o padrão que viria após o fim do dia representado por aquele `UsageData`.

Esse fato impunha uma séria limitação a usabilidade do *LUPA* dentro do *InteGrade*. Isso porque a idéia de uma grade computacional é justamente reunir recursos capazes de executar aplicações pesadas, que exijam uma grande quantidade de computação. Essas aplicações, mesmo executando numa grade, muitas vezes demorarão um tempo considerável para terminar, possivelmente diversas horas (ou até dias). Caso o *LUPA* tivesse uma restrição tão grande no número de horas possíveis de serem previstas, e ainda tão dependente da hora de submissão da aplicação, isso certamente comprometeria seriamente as atividades da grade.

O que propusemos então foi uma modificação no algoritmo anteriormente desenvolvido por Conde, de forma a permitir que previsões fossem realizadas por um número de horas, em teoria, ilimitado. Claro que a pretensão nunca foi realizar previsões para períodos muito grandes no futuro, como semanas, o que poderia comprometer seriamente a taxa de acerto das previsões. A idéia, porém, era contornar a grande limitação imposta anteriormente.

A diferença do novo algoritmo é que ele realiza diversas vezes o procedimento descrito nas seções anteriores, até que o número de horas desejadas na previsão seja atingido. Chamamos essa técnica de **previsões encadeadas**, pois cada nova previsão irá utilizar os resultados da previsão anterior, e no final os resultados serão combinados.

A princípio, uma previsão tradicional é realizada, para o período de tempo máximo permitido pela restrição do algoritmo descrito anteriormente, ou seja, até o fim do dia atual.

Após isso, o *LUPA* passa a agir como se o tempo tivesse sido adiantado para às 00:00 do dia seguinte. Internamente, o tempo é controlado no *LUPA* através da classe `Timestamp`. Ou seja, no algoritmo, um `Timestamp` para esse horário é criado para representar o momento em que foi requisitada a previsão (supostamente no início do próximo dia).

Porém, caso estivéssemos de fato no início do dia seguinte, estaríamos preenchendo um `UsageData` bem no começo de sua segunda metade, ou seja, teríamos a disposição 24 horas de utilização medida para serem comparadas com os dados nos *clusters*. Mas, como sabemos, não podemos ter essa informação, afinal ainda não estamos de fato no horário simulado.

A solução então é utilizar como utilização recente uma combinação do que foi efetivamente medido nas últimas horas, até o instante atual, com as informações devolvidas pela primeira previsão (lembre-se de que ela foi feita exatamente até o final do dia). Na prática, isso significa criar um `UsageData` em que a primeira metade é exatamente a segunda metade do `UsageData` devolvido pela previsão anterior.

Perceba que isso implica que estamos fazendo uma suposição extremamente otimista, já que estamos assumindo que a utilização nas próximas horas de fato será igual ao resultado da previsão. Porém, supondo que as previsões mantêm um bom nível de acerto, podemos contar que a discrepância existente não é tão prejudicial ao resultado final.

A partir então dessa utilização recente "simulada", realizamos uma nova previsão, exatamente do mesmo modo que no método tradicional. Agora teremos a disposição 24 horas para serem previstas. Caso esse tempo, somado ao previsto na primeira etapa, ainda não seja suficiente, poderemos repetir o procedimento quantas vezes forem necessárias. Claro que devemos tomar bastante cuidado com isso, já que encadear diversas previsões pode significar um acúmulo dos erros gerados em cada previsão, a princípio não tão grandes, porém que se somados podem causar uma grande distorção no resultado final.

Ao atingir-se o número de horas desejadas, o algoritmo devolve a resposta que corresponde a uma combinação das respostas de todas as previsões realizadas. Na Seção 3.3 discutiremos esses tipos de respostas, e daremos algum detalhe de como elas são geradas utilizando-se as previsões encadeadas.

A Figura 3.3 demonstra de maneira geral o procedimento realizado.

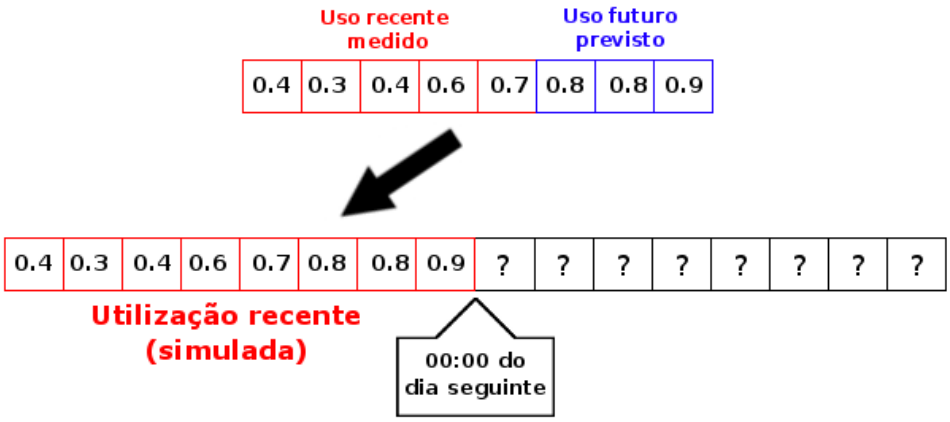


Figura 3.3: Técnica de previsões encadeadas no *LUPA*

## 3.3 Interface do *LUPA*

Como discutido anteriormente, uma grande preocupação na implementação do módulo *LUPA* era quanto à privacidade dos usuários que compartilham suas máquinas com a grade. Afinal, o sistema irá monitorar constantemente a utilização do computador e guardar *logs* com essas informações. É claro que muitas pessoas não veriam com bons olhos esses dados sendo expostos a qualquer um.

Por isso, foi definido que as informações sobre a utilização do computador não deixariam a máquina local. A arquitetura original do *InteGrade* chegou a prever a existência de um outro módulo, chamado *GUPA − Global Usage Pattern Analyzer*, que rodaria no nó gerenciador do aglomerado, e seria responsável por fazer uma análise dos padrões de uso de posse de uma visão global sobre a utilização de todas as máquinas de seu aglomerado. Devido a essa preocupação com a privacidade, no entanto, esse módulo não chegou a ser implementado.

Dessa forma, o *LUPA* realiza todo o processamento localmente. Precisa, portanto, disponibilizar uma interface que permita fazer consultas sobre as previsões realizadas em cima da base de dados coletados. Umas das opções seria o *LUPA* simplesmente devolver a descrição completa da utilização para o período requisitado, ou seja, devolver um vetor (nos moldes de um `UsageData`) que dissesse como estaria o uso da máquina em cada instante dentro de intervalos de, por exemplo, 5 minutos.

Todavia, algumas outras maneiras de consultar-se o *LUPA* foram implementadas, de modo a realizar algum processamento local nessas informações ao invés de simplesmente enviá-las para serem analisadas no *GRM*.

É importante ressaltar que essas consultas ao *LUPA* são feitas sempre levando-se em conta alguma medida de disponibilidade. Isso significa que não perguntamos simplesmente se a máquina estará ociosa pelas próximas, digamos, 8 horas. Devemos informar o que é considerado ociosidade, ou seja, devemos dizer quanto de CPU e memória esperamos que a máquina mantenha disponível pelas próximas 8 horas.

Descreveremos a seguir os métodos de consulta ao *LUPA*, e provavelmente essa idéia ficará mais clara. Porém, uma questão importante é como obter essas informações de disponibilidade, ou seja, como saber o quanto de recursos devem estar livres para rodar uma determinada aplicação num determinado espaço de tempo. Apesar de não ser o foco dessa trabalho, isso será discutido brevemente no Capítulo 4.

### O método *canRunGridApplication*

O primeiro método de consulta ao *LUPA* chama-se *canRunGridApplication*. Como o nome sugere, é um método *booleano*, que devolve apenas **SIM** ou **NÃO** indicando se uma aplicação poderia rodar naquele nó dada sua previsão de disponibilidade. Seu protótipo é:

```
boolean canRunGridApplication(freeCpu, freeRam, hours)
```

Como pode-se notar, devemos passar ao método a quantidade esperada de CPU e memória livres, e o número de horas pelas quais desejamos que essa quantidade de recursos mantenha-se disponível. O algoritmo realizará então uma previsão, como descrita anteriormente, para esse número de horas. Se algum dos valores previstos (eles são dados em intervalos de 5 minutos) não satisfizer os mínimos requisitados, o algoritmo responde **NÃO**; caso contrário, responde **SIM**.

Esse algoritmo já existia na implementação de Conde [5], porém foi adaptado para utilizar a técnica das previsões encadeadas. Nesse caso, a vantagem de realizar-se a análise das previsões no próprio *LUPA* é que o algoritmo é interrompido ao detectar o primeiro valor abaixo do mínimo esperado, respondendo negativamente. Assim, o processo de encadeamento não necessariamente precisa executar até o fim.

### O método *getPrediction*

Esse método também foi implementado por Conde. Como exemplificado anteriormente, esse método devolve a previsão completa para as próximas horas, deixando a cargo de quem o invocou fazer uma análise dos dados de acordo com sua necessidade. Seu protótipo é:

```
double[] getPrediction(resource, hours)
```

Para esse método, é necessário especificar em qual recurso estamos interessados (como dito, no momento os únicos possíveis são CPU e memória RAM). Definimos, também, o número de horas de nosso interesse. O algoritmo realiza então a previsão e devolve um vetor contendo os valores previstos para todo o período requisitado, em intervalos de 5 minutos.

Esse método é o único que ainda não foi adaptado para utilizar a abordagem das previsões encadeadas, ou seja, ele ainda possui a limitação no número de horas a serem previstas. Isso se deve ao fato que ele é o único que não foi utilizado pelos algoritmos do escalonador. Porém essa adaptação não deve demandar um grande trabalho, e será feita em breve.

### O método *howLongCanRunGridApplication*

Esse terceiro método, o primeiro totalmente novo em relação a versão inicial do *LUPA*, funciona de maneira semelhante ao primeiro, *canRunGridApplication*. Porém ele devolve um pouco mais de informação do que apenas uma resposta **SIM** ou **NÃO**. Seu protótipo é:

```
int howLongCanRunGridApplication(freeCpu, freeRam)
```

O método, como visto, não recebe como parâmetro um número de horas. O que ele fará é verificar por quanto tempo o computador irá manter disponíveis as quantidades mínimas de recursos requisitadas. A resposta será então o número de horas, a partir do momento atual, em que o computador irá manter-se ocioso, sendo a ociosidade definida pelos parâmetros recebidos.

Para que o algoritmo não rode indefinidamente, caso por exemplo a quantidade de CPU e memória solicitadas seja muito baixa, estipulamos um número máximo de horas a ser dado como resposta. Esse valor está definido como 72 horas, atualmente, porém pode ser ajustado conforme observe-se a necessidade.

### O método *averageResourceUsage*

Esse método, o último da interface atual do *LUPA*, tenta ser um meio termo entre o método *getPrediction* e os dois outros, fornecendo uma visão um pouco melhor de como estará a utilização de fato nas próximas horas, porém sem devolver todos os valores previstos. Seu protótipo é:

```
double averageResourceUsage(minimum, resource, hours)
```

O que o método fará é devolver a média dos valores previstos de utilização para o recurso especificado, pelo número de horas solicitado. É como se o método representasse uma chamada ao *getPrediction*, seguida de um algoritmo que calculasse a média do vetor devolvido.

Porém, note que um valor mínimo também deve ser passado. Isso implica que, caso algum dos valores previstos fique abaixo desse mínimo, o algoritmo pára imediatamente, devolvendo um valor especial que indica que a máquina não atenderá a ociosidade desejada. Isso evita que uma previsão com muita amplitude entre os picos de disponibilidade gerem uma média que aparentemente atenda os requisitos, porém onde na verdade alguns períodos não apresentarão ociosidade suficiente.

# Capítulo 4

# O escalonador do *InteGrade*

O escalonamento de tarefas é sempre um assunto bastante complexo, além de ser uma atividade de vital importância em muitos sistemas computacionais. Como já discutido anteriormente, o escalonamento em grades computacionais oportunistas acaba tendo características bastante peculiares. A preocupação em garantir a prioridade dos usuários locais acaba acrescentando uma dificuldade extra no processo de escalonar as aplicações para os nós compartilhados.

No capítulo anterior, discutimos uma maneira de obtermos uma informação de grande ajuda para o escalonador do *InteGrade*. Já que queremos preservar os usuários e somente usar seus recursos em momentos de ociosidade, conseguir prever esses momentos é definitivamente algo extremamente relevante.

Nesse capítulo, primeiramente iremos descrever brevemente a versão anterior do escalonador existente no *InteGrade*. Depois, mostraremos a questão da análise dos requisitos de uma aplicação, fator de grande importância para as consultas ao *LUPA*. Finalmente, discutiremos a integração do *LUPA* ao escalonamento, explorando como aproveitar de maneira inteligente suas informações para que o *InteGrade* cumpra sua promessa a quem compartilha seu computador com a grade, ao mesmo tempo em que é eficiente.

## 4.1 Versão anterior do escalonador

Até o começo deste trabalho, o que existia no *InteGrade* era um escalonador bastante simples, porém que continua sendo usado como parte do processo de escalonamento. Basicamente, cada *LRM*, ao registrar-se no *GRM*, informa algumas de suas características, como seu sistema operacional, o tipo de processador e informações sobre a disponibilidade de recursos. Essas últimas, por variarem constantemente, são atualizadas periodicamente.

O *GRM* mantém uma espécie de banco de dados com todas essas informações, sobre cada um dos nós sob seu controle. Ao receber uma submissão de aplicação, o que o escalonador anterior fazia era simplesmente ver qual dos nós, segundo esse banco de dados, satisfaziam as restrições dessa aplicação. Algumas dessas restrições sempre estavam presentes, como sistema operacional e tipo de processador. Porém algumas poderiam ser passadas pelo usuário, como a porcentagem de CPU livre ou a quantidade de *kilobytes* disponíveis na memória RAM. Perceba que, até aqui, essas condições de disponibilidade nada tem a ver com o *LUPA*. As informações sobre disponibilidade que o *GRM* possui são aquelas passadas pelos *LRM's* na última atualização, ou seja, dizem respeito a disponibilidade atual da máquina. Nada conhece-se sobre sua situação futura.

O ponto principal aqui é que, filtradas as máquinas que satisfazem essas restrições

iniciais, elas eram escolhidas numa ordem qualquer, sem fazer nenhuma distinção entre elas. A única política adotada é uma espécie de *round-robin*, onde máquinas escalonadas recebem uma marcação para que evite-se escaloná-las novamente enquanto existirem máquinas ainda não escolhidas. Mas, mesmo após a implementação do *LUPA*, em nada eram usadas as previsões dentro do escalonador.

## 4.2 Definindo os requisitos da aplicação

Como mencionado no Capítulo 3, o *LUPA* funciona esperando que seja dito o quanto de recursos a aplicação irá consumir, para então saber quando uma máquina pode ser considerada disponível. Descobrir esses valores é uma tarefa bastante difícil. Por outro lado, de nada serviriam as previsões se não conseguíssimos estimar esses valores com uma proximidade no mínimo razoável do real. Afinal, sem saber quando considerar ou não uma máquina disponível, o *LUPA* perderia o sentido.

No *InteGrade*, pretende-se desenvolver uma maneira de descobrir automaticamente, no momento da submissão, o quanto uma aplicação irá gastar de recursos, e por quanto tempo. Esses dados seriam passados para o escalonador que, conversando com o *LUPA*, teria tudo que necessita para realizar um escalonamento com grandes chances de sucesso. Porém, essa meta certamente não é fácil de ser alcançada e, atualmente, nada nesse sentido existe no *InteGrade*.

Ao invés disso, tudo o que possuímos hoje é uma maneira de o usuário da grade, no momento de submeter sua aplicação, informar o quanto de recursos ele deseja que estejam livres, e por quanto tempo, para que sua aplicação rode. Não há garantia alguma de que esses dados estão corretos, e portanto o melhor que podemos fazer é contar que o usuário conhece bem sua aplicação, provavelmente de execuções anteriores, e torcer para que os parâmetros passados aproximem-se da realidade.

No *InteGrade*, a submissão de aplicações é feita pelo módulo *ASCT*, como descrito no Capítulo 2. Existe uma interface gráfica, chamada *AsctGUI*, que oferece uma interface amigável ao usuário que deseja submeter e controlar suas aplicações. No *AsctGUI*, o usuário pode configurar alguns parâmetros da execução da aplicação. Dentre eles, há um campo denominado *Constraints*, que serve para que ele passe restrições que deseja que sejam respeitadas pelo escalonador.

Esse campo é, na versão atual do *InteGrade*, a única maneira do usuário passar informações relevantes para o *LUPA*. Basicamente o que será passado é uma frase, num formato específico, que definirá essas restrições. Um exemplo dessa frase seria:

```
cpuUsage < 75 and hours == 6
```

A *constraint* acima informa que deseja-se uma máquina em que o uso de CPU esteja abaixo de 75% (ou seja, com pelo menos 25% de CPU livre) pelas próximas 6 horas. É essa informação que será usada nas consultas feitas ao *LUPA*. Um detalhe importante é que, como nada foi dito sobre memória na *constraint*, assume-se que o mínimo de memória necessária é 0, e portanto esse recurso não irá influir nas decisões do *LUPA* nem do escalonador.

## 4.3 Primeira integração do *LUPA*

Uma das primeiras atividades desse trabalho foi realizar uma integração preliminar do *LUPA*, bastante simples. A idéia era apenas ter algo que usasse o *LUPA*, já que esse módulo estava programado desde o início do *InteGrade*, já possuía uma implementação, mas ainda assim não era usado dentro da grade.

O que foi feito foi basicamente uma maneira de fazer com que o *LUPA* recusasse execuções de aplicações caso verificasse que a máquina não estaria disponível. Essa tarefa, portanto, estaria no passo 3 da Figura 2.2, ou seja, no momento da confirmação de oferta de recursos junto ao *LRM*. Nesse momento, o *LRM* consulta o *LUPA* utilizando o método `canRunGridApplication()`, para verificar se, para os parâmetros da execução, a disponibilidade será mantida. Em caso de resposta negativa, informa o *GRM* que a execução foi recusada, e este procura outro nó disponível.

Perceba que, dessa maneira, o potencial do *LUPA* não está sendo completamente utilizado. O escalonador sequer toma conhecimento de qualquer informação sobre a previsão de disponibilidade. Além disso, a escolha dos nós para executar a aplicação continua sendo feita em ordem qualquer, e essa ordem somente é afetada se algum nó de fato recusar uma execução.

O próximo passo seria trazer essas decisões do nó compartilhado para o nó gerenciador do aglomerado. Ou seja, o escalonador deveria consultar o *LUPA* de cada máquina e, de posse de todas as respostas, decidir quais máquinas seriam escolhidas para executar a aplicação.

## 4.4 Versão melhorada do escalonador

Uma implementação totalmente nova de escalonador foi desenvolvida para ser integrada ao *InteGrade*. Como ela iria rodar no nó gerenciador do aglomerado, junto ao *GRM*, sua implementação foi feita em Java, assim como todo este módulo.

Porém, boa parte do escalonamento feito anteriormente foi mantida. Assim como antes, primeiramente o *GRM* seleciona uma lista inicial de nós candidatos a executar a aplicação, de acordo com alguns pré-requisitos iniciais (mas que nada tem a ver com a disponibilidade futura dada pelo *LUPA*). Então, essa lista é enviada ao escalonador, que irá filtrá-la mais ainda utilizando consultas específicas ao *LUPA* de cada nó. Cada algoritmo de escalonamento possui uma heurística que define como essa filtragem é feita, mas basicamente o que pode ser feito é:

- Retirar da lista nós que não satisfaçam alguma condição desejada, de acordo com os requisitos da aplicação e as informações fornecidas pelo *LUPA*.
- Reordenar a lista, de maneira que, ao devolvê-la para o *GRM*, este submeta a aplicação primeiro para máquinas que, para o escalonador, representam uma melhor escolha para a execução.

Feita essa filtragem, a lista é retornada ao *GRM*, que irá percorrê-la na ordem em que se encontra, solicitando aos *LRM's* a execução da aplicação. A abordagem de *round-robin* permanece existindo, tentando evitar que um mesmo nó execute aplicações repetidas vezes com outros nós disponíveis.

Lembrando que o escalonador encontra-se junto ao *GRM*, e irá consultar o *LUPA* de cada *LRM*, uma comunicação remota precisa ser feita. Assim como em todo o restante do

*InteGrade*, para esse tipo de comunicação aqui também foi usada a tecnologia *CORBA*. Com ela, chamadas a métodos de objetos remotos podem ser feitas como se estes fossem locais. A interface do *LUPA* foi extendida, na verdade, ao *LRM*, ou seja, o *GRM* fazia chamadas a métodos do *LRM*, e esses métodos delegavam a chamada ao *LUPA* local. Isso facilitou bastante o trabalho, já que o *GRM* já possuía uma identificação de cada *LRM* sob seu controle.

### 4.4.1 Análise de requisitos

A primeira coisa que o escalonador deve fazer é definir quais os requisitos de execução da aplicação. Como explicado na Seção 4.2, por hora esses requisitos são definidos única e exclusivamente pelo usuário que a submete, através de uma frase passada no campo *constraints* do *AsctGUI*.

A classe `RequirementsAnalyzer`, parte do escalonador, é encarregada então de receber essa `String` e fazer um *parse*, ou seja, identificar dentro dela campos que são relevantes para o *LUPA*. Esses campos, na versão atual, são `cpuUsage`, `freeRam` e `hours`. O primeiro indica qual a taxa de ocupação máxima de CPU desejada, o segundo a quantidade de memória livre esperada e o último o número de horas pelas quais espera-se que esses valores se mantenham.

O programa gera então uma instância da classe `ApplicationRequirements`, que nada mais é que uma classe com campos para representar os requisitos daquela aplicação. Em particular, cada recurso possui um valor denominado `dontCareValue`, que significa que aquele recurso não deve ser levado em conta na verificação da disponibilidade do computador. Esse valor será usado quando um determinado recurso não aparecer no campo de *constraints*. Uma diferenciação importante, e muitas vezes confusa, é que o `dontCareValue` do recurso `cpuUsage` é dado como 1.0, já que, como estamos falando em termos de uso de CPU, estamos interessados em um uso de CPU abaixo de um determinado valor. Logo, caso esse valor não importe, dizemos que o uso de CPU deve estar abaixo de 1.0 (100%), ou seja, qualquer computador será aceito. Analogamente, como no campo `freeRam` falamos de memória livre, o valor do `dontCareValue` será 0.0.

### 4.4.2 Algoritmos de escalonamento

Após a análise das *constraints* feita pela classe `RequirementsAnalyzer`, a instância de `ApplicationRequirements` gerada é passada como parâmetro ao algoritmo de escalonamento sendo usado no momento, para que ele utilize os requisitos da aplicação nas consultas ao *LUPA*.

De que modo serão feitas essas consultas, e como a lista inicial de nós candidatos a executar a aplicação será modificada, depende da heurística de cada algoritmo. A princípio, para definir o algoritmo a ser utilizado, o escalonador procura por um arquivo específico de configuração, com essa informação. Caso não ache esse arquivo, ele utiliza um dos algoritmos como padrão (decidir qual deles será o padrão ainda dependerá de alguns testes). A idéia para o futuro, porém, é que o escalonador possa decidir automaticamente trocar de algoritmo, caso detecte que os escalonamentos não estão apresentando bons resultados.

Descreveremos a seguir os algoritmos implementados no escalonador.

### *CanRunGridApplication*

Esse algoritmo é o mais simples do escalonador, sendo na prática o descrito na Seção 4.3. A única diferença é que aqui as consultas ao método `canRunGridApplication()` são feitas remotamente, do nó gerenciador do aglomerado. O que o escalonador faz é um *polling* dentre todos os *LRM's*, questionando se eles atendem os requisitos da aplicação. Aqueles *LRM's* cujo *LUPA* responder negativamente serão excluídos da lista de nós candidatos a executar a aplicação.

Na realidade, esse algoritmo traz poucas novidades em relação a primeira versão de integração do *LUPA* ao *InteGrade*. Ele tenta cumprir a obrigação principal do escalonamento, que é garantir que máquinas que não manterão ociosidade suficiente não sejam alocadas para executar uma aplicação. Por outro lado, ele deixa de aproveitar as informações do *LUPA* para tentar um escalonamento mais eficiente, que dê preferência para alguns nós em relação a outros, seguindo alguma heurística.

### *HowLongCanRunGridApplication*

Esse algoritmo usa as respostas do método `howLongCanRunGridApplication()`, do *LUPA*, para decidir quais nós terão preferência para executar a aplicação. Basicamente, os nós serão escolhidos dando preferência àqueles que se manterão ociosos por mais tempo.

Como explicado na Seção 3.3, este método devolve o número de horas pelas quais o computador manterá a disponibilidade solicitada. O que o algoritmo fará então é fazer uma consulta ao método de cada *LUPA*, pesquisando por quanto tempo os requisitos da aplicação serão mantidos.

Primeiramente, caso o *LUPA* devolva 0, isso significa que não haverá disponibilidade. Mais além, caso o número de horas devolvido seja inferior ao número de horas solicitado pelo usuário no campo *constraints* do *AsctGUI*, significa que a ociosidade do computador não será suficiente (na prática, é o mesmo que uma resposta `NÃO` do método `canRunGridApplication()`). Nesse caso, o nó será excluído da lista de nós candidatos a execução, da exata mesma maneira que no algoritmo anterior.

Porém, após fazer essa filtragem inicial da lista de nós candidatos, o algoritmo faz um segundo processamento. Essa lista é reordenada de acordo com as respostas do *LUPA*, de maneira que aqueles em que o número de horas retornado foi maior apareçam primeiro. Isso significa, na prática, que estamos buscando escalonar aplicações para computadores em que a previsão é de que a ociosidade irá se manter por mais tempo. Isso pode evitar, por exemplo, que caso uma aplicação demore um pouco mais do que o esperado, sua execução não seja surpreendida pelo retorno do usuário dono da máquina.

Um algoritmo de ordenação bastante simples foi utilizado, o *Selection Sort*. Como o tamanho do vetor será o número de nós candidatos a executar a aplicação, e ainda depois de uma primeira filtragem, em geral este não será muito grande. Por isso, não pareceu necessário preocupar-se com técnicas mais avançadas de ordenação.

### *GreedyAverageResourceUsage*

Esse algoritmo mantém a mesma idéia do anterior, reordenando a lista de nós candidatos a executar a aplicação. Porém, a informação que ele leva em conta para definir a ordem dos nós não é o tempo previsto de ociosidade, porém a taxa média de utilização prevista para as próximas horas.

Primeiramente o algoritmo realiza consultas ao *LUPA* de cada nó através do método `averageResourceUsage()`, guardando a média de utilização para as próximas horas tanto de CPU quanto de memória. Como explicado na Seção 3.3, este método irá devolver um valor específico caso deseje indicar que, na previsão para essas horas, algum dos valores está abaixo do requisitado para aquele recurso. Então, o algoritmo de escalonamento verifica se foi esse o valor retornado, para, em caso positivo, excluir o nó da lista de nós candidatos.

Após isso, uma ordenação será feita por esses valores de média de cada nó. O primeiro passo é fazer uma normalização nos valores. No caso, os valores de CPU, por serem dados em porcentagem, estão sempre entre 0.0 e 1.0. Já os valores de memória podem assumir, teoricamente, qualquer valor positivo, já que são dados em quantidades absolutas de memória livre. O algoritmo normaliza então essas médias de quantidade de memória livre. Primeiro, encontra-se o maior valor dentre os devolvidos pelo *LUPA*, e então divide-se todos os outros por este, de forma ao maior deles valer 1.0.

Após essa normalização, é preciso combinar os valores de CPU e memória. Isso porque queremos definir uma ordenação dos nós com relação à suas médias de quantidade de recursos disponíveis pelas próximas horas. Mas, cada nó possui uma média de CPU livre e uma média de memória livre. A maneira encontrada foi definir um peso para cada um dos recursos. Por padrão, esse peso é igual para ambos, ou seja, cada recurso tem sua média multiplicada por 0.5 e então os resultados são somados. Porém, esses pesos podem ser lidos de um arquivo de configuração. Pode ser interessante, por exemplo, dar-se um peso maior para o valor médio de CPU disponível nas máquinas, já que esse recurso em geral é o mais crítico na execução de aplicações.

Após definir um único valor que represente a média de utilização prevista para cada nó, o algoritmo ordena a lista de nós candidatos com base nesses valores. No caso desse algoritmo, ele coloca a lista em ordem decrescente de disponibilidade de recursos. Isso significa que os nós em que a previsão é de que uma maior quantidade de recursos estará livre serão escolhidos primeiro no escalonamento. Por isso o nome do algoritmo, já que ele adota uma postura "gulosa", selecionando primeiro as máquinas mais ociosas possíveis.

Essa abordagem pode ter duas conseqüências. Por um lado, a aplicação pode executar mais rapidamente, já que segundo as previsões ela terá mais recursos disponíveis. Por outro lado, ela pode ocupar máquinas com uma quantidade de recursos disponíveis maior do que ela precisaria originalmente. Isso pode fazer com que uma outra aplicação submetida antes de seu término, com maiores requisitos, seja recusada, se os únicos computadores capazes de executá-la estiverem ocupados com a primeira aplicação. Isso pode ser um problema, em particular em grades onde a freqüência de submissões é muito grande, onde talvez fosse interessante preservar máquinas mais disponíveis para aplicações que realmente necessitem de uma grande quantidade de recursos.

#### *BestFitAverageResourceUsage*

Esse algoritmo funciona de forma muito parecida com o anterior. Ele também trabalha realizando consultas ao método `averageResourceUsage()` do *LUPA*. Além disso, o processo de normalização das médias recebidas e a combinação das médias dos diferentes recursos monitorados através do uso de pesos também é idêntico.

A diferença, porém, está no método de ordenação usado depois que um único valor para representar a média de disponibilidade do computador for encontrado. Nesse algoritmo, a lista de nós candidatos é ordenada crescentemente em relação a este valor. Na prática, isso significa que estamos dando preferência primeiramente para as máquinas que estão

com a menor previsão de disponibilidade. É importante deixar claro, porém, que todas as máquinas em que a previsão não era suficiente para os requisitos da aplicação já foram excluídas da lista no momento da ordenação.

Ou seja, como o nome do algoritmo sugere, ele irá procurar as máquinas que satisfazem os requisitos da aplicação com a menor folga possível. Dessa maneira, estamos tentando contornar o problema apontado no algoritmo *GreedyAverageResourceUsage*, onde possivelmente máquinas mais poderosas estariam sendo ocupadas por aplicações não tão necessitadas de recursos, e outras aplicações mais pesadas poderiam ter de ser recusadas. Com esse algoritmo, preservamos esses computadores para que futuras submissões que porventura sejam feitas possam ser aceitas.

# Capítulo 5

# Resultados

Como descrito nos capítulos anteriores, após esse trabalho o *InteGrade* passou a contar, finalmente, com um escalonador que utiliza informações sobre o uso futuro dos nós compartilhados para definir quais destes executarão as aplicações submetidas. Essa tarefa já estava prevista desde o início do projeto *InteGrade* [2] [3] [6], porém nada nesse sentido havia sido efeito.

Porém é importante também que conheçamos a real potencialidade do escalonador desenvolvido e, em particular, os prós e contras de cada um de seus algoritmos. Afinal, por mais atraente que possa ser a idéia de prever o futuro, pode ser quem em certas situações um escalonamento que sequer use o *LUPA* tenha o mesmo efeito, ou mostre-se até mais interessante. Portanto, é importante testar o comportamento deste escalonador num cenário real, ou próximo ao real.

Neste capítulo, descreveremos o que foi feito em relação a experimentos para testar o novo escalonador. Infelizmente, os resultados dos testes não foram tão conclusivos como o esperado, e acreditamos que mais testes, possivelmente em condições diferentes dos aqui realizados, precisariam ser feitos para obtermos conclusões mais concretas.

## 5.1 Ambiente para realização dos testes

Desde meados do mês de agosto de 2008, o *InteGrade* foi instalado e passou a rodar na Rede Linux, uma rede de computadores disponíveis para os alunos de graduação do IME-USP. Houve uma conversa com os responsáveis pela rede, e eles concordaram com essa instalação, já que tudo no *InteGrade* foi projetado para que sua execução não causasse *overhead* no computador, e assim os usuários não deveriam perceber qualquer mudança. Ao todo o *InteGrade* foi instalado em pouco mais de 20 máquinas, número que variava um pouco conforme alguma máquina era retirada para manutenção ou retornava.

O computador escolhido para ser o nó gerenciador do aglomerado foi o `zillertal`, um dos servidores da rede. Todos os outros computadores, distribuídos fisicamente em 3 salas diferentes, faziam parte deste aglomerado, e rodavam os módulos *LRM* e *LUPA*. Vale ressaltar que o computador `zillertal`, além de nó gerenciador, também rodava os módulos *LRM* e *LUPA*. O módulo *ASCT* rodava também no `zillertal`, portanto era preciso logar nessa máquina para submeter aplicações.

A máquina `zillertal` consiste de um *Intel Xeon* de 8 núcleos de 2.66 GHz e 8GB de memória RAM. Já os computadores de uso comum, que serão os compartilhados pela grade, são em sua maioria (se não totalidade) *Intel Pentium 4* de 3 GHz com tecnologia de *Hyper-threading*, e com 512 MB de memória RAM.

Quanto a utilização das máquinas da rede, pode-se dizer que em geral ela segue um padrão diário. As máquinas costumam ser mais utilizadas durante a manhã e tarde, e menos no período da noite. Nas madrugadas e nos finais de semana o uso é quase nulo. Já entre uma máquina e outra, não existe muita diferença nos padrões. Em geral, nos horários de maior uso, a distribuição das máquinas que estão sendo utilizadas é, na prática, aleatória.

## 5.2 Os experimentos

Para testar-se o escalonador desenvolvido, seus algoritmos e a integração com o *LUPA*, foi proposto a submissão e execução de aplicações no *InteGrade*. No fim, isso acabaria por testar o sistema como um todo, já que usaria praticamente tudo que foi feito no *InteGrade* até hoje. Mas, em particular, estávamos interessados em observar diferentes comportamentos nas execuções quando diferentes algoritmos de escalonamento eram usados.

Apenas uma aplicação foi escolhida para ser submetida em todos os experimentos. Precisávamos de uma aplicação que tivesse tempos de execução variados, e o professor Marcelo Finger sugeriu usar o programa `zchaff`[1], que implementa o algoritmo de *Chaff* para resolver o problema da satisfabilidade. O professor Marcelo enviou-me um programa capaz de gerar entradas para o `zchaff`, juntamente com instruções de parâmetros a serem usados que, dependendo de sua relação, criariam entradas "fáceis" ou "difíceis" para o programa. Assim, com esse mesmo programa eu poderia criar tanto programas que executam em poucos segundos como que demoram algumas horas.

Após isso, foi realizado o desenvolvimento de uma maneira de submeter-se aplicações ao *InteGrade* em modo texto, sem usar o *AsctGUI*. Dentro do projeto *InteGrade*, são realizadas as chamadas "Maratonas de Programação", onde todos os envolvidos no projeto juntam-se para resolver problemas e desenvolver melhorias em áreas não totalmente ligadas à sua pesquisa. No caso, essa interface de submissão modo texto já vinha sendo desenvolvida nessas maratonas, e o que foi feito foi apenas sua finalização, para que os testes pudessem ser realizados.

Por fim, foi definida a configuração da execução a ser submetida. O primeiro fator a ser definido é em quantas máquinas a aplicação executaria. A princípio o número usado foi 10, porém por acreditar-se que esse valor talvez não evidenciasse tanto as diferenças entre os algoritmos de escalonamento (já que, ao escolher 10 máquinas dentre 20, provavelmente em todos os algoritmos a escolha seria parecida), ele foi diminuído para 5 e, posteriormente tentando evidenciar mais ainda as diferenças entre os algoritmos, foi reduzido para 3.

Além disso, uma *constraint* também foi definida. Como dito no Capítulo 4, esses valores, no momento, infelizmente têm de ser "chutados". Porém, apenas para que se usasse alguma restrição, algo foi passado, dependendo do tempo aproximadamente esperado da execução.

Um *script* simples foi feito para mudar o escalonamento a ser usado no arquivo de configuração do escalonador, e então submeter a aplicação pelo modo texto criado. Os resultados foram copiados para um diretório separado, com identificação de cada teste. O resultado, no caso, era a saída do `zchaff`, que continha o tempo que a execução havia levado. O que foi feito então foi verificar, para cada algoritmo, os tempos mínimo, máximo e médio de execução.

[1] `http://www.princeton.edu/~chaff/zchaff.html`

A comparação foi realizada, então, pelos tempos de cada execução. É claro que essa não é a única maneira de comparar-se as eficiências dos algoritmos, e talvez nem a melhor, porém por simplicidade foi a utilizada. Seria uma forma de tentar-se comparar a eficiência da grade em si, na execução de aplicações, quando do uso dos diferentes algoritmos.

Um outro lado, importantíssimo de ser testado, seria justamente a taxa de acerto do escalonamento em relação a real ociosidade da máquina. Ou seja, quando uma aplicação foi enviada para ser executada em uma máquina e, antes de seu final, o usuário retornou para utilizá-la. Todavia, como o *InteGrade* ainda não dispõe de um mecanismo para interromper a aplicação nesse caso, e a única garantia é a prioridade mínima com que esta executa, para garantir o bom uso da rede realizamos os testes apenas nos períodos da madrugada ou final de semana. Nesses períodos, como as máquinas em geral estão de fato ociosas, ficaria mais complicado testar esse outro lado. Contudo, esses testes estariam, na verdade, basicamente testando a correção ou não das previsões do *LUPA*, e resultados nesse sentido podem ser observados em [4] e [5].

## 5.3 Resultados

O primeiro passo foi realizar diversas execuções do `zchaff`, procurando tempos de execução que se adequassem aos nossos interesses. A primeira execução foi de apenas alguns segundos, apenas para ver se alguma coisa poderia ser observada. Porém, parecia bastante óbvio que nesse caso nenhuma diferença seria notada, primeiro porque as previsões do *LUPA* sequer tratam precisões da ordem de segundos, e também porque uma execução assim tão rápida não deveria sofrer muito com mudanças na disponibilidade da máquina. De toda forma, até a título de curiosidade, os resultados estão mostrados na Tabela 5.1.

Como um nó pode terminar a execução antes de outro, a tabela mostra o tempo mínimo para a aplicação executar (ou seja, o tempo que ela levou no nó mais rápido), o máximo e o médio. Como o teste foi executado 5 vezes, os resultados de tempo são as médias de todas as execuções. Além disso, mostramos também quais os *hosts* escolhidos para executar cada aplicação, dependendo do algoritmo. Os algoritmos estão com um nome simbólico que identifica cada um, e o primeiro algoritmo é o escalonamento sem consultar o *LUPA*, da exata maneira que acontecia anteriormente.

| | **Tempo mínimo** | **Tempo médio** | **Tempo máximo** | ***Hosts* escolhidos** |
|---|---|---|---|---|
| `NO LUPA` | 21s | 32s | 51s | `superman lisa homer` |
| `CAN RUN` | 21s | 31s | 50s | `superman lisa homer` |
| `HOW LONG` | 49s | 50s | 50s | `homer milhouse aquaman` |
| `GREEDY` | 41s | 47s | 50s | `cyclops burns wolverine` |
| `BEST FIT` | 26s | 44s | 57s | `skinner flash superman` |

Tabela 5.1: Primeiro experimento realizado, aplicações muito rápidas

Para essa execução foram solicitadas máquinas onde o uso de CPU estivesse abaixo de

70% e fosse se manter assim pelas próximas 2 horas pelo menos. Para representar isso, a seguinte *constraint*[2] foi passada:

```
cpuUsage < 70 and hours == 2
```

Em geral, o maior interesse é no tempo máximo, porque ele diz quando o último nó terminou de executar e a grade realmente devolverá os resultados para o usuário. Nesse caso, podemos ver que a média em geral foi a mesma, tirando um valor um pouco maior para o último algoritmo, *BestFitAverageResourceUsage*. Todavia, os tempos mínimos aparecem justamente nos algoritmos mais simples. A verdade é que, para uma execução tão rápida, isso provavelmente não tenha significado algum. Um fato aparentemente constante é que em geral a primeira execução era sempre mais rápida (no caso, a primeira era sempre do algoritmo que não usava o *LUPA*). Não podemos dizer, no entanto, se isso é realmente um fato consumado e qual seria o motivo (alguma particularidade do *InteGrade* em si, talvez).

A seguir, realizamos um outro teste com uma aplicação rápida, da ordem de alguns minutos. Ainda era pouco perto do que se espera de aplicações numa grade computacional, mas a idéia era ir avançando os tempos aos poucos. Vale ressaltar que a mesma *constraint* da execução anterior foi utilizada. Os resultados podem ser observados na Tabela 5.2.

| | **Tempo mínimo** | **Tempo médio** | **Tempo máximo** | ***Hosts* escolhidos** |
|---|---|---|---|---|
| `NO LUPA` | 6min57s | 10min12s | 15min23s | `superman lisa homer` |
| `CAN RUN` | 7min16s | 9min59s | 15min18s | `superman lisa homer` |
| `HOW LONG` | 15min8s | 15min13s | 15min22s | `homer milhouse aquaman` |
| `GREEDY` | 11min57s | 14min2s | 15min5s | `cyclops burns krusty` |
| `BEST FIT` | 6min30s | 9min40s | 15min3s | `skinner lisa superman` |

Tabela 5.2: Segundo experimento realizado, execuções da ordem de alguns minutos

Novamente, acreditamos que os resultados aqui não são tão significativos no que diz respeito a diferenciação dos algoritmos do escalonador. Isso porque o tempo das execuções ainda não é grande o bastante para que as previsões do *LUPA* realmente mostrem sua importância. Vale notar, porém, que novamente os nós que mais demoraram, demoraram mais ou menos o mesmo tempo. E, estranhamente, novamente os 2 primeiros algoritmos, os mais simples, saíram-se muito bem. O algoritmo *BestFitAverageResourceUsage*, porém, foi quem apresentou os menores valores de tempo.

A seguir, um terceiro teste foi realizado, seguindo exatamente o mesmo padrão. Dessa vez, foi escolhida uma entrada para o `zchaff` que demorasse um tempo um pouco maior, ainda não tão grande porém já mais razoável tendo em vista ambientes de grades computacionais. Em média, era esperado que a aplicação levasse de 30 a 60 minutos para encerrar, um tempo ainda pequeno se comparado a determinadas aplicações científicas, por exemplo, mas já razoável considerando uma grande gama de aplicações. É verdade, porém, que esse tempo ainda não serve tanto assim para as pretensões do *LUPA*, que trabalha os

[2]Para maiores detalhes sobre *constraints*, consulte o Capítulo 4

intervalos de tempo em horas. Dessa forma, a mesma *constraint* das execuções anteriores foi usada, requisitando uma utilização de menos que 70% de CPU pelas próximas 2 horas. Os resultados desse teste podem ser vistos na Tabela 5.3

| | **Tempo mínimo** | **Tempo médio** | **Tempo máximo** | ***Hosts* escolhidos** |
|---|---|---|---|---|
| `NO LUPA` | 57min56s | 58min25s | 58min45s | `superman lisa homer` |
| `CAN RUN` | 58min54s | 59min14s | 59min33s | `superman lisa homer` |
| `HOW LONG` | 52min44s | 57min11s | 59min51s | `homer lisa superman` |
| `GREEDY` | 34min1s | 50min6s | 59min16s | `cyclops flanders milhouse` |
| `BEST FIT` | 26min25s | 50min55s | 1h8min32s | `patty flash superman` |

Tabela 5.3: Terceiro experimento, execuções que demoram entre meia e uma hora

Aqui, algumas diferenças já começam a ser notadas. O algoritmo em que uma execução terminou mais rapidamente foi o *BestFitAverageResourceUsage*, porém foi com ele também que ocorreu a execução mais demorada. Ainda assim, na média esse algoritmo e o *GreedyAverageResourceUsage* foram os que apresentaram menor tempo de execução. Já os outros algoritmos tiveram um tempo um pouco superior, como notado. Ainda assim, há de se ter cuidado antes de chegar-se a conclusões, já que isso não necessariamente indica uma prevalência dos métodos mais sofisticados que usam o *LUPA*, já que o tempo de execução das aplicações ainda não é assim tão significativo. Porém, há um indicativo que possivelmente o escalonamento usando o *LUPA* já consegue tomar decisões adequadas. Uma coisa pelo menos é clara: os 3 primeiros algoritmos, em geral, escolhem muitas máquinas em comum, e por isso os tempos tão parecidos.

O quarto e último teste a ser realizado foi com uma entrada para o `zchaff` onde esperava-se um tempo de execução de aproximadamente 2 horas, um tempo mais razoável para as pretensões do *LUPA*. Aqui uma *constraint* um pouco diferente foi utilizada, ainda que novamente isso não passou de um "chute". Foram requisitadas máquinas com um uso de CPU que não ultrapassasse 80% pelas próximas 4 horas. A restrição sobre a CPU foi relaxada um pouco de maneira a evitar que máquinas fossem recusadas sem grande necessidade. A *constraint* passada foi, portanto:

```
cpuUsage < 80 and hours == 4
```

Os resultados desses experimentos podem ser vistos na Tabela 5.4. Aqui, esses resultados já devem ser analisados com mais calma, já que são execuções mais realistas num ambiente de grade, e o algoritmo que não usa o *LUPA* levou uma razoável vantagem no que diz respeito ao tempo. Porém, é preciso tomar cuidado com as conclusões, já que observe, por exemplo, que o algoritmo `CAN RUN` escolheu exatamente as mesmas máquinas que o algoritmo `NO LUPA`, e suas execuções já demoraram um pouco mais. Na Seção 5.4 tentaremos dar algumas idéias do porquê dessas diferenças de tempo, e o que isso representa na escolha dos algoritmos de escalonamento. Outro ponto interessante a se notar é que os algoritmos *GreedyAverageResourceUsage* e *BestFitAverageResourceUsage*, apesar de escolherem as máquinas por heurísticas completamente opostas, apresentaram tempos de execução muito semelhantes.

| | **Tempo mínimo** | **Tempo médio** | **Tempo máximo** | ***Hosts* escolhidos** |
|---|---|---|---|---|
| `NO LUPA` | 2h17min54s | 2h32min25s | 2h42min25s | `homer lisa krusty` |
| `CAN RUN` | 2h50min34s | 2h51min5s | 2h51min41s | `homer lisa krusty` |
| `HOW LONG` | 2h18min17s | 2h40min16s | 2h51min32s | `homer lisa krusty` |
| `GREEDY` | 2h50min38s | 2h51min3s | 2h51min27s | `cyclops krusty milhouse` |
| `BEST FIT` | 2h51min13s | 2h51min31s | 2h51min55s | `lantern hulk wolverine` |

Tabela 5.4: Quarto experimento, execuções de algumas horas

## 5.4 Análise dos resultados

Como dito, infelizmente não obtivemos o sucesso esperado na realização dos testes. Isso porque não possuímos, no momento, dados realmente conclusivos sobre as melhorias obtidas ou não com o novo escalonador. Além disso, alguns resultados parecem sugerir, inclusive, que usar o *LUPA* seria na verdade pior do que manter o escalonamento que existia anteriormente, bastante simples. Porém tentaremos aqui explorar alguns fatores que podem desmentir essa conclusão inicial.

Primeiramente, alguns comentários merecem ser feitos sobre o ambiente onde os testes foram realizados, que talvez não represente tão bem o cenário que seria o foco principal do *InteGrade*. Apesar de a Rede Linux ser, de fato, uma rede de uso comum, onde a grade aproveitaria de seus momentos de ociosidade, ela não favorece tanto assim a descoberta de padrões de uso, e o aproveitamento dessa informação. Isso porque, na realidade, o que se vê em geral é um padrão geral das máquinas, e não particular de máquina para máquina. Por exemplo, as máquinas em geral estão ocupadas nos dias da semana a tarde, porém se é a máquina A ou a máquina B que estará ocupada, isso em geral não segue um padrão. Isso causou uma certa preocupação, se de fato o uso do *LUPA* escolheria máquinas mais aptas a executar a aplicação, ou se dada uma aparente "aleatoriedade" na utilização das máquinas, no fundo o *LUPA* não faria tanta diferença.

Outro fator que pode ter contribuído um pouco para que os resultados não representassem com tanta fidelidade um cenário real é o horário em que os experimentos foram realizados. Para garantir que os testes não atrapalhassem os usuários da Rede Linux, já que o *InteGrade* ainda não dispõe de mecanismos confiáveis para garantir que, em caso de erro do *LUPA*, a execução seja interrompida para preservar o dono do computador, todos os testes foram realizados durante a madrugada e nos finais de semana. Assim, outro problema é que, nesses horários os computadores estão todos ociosos, salvo algumas exceções. Logo, novamente talvez o escalonamento com ou sem o *LUPA* não apresente, nesse caso, um diferencial tão grande assim.

Dessa forma, acreditamos que esses testes ainda não expõe de fato os benefícios advindos ou não do uso do *LUPA* no *InteGrade*. Talvez se testado num outro ambiente, onde o uso das máquinas apresentasse padrões mais característicos, e realizando os experimentos em períodos onde os computadores estão sendo usados localmente, o escalonador poderia responder de maneira diferente. E, dependendo dos resultados, ajustes e modificações poderiam ser feitas em sua implementação.

Por hora, o que podemos fazer é tentar analisar especialmente os dois últimos testes realizados, porém mantendo em mente tudo o que foi dito nos parágrafos anteriores, para não chegar a conclusões precipitadas. Num desses testes, os dois algoritmos que supostamente seriam os mais sofisticados de fato apresentaram um tempo de execução em geral mais baixo do que os outros. Já no último teste, o que mais se aproximaria de uma utilização real da grade, pelo menos no que diz respeito ao tempo que a aplicação demora para terminar, esses dois algoritmos, ao contrário, apresentaram um tempo razoavelmente pior do que o dos mais simples, inclusive o que não usa o *LUPA*.

Porém, algo nesse teste chama a atenção. Perceba que os tempos dos algoritmos `NO LUPA` e `CAN RUN` apresentam uma razoável diferença, sendo os do primeiro inferiores. Porém, os computadores escolhidos por ambos são exatamente os mesmos. Faz sentido, já que o algoritmo que não usa o *LUPA* não modifica a lista de nós candidatos a execução, enquanto o segundo algoritmo apenas exclui dessa lista nós em que o *LUPA* acusa que não haverá disponibilidade, mas isso é pouco provável de ocorrer durante a madrugada. Uma das explicações possíveis para essa diferença é que o escalonamento com o algoritmo `CAN RUN`, executado primeiro, foi feito no período entre fim de noite e começo de madrugada, enquanto o outro foi executado no meio da madrugada. Isso pode, talvez, significar que a máquina estava completamente ociosa na segunda execução, e não totalmente na primeira. Isso, porém, é apenas uma possibilidade, sendo necessários mais testes, possivelmente em outras redes, para ter certeza da comparação entre os dois algoritmos.

No geral, o que se viu é que o tempo total de execução da aplicação, definido quando o último nó termina, foi em geral muito parecido entre todos os algoritmos. O que mais variou, na maior parte dos casos, foi o tempo que o nó mais rápido levou para encerrar sua execução. Talvez experimentos com aplicações mais longas acentuem mais essa diferença, e algum padrão possa ser notado, porém devido a necessidade de execução dos testes em horários restritos, isso ainda não é totalmente viável na Rede Linux.

# Capítulo 6

# Conclusão

Acreditamos que esse trabalho tenha sido importante para o projeto *InteGrade* em si, já que fez progressos numa área há muito precisando ser desenvolvida no sistema. Há muito tempo que esperava-se, por parte de todos os envolvidos, a integração do *LUPA* ao resto do sistema. Claro que muito ainda precisa ser feito nesse sentido, as possibilidades de melhorias são muitas, porém o primeiro passo já foi dado.

Apesar de os testes, como dito anteriormente, não terem tido todo o sucesso esperado, a implementação está pronta para ser usada. O próprio *InteGrade* sofre do mesmo problema que sofremos na hora de realizar os testes, de não possuir uma rede de computadores com as características ideias para seu funcionamento como uma grade realmente oportunista. Porém, caso ele venha a ser finalmente instalado numa rede desse tipo, o escalonador aqui desenvolvido poderá ser testado na prática, em condições reais, e finalmente todo o trabalho envolvendo o *LUPA* e o escalonador poderá ser validado de fato.

De toda forma, o fato é que agora o *InteGrade* conta com um escalonador mais inteligente e, em particular, mais expansível. Isso porque novos algoritmos podem ser desenvolvidos, assim como os descritos nesse trabalho, para trabalhar com diferentes heurísticas. Até mesmo algoritmos que sequer usem o *LUPA* podem ser colocados no escalonador desenvolvido de maneira muito simples. Isso certamente facilita muito a realização de melhorias no escalonamento, já que antes esse era feito com trechos de código "implícitos", inseridos dentro do próprio código do *GRM*.

Quanto ao código do *LUPA*, as melhorias também foram de grande importância. Esse módulo, apesar de já ter sido extensivamente desenvolvido, ainda tem muito potencial para ser melhorado e expandido. Todavia, as modificações praticadas nesse trabalho certamente já conferiram um pouco mais de robustez ao módulo. Em particular, o fato de não existir mais restrições no número de horas das previsões a serem realizadas com certeza representa um grande diferencial ao módulo.

Por fim, além do trabalho principal envolvendo o escalonador e o *LUPA*, diversas foram as contribuições em várias partes distintas do *InteGrade*. A maioria delas de impacto bem pequeno, é verdade, mas acreditamos que no fim, como fruto indireto desse trabalho, o sistema *InteGrade* como um todo também acabou evoluindo de diversas maneiras.

## 6.1 Trabalhos futuros

As possibilidades de continuação desse trabalho são imensas, tanto no que diz respeito a melhorias no escalonamento, como no funcionamento do *LUPA* em si. Inclusive, algumas atividades nesse sentido já estão sendo realizadas por outros alunos do projeto *InteGrade*.

No que diz respeito ao escalonador, algumas idéias do que poderia ser feito:

- Permitir agendamento de execuções quando não houvessem máquinas disponíveis no momento. Atualmente, as aplicações são simplesmente recusadas. Porém, utilizando o *LUPA*, poderíamos não só agendar a execução, como já planejar de antemão quando e em que máquinas ela ocorreria. No momento planejado, faríamos uma confirmação da disponibilidade, para garantir que a previsão do *LUPA* tinha sido correta, e então submeteríamos a aplicação aos nós.

- Integrar o escalonamento, as previsões do *LUPA* e migrações de processos. Inclusive, já existe hoje no *InteGrade* uma plataforma que permite a migração de processos [7] [8] [9]. No momento do escalonamento, poderíamos definir não simplesmente uma máquina para realizar a execução, porém um "roteiro" por onde a aplicação passaria. Isso significa que caso uma única máquina não apresentasse a disponibilidade por tempo suficiente, poderíamos escalonar para uma máquina por um tempo específico, já planejando que, próximo do fim do período de ociosidade, a aplicação seria interrompida e migrada para uma máquina que agora está ociosa. Com o *LUPA*, essa próxima máquina que receberia a aplicação daqui algumas horas já poderia estar previamente definida.

- Uma melhoria interessante no escalonador seria fazer com que ele monitorasse e analisasse suas próprias decisões de escalonamento. Caso verificasse, por exemplo, que muitos escalonamentos estão sendo malsucedidos, ele poderia trocar o algoritmo de escalonamento sendo usado automaticamente, para tentar obter melhores resultados.

Já quanto ao *LUPA*, ele está já num estado bem mais avançado, até por estar sendo desenvolvido há mais tempo. Porém, não é difícil pensar em alguns aprimoramentos a serem feitos também, como por exemplo:

- Na versão atual, ao monitorar o uso dos recursos da máquina, o *LUPA* não diferencia se estes estão sendo usados por processos locais ou do próprio *InteGrade*. Isso significa que, caso uma máquina esteja sempre ociosa (sem uso local), porém rodando aplicações da grade, o *LUPA* irá entender que ela está bastante ocupada, e, portanto, para as próximas previsões não irá dar aquela máquina como disponível. É preciso fazer uma diferenciação nesse monitoramento, entre o que está sendo usado localmente, e o que está sendo usado pelo *InteGrade*. Essa é provavelmente a modificação mais urgente na implementação do *LUPA*, e está sendo estudada por um outro aluno do projeto.

- Extender o número de recursos a serem monitorados e terem seu uso futuro previsto. Em particular, o *LUPA* poderia realizar um monitoramento do estado da rede, fator de extrema importância numa grade, principalmente na execução de aplicações distribuídas que realizam intensa comunicação.

- Modificar a implementação de modo a desacoplar o *LUPA* do *LRM*. Como descrito na Seção 3.2, uma instância do *LUPA* é criada no momento que o *LRM* é iniciado. Isso significa que, por exemplo, se tivéssemos 2 *LRM's* executando numa mesma máquina, também teríamos 2 *LUPA's*. Isso estaria errado, já que um único *LUPA* deve executar em cada máquina, independente de quantos *LRM's* estão rodando

ali. Certamente não é comum ter mais de um *LRM* por nó compartilhado, porém esse desacoplamento não deixa de ser algo interessante de ser feito.

Como visto, as possibilidades de dar prosseguimento ao trabalho são muitas. As modificações citadas são mais práticas, dizendo respeito principalmente a modificações na implementação. Porém, certamente existe bastante pesquisa a ser feita nessa área, como por exemplo o estudo de outras possíveis maneiras de se realizar previsões de utilização, e também a investigação de outras heurísticas para realizar o escalonamento de aplicações nesse ambiente de grades computacionais oportunistas.

# Parte II

# Parte subjetiva

# Capítulo 7

## 7.1 O trabalho no projeto *InteGrade*

Meu trabalho no projeto *InteGrade* iniciou-se com uma iniciação científica na segunda metade de meu terceiro ano como aluno do BCC. Apesar de ter como orientador o professor Marcelo Finger, interagi também bastante com o professor Fabio Kon, o coordenador geral do projeto. Reuniões quinzenais são realizadas com toda a equipe do IME-USP envolvida no *InteGrade*, para discutirmos os avanços, os problemas que aparecem, e tudo que tiver relevância para o sistema. Isso permitiu uma grande interação com outros alunos, tanto de graduação quanto de mestrado, além de professores. Houve também interação com participantes do projeto de outros estados, em especial durante o *V Workshop InteGrade*, realizado em Julho de 2008 no próprio IME-USP.

O fato de meu trabalho estar inserido num sistema bem maior teve pontos positivos e negativos, em minha opinião. Por um lado, eu tomei contato com diversas tecnologias e linguagens de programação, como C++, Lua e CORBA. Além disso, pude observar códigos feitos por alunos de mestrado e doutorado, altamente competentes, onde com certeza o aprendizado foi muito grande. Por fim, ver como um sistema dessas proporções, com diversos módulos e milhares de linhas de código, funciona internamente, como é feita a integração de códigos em diferentes linguagens, como é feito o processo de configuração e compilação, etc, pode me proporcionar uma experiência que não costumamos ter nas disciplinas de graduação.

Por outro lado, porém, isso também acarretou uma curva de aprendizado razoavelmente acentuada. Como era de se esperar, a primeira vista a implementação de um sistema desse tamanho pareceu-me extremamente complexa e pouco (ou nada) eu entendia do que estava acontecendo. Mesmo tarefas como a compilação e execução do *InteGrade* eram executadas com certa dificuldade no começo. Um bom tempo passou até que eu conhecesse razoavelmente bem os módulos com os quais meu trabalho se relacionaria, e algumas partes do sistema, até hoje, permanecem desconhecidas por mim.

Além disso, um outro problema no *InteGrade* é que o sistema ainda apresenta problemas de robustez (o que é compreensível, dada sua complexidade). Muitas vezes dava de cara com problemas "inexplicáveis" na execução do *InteGrade*, e muito tempo era perdido até detectar o que havia acontecido, ou pelo menos descobrir como voltar o sistema ao seu funcionamento normal. Claro que, com o tempo, as formas de resolver os erros mais comuns foram sendo aprendidas, mas ainda assim, até hoje ainda acontece de deparar-me com alguma problema não esperado que demoro a conseguir resolver.

Por fim, a conclusão que chego é que fazer um trabalho em um sistema de grande porte como esse tem muitas vantagens, especialmente no que diz respeito a aprendizado e interação. Por outro lado, com o passar do tempo eu fui cada vez mais sentindo falta de

trabalhar em um projeto "exclusivamente meu", ou seja, algo (implementado ou não) que eu desenvolvesse do zero, e onde eu tivesse total controle e conhecimento do que estava acontecendo. Claro que não se pode esperar, de um trabalho desse, os mesmos resultados do que de um trabalho realizado por várias pessoas. Porém acho que também é algo válido, e algo que eu gostaria de experimentar fazer, no IME, no futuro.

## 7.2 Desafios e frustrações

Primeiramente, quanto ao trabalho em si, os principais desafios foram os mencionados na seção anterior: estudar e conhecer o código com o qual iria trabalhar, bastante extenso e muitas vezes utilizando tecnologias completamente desconhecidas por mim (como CORBA, por exemplo); basear todo o trabalho no sistema *InteGrade*, muitas vezes sujeito a falhas ou mesmo com algumas partes, de onde o trabalho de certa forma dependia, ainda incompletas.

E a principal frustração foi de não ter conseguido, da forma esperada, realizar experimentos conclusivos sobre o trabalho realizado. Algo que deu uma certa dor de cabeça foi a execução do *InteGrade* na Rede Linux. Para começar, o *LUPA*, para conseguir realizar previsões confiáveis, precisa de um tempo de "treinamento" [5]. É interessante que as máquinas já estejam coletando dados há 1 mês, aproximadamente, para que as informações do *LUPA* tornem-se mais confiáveis. Porém, quando o *InteGrade* rodava há exatamente 1 mês na Rede Linux, esta começou a ficar extremamente lenta. Diversos alunos começaram a reclamar, já que a piora era notória. Mesmo sem ter dados que indicassem que o *InteGrade* fosse o responsável, os administradores solicitaram que ele fosse desligado para que observassem os resultados. Coincidência ou não, a rede, poucos dias depois, voltou ao normal. Ainda assim, nada indicava que o problema tivesse sido causado pelo *InteGrade*.

Porém, esse período em que o *InteGrade* ficou desligado acabou prejudicando um pouco o trabalho do *LUPA*, já que, como dito, ele precisa da maior quantidade de dados possível para fazer boas previsões. Aproximadamente 15 dias depois, os administradores solicitarem que se subisse o *InteGrade* em metade das máquinas, e depois de mais 15 dias, em todas elas. Mais recentemente a rede tornou a ficar lenta, e novamente pediu-se que desligasse o *InteGrade* para ver se alguma correlação realmente existe. Como nisso faltava muito pouco para o término do TCC, acabei pedindo apenas mais alguns dias antes de desligá-lo novamente.

Porém, é claro que esses problemas na Rede Linux não foram os únicos que impediram a realização dos testes como esperado. Não posso negar que errei no planejamento e estimação de tempo de minhas atividades, quando acabei levando bem mais tempo do que achei que levaria para a conclusão das implementações. Isso se deveu tanto a imprevistos que acabaram aparecendo no meio do caminho, como a períodos de tempo em que me dediquei menos a essas atividades, devido a outras obrigações acadêmicas. Porém, como essas atribuições eram esperadas, talvez pudesse ter organizado melhor o tempo. De toda forma, com isso comecei a fase de testes tarde, tendo menos tempo para enfrentar os problemas encontrados nessa fase.

Falando agora em relação ao curso de BCC em si, poderia dizer que foi um desafio ter feito o curso "ideal", seguindo praticamente a risca a sugestão de carga horária e sem reprovar em nenhuma matéria. Tive grandes problemas, em especial, para administrar meu tempo, onde em períodos mais tranqüilos eu dava pouca atenção às disciplinas, porém depois tinha que suar bastante para concluir todas as tarefas. E, com isso, um

outro grande desafio foi controlar o nível de estresse que a cada semestre acumulava-se mais. A verdade é que, no fim das contas, o curso de BCC é bem puxado, e se não tomar cuidado com a administração do tempo e das atividades, o aluno pode ver-se numa situação onde dedica praticamente todo seu tempo à faculdade, se quiser cumprir com suas obrigações e ser aprovado em todas as disciplinas. E, sem muito tempo para atividades de relaxamento e lazer, realmente o estresse só tende a aumentar.

Não fica difícil imaginar, então, qual minha principal frustração com o curso de BCC. Quando entrei no IME, não tinha a menor idéia do que consistia o curso. Poderia, facilmente, ter "quebrado a cara", me arrependendo profundamente da escolha e possivelmente desistindo do curso. Afinal, o conteúdo extremamente teórico e recheado de matemática dos 2 primeiros anos pode assustar qualquer um. Porém, mesmo sem esperar isso, eu acabei me identificando e gostando muito do curso. Sempre me empolguei com muitas coisas, em especial com a parte prática, como a realização de EPs (confesso que me diverti muito fazendo alguns deles). Porém, dado o nível de estresse e cansaço que acumulei com o passar dos semestres, como dito no parágrafo anterior, fui cada vez mais perdendo essa empolgação com o curso. Claro que isso é possivelmente passageiro, fruto do cansaço que de fato deve ser o maior possível no fim do curso, mas a verdade é que a computação não me fascina mais tanto como há 2 anos atrás. Mas, espero que depois de algum tempo de descanso, eu possa retomar o mesmo interesse de antes, e principalmente me organizar melhor para não sofrer mais tanto com esse excesso de cansaço, agora no mestrado onde espero ser aprovado.

## 7.3 Disciplinas mais relevantes

**MAC0122 – Princípios de Desenvolvimento de Algoritmos**
**Professor: Carlinhos**

Essa disciplina é uma das mais importantes, se não a mais importante, não só para esse como para qualquer outro trabalho de conclusão de curso. Claro que disciplinas muito mais específicas de cada assunto são ministradas posteriormente, porém é nessa disciplina que todos tomam contato pela primeira vez com o que é Computação de verdade. E, os conceitos aqui vistos, mesmo que básicos, estão em absolutamente todos os lugares onde haja algum código sendo produzido. No meu trabalho não foi diferente, direta ou indiretamente muito das estruturas de dados e algoritmos ali vistos foram sendo aplicados.

**MAC0323 – Estruturas de Dados**
**Professora: Cris**

Assim como no caso de MAC0122, essa disciplina também aborda diversos assuntos que simplesmente formam toda a base da Ciência da Computação. Aqui, já são vistos algoritmos e estruturas mais complexas, a maioria (ou talvez todos) não usados por mim diretamente no meu trabalho. Porém, novamente, os conceitos vistos aqui servem principalmente para aprendermos como pensar de verdade na hora de resolver problemas na Computação.

**MAC0211 – Laboratório de Programação I**
**Professor: Roberto Hirata**

Aqui, tivemos o primeiro contato com o desenvolvimento de um *software* de grande porte. Apesar de ter sido um jogo o programa feito, muito pôde ser aprendido sobre as dificuldades em projetos de tamanho maior. E isso certamente serviu como uma base para o que eu encontraria no projeto *InteGrade*. Muito importante também foi aprender a mexer em ferramentas como o `SVN` e o `Makefile`. Ambos são muito importantes dentro do projeto *InteGrade*, e a primeira vez que tomei contato com ambos foi justamente em MAC0211.

**MAC0441 – Programação Orientada a Objetos**
**Professor: Alfredo Goldman**

Essa disciplina, que na minha opinião deveria tornar-se obrigatória no curso de BCC, é extremamente importante atualmente. Afinal, o mundo da orientação a objetos está cada vez mais dominando o mercado, impulsionado principalmente pela popularização da linguagem Java. No *InteGrade*, especificamente, todo o sistema é orientado a objetos, seja utilizando a linguagem Java, no servidor, ou C++, nos nós compartilhados. Portanto, todo o desenvolvimento que realizei foi utilizando Orientação a Objetos, e muitos dos conceitos dessa disciplina foram importantíssimos para meu trabalho.

**MAC0422 – Sistemas Operacionais**
**Professor: Roberto Hirata**

No meu trabalho com o *LUPA* e o *InteGrade*, muitas vezes tive de lidar com conceitos relacionados ao sistema operacional Linux. Principalmente, no que diz respeito ao monitoramento do uso de recursos como CPU e memória, e na execução de processos. Muito disso foi abordado na disciplina de MAC0422, o que ajudou bastante como um conhecimento prévio de como as coisas estão organizadas e como são realizadas num sistema operacional.

**MAC0431 – Introdução à Computação Paralela e Distribuída**
**Professor: Gubi**

Apesar de não ter influência direta em meu trabalho em si, estudar conceitos da área de sistemas distribuídos foi bastante importante para um melhor entendimento do sistema *InteGrade* como um todo. O *InteGrade* está apto a rodar, por exemplo, aplicações MPI, tecnologia essa apresentada pelo professor Gubi em MAC0431. Como dito, conhecer mais sobre isso não foi vital para meu trabalho, mas ajudou bastante a situar-me no ambiente onde estava desenvolvendo.

## 7.4 Estudos para trabalho futuro

Caso resolvesse continuar o trabalho aqui realizado, provavelmente poderia tomar mais de um caminho. Poderia aprofundar meus estudos em lógica e inteligência artificial, para retomar a pesquisa mais teórica de [4] para estudar melhorias ou mesmo outros métodos de se realizar previsões de usabilidade. Por outro lado, poderia seguir também um encaminhamento mais direcionado a própria área de desenvolvimento de sistemas,

para continuar no desenvolvimento do escalonador, adicionando recursos mais avançados a ele.

Pretendo, no próximo ano, ingressar no mestrado dentro do próprio IME-USP. Não tenho definido ainda qual será o tema de meus estudos nessa pós-graduação, porém o mais provável é que eu não prossiga no *InteGrade*, pelo desejo de começar algo novo como comentado mais acima. Porém, o mais provável é que eu siga mesmo para algum projeto voltado a área de sistemas.

# Referências Bibliográficas


[1] **Website do InteGrade:** `http://www.integrade.org.br`.

[2] A. Goldchleger, *InteGrade: Um Sistema de Middleware para Computação em Grade Oportunista*. PhD thesis, Universidade de São Paulo, 2004.

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

[4] G. Bezerra, *Análise de Conglomerados Aplicada ao Reconhecimento de Padrões de Uso de Recursos Computacionais*. PhD thesis, Dissertação de mestrado. Universidade de São Paulo, 2006.

[5] D. M. R. Conde, *Análise de Padrões de Uso em Grades Computacionais*. PhD thesis, Dissertação de mestrado. Universidade de São Paulo, 2008.

[6] A. Goldchleger, F. Kon, A. Lejbman, M. Finger, and S. Song, "Integrade: rumo a um sistema de computação em grade para aproveitamento de recursos ociosos em máquinas compartilhadas," tech. rep., IME-USP, 2002.

[7] R. F. Lopes and F. J. da Silva e Silva, "Migration Transparency in a Mobile Agent Based Computational Grid," in *Proceedings of the 5th WSEAS International Conference on SIMULATION, MODELING AND OPTIMIZATION. 1st WSEAS International Symposium on GRID COMPUTING*, (Corfu, Greece), pp. 31–36, August 2005.

[8] R. F. Lopes and F. J. da Silva e Silva, "Strong Migration in a Grid based on Mobile Agents," *WSEAS Transactions On Systems, Greece*, vol. 4, pp. 1687–1694, October 2005. ISSN 1109-2777.

[9] F. J. da Silva e Silva, R. F. Lopes, B. B. de Sousa, A. E. B. Viana, and S. A. de Sousa, "MAG, um middleware de grade baseado em agentes: estado atual e perspectivas futuras," in *In WCGA 2006: Anais do IV Workshop on Computational Grids and Applications. Simpósio Brasileiro de Redes de Computadores (SBRC2006)*, (Curitiba), Sociedade Brasileira de Computação, Maio 2006.