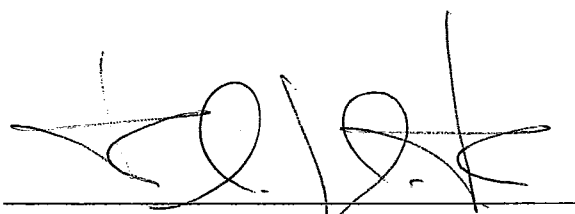
APRENDIZADO POR REFORÇO APLICADO À META-ESCALONAMENTO

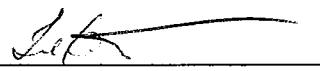
Bernardo Fortunato Costa

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:



Prof. Felipe Maia Galvão França, Ph.D.



Profa. Inês de Castro Dutra, Ph.D.

Prof. Eugene Francis Vinod Rebello, Ph.D.

RIO DE JANEIRO, RJ - BRASIL

SETEMBRO DE 2007

COSTA, BERNARDO FORTUNATO

Aprendizado por Reforço aplicado à Meta-escalonamento [Rio de Janeiro] 2007

XI, 84 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2007)

Dissertação – Universidade Federal do Rio de Janeiro, COPPE

1 - Meta-escalonamento em grids

2 - Heurísticas de escalonamento

3 - Sistemas distribuídos

I. COPPE/UFRJ II. Título (série)

À todos aqueles que ainda não encontraram a luz no fim do túnel.

Agradecimentos

Antes de mais nada é necessário fazer um registro de agradecimento à todos aqueles que tornaram este trabalho possível de ser realizado. Sendo assim, eu gostaria de iniciar minha lista de agradecimentos por todos os integrantes do GRIDS Laboratory na Universidade de Melbourne, citando aqui nominalmente Srikumar Venugopal, por ter desenvolvido e dado continuidade ao GridbusBroker como ferramenta de código aberto, além de um agradecimento especial à Krishna Nadiminti, pelo apoio dado a mim pessoalmente na tarefa que tive de entender e alterar a referida ferramenta, o que foi parte substancial deste trabalho, além de também terem aceitado algumas alterações por mim propostas. Aos demais integrantes, toda a minha admiração e respeito pelo belo trabalho realizado.

À Lee David Painter, por ter possibilitado a continuação do projeto SSHTools sob direção de um novo grupo de desenvolvedores e sob novos termos de distribuição, possibilitando que um projeto vital para o GridbusBroker continue vivo. À Chris Alex Thomas, por ter tomado a iniciativa e coordenado a reativação do projeto SSHTools juntamente com Sascha Hunold. Ao próprio Sascha Hunold um agradecimento muito especial, por ter realizado diversas correções por conta própria na ferramenta SSHTools disponibilizando-as ao público, além de ter pessoalmente me ajudado na correção e validação desta ferramenta em um momento crucial para o desenvolvimento do meu trabalho, onde, sem sua ajuda, eu teria sido forçado a mudar todo o escopo deste trabalho.

Um agradecimento muito especial também a todos aqueles que de alguma forma me disponibilizaram recursos computacionais com os quais eu pude montar um ambiente para meus experimentos. Sendo assim, eu gostaria de agradecer ao Núcleo de Computação de Alto Desempenho (NACAD) da UFRJ, citando nominalmente Albino Aveleda e Miriam Costa, pela disponibilização de uma conta em seus domínios para meus experimentos. Gostaria de agradecer ao Prof. Alberto Santoro da UERJ e citar nominalmente Patrícia Bittencourt Sampaio, pela ajuda e pela disponibilização de mais um recurso computacional na fase final dos experimentos. Da mesma forma, agradeço ao Prof. Cláudio Geyer

da UFRGS pela disponibilização de uma conta em um cluster nesta universidade, assim como não poderia deixar de citar nominalmente Patrícia Kayser Mangan Vargas e Éder Fontoura, além de todo o resto de sua equipe pela assistência prestada tanto em momentos de dúvidas quanto na solução de problemas. Um agradecimento especialíssimo também à Prof. Marluce Rodrigues Pereira da UFLA, não só por ter me oferecido uma conta no Laboratório de Computação Científica em sua universidade, como por tê-la bancado em vários momentos durante os experimentos. Ao Prof. Cláudio Luis Amorim do Laboratório de Computação Paralela (LCP) da COPPE/UFRJ por ter disponibilizado uma conta no referido laboratório. Aos Prof. Valmir Carneiro Barbosa, Inês de Castro Dutra e Felipe Maia França Galvão por terem cedido uma parte das máquinas disponíveis do Laboratório de Inteligência Artificial para a construção de um cluster que veio a servir aos meus experimentos.

Um agradecimento também aos meus familiares do Rio de Janeiro que possibilitaram que este trabalho tenha sido realizado num ambiente mais tranquilo do que teria sido sem sua presença. Estendo este agradecimento também a Fundação Coppetec e ao LCP em conjunto com o CNPQ pela concessão de bolsas que possibilitaram a continuação deste trabalho. Um agradecimento também à Prof. Marta Mattoso por ter aceitado me ajudar num período conturbado.

Por fim agradeço aos meus amigos. Aos amigos cuja amizade vem do período de graduação, muitos dos quais são uma referência para mim tanto de alunos, como profissionais ou pessoas humanas até hoje, e que sem esta referência eu possivelmente jamais teria cogitado a hipótese de entrar para o mestrado. Aos amigos cuja amizade se fez aqui dentro desta instituição, que também serviram como referência, e que além disso me ajudaram diretamente na realização deste trabalho, principalmente na solução de dúvidas referentes a edição de documentos LaTeX. Seria injusto e contraproducente citar nomes aqui pois seriam muitos, desde integrantes do Laboratório de Algoritmos e Combinatória até o pessoal do Laboratório da Star One, passando por todos os demais laboratórios das salas H-317 e I-246, LCP, além dos integrantes de outros Programas de pesquisa da COPPE ou mesmo da graduação.

À todos vocês, meu muito obrigado !

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

APRENDIZADO POR REFORÇO APLICADO À META-ESCALONAMENTO

Bernardo Fortunato Costa

Setembro/2007

Orientador: Inês de Castro Dutra
Programa: Engenharia de Sistemas e Computação

Algoritmos de escalonamento de tarefas em grids são um dos fatores primordiais da utilização eficiente deste tipo de infraestrutura. Estes sofrem das limitações que um ambiente de grid impõe sobre a real estimação do estado de seus recursos computacionais. Neste contexto, este trabalho realiza um estudo em ambiente real de execução sobre dois possíveis algoritmos de escalonamento baseados na estimação da eficiência dos recursos computacionais, feita a partir da técnica de aprendizado por reforço. É realizada uma discussão sobre o ajuste possível de parâmetros internos à estimação da eficiência dos recursos computacionais, e sobre o comportamento destes algoritmos na presença de reescalonamento. Os resultados mostram os ganhos ou perdas reais obtidas nos cenários avaliados e servem também para validar trabalhos já realizados em ambiente simulado.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

REINFORCEMENT LEARNING APPLIED TO META-SCHEDULING

Bernardo Fortunato Costa

September/2007

Advisor: Inês de Castro Dutra
Department: Systems Engineering and Computer Science

Scheduling of jobs on grids is one of the most important tasks to use resources efficiently. In such environments, the resource's state is estimated with great uncertainity and the information is not completely reliable. This work studies two scheduling algorithms based on resource's efficiency, which use the reinforcement learning technique. A discussion of internal parameter's tuning of this technique is performed as well as the behaviour of these two heuristics in the presence of rescheduling. The results here presented show the gains and losses on the built scenarios and validate other related works produced on simulated environments.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Este capítulo fará uma breve apresentação do trabalho que foi realizado. A área de computação em grade ou simplesmente grid vem atraindo grande interesse tanto por parte da comunidade científica quanto da indústria. A primeira, um tradicional consumidor de computação de alto desempenho desde que esta surgiu, iniciou os estudos desta área para seu próprio consumo, e recentemente intensificou o seu uso, pois a medida que a computação torna-se popular, mais este tipo de infraestrutura vem a ser cada vez mais utilizado.

Com o passar do tempo, a indústria viu a possibilidade de ganhos econômicos nesta área, ao visualizar futuramente a construção de um ambiente computacional de alto desempenho, onde os usuários pagariam para sua utilização ao invés de tentarem construí-lo e posteriormente ter que arcar com o custo de mantê-lo. Há quem chegue a prever o dia em que computação será um serviço de utilidade pública como a eletricidade, por exemplo. E para que isso seja possível, caso venha a se tornar realidade, imagina-se que a infraestrutura necessária seja criada a partir de grids.

Por conta destes fatores, a área de computação em grade é uma área interessante para estudo e de grande atividade acadêmica. Os esforços para construir um ambiente computacional dedicado a grids acabaram por produzir o Globus [24], que entre outros softwares relacionados a grids, se tornou o mais popular na constituição deste tipo de ambiente, muito porque é, também, um dos mais completos. Embora tenha se tornado um padrão de fato como constituinte de grids, este é muito extenso e de difícil configuração. Não à toa, existem esforços na criação de outros softwares de infraestrutura de grid, sendo estes em geral mais simples e de uso mais fácil que o Globus.

Mesmo que boa parte da infraestrutura esteja desenvolvida, ou venha sendo continu-

amente desenvolvida, ainda persistem questões importantes a serem desvendadas. Uma delas, particularmente difícil, é o escalonamento de jobs neste ambiente. Este se trata de encontrar uma alocação entre jobs para execução e recursos disponíveis de maneira a minimizar o tempo total de execução. Um outro objetivo desejável de algoritmos de escalonamento é que estes evitem a sobrecarga de alguns recursos computacionais em detrimento de outros.

O problema do escalonamento de jobs em um ambiente computacional é reconhecidamente difícil até por pertencer à classe de problemas NP-completos, para os quais não é esperado que se encontre um algoritmo polinomial que resolva deterministicamente o problema apontando o máximo ou mínimo desejado. Além disso, o ambiente de grid traz complicadores que tornam as informações obtidas a respeito do estado e monitoramento dos recursos computacionais pouco confiáveis. Isto torna este ambiente extremamente adverso para o desenvolvimento de heurísticas em algoritmos de escalonamento, pois se um problema NP-completo já é difícil de ser tratado mesmo quando existe boa quantidade de informações confiáveis sobre o problema, na sua ausência ou escassez o quadro fica ainda mais complicado.

Muitos trabalhos foram feitos no estudo de algoritmos de escalonamento em grids. O termo meta-escalonamento substitui escalonamento em grids neste trabalho, pois trata-se neste ambiente muitas vezes de se alocar jobs não diretamente a recursos, mas sim a outros escalonadores pertencentes à grade computacional. Apenas para citar alguns, temos o MARS [2], Nimrod/G [3], AppLeS [5], entre outros. Estes e outros estão descritos no capítulo 4 que trata deste assunto em mais detalhes.

Vários dos trabalhos sobre meta-escalonamento neste ambiente é realizado em simuladores, muito por conta da dificuldade, em alguns casos, de se possuir ou construir um ambiente computacional de grid, ou até mesmo pela comodidade de execução, repetição e rapidez dos testes simulados. Muitos destes simuladores são bastante complexos e bem trabalhados, simulando várias situações vistas em ambiente real. Mas mesmo o melhor entre todos os simuladores é incapaz de nos fornecer certeza do que ocorreria em um ambiente real, tornando interessante sempre verificar o que de fato ocorre com os algoritmos quando estes enfrentam situações reais.

E é neste ponto que encontramos o objetivo e a contribuição primordial deste trabalho: verificar o desempenho de dois algoritmos de escalonamento em grid [14, 18], em um ambiente real. Estes foram implementados no GridbusBroker que se trata de um ambiente

de trabalho mais simples que o Globus, mas também voltado para grid. O que está por trás da escolha destas duas heurísticas em questão e da forma como elas foram implementadas, está relacionada à simplicidade da heurística de Aprendizado por Reforço e sua possível utilização como ferramenta de análise para algoritmos de escalonamento em grids, haja visto os problemas relativos ao escalonamento de jobs em grids que serão detalhados mais a frente.

Isto posto, este trabalho está organizado da forma descrita a seguir. O capítulo 2 faz uma revisão teórica sobre conceitos e trabalhos desenvolvidos na área de grids e sistemas distribuídos. O capítulo 5 explica em maiores detalhes o trabalho realizado tanto em termos de desenvolvimento e programação dos algoritmos, quanto ao ambiente disponível para realização dos experimentos além de detalhar as ferramentas utilizadas. O capítulo 6 mostra os resultados. Ao fim, o capítulo 7 resume as principais conclusões possíveis deste trabalho e mostra possíveis direções para trabalhos futuros, tendo como base o que foi aqui realizado.

Capítulo 2

Revisão Bibliográfica

Neste capítulo serão revisados os conceitos mais importantes na área, revendo as principais definições de grid, estudando o contexto de escalonamento nesse tipo de ambiente, e posteriormente, fazendo uma pequena apresentação de escalonadores e heurísticas relacionadas ao tema, existentes na literatura. Grid ou grade computacional é uma generalização de um sistema de computação distribuída em larga escala. Uma definição mais completa sobre sistema de computação distribuído será dada na seção 2.2 sobre taxonomia.

Segundo Foster *et al.* [13], os problemas reais e específicos que envolvem a concepção de grid dizem respeito a coordenação de recursos compartilhados e resolução de problemas em organizações virtuais dinâmicas e multi-institucionais. Um conjunto de indivíduos e/ou instituições definidas por suas regras de compartilhamento formam o que chamamos de organizações virtuais (VO[1]). As VOs permitem que grupos heterogêneos de organizações e/ou indivíduos compartilhem recursos de uma maneira controlada, de forma que seus membros alcancem um objetivo comum.

O cenário genérico de utilização de um grid remete a um número de usuários não-confiáveis entre si, com algum ou até mesmo nenhum grau de relacionamento, os quais desejam compartilhar recursos para executarem alguma tarefa. Esse compartilhamento vai além da simples troca de documentos e pode envolver acesso direto a software remoto, computadores, dados, sensores e outros recursos. A natureza dinâmica dos relacionamentos de compartilhamento torna necessária a existência de mecanismos para descoberta e caracterização da sua natureza em um determinado instante de tempo. Uma caracterís-

[1] virtual organizations

tica importante nestes sistemas é que estes sejam interoperáveis[2] pois supõe-se que estes deverão cooperar entre si.

Ainda segundo Foster *et al.* [13], uma arquitetura de grid é antes de tudo e principalmente uma arquitetura de protocolos, com estes definindo os mecanismos básicos pelos quais os usuários e os recursos da VO negociam, administram, estabelecem e exploram seus relacionamentos de compartilhamento. Neste contexto, uma arquitetura de grid enfatiza a identificação e a definição de protocolos e serviços em primeiro plano, e interfaces (APIs[3]) e bibliotecas (SDKs[4]) em segundo plano.

Os protocolos de conectividade e utilização de recursos facilitam o compartilhamento de recursos individuais e também são o gargalo da aplicação. Neste ponto, surge uma classificação de cinco camadas na arquitetura de protocolos do grid. Estes seriam referentes à aplicação, coleção, recursos, conectividade e fábrica. A figura 2.1 estabelece uma relação entre estas camadas da arquitetura de protocolos do grid com as camadas do modelo OSI de rede, fazendo um mapeamento das camadas de grid para as camadas de rede.

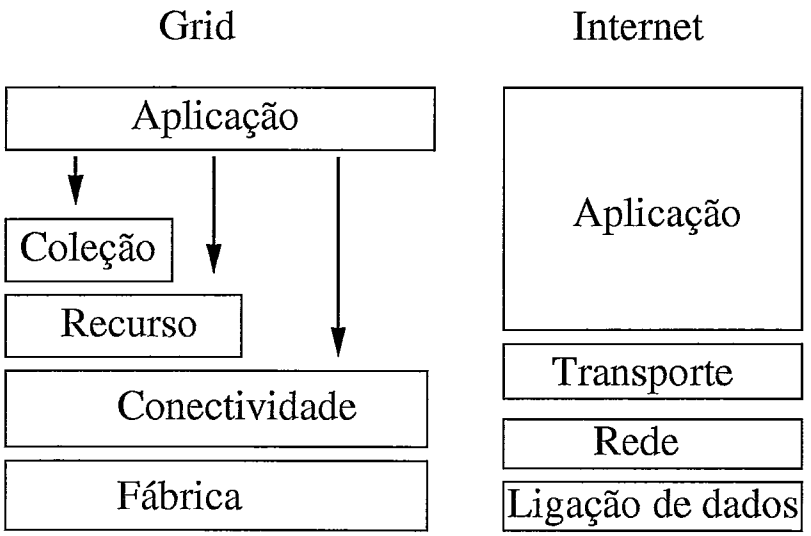


Figura 2.1: Camadas da arquitetura de Grid e sua relação com as camadas do Modelo OSI

Componentes de fábrica implementam as operações locais e específicas do recurso como resultado de operações de compartilhamento em níveis mais altos. A camada de conectividade define os protocolos de comunicação e autenticação, que possibilitam a

[2]podemos entender interoperabilidade como a capacidade destes sistemas constituintes da grade de comunicarem entre si de maneira transparente

[3]Application Programming Interface

[4]Standard Development Kit

troca de dados entre os componentes da camada de fábrica. A camada de recursos constrói sobre a camada de conectividade protocolos para negociação, iniciação, monitoramento, controle, contabilidade e pagamento de operações compartilhadas sobre recursos individuais. Estes podem ser divididos em protocolos de informação e de administração. A camada de coleção contém protocolos que não são associados a nenhum recurso individual, mas sim com recursos globais em sua natureza que capturam interações através de coleções de recursos. A última camada, de aplicação, representa as aplicações dos usuários que operam dentro do ambiente da VO.

O desenvolvimento de programas em ambiente de grid introduz desafios que não são encontrados na computação tradicional sequencial ou mesmo paralela, tal como variedade de domínios administrativos, novos modos de falhas e uma grande variabilidade de desempenho.

2.1 Ambientes distribuídos e grid

Faremos nesta seção uma comparação entre ambientes distribuídos tradicionais e grid, revelando as diferenças intrínsecas entre estas categorias de ambientes de trabalho distribuídos. Será realizada uma análise *bottom-up* das camadas virtuais apresentadas e por fim veremos que a camada virtual é bastante diferente nestas duas categorias de ambientes distribuídos.

Aplicações distribuídas são constituídas de um número de processos cooperativos que exploram os recursos fracamente acoplados de sistemas de computação. Em geral, os processos nestes ambientes se comunicam por bibliotecas de passagem de mensagem. Um ambiente distribuído convencional assume um conjunto (*pool*) de recursos computacionais. Um nó (*node*) é uma coleção de recursos tratada como uma unidade deste. O pool de nós é um conjunto de computadores pessoais e alguns supercomputadores, dado que o usuário tem acesso pessoal a eles.

Segundo Németh e Sunderam [21], sistemas distribuídos convencionais assumem a existência de um conjunto (*pool*) de nós computacionais, que formam uma máquina paralela virtual. Nestes ambientes, o conjunto de nós é estático ou muda raramente. O usuário uma vez logado em um nó tem permissão ou pode executar seus jobs nos recursos pertencentes sem necessidade de outras autorizações. Já em ambientes de grid, é assumido a existência

Sistemas distribuídos convencionais	Grids
conjunto de nós computacionais virtuais	conjunto de recursos virtuais
usuário possui acesso a todos os nós do conjunto	usuário possui acesso ao conjunto mas não aos sítios individuais
acesso a um nó significa acesso a todos os recursos deste	acesso aos recursos podem sofrer restrições
usuário tem conhecimento das características do nó	usuário possui pouco conhecimento sobre os sítios
nós pertencem a um mesmo domínio	recursos estão espalhados por diversos domínios
conjunto de nós entre 10 a 100, mais ou menos estáticos	conjunto de nós entre 1000 a 10000, dinâmicos

Tabela 2.1: Comparação entre grids e sistemas distribuídos convencionais

de um conjunto de recursos virtuais, ao invés de simples nós computacionais, para os quais a infraestrutura de grid torna transparente este processo.

Um grid assume um pool virtual de recursos ao invés de um pool de nós computacionais. Os recursos em geral existem dentro dos nós que ficam distribuídos geograficamente ou espalhados em diversos domínios administrativos. Em um grid, o pool virtual é dinâmico e diverso. Devido a estas características, o usuário em geral não possui conhecimento sobre o estado, tipo ou qualidade dos recursos constituintes do pool. Como o conjunto de recursos é dinâmico em um grid, o usuário não tem conhecimento do tipo, estado ou características dos recursos pertencentes ao pool. A tabela 2.1 resume as diferenças entre sistemas distribuídos convencionais e grids.

Os processos distribuídos a serem executados devem ser mapeados em nós escolhidos no pool. E assim sendo, o usuário deve ter um acesso de login em todos os nós do pool. Ao contrário de outros tipos de sistema, que tentam satisfazer as necessidades dos recursos localmente, em grids a seleção de recursos é realizada tendo como hipótese um conjunto abundante e comum de recursos. Isto significa que os recursos são primeiramente selecionados globalmente e depois é que é feito o mapeamento entre a seleção de recursos e os jobs.

Em um ambiente de grid, evita-se que acesso aos nós seja controlado via login, e por isso é necessário possuir credenciais de alto nível na camada virtual para identificar e au-

torizar a execução dos jobs como se eles pertencessem a usuários locais. O mapeamento entre estas partes tem que ser explícito, e é realizado pelo processo de associação de recursos. Em sistemas distribuídos convencionais, há um mapeamento implícito entre os recursos abstratos e a sua parte física real.

Em ambientes de grid, o usuário da máquina virtual é diferente do usuário em nível físico, e desta forma, nele deve haver uma maneira de achar um mapeamento entre as credenciais do usuário e sua identidade local nos nós. Em outras palavras, os níveis físicos e virtuais em um grid são completamente distintos. O usuário não possui conhecimento a respeito da disponibilidade e especificações dos recursos disponíveis, e sendo assim, são necessários alguns componentes tais como um sistema de informação capaz de descobrir e procurar recursos e um provedor local de informação que conhece as características dos recursos locais e que seja capaz de informar o catálogo geral sobre estes recursos.

O usuário em um grid é também uma abstração dada pelo mapeamento entre credenciais válidas e contas locais. Para isso, é necessário um mecanismo de segurança que aceite usuários globais e os autentique, assim como gerenciadores de recursos que autorizem usuários autenticados a utilizarem certos recursos. Alguns outros serviços também são necessários como transferência de arquivos ou co-alocação, por exemplo.

O objetivo do uso de grids não é exatamente em aplicações de alto desempenho, mas sim em aplicações que necessitem de um ambiente distribuído grande e em larga escala. Obviamente que uma grande quantidade de recursos disponíveis permite também a realização de computação de alta performance, ainda que a infraestrutura montada para o grid introduza uma série de atrasos[5] na execução de tais aplicações.

De acordo com [21], temos ainda um conjunto de regras que é válido para sistemas distribuídos convencionais, mas não para grids. Abaixo segue a caracterização de algumas destas.

Em sistemas distribuídos convencionais, existe um pool virtual de nós para cada usuário, nos quais o usuário possui um login válido em cada nó. nesse contexto, as tarefas dos processos foram pré-instaladas em alguns nós. A seleção de recursos possui um nível de abstração irrelevante, seja ela feita pelo usuário ou por algum escalonador. Os recursos pendentes de alocação poderão ser satisfeitos se pertencerem ao mesmo nó onde executa o processo.

[5]na literatura, este aparece definido como overhead

Há, ainda, um conjunto de estados pelos quais a tarefa passa durante a sua execução em sistemas distribuídos convencionais. Quando todas as requisições são satisfeitas e não há bloqueio de comunicação, o processo pode entrar no estado de execução. Caso este necessite de recursos adicionais durante a execução, o processo passará ao estado de espera[6]; um novo processo criado por duplicação pertence sempre ao mesmo usuário e nó e não possui nenhum novo recurso. Ao enviar uma mensagem, este entrará ou não no estado de espera, caso a mensagem seja bloqueante ou não, respectivamente. O mesmo acontece no recebimento, caso o recebimento seja bloqueante ou não, fazendo o processo entrar no estado de espera por resposta[7]. Por fim, o evento finalização[8] descreve o fim do processo.

Feita essa caracterização de sistemas distribuídos convencionais, contrasta-se agora estas regras para o caso de um grid. Nele, o estado inicial é o mesmo, embora haja uma diferença entre como é feita a aquisição de recursos e nós, pois nele os recursos é quem determinam os nós e não o contrário. Na fase de seleção de recursos, a idéia é trocar a ordem entre alocação de nós e recursos, mas mantendo a regra ou critérios de seleção que podem avaliar nos recursos seu desempenho, disponibilidade, carga atual ou prevista, entre outros. Deve haver um mapeamento explícito em um grid entre os recursos abstratos e físicos, em contraposição a um mapeamento implícito nos sistemas convencionais distribuídos, no que se refere à abstração dos recursos. Em um grid existem usuários globais e locais. Um usuário global não possui necessariamente acesso de login em nós, embora o usuário local possua este privilégio, no que se refere aos mecanismos de controle de acesso.

De fato, o mínimo que um grid deve prover são as abstrações entre recursos e usuários, com certo nível de equivalência entre elas. A abstração de recursos requer um serviço de informação local que possua conhecimento dos recursos locais. Da mesma maneira, é necessário um serviço que aceite e autentique usuários globais, assim como um gerenciador local que autorize os usuários a certos recursos.

Ainda segundo Németh e Sunderam [20], existem duas funcionalidades essenciais que um grid deve suportar: a separação do mundo real do virtual. Sistemas distribuídos convencionais são baseados na propriedade dos recursos, enquanto que os sistemas de grid se focam no compartilhamento destes. Podemos dizer que em sistemas distribuídos con-

[6] waiting
[7] receive waiting
[8] termination

vencionais a camada virtual é apenas uma visão diferente da camada física, enquanto que em sistemas de grid tanto os usuários quanto os recursos aparecem de maneira diferente nestas camadas, sendo necessário um mapeamento apropriado entre elas.

2.2 Taxonomia e Terminologia

Esta sessão pretende introduzir ao leitor conceitos básicos, classificações e terminologias comuns a grids ou computação em grade. Utilizaremos a taxonomia proposta em [17], a qual é focada no sistema de gerenciamento de recursos da grade. Serão discutidas a definição de grid, de recurso no contexto de grid e de sistemas de gerenciamento de recursos. Em seguida, será realizada uma classificação de sistemas de gerenciamento de recursos, com uma pequena discussão sobre o papel destes no escalonamento geral do sistema.

Recapitulando, grid ou grade computacional foi definido como uma generalização de um sistema de computação distribuída em larga escala. Um sistema de computação distribuído pode ser visto como um conjunto de máquinas heterogêneas que acordam entre si o compartilhamento de seus recursos locais, com o objetivo de formar um computador virtual. Suas máquinas podem estar distribuídas através de vários domínios organizacionais e administrativos. Sua arquitetura é determinada tanto pelos objetivos de sua construção quanto pelas características da aplicação ao qual este é destinado.

Podemos distinguir três objetivos diferentes na construção de uma grade computacional e assim, classificarmos os grids em três tipos: grid computacional, grid de dados e grid de serviços. Um grid computacional é aquele que possui uma elevada capacidade computacional, maior que qualquer uma das máquinas disponíveis na grade sozinhas, destinado a executar aplicações simples, com o objetivo de diminuir seu tempo de execução através da paralelização da tarefa[9]. Um grid de dados é designado para sistemas que provem infraestrutura de repositório de dados tal como bibliotecas digitais ou *data warehouses*. Aplicações típicas destes sistemas são a mineração de dados que correlacionam informações de várias fontes de dados diferentes[10]. Já uma grid de serviços se caracteriza por sistemas cujos serviços não podem ser providos por uma única máquina. São exemplos deste tipo de grid aplicações multimídia em tempo real como vídeo sob demanda,

[9] podemos entender aqui tarefa como o conjunto de execuções desejadas pelo usuário

[10] também conhecido como data sources

aplicações de simulação de alta fidelidade, nas quais é necessário adicionar mais máquinas para manter ou incrementar a fidelidade, ou sistemas de interação humana com realidade virtual.

Recurso é uma entidade reutilizável do grid que é empregada para satisfazer um job ou uma requisição de um outro recurso. Pode se tratar de máquinas, banda de transmissão, espaço em disco ou memória, ou mesmo um conjunto de máquinas, rede e aplicativos ou serviços. Um provedor de recurso é um agente, o qual tem controle sobre o recurso. Consumidor de recurso é também definido como um agente, o qual controla o consumo do recurso. Um sistema de gerenciamento de recursos (RMS[11]) pode ser definido como um serviço que provê, para um sistema distribuído, em uma rede de computadores, o gerenciamento dos recursos conhecidos que estão disponíveis para este, de tal forma que alguma métrica centrada ou no sistema ou na unidade de trabalho (*job*) ou em ambos é otimizada.

O sistema de gerenciamento de recursos é quem administra os recursos disponíveis para a grade, ou seja, o escalonamento de processadores, largura de banda e espaço em disco, entre outros. Alguns recursos poderão estar vinculados a mais de um provedor. Estes, por sua vez, podem possuir políticas administrativas diferentes ou terem recursos heterogêneos. Em alguns casos, devido a esta diversidade de políticas administrativas e heterogeneidade de recursos, é necessário trabalhar com uma federação de RMS ao invés de um RMS centralizado, nos quais se farão presentes um conjunto de protocolos capazes de interoperar estes recursos compartilhados.

Generalizando, podemos dizer que dar suporte à variedade de políticas possíveis dentro de cada domínio administrativo, implica em se ter mecanismos de escalonamento que levem em consideração a otimização de um problema com vários critérios e restrições. Além de escalabilidade, tolerância a falhas, estabilidade, tempo de resposta, é necessário levar em consideração também a propriedade e o responsável pelos recursos distribuídos. De maneira geral, uma aplicação em um grid pode possuir diversos componentes sendo que cada um destes pode estar associado à um recurso diferente na grade. Por isso, de um RMS é esperado possuir uma abstração global da aplicação para satisfazer seus requisitos, como co-alocação, qualidade de serviço (*QoS*), ou prazos para execução[12], o que torna os RMS de grids bastante complexos. Tal é a variedade de parâmetros a serem otimizados

[11] de Resource Management System
[12] deadlines

que pode ser necessário se utilizar mais de um RMS, num ambiente em que estes cooperem entre si, ou um RMS de várias camadas, de forma a achar uma solução para o problema de alocação de recursos em grade.

De acordo com o modelo proposto na taxonomia utilizada por Krauter *et* al. [17], um RMS deve conter várias unidades funcionais e quatro interfaces: consumo de recursos, para gerenciar seu consumo; provimento de recursos para gerenciar sua oferta; suporte a administração de recursos, para gerenciar as operações de administração e segurança dos recursos; e administração de domínio (*peer*) para fornecer interconexão com outros RMS.

Percebe-se que se faz necessário o uso de protocolos de descoberta e disseminação de informações sobre seus recursos a outros RMS, pois esta é uma maneira de estimar o estado dos recursos que são administrados pelo próprio RMS, como também interoperado pelos demais RMS. Outras tarefas de responsabilidade do RMS seriam examinar os pedidos de recursos que lhe foram enviados por meio de alguma linguagem de especificação de recursos, e traduzir estes pedidos em protocolos internos de administração de recursos para satisfazê-los.

Um RMS pode ser classificado de acordo com vários critérios como os que seguem: organização de máquinas no grid, modelo dos recursos, disseminação de protocolos, organização na hierarquia de nomes[13], organização do repositório de dados, suporte à qualidade de serviços, organização do escalonador, política de escalonamento, estimação do estado e reescalonamento de tarefas.

A organização de máquinas no grid descreve como as máquinas envolvidas na administração dos recursos realizam as decisões de escalonamento, sua estrutura de comunicação e quais são os diferentes papéis que cada um tem no processo de escalonamento. Estas podem ser classificadas como horizontais, quando a comunicação entre elas é feita diretamente, hierarquizadas, quando a comunicação é realizada diretamente com um líder, ou por células, onde um misto de ambas anteriores ocorre, pois dentro da célula a comunicação é horizontal, e entre células, hierarquizada.

O modelo dos recursos determina como estes e as aplicações nele executadas são descritos pelo RMS no grid. Podem ser descritas por esquemas[14], onde estes são descritos através de uma linguagem com restrições de integridade, ou por objetos, onde as operações sobre os recursos são definidas como parte do modelo de recursos.

[13] namespace organization
[14] schema based approach

A organização da hierarquia de nomes pode ser feita respeitando-se a lógica relacional, onde os recursos são organizados por relacionamentos, algo tipicamente utilizado em bases de dados relacionais. Outra forma de organização é por hierarquia, onde o exemplo típico é a estrutura física ou lógica de uma topologia de rede. Além dessas, é possível adotar a organização por grafos, onde a organização é dada por nós e arestas ou ponteiros entre nós.

O suporte à qualidade de serviço pode ser implementado através de políticas como SLA[15], reserva de recurso e controle de admissão. A existência ou não destas políticas e a maneira como estas foram incorporadas ao RMS, nos permite classificá-los como sem QoS, QoS de fácil implementação (*soft*), e QoS de difícil implementação (*hard*).

Por fim, vejamos os critérios de classificação restantes. Um destes é a organização do repositório de dados que podem ser organizados por diretórios em rede ou por objetos distribuídos nos diversos repositórios de dados. Quanto à descoberta e disseminação de informação de recursos, os RMS podem realizar a descoberta com o auxilio de agentes ou diretamente por meio de queries em uma base de dados centralizada ou distribuída. Já a disseminação pode ser feita periodicamente ou então sob demanda.

Com relação ao escalonamento propriamente dito, os RMS podem se distinguir, pela sua organização, estimação de estado, política de escalonamento e reescalonamento. Com relação a sua organização, podemos classificar os escalonadores como centralizados, hierarquizados ou descentralizados.

Os RMS centralizados possuem um único controlador que efetivamente escalona, tomando para si todas as responsabilidades de decisões, tornando simples a tarefa de administração do escalonamento assim como facilitando a tarefa de co-alocação de recursos. Num esquema de organização hierarquizado, existe uma hierarquia entre os escalonadores mas esta não provê autonomia de sítio ou escalonamento com mais de uma política, também conhecida como multi-escalonamento. Porém, torna mais fácil a implementação de escalabilidade e tolerância a falhas, e é recomendado a grids que possuem políticas dinâmicas de uso de recursos.

O modo descentralizado é escalável para grandes redes, além de tornar possível a autonomia de sítio e a implementação de várias políticas de escalonamento simultâneas, sendo uma tarefa não trivial a administração e co-alocação de recursos, pois é necessário

[15]service level agreement

implementar um protocolo que faça os escalonadores coordenarem-se entre si na descoberta e troca de recursos.

Sobre a estimação de estado, esta é sempre realizada de modo parcial ou não confiável, quando não invalidada, dada as características de latência sobre a rede. Dado que não é possível ter absoluta certeza do estado atual, utilizam-se técnicas preditivas ou não preditivas para se estimar o estado corrente.

A estimação de estado não preditiva utiliza apenas as informações do job e do recurso corrente, sem analisar seu histórico. Sob as características do recurso ou job podem ser realizadas heurísticas ou aplicada uma função de distribuição de probabilidade que estime seu estado. Ao levarmos em consideração o histórico do recurso, utilizamos métricas preditivas de estimação de recursos, as quais podem utilizar técnicas de aprendizado de máquinas, modelos de preços ou outras heurísticas.

O reescalonamento é a capacidade que tem o escalonador de reordenar a execução de jobs, característica que pode estar presente ou não no escalonador. Caso esteja, o reescalonamento pode ser realizado de maneira periódica pelo escalonador ou então por orientação de eventos que sinalizem ao escalonador a necessidade de reordenar a fila de execução. Em geral, os sistemas de grid não são explícitos quanto à sua política de reescalonamento, o que torna difícil classificá-los.

A classificação da política de escalonamento foi produzida em relação à capacidade que o RMS possui de alterar suas políticas pelo comportamento de entidades externas a este. Este pode ser classificado como fixo, onde a política de escalonamento é pré-determinada, ainda que seja possível posteriormente um ajuste fino ou extensível, na qual as entidades externas ao RMS podem alterar a política de escalonamento do próprio RMS. Na estratégia fixa, a política pré-determinada tem como objetivo a maximização de alguma métrica relacionada ao sistema ou à aplicação a ser executada, abrindo assim a possibilidade desta estratégia fixa ser orientada ao sistema ou à aplicação. Caso seja adotada a estratégia extensível, os agentes externos podem realizar mudanças na política de escalonamento de maneira estruturada, alterando valores e significados internos à computação do escalonador, ou de modo *ad-hoc*, onde um agente externo muda uma política pré-determinada por uma de sua preferência.

2.3 Grade Computacional e Escalonamento

O problema real e específico que permeia o conceito de grid é a coordenação do compartilhamento de recursos e a solução de problemas em organizações virtuais dinâmicas e multi-institucionais. Particularmente, no escalonamento de tarefas estes problemas se fazem presentes. Podemos, de certa forma, generalizar o processo de escalonamento em um grid em três estágios: descoberta de recursos e filtro, seleção de recursos e escalonamento de acordo com os objetivos propostos, e submissão de tarefas. A fonte principal de informações sobre escalonamento em grid no qual este trabalho se baseia está em [10].

Um escalonador em nível de grid ou meta-escalonador não é um componente indispensável em uma infraestrutura de grid. Um exemplo claro disto é a ausência de um meta-escalonador no Globus, tido como um padrão *de facto* de software para grid. Porém, não há dúvida de que tal componente seja crucial para utilização plena do potencial dos grids em termos de sua expansão, incorporando recursos desde supercomputadores a estações de trabalho.

Um requisito de um escalonador de grid é que este seja adaptável às diferentes políticas locais dos nós. Do ponto de vista do escalonamento de tarefas, a variação no desempenho talvez seja a característica mais marcante dos sistemas de grid em comparação com os sistemas tradicionais. Um outro problema, também trazido à tona em sistemas de grid, é uma eventual separação entre nós de computação e de dados, pois a vantagem adquirida pela seleção de um recurso computacional com custo baixo pode ser neutralizado pelo alto custo de acesso à localização e busca dos dados.

Antes de seguirmos em prosseguimento na discussão sobre escalonamento, é necessário fazer aqui uma distinção entre escalonamento global e local em um grid. O escalonamento local determina como o processo residente em uma única CPU é alocado e executado. Já uma política de escalonamento global usa informações do sistema para alocar processos para vários processadores, com o objetivo de otimizar seu desempenho como um todo, o que é o caso do escalonamento em grid.

Uma outra possível classificação entre as formas de escalonamento é a forma dinâmica e estática. No escalonamento estático, cada tarefa é associada uma única vez a um recurso, tornando necessário uma estimativa do custo de computação antes de sua execução. No escalonamento dinâmico, existem dois componentes: estimador de estados e um realizador de decisões. Tanto o escalonamento global quanto o dinâmico são amplamente adotados

em grids.

Estes últimos buscam o balanceamento de carga dinâmico, adotando uma de quatro possíveis políticas: fila sem restrições, técnicas de balanceamento de carga restritas, técnicas de custo de comunicação restritas, e híbridos entre técnicas dinâmicas e estáticas. Há uma enorme dificuldade de se encontrar soluções ótimas para o problema do escalonamento em grid, seja pela natureza do problema dos algoritmos de escalonamento ser NP-completo, seja pelas restrições de informações confiáveis existentes, e pela dificuldade no ambiente de grid de assumir premissas que provem a optimalidade de um algoritmo.

Por isso, as pesquisas nesta área procuram achar soluções sub-ótimas que podem ser divididas em aproximativas ou heurísticas. Algoritmos aproximativos usam modelos formais de computação que ao invés de buscarem uma solução ótima, ficam satisfeitos com uma solução suficientemente boa[16]. Já os algoritmos heurísticos representam a classe de algoritmos que assumem as hipóteses mais realistas possíveis sobre o conhecimento existente e as características de carga do sistema.

Uma das estratégias possíveis para o escalonamento dinâmico é a centralização das decisões. Esta possui a vantagem de ser de fácil implementação, mas tem dificuldades em oferecer escalabilidade, tolerância a falhas, além de possibilitar a formação de gargalos de desempenho. Outra possível estratégia é utilizar sistemas não-cooperativos, onde os escalonadores individuais agem como se fossem entidades autônomas, e chegam a decisões procurando seu próprio ótimo, sem se preocupar com os efeitos que estas possam ter sobre o resto do sistema. Em sistemas cooperativos, cada escalonador tem a responsabilidade de carregar a sua porção da tarefa de escalonamento, mas todos estão trabalhando para um mesmo objetivo global.

Podemos definir dois participantes nos sistemas de escalonamento em grid: os consumidores de recursos, que submetem as suas aplicações, e os provedores de recursos, que compartilham seus recursos quando se juntam ao grid. Estas entidades possuem objetivos distintos a serem satisfeitos, os quais poderiam ser descritos na função objetivo do escalonamento. As funções objetivo podem ser classificadas como centradas na aplicação, ou centrada nos recursos. Os algoritmos centrados na aplicação buscam otimizar o desempenho de cada aplicação individual. Já os objetivos dos algoritmos que são centrados nos recursos estão relacionados à utilização e produtividade[17]. Uma métrica muito utilizada

[16] seja esta um máximo local ou uma solução acima de um limitante

[17] throughput

entre escalonadores é o TPCC[18], sendo este o número total de instruções que um grid pode computar, no intervalo compreendido entre o começo de execução do escalonamento e o seu final.

Existe no escalonamento em grid uma demanda por adaptações, cuja fonte pode ser dita vinda de três pontos: heterogeneidade dos recursos, o dinamismo do desempenho dos recursos e a diversidade das aplicações. Para cada um deles, há um correspondente algoritmo de escalonamento adaptativo. A adaptação ao desempenho dinâmico dos recursos é realizada principalmente nas mudanças de políticas de escalonamento ou reescalonamento, na distribuição da carga de trabalho de acordo com modelos específicos de desempenho da aplicação, e ao achar um número apropriado de recursos a serem utilizados. Já a adaptação a uma aplicação específica exige forte integração do escalonador com a aplicação. No escalonamento de tarefas independentes, uma estratégia comum é associá-las de acordo com a carga nos recursos, de maneira a alcançar uma alta produtividade[19] do sistema.

Temos aqui alguns exemplos de algoritmos estáticos com estimação de desempenho para tarefas independentes. MET[20] associa cada tarefa ao recurso que possui o melhor tempo de execução esperado para a tarefa, ou seja, o menor MET, não importando o recurso está disponível ou não. MCT[21] associa cada tarefa ao recurso com o mínimo tempo de completude esperado do recurso para a tarefa. É bom salientar que o tempo de completude inclui, além do tempo de execução, o total de tempo a ser esperado para que a execução inicie.

O Min-min também se baseia no mínimo tempo de completude da tarefa. Sua diferença para o algoritmo MCT é que o Min-min considera todas as tarefas não alocadas no momento da decisão, ou seja, toma o mínimo entre as tarefas ainda não alocadas. Já o MCT toma suas decisões de escalonamento baseado em cada tarefa individualmente sem levar em consideração as demais. O Max-min é, por sua vez, uma variação sobre o Min-min. Sua diferença em relação à este é que ele escolhe o máximo, ao invés do mínimo,

[18]Total Processor Cycle Consumption

[19]throughput

[20]Minimum Execution Time. Embora o algoritmo tenha este nome, MET é ao mesmo tempo uma propriedade relativa à tarefa em associação com algum recurso computacional que, se neste executado, irá produzir o referido tempo de execução

[21]Minimum Completion Time. Da mesma forma que no caso do algoritmo MET, o MCT é também uma propriedade da tarefa em relação aos recursos computacionais, levando-se em consideração seu estado de disponibilidade, ou momento em que estará disponível para execução

dentre os mínimos tempos de completude das tarefas ainda não alocadas, para associá-la ao recurso computacional correspondente.

Um outro algoritmo é o *suffrage*, com uma possível extensão chamada *Xsuffrage*. O primeiro trata de escolher a máquina cuja diferença entre o seu mínimo tempo de completude e o segundo menor tempo de completude seja a maior possível[22]. Se ao invés de máquina tivermos essa diferença associada ao cluster, temos a implementação do Xsuffrage.

Estudos mostram que nenhum algoritmo leva vantagem em todos os cenários, o que aponta para a necessidade de adaptação dos algoritmos a recursos e aplicações heterogêneas. O agrupamento de tarefas pode ser utilizado, se estas possuem características de fina granularidade, reduzindo assim o tempo de escalonamento e lançamento das tarefas, e aumentando a utilização dos recursos.

Quando se verifica a existência de precedência entre as tarefas, um modelo popular é utilizado para representar a sequência de tarefas: os grafos acíclicos direcionados. Existem dois problemas a serem considerados na execução de fluxos de trabalho: como as tarefas dentro do fluxo são escalonadas, e como submeter as tarefas escalonadas sem violar a estrutura de precedência das tarefas. Geradores de fluxos de grid (*grid workflow generators*) tratam do primeiro problema, enquanto que os executores de fluxos de grid (*grid workflow engines*) tratam do segundo problema. Há um dilema implícito nestes sistemas entre tomar vantagem máxima do paralelismo, utilizando o maior número de recursos disponíveis, e minimizar os atrasos de comunicação, agrupando as tarefas em poucos recursos.

Existem três tipos de heurísticas propostas para tentar resolver este tipo de problema. O escalonamento por listagem é uma classe de heurística na qual as tarefas são associadas a prioridades e colocadas em uma lista ordenada. As diferenças entre heurísticas deste tipo se dão na forma como é elencada a lista e quando a tarefa é considerada pronta para ser associada a um recurso. Um exemplo deste tipo de heurística é o HEFT[23], onde são selecionadas tarefas com um mínimo custo conjunto de computação e comunicação. Outro exemplo é o FCP[24], que mantém um número limitado de tarefas ordenadas em um determinado instante, reduzindo assim sua complexidade de tempo.

Uma outra técnica possível é a duplicação, que utiliza o tempo de recursos livres para

[22]A idéia é que esta diferença de tempo meça o "sofrimento" da tarefa de não poder ser alocada ao recurso ao qual está associado seu mínimo tempo de completude

[23]Heterogeneous Earliest Finish Time

[24]Fast Critical Path

duplicar tarefas predecessoras. Ainda existe a heurística de agrupamento (*clustering*), que particiona o grafo em clusters para realizar a associação de clusters a recursos. O agrupamento pode ser não linear ou linear, se duas tarefas independentes pertencerem ao mesmo agrupamento ou não, respectivamente. Para a junção posterior dos grupos, são possíveis estratégias para o balanceamento de carga, a minimização do tráfego de comunicação e a escolha aleatória.

Ao se levar em consideração a localização dos dados durante o escalonamento, é necessário ter em mente que associar uma tarefa a uma máquina que lhe dê o melhor tempo de execução, pode comprometer seu desempenho se houver alto custo para obtenção dos dados de entrada. Sendo assim, duas situações podem ocorrer: pode-se ou não permitir a replicação de dados. As estratégias de replicação de dados ocorrem em nível de sistema ao invés de em nível de aplicação, procurando sempre reduzir o consumo de banda nas transferências e aumento excessivo de carga de dados em determinados pontos. Quando existe interação entre os escalonamentos de computação e dados, duas maneiras de lidar com o problema surgem: separar o escalonamento de computação e de dados ou conduzir um escalonamento combinatório.

Uma maneira possível de enfrentar o problema do escalonamento é tomar o modelo de um grid como um conjunto de agentes econômicos, o que faz surgir novas funções objetivo a serem otimizadas. A dificuldade em otimizar o custo e o lucro dos usuários do grid e seus recursos reside no fato de que as unidades de custo e tempo são diferentes e seus objetivos são quase sempre conflitantes, ou seja, quão melhor o desempenho de um recurso, maior será o seu custo.

Outro tipo de modelagem possível está ligado à métodos de escalonamento inspirados em leis da natureza, tais como algoritmos genéticos (GA), *simulated annealing* (SA) e busca tabu (TS) ou uma combinação entre estes. Também é possível adotar parâmetros de qualidade de serviço (QoS), impondo condições para que as aplicações executem com sucesso, o que vem a se tornar uma restrição a ser considerada nos escalonadores. Alguns escalonadores atuais de grid adotam outras técnicas para solução de problemas dinâmicos tais como escalonamento baseado em informações recentes (*just-in-time*), predição de desempenho, e reescalonamento dinâmico durante a execução.

Antes de finalizar esta seção a respeito de escalonamento em grids, cabe uma palavra a respeito de ferramentas que auxiliam na tarefa de descobrir que heurística é mais apropriada para cada tipo de aplicação. Um bom exemplo de ferramenta nesta área é o portal

de escalonamento *EasyGrid*. Esta pode ser vista em mais detalhes em Boeres *et al.* [1].

A metodologia *EasyGrid* tem seu foco em fornecer aos programadores e usuários a tarefa de tornar a execução das aplicações eficiente em ambientes de grid. Esta metodologia é baseada em um middleware centrado na aplicação para prover serviços ajustados especificamente para as necessidades de cada aplicação individual. A idéia é que aspectos relacionados ao grid permaneçam transparentes ao programador, evitando assim que este desenvolva mais de uma versão de sua aplicação para executar localmente ou em plataformas remotas.

Isto é alcançado transformando automaticamente as aplicações paralelas em aplicações com conhecimento do sistema[25], pela incorporação de um middleware específico das aplicações nestas na forma de um sistema de gerenciamento das aplicações chamado AMS[26]. Estas aplicações com conhecimento de sistema são robustas a falhas, adaptativas e auto-escalonáveis, além de capazes de reagir às mudanças ocorridas em ambientes dinâmicos, distribuídos, instáveis e compartilhados.

O portal de escalonamento EasyGrid permite a análise de *middlewares* de escalonamento de tarefas para aplicações com conhecimento de sistema. Estes incluem algoritmos de escalonamento estáticos para uma alocação inicial no grid e algoritmos de escalonamento dinâmicos para adaptação em tempo de execução. Os objetivos principais da ferramenta são: facilitar a investigação de técnicas de escalonamento, avaliar os algoritmos em termos de alguns critérios pré-definidos e validar o escalonamento gerado pelas heurísticas propostas, seja em ambiente simulado ou real.

Numa fase estática ou inicial, informações relativas a aplicação são produzidas por um escalonador estático, as quais posteriormente serão utilizadas em conjunto com dados de execução que influenciam as decisões de um escalonador dinâmico. De maneira a analisar o desempenho das estratégias de escalonamento híbrido nas diversas arquiteturas de grid, o portal *EasyGrid* utiliza-se da ferramenta *SimGrid* como um simulador de execuções. O *SimGrid* é possivelmente o simulador mais famoso e utilizado para estudos de escalonamento em ambiente de grid. Maiores informações sobre esta ferramenta pode ser encontrada em [4].

Ainda que o portal antes de mais nada seja uma ferramenta para construção e avaliação de estratégias estáticas e híbridas de escalonamento em sistemas distribuídos, este também

[25]no contexto da ferramenta *EasyGrid* conhecimento de sistema significa também transparência em relação à arquitetura das máquinas do grid

[26]de *Applicaction Management System*

pode ser utilizado como um facilitador de acesso dos recursos de grid aos usuários. Ela também é capaz de escalonar aplicações MPI em recursos computacionais baseados em Globus. A arquitetura de execução tanto pode ser definida pelo usuário com valores artificiais lidos de um arquivo (no caso de a execução ser realizada em ambiente simulado) tanto quanto clusters ou grids reais. Uma interface visual e amigável oferece ao usuário a oportunidade de escolha de alguns parâmetros de desenvolvimento e análise sobre as estratégias de escalonamento escolhidas. Esta também fornece ao usuário os resultados do escalonamento para avaliação gráfica, na forma visual de um diagrama de Gantt.

Concluindo esta parte, alguns dos principais desafios na área de escalonamento em grid podem ser elencados aqui: heterogeneidade do ambiente computacional, dinamismo do estado, separação existente entre recursos de dados e de computação. Muitas das soluções de escalonamento em grid foram inspiradas em algoritmos tradicionais, e os potenciais destes algoritmos dentro do ambiente de grid ainda não foram completamente explorados, permanecendo um desafio sua melhor utilização. Muitos destes algoritmos em geral consideram apenas retratos estáticos de valores previstos, sendo um desafio também, utilizar predição dinâmica de desempenho.

Existem áreas pouco estudadas neste contexto de escalonamento em grid. Um ponto a ser melhor estudado é o reescalonamento de grafos acíclicos. Outra área também pouco estudada é como os requisitos de QoS afetam a alocação de recursos e o desempenho das outras partes da aplicação. Existe também pouca pesquisa na combinação simultânea de otimização de computação e transferência de dados. O estudo de modelos novos também reserva muito esforço a ser realizado na área de pesquisa, assim como novos algoritmos que se utilizem de particularidades da infraestrutura de grid existente.

Capítulo 3

Heurísticas

Nesta seção, faremos uma breve descrição de algumas heurísticas presentes na literatura de meta-escalonamento em ambientes de grid. Embora o capítulo anterior já tenha introduzido algumas heurísticas relevantes em 2.3, acreditamos que o detalhamento de algumas heurísticas aqui citadas será enriquecedor, além de que o detalhamento de duas dessas heurísticas, notadamente 3.3 e 3.4 se fará necessário para a compreensão do trabalho realizado. Faremos também, antes, uma breve explanação de um tipo de aplicação costumeiramente executado em grids, assim como características do ambiente.

3.1 Aplicações de Troca de Parâmetros

Uma classe de aplicação bastante frequente neste tipo de ambiente estudado são as aplicações de troca de parâmetros[1]. Podemos definí-las como um conjunto de tarefas sequenciais e independentes onde não existe comunicação entre estas, nem dependência de dados ou precedência de tarefas. Este tipo de aplicação é estruturado de maneira que cada execução equivale a um experimento, dentro de um conjunto destes, realizado com seus próprios parâmetros. Maiores detalhes sobre escalonamento deste tipo de aplicação pode ser encontrado em [6].

Neste cenário, podemos imaginar, sem perda de generalidade, um grid como sendo um conjunto de k clusters, acessíveis ao usuário, por meio de k diferentes caminhos. Arquivos de entrada estariam localizados na máquina do usuário, da mesma forma que os arquivos de saída para lá deveriam retornar, após finalizada a execução. Podemos desconsiderar a existência de migração de tarefas entre os clusters executores, assim como qualquer possível troca de arquivos entre estes. Da mesma forma, o acesso de rede é feito de maneira

[1]Parameter Sweep Applications - PSA

paralela a cada cluster, sem interferência entre estes. O escalonador tem conhecimento da localização e número de cópias de todos os arquivos de entrada.

Recapitulando, neste tipo de ambiente existem quatro heurísticas bastante populares. São elas o Min-min, Max-min, Sufferage e Xsufferage. Estes foram descritos em 2.3 e também podem ser encontrados em [6]. Considerando-se experimentos onde haja tráfego de grandes arquivos de entrada, os resultados de simulações demonstraram que a heurística Xsufferage se sobressai entre as demais, chegando em alguns casos a diminuir em 50 % o tempo total de execução. Isto ocorre pois este tipo de heurística captura melhor as dependências entre arquivos de entrada e tarefas, tornando mais frequente seu reuso e consequentemente diminuindo os atrasos na execução, devido ao tráfego destes arquivos.

3.2 On-line baseado em custo

O algoritmo de escalonamento on-line baseado em custo associa um job a uma máquina, onde o seu recurso a ser consumido tem o custo marginal mínimo. O conceito de custo pode ser adotado para o problema do escalonamento baseado em princípios econômicos. O método chave é converter o uso total de tipos diferentes de recursos, tal como CPU, memória e banda em um custo homogêneo. Maiores detalhes sobre este algoritmo podem ser encontrados em [7].

Para medir o desempenho deste algoritmo, foi adotado como métrica de desempenho o *makespan*, que é o tempo total de escalonamento, ou a diferença de tempo entre o início da execução do primeiro job e a finalização do último job. O objetivo deste algoritmo é minimizar o máximo da carga nas máquinas e na rede dentro do grid, para que a minimização total do *makespan* possa ser alcançada.

Foi adotado o paradigma de chegada de jobs, onde o atributo de um job é conhecido quando este será executado. Este também é conhecido como *jobs arrive over time*. Cada job é definido pelos seguintes parâmetros: número de ciclos de CPU necessários para o job e a quantidade de dados necessária para comunicação.

Este algoritmo foi comparado com outros três para fins de verificação de seu desempenho: o algoritmo guloso, o algoritmo Round-robin e o aleatório. No algoritmo guloso, para cada job adicionado a fila, o meta-escalonador imediatamente o associa a uma máquina que possui a mínima soma das cargas de rede e da máquina e o tempo de início de execução do job na máquina. No Round-robin, o job que chega na fila é associado a

cada domínio de recurso em uma sequência predefinida deste último. No caso do algoritmo aleatório, os jobs são associados a máquinas selecionadas aleatoriamente no grid computacional.

Os resultados obtidos deste algoritmo revelam que este conseguiu realizar melhorias em relação aos outros algoritmos contra os quais ele foi comparado. Quando o intervalo de chegada de jobs varia entre 2 e 10 segundos com tamanho dos dados de entrada fixos em 60 MB, o algoritmo baseado em custo consegue ter o menor *makespan* que todos os outros três. Quando o intervalo de chegada de jobs é fixado em 6 segundos e o tamanho dos dados de entrada varia, o algoritmo baseado em custo ainda possui o menor makespan que todos os outros três.

3.3 Algoritmo de Galstyan

Os sistemas multi-agentes (MAS[2]) e comunidades sobre inteligência artificial distribuída mostraram que grupos de agentes autônomos de aprendizado podem resolver com sucesso diferentes problemas de balanceamento de carga e alocação de recursos. Segundo Galstyan *et* al. [14], a modelagem em sistemas multi-agentes é apropriada para a descrição de um grid, porque a natureza autônoma e distribuída de seus agentes, que podem ser usuários ou recursos, reflete a natureza federativa deste. Devido a políticas de grid descentralizadas, as partes de um grid podem utilizar estratégias de alocação diferentes, tornando um gerenciador de alocação centralizado não factível ou desejado. Entender os efeitos dos vários mecanismos de alocação de recursos no comportamento global do sistema irá influenciar as decisões arquiteturais assim como as políticas escolhidas dentro de uma VO federativa.

Galstyan *et* al. apresentam um escalonador que examina um caso específico, quando as decisões de alocação são feitas pelos usuários individualmente e são baseadas em eficiência computacional medida pela técnica de aprendizado por reforço (RL[3]). Seu foco é o estudo do desempenho global de uma VO que implementa este mecanismo. Procura-se vencer um desafio em particular de alocação de recursos em grid: a falta de precisão nas informações de estado de um recurso em escala global. Por conta disso, os mecanismos de alocação não devem depender fortemente na disponibilidade do conhecimento global

[2]multi agent systems
[3]Reinforcement Learning

corrente dos recursos. Caso seja necessária ou de interesse do leitor se aprofundar no tema Aprendizado por Reforço, sugerimos aqui a leitura de [16], [23] ou em [9], que tratam do tema em mais detalhes.

Aprendizado por reforço é uma poderosa idéia na qual um agente, como um usuário de grid, aprende ações ótimas por meio da exploração de tentativa e erro no ambiente e pelo recebimento de recompensas pelas suas ações. Neste sistema, um grande número de usuários submete jobs para algum dos recursos que são escalonados por meio de um escalonador local de acordo com suas políticas locais. Os usuários são modelados como agentes racionais, egoístas que tentam maximizar as suas utilidades, ou seja, minimizar o tempo de execução de suas tarefas. A análise do comportamento global do sistema foi feito por meio de uma simulação numérica, e posteriormente foi realizada uma comparação com um algoritmo base que faz uso de conhecimento global das cargas correntes dos recursos.

Neste modelo, foi negligenciada a necessidade de co-alocação[4], e é assumido que as tarefas geradas pelos usuários necessitam de apenas de um certo período de CPU, de tal forma que elas são caracterizadas pela sua duração. No modelo proposto, foi considerada uma representação simplificada dos recursos e dos escalonadores locais. Cada recurso foi caracterizado com seu poder de processamento P, que é definido pelo tempo de CPU necessário para completar um job com uma unidade de medida. Da mesma forma, só existe um único job em execução no sistema em um determinado instante. Por simplicidade, assumimos que todos os escalonadores locais utilizam uma fila que prioriza os jobs pela sua ordem de chegada. Não foi implementado também a possibilidade de reescalonamento da tarefa.

Os agentes estão interessados em minimizar o tempo de espera das tarefas que eles submeteram, e sendo assim, preferirão recursos com menor tamanho de fila possível. Outra métrica centrada no usuário seria o tempo de resposta, que é o intervalo entre a geração do job e sua finalização, ou então métricas baseadas na acurácia da predição do período de finalização. Foram utilizadas a ponderação de duas métricas: tempo de fila e de execução, onde as ponderações entre estas métricas varia de agente para agente. Essa ponderação impõe um limite mínimo de participação, associado ao tempo de fila nunca menor que 0,2. Os vários agentes presentes nos experimentos realizados, de modo a simular um ambiente heterogêneo destes, tiveram este fator de ponderação sorteado aleatoriamente, porém

[4]entenda-se co-alocação como a alocação conjunta de um conjunto de recursos que existem de maneira independente entre si

seguindo a regra anteriormente descrita.

Dentre as várias maneiras de se utilizar aprendizado por reforço, foi escolhido o aprendizado Q[5]. Esta técnica busca dar as recompensas (ou punições) aos agentes de acordo com informações adquiridas do ambiente considerando-se um comportamento esperado para este. Maiores detalhes sobre *Q-learning* pode ser encontrado em [27], além de [16] e [23]. Para comparação e análise do algoritmo, foram utilizados outros dois algoritmos: seleção aleatória e recurso com menor carga.

Sendo assim, para taxas de chegada ou geração de jobs pequenas, o algoritmo aleatório tem o melhor desempenho de todos. Porém a situação se altera quando aumenta a taxa de chegada ou geração de jobs, e neste caso o algoritmo aleatório tem piora significativa, enquanto os demais algoritmos conseguem manter seu desempenho. Percebe-se pelas simulações que o algoritmo de Galstyan permite aos agentes uma distribuição mais eficiente das tarefas pelos recursos do que a seleção aleatória, em termos de balanceamento de carga. Após um período pequeno de aprendizado, o tempo médio de espera para o algoritmo de Galstyan cai abaixo do nível do algoritmo de seleção do recurso com menor carga. Perceba que não existe nenhuma comunicação entre os agentes que os façam tomar decisões de distribuir a carga pelos recursos.

Outro ponto analisado foi como o sistema se comporta com a troca de antigos usuários por novos. É verificado que, se a taxa de troca de usuários novos por velhos é pequena, não há impacto significativo sobre o desempenho global do sistema ou dos usuários antigos. Isto é importante pois usuários novos necessitarão de um período de treinamento para perceberem o ambiente de execução a eles oferecidos. Isto, porém, muda quando um grande número de novos usuários chega ao sistema trazendo impacto negativo tanto sobre o desempenho global do sistema quanto sobre jobs a serem escalonados de usuários antigos. Outro ponto interessante a ser mencionado é que o período de treinamento aumenta se existe demora na obtenção da recompensa no algoritmo de Galstyan.

3.4 Multiple Queues with Duplication

A avaliação do algoritmo Filas Múltiplas com Duplicação (MQD[6]) foi realizada com aplicações *bag-of-tasks* no simulador SimGrid. Este escalonador está descrito em mais

[5]Q-learning
[6]Multiple Queue with Duplication

detalhes em [18]. Fazem parte do modelo da simulação realizada duas hipóteses: cada domínio administrativo tem seus próprios usuários locais que utilizam seus recursos e disponibilidade e a capacidade de processamento de cada nó flutua através do tempo. Assume-se também que os tempos de execução das tarefas são conhecidos, além de serem intensivos em uso de CPU e com tempo de transferência de dados desprezível. Considera-se o objetivo do escalonador minimizar o tempo total de escalonamento[7].

O algoritmo de escalonamento em si pode ser dividido em duas fases: inicial (fase de avaliação) e uma corrente, no qual os princípios do algoritmo são aplicados. Na fase inicial ou de avaliação, são escolhidos os jobs de menor tamanho possível para serem escalonados e estes serão alocados até que a totalidade dos recursos tenha sido alocada para este grupo de jobs, fato que caracteriza a finalização desta fase. Este grupo inicial de jobs menores serve como parâmetro para classificação dos recursos[8] que fazem parte do pool de execução, segundo seu poder de computação. Sendo assim, após esta fase, saberemos quem são os recursos de melhor e pior desempenho para o conjunto de jobs.

A seguir, ordenamos e agrupamos os jobs em grupos que serão associados a cada um dos recursos computacionais, na ordem direta entre tempo previsto de execução para cada grupo de jobs e o poder computacional aferido na fase inicial do algoritmo. Em outras palavras, o grupo de jobs mais longos é associado ao recurso computacional de melhor desempenho, e assim por diante até que o grupo de jobs mais curtos restantes seja associado ao recurso computacional de pior desempenho. O esquema utilizado espera reunir tanto balanceamento de carga quanto minimização de tempo total dentro da heurística. A duplicação proposta na heurística tem papel importante na redução do tempo total de escalonamento, principalmente na finalização de execuções de tarefas quando há mais recursos disponíveis do que tarefas.

A figura 3.1 mostra um exemplo de alocação utilizando o algoritmo MQD. Nas caixas que identificam as tarefas numeradas de 1 a 9 estão também o tempo de execução da tarefa. Na fase inicial[9], os jobs de tamanho mais curto são enviados aos recursos computacionais. Posteriormente, os recursos Q1, Q2 e Q3 são ordenados e a eles são associados os jobs de maior ou menor exigência, conforme a classificação do primeiro. Perceba que os recursos computacionais podem ser ordenados de maneira crescente em produtividade como Q2 < Q3 < Q1.

[7] também conhecido como makespan
[8] hosts de execução
[9] na figura, aparece com o rótulo initQ

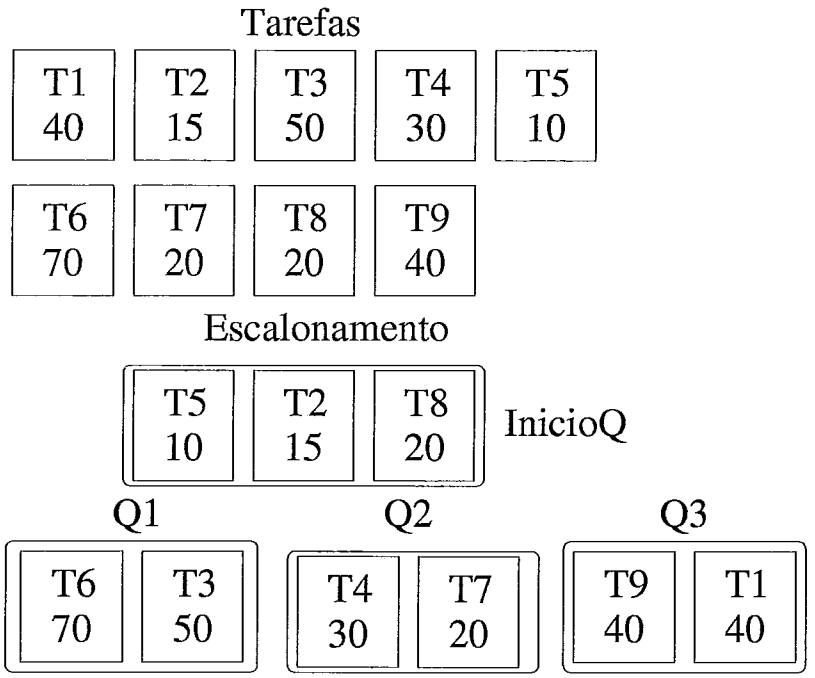


Figura 3.1: Exemplo de alocação utilizando o algoritmo MQD

Os resultados foram retirados de simulação realizada no ambiente SimGrid. Nele, o algoritmo MQD foi comparado com outras quatro heurísticas a seguir: Max-min, Min-min, Sufferage e Round-robin. A métrica de comparação é o próprio *makespan*. O tempo de computação das tarefas, o número de tarefas, número de sítios disponíveis para computação e o número de máquina em cada sítio são todas grandezas distribuídas uniformemente dentro de intervalo considerado. Os experimentos mostram que o algoritmo MQD consegue uma economia de 3 a 10% em relação ao makespan de Round-Robin. Quando há uma taxa de erro de 30 % na previsão de desempenho dos algoritmos como Max-Min, Min-Min ou suffrage que possuem dependência em relação a previsão de desempenho, o MQD consegue atingir um uma melhora de até 20% se comparado com estes.

3.5 Qsuffrage

Apresentaremos agora o algoritmo Qsuffrage para escalonamento de aplicações *Bag-of-Tasks*, onde a localização dos dados de entrada é utilizada como parâmetro para o escalonamento. Para avaliá-lo, teremos como parâmetro as métricas *makespan* ou tempo total de escalonamento e a taxa de resposta. Nele, a taxa de resposta é definida como sendo

a razão entre o tempo de duração de execução da tarefa e a diferença entre os tempos de submissão e término da tarefa. Este se encontra descrito em mais detalhes em [28].

Existem duas classes de heurísticas de escalonamento: as que operam em modo on-line e as que operam em lotes[10]. No modo on-line, a tarefa é associada ao nó de computação assim que este se torna disponível ao escalonador. No caso do escalonamento em lote, ela é mapeada ao seu nó de computação assim que ela chega ao escalonador. A heurística proposta pelo Qsuffrage possui estratégia de escalonamento em lotes, e é bastante próxima de outras heurísticas relacionadas a grid como Max-min, Min-min, Sufferage e Xsufferage.

O objetivo da heurística Qsuffrage não é apenas minimizar o tempo total de escalonamento ou *makespan*, mas também minimizar o tempo médio de espera para execução de uma tarefa individual. O algoritmo Qsuffrage pode ser descrito em três passos. O primeiro passo é calcular o tempo esperado de completude de cada tarefa em cada nó da grade de computação. Este cálculo é feito com base na quantidade de dados de entrada que o sítio possui. O segundo passo é calcular o valor *suffrage*[11] de cada tarefa, tendo como base os valores calculados no primeiro passo. O último passo é o de determinar a tarefa com o máximo valor *suffrage* e associá-la ao seu nó de computação correspondente.

Para os experimentos realizados com este algoritmo, foram geradas 1000 tarefas que chegam ao meta-escalonador de acordo com uma distribuição de poisson, tendo como média entre as chegadas 6 segundos. A distribuição de tempo entre as tarefas submetidas é uniforme dentro do intervalo de 100 a 300 segundos para uma velocidade média de CPU dentro do grid. O intervalo entre os eventos de mapeamento é de 500 segundos. Foram feitas 500 simulações variando o tamanho dos arquivos de entrada, comparando-se o desempenho entre o Qsuffrage e os demais algoritmos Min-min, Max-mix, Suffrage e Xsuffrage.

Os experimento mostram que o Qsuffrage tem melhor desempenho que os demais algoritmos não só em temos de *makespan* mas também em taxa média de resposta, apresentando ganhos significativos nestas métricas. Os resultados dos experimentos também demonstram que o tamanho dos dados de entrada influencia fortemente o desempenho de todas as heurísticas. Max-min, por exemplo, tem uma piora significativa em termos de *makespan* e taxa de resposta quando cresce o tamanho dos arquivos de entrada. Já o Qsuf-

[10] modo batch

[11] este valor se refere ao mesmo valor que é calculado para o algoritmo de escalonamento Suffrage, ou seja a diferença entre o mínimo tempo de completude da tarefa (MCT) e o segundo menor tempo de completude da tarefa, tal como descrito em 2.3.

frage se revelou constantemente a melhor das heurísticas quando o tamanho dos arquivos de entrada varia.

Capítulo 4

Meta-escalonadores

Nesta seção, faremos uma breve descrição de alguns meta-escalonadores presentes na literatura, na tentativa de resolução do problema de alocação de recursos de grids a jobs. Os trabalhos aqui relatados darão um panorama do estado da arte na confecção de meta-escalonadores em ambientes de grid, ainda que muitos destes não sejam popularmente utilizados nos ambientes de grid construídos. De qualquer maneira, todos possuem relevância de contribuição ao estudo de meta-escalonadores. Em particular, para este trabalho é interessante explicitar que o meta-escalonador descrito em 4.5 será utilizado como ferramenta nos trabalhos desta dissertação.

4.1 AppLeS

É de conhecimento que, para se ter um bom desempenho de aplicações parametrizadas, seja fundamental possuir escalonamento adaptativo. Um dos primeiros escalonadores que surgiram para aplicações parametrizadas com escalonamento adaptativo é o AppLeS. Nesta seção, vamos descrever brevemente a arquitetura e implementação do AppLeS. Maiores detalhes sobre este aplicativo podem ser encontrados em [5].

Defina-se uma aplicação parametrizada como um conjunto de tarefas sequenciais independentes ou sem precedência (como descrito na seção anterior). Assumimos que a entrada de cada tarefa é um conjunto de arquivos e que um arquivo qualquer pode ser entrada de mais de uma tarefa. Tipicamente, o número de tarefas na aplicação será de algumas ordens de magnitude maior que o número de processadores disponíveis. O acesso a recursos computacionais remotos pode ser facilitado por projetos de infraestrutura de grid (ex: GASS[1]). Para alguns algoritmos de escalonamento será necessário uma estimativa do

[1]Grid Application Support Services

tempo de computação e de transferência de arquivos.

Um primeiro algoritmo de escalonamento para o AppLeS é a fila auto-escalonável. Nela, cada job é associado a algum nó tão logo este esteja disponível, na forma de um algoritmo guloso, e não se considera a hipótese de compartilhamento de grandes arquivos de entrada. Essa última hipótese leva este algoritmo a ter desempenho ruim quando é verificado a existência de compartilhamento de grandes arquivos, mesmo que para um número pequeno de tarefas. Um outro problema com este algoritmo é que ele não realiza a seleção de recursos em prol do desempenho da aplicação.

Em aplicações parametrizadas, os algoritmos de escalonamento têm como objetivo minimizar a diferença entre o início e o fim da aplicação ou *makespan*. Quão maior for a frequência de eventos de escalonamento, mais adaptativo o algoritmo se tornará.

Existem dois objetivos neste escalonador. Um é investigar os problemas de escalonamento adaptativo e desenvolvimento de aplicações paramétricas. Outro é prover os usuários uma maneira eficiente e conveniente de executar as aplicações paramétricas na maioria dos recursos de grid. Para atingir estes objetivos, a arquitetura do software deve possibilitar ao usuário habilitar e desabilitar diferentes módulos, assim como ajustar alguns parâmetros do algoritmo de escalonamento.

O software é composto de um cliente e um servidor (*daemon*). Para submeter novas tarefas, o usuário deve ter um arquivo de descrição da tarefa que conterá uma descrição de tarefa por linha. O *daemon* é composto de quatro subsistemas: controlador, escalonador, atuador e o registrador[2]. O escalonador é a parte central do *daemon*. Ele é utilizado para notificação de eventos sobre a estrutura da aplicação, estado dos recursos e das tarefas. O controlador transita informações entre o cliente e o servidor e notifica o escalonador de novas tarefas a serem executadas ou canceladas. O atuador implementa toda a interação com a infraestrutura de software do grid para acesso aos recursos computacionais, de rede e de disco. A implementação de um algoritmo de escalonamento qualquer é completamente isolada da infraestrutura de grid utilizada para construir as tarefas. O registrador fica a cargo de manter o registro de meta-dados estáticos e dinâmicos, tanto dos recursos quanto das aplicações.

Foram implementadas três APIs para cada algoritmo de escalonamento: uma para a fila padrão, outra para uma fila que permite duplicação de tarefas e roubo de recursos, e uma outra que se utiliza de diagramas de Gantt para implementar seus algoritmos. O

[2]meta-data bookkeeper

componente registrador interage com o NWS[3] para obter informações dinâmicas do grid, como cargas na CPU e latência ou largura de banda. O atuador utiliza três APIs de transporte: GASS do Globus, IBP[4], e NFS[5]. Ele pode se utilizar simultaneamente de servidores Globus ou NetSolve para tarefas, assim como IBP, GASS ou NFS para arquivos.

Foi utilizado para avaliação do sistema o software MCell, uma aplicação que utiliza técnicas de simulação de Monte Carlo em três dimensões para estudar interações bioquímicas moleculares com células vivas. A avaliação do APST[6] foi feita utilizando-se como critérios a usabilidade, a capacidade de escalonamento em melhorar o desempenho do ambiente de grid e como a utilização de várias threads[7] melhora a utilização de recursos. Várias tarefas como tráfego de arquivos e mecanismos de tolerância à falhas são implementadas transparentemente pelos serviços do Grid, o que adiciona quesitos de usabilidade ao sistema.

As duas heurísticas de escalonamento aqui descritas, fila auto-escalonável e Gantt, se comportam diferentemente quando a topologia do grid contém um grande número de sítios. E neste caso, a utilização da heurística de diagrama Gantt leva a um melhor escalonamento, pois leva em consideração os padrões de compartilhamento e localização dos dados. A utilização de caching para o reuso de previsões, até que novas previsões sejam refeitas, torna o escalonamento menos custoso também. Foi verificado também que o agrupamento de até 20 threads, na realização de chamadas simultâneas aos softwares de grid, torna o tempo de execução 48% menor que uma implementação sem suporte à *multi-threading*.

4.2 GridWay

O centro do ambiente de trabalho GridWay é seu agente de submissão pessoal que realiza passos de escalonamento e monitora a correta e eficiente execução dos jobs. Este se encontra descrito com mais detalhes em [11]. Seu ponto importante diz respeito a sua capacidade de adaptação à mudanças. Esta capacidade é alcançada pelo reescalonamento dinâmico, ou seja, uma vez alocado o job, este pode ser reescalonado dada a existência

[3]Network Weather Service
[4]Internet Backplane Protocol
[5]Network File System
[6]Apples Parameter Sweep Template
[7]ou simplesmente multi-threading

de uma circunstância de migração. Circunstância de migração se refere a uma migração iniciada pelo grid ou pela aplicação.

A execução do job é feita em três estágios, cada um com um módulo responsável correspondente, estando sempre presente na base de um por job: módulo prólogo, que prepara o sistema remoto e os arquivos de entrada; o módulo interpretador[8] que executa e retorna o seu código de saída; e o módulo epílogo, que adquire os arquivos de saída e limpa o sistema remoto.

Os agentes de submissão utilizam outros módulos para dar suporte necessário de funcionalidades auto-adaptáveis para a aplicação. O módulo de seleção de recursos avalia os requisitos e as expressões de ranqueamento quando o job for escalonado ou reescalonado. O módulo de avaliação de desempenho avalia periodicamente o perfil de desempenho da aplicação e tenta detectar desacelerações para requisitar um reescalonamento. Para a implementação corrente, tanto o seletor de recursos quanto o avaliador de desempenho são implementados como *scripts shell.* O agente de submissão também provê suporte à tolerância a falhas em ambientes propensos a sua ocorrência, como notificar os gerenciadores de jobs da ocorrência de falhas, verificar periodicamente sua execução, e tentar detectar falhas nas saídas dos módulos prólogo, epílogo e interpretador.

A ocorrência de falhas irrecuperáveis tenta ser resolvida da seguinte maneira: o módulo falho é ressubmetido até um limite de novas tentativas, as quais se não obtiverem resultado positivo farão o job ser reescalonado ou ser terminado para só voltar a executar após intervenção manual, dependendo da escolha que o usuário tiver feito. O agente de submissão possui tanto uma API[9] quanto uma aplicação em linha de comando que serve como meio de interação com o usuário.

Os resultados dos testes realizados mostram que, mesmo com as perdas decorrentes de migração de jobs, calculadas em torno de 6 % do tempo de execução, os jobs migrados conseguem finalizar antes sua execução do que os não migrados neste ambiente.

4.3 Nimrod/G

Para resolver complexidades associadas com a computação paramétrica em clusters de sistemas distribuídos, tais como acesso pervasivo ao ambiente computacional e a exe-

[8] wrapper
[9] Application Programming Interface

cução de um grande número de tarefas associadas à variação de parâmetros de execução destas, foi desenvolvido um sistema chamado Nimrod. Ele provê uma linguagem simples de modelagem paramétrica declarativa para expressar o experimento paramétrico. Nimrod foi desenvolvido para ambientes estáticos, enquanto que, para ambientes dinâmicos foi desenvolvido Nimrod/G que utiliza o sistema Globus para descoberta dinâmica de recursos e envio[10] de tarefas em grids computacionais. Aqui será discutida a arquitetura de uma versão portátil, extensível e modularizada de Nimrod/G. Esta se encontra descrita em maiores detalhes em [3].

O seus principais componentes são: estação cliente ou de usuário, máquina paramétrica[11], escalonador, despachante[12] e o executor[13]. A estação cliente ou usuária age como uma interface de usuário para controle e supervisão de um experimento a ser considerado. Também serve como um console de monitoramento, listando o estado de todos os jobs que um usuário pode ver e controlar. É possível executar várias instâncias do mesmo cliente em diferentes locais.

A máquina paramétrica é responsável pela parametrização do experimento, criação dos jobs, manutenção de seu estado, interação com os clientes, enviar avisos ao escalonador e despachar jobs. Ela mantém o estado completo do experimento e assegura que este fica guardado de maneira persistente, o que permite que o experimento seja recomeçado caso haja a queda de algum nó. O escalonador é responsável pela descoberta e seleção de recursos e sua associação a jobs. O algoritmo de seleção de recursos é responsável por selecionar recursos que cumpram os limites de prazos e minimizem os custos da computação.

O despachante inicia a execução de uma tarefa no recurso selecionado, como instrução do escalonador. Periodicamente, ele realiza a atualização do estado de execução da tarefa na máquina paramétrica. Por fim, o executor é o responsável por enviar e receber as tarefas e seus dados, assim como iniciar a execução da tarefa no recurso a ela associado, e enviar de volta os resultados para a máquina paramétrica por meio do despachante.

Segundo [3], futuramente os grids computacionais irão popularizar um modelo chamado de economia computacional. Este prevê a existência de um mercado com demanda e oferta de aluguel de serviços computacionais rentáveis, tal como hoje existem para serviços como

[10] no Globus, este possui o nome de dispatching
[11] parametric engine
[12] dispatcher
[13] job-wrapper

energia elétrica, por exemplo. Neste contexto, usuários pagariam pelo aluguel de recursos computacionais por eles demandados, à medida que estes fossem a eles necessários, como possivelmente também, migrariam de um recurso ou provedor para outro, conforme houvesse uma variação de preços vantajosa. Entenda-se este ambiente, também, como o mais genérico e realista possível do ponto de vista econômico. Em outras palavras, haveria espaço para negociação entre as partes, os preços variam conforme o tempo e podem ser diferentes para os diversos usuários, apenas para citar alguns exemplos.

A integração de economia computacional e seus conceitos relacionados (ex: negociação, preços) como parte do sistema de escalonamento influencia enormemente a maneira como os recursos computacionais são selecionados para prover os requisitos de usuário. Isto pode ser feito de duas maneiras. Sistemas como o Nimrod/G podem trabalhar em benefício do usuário e tentar completar a tarefa associada dentro de um prazo e custo estipulados. De outra maneira, seria necessário o usuário assinar um contrato onde fica previamente estabelecido por ambas as partes o custo e o prazo de execução, de forma que o usuário saiba de antemão se é possível ter o resultado e a que custo.

O escalonador utiliza vários tipos de parâmetros para achar uma política de escalonamento que complete a execução da aplicação. São exemplos destes parâmetros: arquitetura, configuração, estado e capacidade dos recursos, requisitos de velocidade de acesso, número de nós livres ou disponíveis, prioridade, tipo e tamanho de fila utilizada, largura de banda, carga e latência de rede, confiabilidade, preferência do usuário, capacidade de pagamento do usuário, prazos, custo do recurso e sua variação no tempo e histórico. Destes os mais importantes que influenciam a maneira como o escalonamento é feito é o custo do recurso, preço oferecido pelo usuário, e o prazo de execução. O escalonador pode usar todo o tipo de informação reunida pelo descobridor de recursos e ainda negociar com os proprietários destes para obter a melhor relação custo-benefício.

Como foi construído sobre o Globus Toolkit[14], ele necessita dos seguintes componentes deste: GRAM[15], MDS[16], GSI[17] e GASS[18]. O escalonador utiliza estes serviços para dar suporte ao modelo de computação econômica, baseada em mercado. A nova versão de Nimrod/G se comparada a sua versão monolítica anterior apresenta ganhos de

[14] versão 1.0
[15] Globus Resource Allocation Manager
[16] Monitoring and Discovery System
[17] Grid Security Infrastructure
[18] Globus Access to Secondary Storage

escalabilidade e na habilidade de escalonar tarefas com restrições de tempo e custo em uma grade de recursos.

4.4 MARS

Aqui será descrito o MARS[19], um framework aberto de meta-escalonamento, com uma arquitetura modular, flexível e orientada a objeto. Maiores detalhes sobre este escalonador podem ser encontrados em [2].

São objetivos do MARS implementar uma arquitetura extensível, escalonamento sob demanda, previsão sobre recursos, interoperabilidade e possuir uma interface via API. Possuir uma arquitetura extensível significa que o escalonador deve ser extensível aos novos padrões e protocolos que surgem continuamente de forma serem facilmente incorporados. Escalonamento sob demanda se refere à descoberta e associação de recursos requisitados por tarefas críticas ou prioritárias para começarem imediatamente, logicamente levando-se em consideração a latência decorrente das operações de escalonamento.

A previsão do estado dos recursos é importante pois estes nem sempre podem ser monitorados em tempo real devido as latências intrínsecas de rede ou do próprio processamento e por perda de mensagens. Por fim, a interface oferecida pelo MARS por meio de uma API suporta infra-estrutura necessária para disponibilizar autorização, submissão, monitoramento, uso e contabilização.

Existem dois níveis de interoperabilidade necessários a um escalonador de grid. Ele tanto deve negociar com outros meta-escalonadores quanto com escalonadores locais de recursos. Para realizar suas tarefas, o escalonador do MARS possui os seguintes componentes: um conjunto de filas de jobs priorizadas e de ingresso, monitor de recursos que autoriza, atualiza e prevê seus estados, e um submissor e monitor de jobs. Cada tarefa ingressante é posta em uma fila de prioridades e é criado um bloco de recursos para a tarefa (TRB[20]). O TRB consiste no seguinte conjunto de informações: identificador da tarefa, atributos, estado, lista de recursos e momento de criação[21]. A classificação das tarefas é feita tendo por base alguns parâmetros como requisitos de recursos, domínio de origem, recompensa[22] e prazos de execução.

[19] Michigan Advanced Resource Scheduler

[20] Task Resource Block

[21] timestamp

[22] recompensa, no contexto deste escalonador, embora não relatado explicitamente em [2], trata-se de uma maneira de se priorizar algumas tarefas sobre as outras. Esta recompensa não tem relação e nem é utilizada

São conhecidos três tipos de tarefas. Regulares[23] já trazem consigo a descrição de todos os requisitos necessários e permitem reserva de recursos. Sob-demanda não trazem todos os requisitos necessários previamente pois alguns terão que ser descobertos durante a execução. As tarefas de tempo real são dirigidas a eventos e necessitam de cumprimento estrito de requisitos. Com isso, foram implementadas duas filas para tarefas sob demanda, chamadas *crítica* e *tempo real*, e um conjunto de filas regulares.

O núcleo do escalonador consiste de dois agentes, cada um executando em uma *thread* diferente: o agente do algoritmo de escalonamento que trata de retirar as tarefas das filas e invocar os algoritmos de escalonamento registrados; e o agente de submissão de tarefas que é responsável por submeter a tarefa ao recurso a ela associado e atualizar seu estado na TRB. A monitoração de recursos é feita através do módulo MDS do Globus toolkit. Um agente de atualização de recursos é configurado para entrar em contato com um servidor GIIS[24] que contenha todos os nós, ou com cada um dos escalonadores locais do grid que mantém o estado local de seu recurso. A previsão do estado do recurso é feita com atenuação exponencial[25] sobre os dados coletados dos recursos.

São dois os tipos de algoritmos de escalonamento utilizados: tempo mínimo de execução (MCT[26]) e evolutivo. No caso do MCT, a tarefa é alocada para o recurso o qual imagina-se que terminará primeiro a sua execução, seja por ter um tempo de execução menor ou por estar livre enquanto que outros mais rápidos demorarão a ficar disponíveis. No caso do algoritmo evolutivo, utiliza-se algoritmos genéticos e, com base nas previsões de estado dos recursos, tenta-se maximizar o casamento entre alocação de recursos e associação de tarefas [27].

Os testes realizados com estes escalonadores mostram grande variabilidade de resultados quando o algoritmo evolutivo é utilizado, e ao mesmo tempo este algoritmo consegue alcançar um resultado sub-ótimo de alocação de recursos em poucas iterações.

---

por nenhum algoritmo de Aprendizado por Reforço neste contexto

[23] também conhecidas como best effort

[24] Grid Information Index Server

[25] exponential smoothing

[26] Minimum Completion Time

[27] métrica conhecida como fitness

4.5 Gridbus

Aqui será apresentado o GridbusBroker. O GridbusBroker é a extensão do Nimrod/G, um negociador[28] de recursos de um grid computacional para grids distribuídos orientados a dados. Também é extensível do Nimrod/G uma linguagem de modelagem paramétrica para suportar parâmetros dinâmicos, ou seja, cujos valores são determinados em tempo de execução. É semelhante ao AppLeS, que também suporta o desenvolvimento de aplicações de trocas de parâmetros[29], sendo que o seu algoritmo de escalonamento mantém ênfase no reuso de dados. Maiores detalhes sobre este escalonador e os experimentos podem ser encontrados em [25] e [26].

O trabalho realizado no GridbusBroker foca em estratégias de escalonamento de recursos dentro de um grid de dados, particularmente em algoritmos adaptativos de escalonamento e negociação de recursos heterogêneos compartilhados por múltiplos jobs de usuários. Um ambiente de computação intensivo em dados pode ser tomado como um sistema econômico do mundo real onde existem consumidores e produtores de dados. Os produtores seriam as entidades que produzem os dados e controlam a sua distribuição, espelhando-as em diversas localidades. Os consumidores são os usuários que precisam investigar conjuntos de dados específicos dentre centenas ou milhares.

São entradas para o GridbusBroker as tarefas e seus parâmetros correspondentes. Estes podem ser especificados em um arquivo texto com as especificações de tarefas, tipos de parâmetros e seus valores. Estes parâmetros podem ser dinâmicos ou estáticos, caso sejam determinados ou não durante a execução.

Uma tarefa é uma sequência de comandos descrita nos requisitos do usuário. Os requisitos da tarefa norteiam a descoberta de recursos tais como nós computacionais e dados. Os arquivos de dados podem ser descobertos como nomes lógicos de arquivos (LFN[30]), dentro de uma estrutura virtual de diretórios, usando um serviço de catálogo de réplica/dados. Cada LFN é traduzido em um ou mais nomes de arquivos físicos (PFN[31]) localizados em algum lugar do grid.

Uma vez descrita a tarefa, ela é decomposta em jobs, que são nada mais que uma instância da tarefa com uma combinação única de seus parâmetros. Um job é uma instan-

[28] broker
[29] parameter-sweep
[30] Logical File Names
[31] Physical File Names

ciação de uma tarefa com uma única combinação de valores de parâmetros. Esta é também a unidade de trabalho que é enviada a cada nó da grade. Os jobs são despachados para os nós remotos através do atuador[32]. Este submete o job ao nó remoto usando a funcionalidade dada pelo software/middleware que é executado no próprio recurso computacional. O componente de monitoração se encarrega de manter informação do estado do job. Após a finalização do job, este agente retorna os resultados para o broker juntamente com algumas informações para depuração. As informações a respeito do job e dos recursos são mantidas de maneira persistente durante toda a sua execução. Cabe ao registrador[33] gravar os registros relativos aos jobs e recursos durante toda a execução.

As principais entidades constituintes do broker são os descritos a seguir. O servidor de computação[34] descreve o nó do grid. O Job é uma abstração da unidade de trabalho enviada ao nó. O job possui a seguinte composição: variáveis, ou simplesmente o valor de um parâmetro; tarefa, ou a descrição dos comandos a serem feitos pelo job, sendo que estes podem ser comandos de cópia, execução ou substituição de parâmetros de configuração. As máquinas de dados (*Data Hosts*) são nós nos quais os arquivos de dados estão guardados. Os arquivos de dados (*Data Files*) representam os atributos dos arquivos de entrada, tais como tamanho e localização.

O escalonador olha para o grid de dados sob o ponto de vista dos dados. Proximidade de rede entre um recurso computacional e um nó de dados é uma medida de banda disponível entre estes recursos. A heurística utilizada como base para o escalonamento é minimizar a quantidade de dados transferidos durante a execução dos jobs, despachando-os para nós que estejam próximos da fonte de dados. Sob o ponto de vista do usuário, a medida mais importante é o quão rápido os seus jobs são terminados. O escalonador, então, utiliza a razão entre número de jobs completados[35] pelo número de jobs alocados para avaliar o desempenho dos recursos computacionais. A média desta razão é um indicador aproximativo do desempenho deste recurso. Para cada recurso é associado um limite de jobs, ou seja, o número máximo de jobs que podem ser alocados, vindos da lista de jobs em espera, é proporcional a sua média de razão de completude.

Para a validação da plataforma, foi realizada uma simulação de um experimento de

[32] actuator
[33] bookkeeper
[34] ComputeServer
[35] ou simplesmente razão de completude

cadeia de queda de partículas[36]. Uma cadeia de queda ocorre quando uma partícula instável quebra-se em outra, e assim por diante, até que um estado de partículas estáveis é atingida. O experimento envolve a cópia de arquivos de configuração, módulos de análise, arquivos de dados, e produção e cópia de histogramas entre os nós. Para a monitoração de recursos de banda, cada recurso teve um sensor NWS[37], reportando ao servidor NWS o estado dos recursos.

Nos experimentos realizados foram comparadas três estratégias de escalonamento: apenas recursos com dados são escalonados, escalona-se qualquer recurso ignorando a fonte dos dados, e o escalonamento adaptativo que tenta otimizar a computação baseada na localização dos recursos. Maiores detalhes sobre os experimentos, incluído os gráficos com os resultados podem ser vistos tanto em [25] quanto em [26]. A primeira estratégia, mesmo eliminando qualquer possibilidade de transferência de dados, possui limites tais como a impossibilidade de reassociar o job a algum outro recurso disponível, uma vez que este está preso aos dados onde estão localizados. A segunda estratégia implica em ter uma quantidade muito grande de transferências, às vezes desnecessárias, e que com dados de grande escala, tornam inviável a sua execução. A última estratégia, otimização pela localização de dados, é a que melhor consegue escalonar recursos, tendo o menor tempo de todas as três.

4.6 PAUÁ

Em um ambiente de grid, o escalonador global sempre deve interagir ou pelo menos levar em consideração os escalonadores locais ao tentar atingir seus objetivos. Aplicações do tipo *Bag-of-Tasks* são aplicações paralelas cujas tarefas são independentes umas das outras. Manter o foco em tal tipo de aplicação é interessante porque o problema pode ser simplificado, mas ainda sim se mantém útil e relevante. Podemos considerar o escalonamento em grid como sendo o resultado de decisões de vários agentes autônomos ainda que relacionados. Entre os aspectos considerados na heurística de escalonamento, podemos citar a dinamicidade da disponibilidade de recursos, o impacto de grande transferência de dados, a possível coordenação entre diversos escalonadores para a entrega de um serviço combinado e a virtualização como uma maneira de transparecer o escalonamento.

[36] decay chain of particles
[37] Network Weather Service

Alguns esforços foram realizados em escalonamento de aplicações *Bag-of-Tasks* tais como nos escalonadores APST e Nimrod/G. Podemos citar também o OurGrid e o MyGrid. Sendo assim, tanto o Condor quanto o MyGrid são escalonadores centrados em sistema[38], tal qual MyGrid, Nimrod/G e APST são escalonadores centrados em usuários[39].

PAUÁ é uma iniciativa brasileira de construção de um grid nacional. Maiores detalhes deste escalonador podem ser encontrados em [8]. Um de seus objetivos é usufruir da infraestrutura de numerosos recursos computacionais instalados em diversos centros de pesquisa, criando um grid geograficamente distribuído em todo o país. Um outro objetivo seu é incentivar a pesquisa em grid, de forma que as soluções propostas sejam continuamente melhoradas.

Para o PAUÁ, foram desenvolvidos um conjunto de escalonadores conjuntamente responsáveis pelo escalonamento. O usuário submete um job do tipo *Bag-of-Tasks* para um escalonador, o qual envia requisições aos vários sítios onde o usuário possui conta ou acesso. Assim que este começa a receber o resultado das requisições, os jobs que compõem a aplicação são enviados para execução remota. O objetivo primário deste escalonador é o de minimizar o tempo de execução.

A infraestrutura de trabalho foi construída na forma de uma rede de favores, chamada de OurGrid. Nela, temos que cada sitio escalonador na verdade é um *peer* OurGrid, sendo que o escalonador de jobs é chamado de MyGrid. Ainda que as aplicações do tipo *Bag-of-Tasks* sejam as mais simples possíveis, por se tratar de escalonamento em grid, ela se torna uma tarefa bastante complexa pois, em um ambiente de grid, não só a informação é difícil de ser obtida com precisão ou confiabilidade, como também, é necessário considerar o tamanho das transferência de dados como parâmetro do escalonamento. São poucos os escalonadores de grid que consideram as transferências de dados na realização de suas heurísticas, tal como o Xsuffrage.

Neste trabalho, foi desenvolvida uma heurística chamada de afinidade de repositórios[40], que tenta levar em consideração a falta de informações confiáveis e troca de dados entre os sítios. Ela é na verdade um índice que mede a quantidade de bytes para a entrada da tarefa que já estão localizados no sítio em questão para executar a tarefa. Como esta heurística não utiliza informações dinâmicas para escalonar tarefas, é esperado que ela faça escolhas ineficientes de processadores.

[38] system-centric
[39] user-centric
[40] originalmente, storage affinity

O experimentos mostram que tanto as heurísticas afinidade de repositórios e Xsuffrage têm melhor desempenho que a fila com duplicação[41], sendo equivalentes entre si. Utilizando-se uma estratégia de replicação em conjunto com a heurística de afinidade de repositório, temos resultados factíveis ainda que com grande consumo não aproveitado de CPU.

Em geral, para grids de aplicações *Bag-of-Tasks*, temos dois tipos de recursos: máquinas MPP[42] ou clusters, que compartilham recursos em espaço sendo dedicadas em tempo a tarefa, e máquinas desktops, que compartilham recursos em tempo. Aqui, um outro tipo de recurso foi adicionado: o escalonador de sítio[43]. Este nada mais é do que a representação dos recursos do sítio, que serão disponibilizados aos demais escalonadores em nível acima dele próprio. São atribuições do escalonador do sítio a verificação dos direitos de acesso dos jobs ao grid, a abstração dos recursos constituintes do sítio, e a arbitragem entre as demandas do sítio e as do grid.

Há também uma outra inovação dentro do OurGrid que é a rede de favores[44]. Esta trata-se de uma maneira de incentivar os componentes do grid em oferecerem recursos a serem compartilhados, promovendo uma política de justiça e equidade na distribuição dos recursos.

A aplicação utilizada para avaliação é a divisão entre números muito grandes, bastante intensiva em processamento. Os experimentos foram realizados em duas configurações diferentes. Num primeiro instante, as tarefas submetidas eram mais curtas, com duração próxima a 1 minuto, na configuração de hardware. Depois, um segundo conjunto de tarefas cerca de 5,2 vezes mais longo foi submetido. A aceleração[45] obtida foi em média de 7,5 para as tarefas curtas e de 22,3 para as tarefas longas. O fato de as tarefas mais longas possuírem uma aceleração maior revela que o ambiente PAUÁ requer que as aplicações possuam grande granularidade para apresentarem um melhor desempenho.

[41] originalmente, workqueue with duplication
[42] massive parallel processors
[43] originalmente, SiteScheduler
[44] originalmente, network of favors
[45] speed-up

Capítulo 5

Implementação

Nesta seção, iremos apresentar dois dos principais componentes para realização dos experimentos deste trabalho: o GridbusBroker, que é a ferramenta principal no qual o trabalho foi realizado e clusters, pois estes serão utilizados pelo GridbusBroker como unidades disponíveis de recurso computacional para sua alocação à jobs. Além destes dois componentes, será mostrada o cerne da contribuição deste trabalho na última sessão, ao se revelar os detalhes de implementação dos algoritmos desenvolvidos dentro do ambiente GridbusBroker. Este é a principal contribuição deste trabalho, pois o objetivo é justamente verificar como os algoritmos portados para o GridbusBroker se comportam em um ambiente real de grid. A primeira sessão trata da ferramenta de trabalho, os conceitos nela embutidos e alguns motivos pelos quais esta foi utilizada aqui. A segunda sessão trata dos conceitos relacionados a cluster mais importantes a este trabalho. A terceira e última sessão detalha a implementação dos algoritmos no ambiente escolhido para realização dos experimentos.

5.1 GridbusBroker como ambiente de trabalho

Esta subseção irá dar ao leitor uma visão mais detalhada do que vem a ser o GridbusBroker como ferramenta e como este foi utilizado neste trabalho. Por isso, o foco aqui serão os principais conceitos envolvidos na utilização da ferramenta e seus aspectos técnicos de implementação e uso, além dos algoritmos desenvolvidos. Como o que será descrito aqui é apenas um resumo de tópicos importantes da ferramenta, é recomendado aos leitores que queiram se aprofundar que consultem o manual desta, que se encontra disponível em [19].

Gridbus é um projeto da Universidade de Melbourne para criação de uma infraestrutura de grid. Este possui um conjunto de várias aplicações comuns ao tipo de infraestrutura mencionado, tais como simuladores (GridSim), monitores de nós (G-Monitor), autenticadores (Grid Bank), entre outros. Várias destas aplicações não foram utilizadas neste trabalho, e por isso não serão analisadas aqui.

Uma aplicação que foi utilizada e faz parte deste projeto é o GridbusBroker, que é responsável por realizar a associação e submissão de jobs aos recursos computacionais disponíveis. Isso significa acompanhar o percurso do job em seu histórico de execução, desde que este é escalonado a um host em particular até a sua completa finalização, passando pelas tarefas de enviar arquivos de entrada, executar o job no recurso computacional ou submetê-lo a uma fila de execução, adquirir os arquivos de saída e de resultado e computar suas estatísticas. Trata-se de uma aplicação *multi-threaded* onde as informações são guardadas em um banco de dados, e as threads se comunicam por meio de consultas a uma base de dados própria do aplicativo.

O GridbusBroker (GBB) foi escrito em linguagem java, compatível com as JVMs[1] iguais ou superiores a versão 1.4. Ele consiste em uma aplicação que executa um conjunto de jobs definidos pelo usuário em um determinado ambiente também descrito por este, os quais são descritos em três arquivos de entrada no formato XML[2]. Um arquivo descreve o conjunto de jobs a ser executado junto com os parâmetros de execução a ele relacionados, que pode ser chamado de arquivo de descrição da aplicação[3]. Os dois demais arquivos estão relacionados à descrição do ambiente de computação: arquivo de descrição de serviços[4] especifica os recursos computacionais disponíveis para utilização, em geral, hosts onde existe instalado um RMS[5] para o qual o job será submetido; e o arquivo de descrição de credenciais, que nada mais é que a descrição de como é possível realizar acesso aos recursos computacionais descritos no arquivo de serviços. É de responsabilidade do usuário a edição destes arquivos para montagem correta da infraestrutura.

Um dos possíveis parâmetros a ser especificado no arquivo de descrição da aplicação é o tipo de otimização que se quer utilizar para a alocação e submissão dos jobs. A especificação do tipo de otimização determina o escalonador que será utilizado para a alocação dos

[1] Java Virtual Machine
[2] Extensible Format Language
[3] na documentação, o termo em inglês correspondente é application file
[4] na documentação, service file
[5] Resource Management System

jobs aos nós da grade[6]. Os valores possíveis para o tipo de otimização são sem otimização (*NONE*), otimização por custo (*COST*), por tempo (*TIME*) e por ambos os parâmetros anteriores (*COST_TIME*). Embora haja quatro tipo de otimizações possíveis, sendo apenas três destas realmente efetivas, estas se referem a dois tipos de escalonadores, sendo um deles responsável por implementar os três tipos de otimizações ativas.

A figura 5.1 mostra o conteúdo de um arquivo de descrição da aplicação a ser executada no GridbusBroker. Trata-se de executar 1000 jobs do executável *tiny*, tendo como arquivo de entrada *arq1000.txt* e de saída *mc.txt*. Para maiores informações sobre a sintaxe deste e dos demais arquivos citados, é necessário ler o manual[7] do GridbusBroker em [19].

Sobre este escalonador ainda é possível se aplicar uma outra otimização relativa a transferência de arquivos entre os nós, caso a aplicação possua arquivos guardados em repositórios de dados de um grid. Neste caso, um terceiro escalonador é atribuído a aplicação, o qual dentro da aplicação é chamado de *DBDataScheduler*. Os demais escalonadores são o *RoundRobinScheduler*, que não possui otimização ativa, e o *DBScheduler*, que possui este nome pois tenta realizar as otimizações descritas anteriormente tendo como fundo informações guardadas na base de dados da aplicação.

Existe um conjunto de razões que tornaram o GridbusBroker atraente para uso neste trabalho, e que por conta disso acabou sendo escolhida como a ferramenta no qual este trabalho foi desenvolvido. Uma primeira razão é que se trata de uma ferramenta de código aberto e disponibilizada nos termos da licença GPL[8] versão II, que garante o direito de alteração dos fontes. Uma outra razão importante é o fato de o seu uso não implicar na necessidade de instalação de software em qualquer nó que pudesse ser agregado à grade. A aplicação executa em uma máquina apenas e as demais máquinas constituintes da grade não necessitam de nenhum serviço especial do GBB instalado remotamente para fazer parte da grade. Basta a máquina da grade ser acessível por meio de *ssh*[9] e o usuário ter uma conta nela com login disponível por este serviço, o que simplifica bastante a montagem de um ambiente de execução. Uma terceira razão é que a aplicação possui interface disponível com alguns RMS bastante comuns para submissão de jobs tais como Torque/OpenPBS e SGE, além do próprio Globus.

[6]dentro da aplicação, estes aparecem como conceitos de serviços

[7]a tag queueWaitTime não terá referência dentro do manual, pois se trata de uma alteração realizada por este trabalho. Ela especifica o tempo máximo de espera em fila que um job aguardará até ser desassociado do RMS e posteriormente reescalonado.

[8]Gnu Public License

[9]secure shell

```
<?xml version="1.0" encoding="UTF-8"?>
<xpml xmlns="http://schemas.gridbus.org/xpml/2006/01/xpml"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://schemas.gridbus.org/xpml/2006/01/xpml
  XPMLSchema.xsd">

<qos>
        <deadline value="2007-09-27T22:59:59"/>
        <budget value="789.0" />
        <queueWaitTime value="750"/>
        <optimisation value="NONE" />
</qos>
<parameter name="X" type="integer" domain="range">
        <range from="1" to="1000" interval="1"/>
</parameter>
<parameter name="file" type="string" domain="single">
        <single value="arq1000.txt"/>
</parameter>
<task>
        <copy>
                <source location="local" file="user/inputs/tiny.$arch"/>
                <destination location="node" file="tiny.$arch"/>
        </copy>
        <copy>
                <source location="local" file="user/inputs/arq1000.txt"/>
                <destination location="node" file="arq1000.txt"/>
        </copy>
        <execute>
                <command value="./tiny.$arch"/>
                <arg value="$file"/>
                <arg value="$X"/>
        </execute>
        <copy>
                <source location="node" file="mc.txt"/>
                <destination location="local" file="mc.txt.$jobname"/>
        </copy>
</task>
</xpml>
```

Figura 5.1: Conteúdo exemplo de um arquivo descritor das tarefas executadas

Cabe ainda uma última palavra sobre outros meta-escalonadores que poderiam vir a ser utilizados neste trabalho, assim como as razões que fizeram com que estas ferramentas fossem preteridas pelo GridbusBroker. A avaliação das ferramentas disponíveis teve por base um estudo publicado em [15], onde foram citados os meta-escalonadores mais populares. Podemos citar nominalmente algumas plataformas que foram avaliadas inicialmente: Community Scheduler Framework (CSF), GridWay, EGEE Workload Manager Service (EGEE-WMS), Condor-G, Nimrod/G, MP Synergy e Moab Grid Scheduler. As duas últimas opções citadas anteriormente pertencem a produtos comerciais fechados, do qual não é possível obter os fontes para posterior modificação, e por isso foram descartadas. Caso parecido acontece com o Condor-G, onde os fontes também não são disponibilizados ainda que a plataforma não tenha restrição de uso. O Nimrod/G oferece os fontes aos usuários que o requisitarem, por meio de mensagem eletrônica, mas possui restrições com relação a sua posterior modificação. Além disso, como este fez parte da árvore genealógica do GridbusBroker, este último já possui grande parte de suas funcionalidades agregadas ao próprio. O GridWay e o EGEE-WMS necessitam da instalação de uma infraestrutura mais complexa nos nós constituintes da grade, para que possam ser utilizados. No caso do GridWay, é necessário a utilização do Globus, e para o EGEE-WMS o ambiente EGEE, que aliás, é bastante parecido com o Globus em complexidade e funções. Caso semelhante ocorre com o CSF, que necessita da instalação do Globus, além de ser restrito ao uso de um único RMS, no caso do LSF[10], que por ser um produto comercial fechado, não possui o uso tão difundido quanto outros RMS, tais como OpenPBS, Condor ou SGE.

5.2 Clusters, Gerenciadores de Recursos e filas locais

Podemos considerar um cluster como sendo um conjunto de estações de trabalho, ou sistemas com múltiplas CPUs, ou ainda um conjunto de nós em um computador paralelo. Geralmente, este conjunto de nós, CPUs ou estações de trabalho possuem um único servidor para processamento em lotes [11] de seus jobs. Estão relacionados ao conceito de processamento em lotes outros conceitos que serão descritos a seguir. Um estudo mais detalhado sobre este tema pode ser encontrado em [12].

[10] Platform LSF

[11] batch processing

Uma fila, no âmbito de um sistema de processamento em lotes, é um conjunto de entidades escalonáveis, tais como jobs ou tarefas a ele relacionadas. A fila é um dos conceitos mais importantes em um ambiente de cluster. A ela estão associados atributos tais como, tempo máximo do processamento do job, prioridades entre as filas[12], máquinas a ela associadas, número máximo e mínimo de recursos computacionais, por exemplo CPUs, entre outras.

As políticas de escalonamento em um sistema de processamento em lotes são implementadas tanto dentro das filas (ordenação da seleção dos jobs pertencentes a uma fila) quanto na relação entre as filas (seleção entre jobs de diferentes filas para ocupar um recurso disponível). O nome fila pode dar a impressão de que as políticas de escalonamento a ela associadas se restringem apenas à política FIFO[13]. De fato, esta é uma possível política implementada em filas, porém não se trata da única possível. Por conta disso, neste contexto deve-se desassociar a idéia de filas com algoritmos FIFO necessariamente.

Outros conceitos importantes são o de lote, processamento em lote, servidor e sistema. O lote em si pode ser definido como um conjunto de jobs que são submetidos para processamento, para os quais o resultado será obtido em um momento futuro. Processamento em lote é a capacidade que estes sistemas possuem de executar jobs sem a necessidade de um ambiente com shell interativa, além de proverem controle sobre os recursos e o escalonamento dos jobs. O servidor é um serviço ativo em uma única máquina que provê a capacidade de processamento em lote. O sistema de processamento em lotes é o conjunto de servidores que está configurado para o processamento.

Os sistemas de processamento em lotes devem executar um conjunto de tarefas como as descritas a seguir: monitorar o estado dos recursos disponíveis; aceitar jobs submetidos por usuários juntamente com os requisitos associados a cada um destes; realizar o escalonamento centralizado dos jobs de acordo com a política definida e os recursos disponíveis; alocar recursos a jobs e iniciar sua execução; e monitorar o estado dos jobs coletando suas informações.

Um cluster não é necessariamente um ambiente homogêneo. Pode existir algum grau de heterogeneidade, como por exemplo, a existência de filas que submetem jobs para máquinas arquiteturalmente diferentes. Isto porém não ocorre de forma tão ampla e genérica quanto em um grid que torna isso transparente ao usuário. Uma outra limitação

[12] existem também prioridades associadas aos jobs dentro de uma fila

[13] First in first out

dos clusters é que estes possuem problemas de escalabilidade, além de vulnerabilidade à falhas e mesmo à segurança, no que diz respeito ao escalonamento entre vários domínios, o que torna seu uso inadequado para o ambiente de grid. A importância destes sistemas é que, neste estudo, estes serão tomados como parte do ambiente de grid, tal como um nó da grade.

5.3 Trabalho Realizado e Contribuição

O trabalho desta dissertação consistiu em inserir duas heurísticas de meta-escalonamento na ferramenta GridbusBroker, como novos escalonadores disponíveis para esta ferramenta. Ambas as heurísticas se baseiam na classificação dos recursos, utilizando-se o aprendizado por reforço[14] para classificação dos recursos computacionais disponíveis. As heurísticas aqui verificadas tiveram inspiração nos algoritmos descritos na revisão bibliográfica deste trabalho como Algoritmo de Galstyan[15] (AG) e *Multiple Queue with Duplication* [16] (MQD). Este último teve o algoritmo de aprendizado por reforço como fator de classificação de eficiência entre os recursos computacionais disponíveis. Recursos computacionais, no contexto deste trabalho, serão entendidos como filas para submissão de jobs em diversos clusters onde havia disponível uma conta de usuário própria para execução de jobs, ou seja, um RMS[17]. Recursos computacionais também podem ser entendidos como serviços computacionais[18] dentro da ferramenta GridbusBroker.

Assim sendo, a estrutura do GridbusBroker foi alterada de modo que cada recurso computacional estivesse relacionado a um parâmetro que medisse a sua eficiência. No início, todos os recursos computacionais têm atribuídos uma eficiência inicial igual a zero. Esta, porém, será continuamente atualizada a medida em que os jobs forem completando a sua execução. O algoritmo de atualização é simples. Ao término da execução de um job, é feito um cálculo, tendo por base tanto o tempo total de execução deste no sítio em que fora atribuído para execução, quanto o tempo total para completude do job, para atualizar a eficiência. Os detalhes de como é calculada a eficiência seguem logo abaixo.

[14]Reinforcement Learning

[15]originalmente, na referência bibliográfica [14], este aparece com o nome de *Reinforcement Learning*. Como porém esta designação é confusa para o leitor, pois faz referência direta a técnica de Aprendizado por Reforço, achamos por bem renomear o algoritmo para Algoritmo de Galstyan.

[16]em português, Filas Múltiplas com Duplicação

[17]Resource Management System

[18]Compute servers

Chamemos as duas grandezas de tempo relevantes para análise de: tempo de execução (te) e tempo total (tto). O que diferencia um do outro é que o tempo total (tto) inclui, além do tempo de execução (te), o tempo gasto remotamente na fila do RMS para o qual este foi submetido ou tempo de fila (tf), junto com o tempo gasto com a transferência de arquivos de entrada e saída ou tempo de transferência (ttr). Outras atividades como a submissão propriamente dita do job ao recurso, ou a atualização do estado interno em que se encontra o job no GridbusBroker, por exemplo, ainda que desprezíveis em termos de tempo gasto, também são computáveis no tempo total. Tempo total (tto) é medido pelo GridbusBroker, sendo o início de sua medição o momento em que o job entra no estado escalonado[19] e o fim o momento em que este entra no estado finalizado[20]. Tempo de execução é medido dentro do script de submissão, sendo seu início o momento antes da chamada do executável do job e a sua finalização logo após o término do executável em questão. Em geral, podemos afirmar que:

$$tto = te + tf + ttr \tag{5.1}$$

Como vimos, ao final da execução de um job, temos do GridbusBroker o tempo de execução (te) e o tempo total do job (tto). O cálculo da eficiência do recurso se faz como descrito a seguir. Calcula-se a média do tempo de execução (tem), seu desvio padrão ($stdtem$) e a média do tempo total (ttm) de todos os jobs até então finalizados. Em seguida, é atribuído um peso α para o cálculo de um índice de tempo de execução (pi) relativo ao job recém terminado, onde:

$$pi = te \cdot \alpha + (1 - \alpha) \cdot (tto - te) \tag{5.2}$$

O parâmetro α tem a função de especificar o quanto de importância será dado ao tempo de execução, considerada a parte menos volátil dentre os componentes do tempo total de execução, no índice (pi) de tempo. De forma análoga, é calculado um índice de tempo médio de execução (pim), onde:

$$pim = tem \cdot \alpha + (1 - \alpha) \cdot (ttm - tem) \tag{5.3}$$

Tendo conhecimento destes dois índices de tempo e o desvio padrão do tempo médio de execução ($stdtem$), compara-se pi com pim. Se pi é maior que pim mais um desvio

[19] scheduled, momento em que este é atribuído a um recurso computacional

[20] finished, momento em que não há mais nada a ser feito com o job

padrão, o resultado é uma recompensa negativa à eficiência do recurso. Se pi é menor que pim menos um desvio padrão em questão, o resultado é uma recompensa positiva à eficiência do recurso. Caso nenhuma das duas situações ocorra, ou seja pi está dentro do intervalo entre pim, considerando-se um $stdtem$, a eficiência do recurso fica inalterada.

A atualização da eficiência, por sua vez, não é feita de forma direta, atribuindo o valor da recompensa, positivo ou negativo, ou somando-se a recompensa diretamente à eficiência. Para atualizar a eficiência, é utilizado um parâmetro de aprendizado (l), para levar em consideração o histórico desta. A nova eficiência que será atribuída ao recurso computacional ($nrle$) será igual a uma ponderação entre a eficiência anteriormente atribuída ao recurso computacional (rle) e a recompensa (r) a ele atribuída, ponderada pelo parâmetro de aprendizado (l), da seguinte forma:

$$nrle = rle + l \cdot (r - rle) \tag{5.4}$$

Obviamente, se não há recompensa, não há atualização da eficiência do recurso, e nesse caso $nrle = rle$.

Algorithm 1: getNewRlEfficiency

Data:
tto, *// tempo total*
te, *// tempo de execução*
tem, *// tempo médio de execução*
stdtem, *// desvio padrão do tempo médio de execução*
rle *// índice atual da eficiência do recurso*
Result:
nrle *//novo valor de eficiência do recurso computacional*

inicialização dos parâmetros α e l;
$pi \leftarrow te * \alpha + (1 - \alpha) * (tto - te)$;
$pim \leftarrow tem * \alpha + (1 - \alpha) * (ttm - tem)$;
if $(pim - stdtem) > pi$ **then**
  $r \leftarrow 1$;
  $nrle \leftarrow rle + l * (r - rle)$;
else
  if $(pim + stdtem) < pi$ **then**
    $r \leftarrow -1$;
    $nrle \leftarrow rle + l * (r - rle)$;
  else
    $nrle \leftarrow rle$;
  end
end
return $nrle$

Após a descrição mais detalhada de como é feita a atualização do parâmetro principal dos algoritmos utilizados neste trabalho, resta agora explicar como ela é utilizada pelos algoritmos que foram implementados. Recapitulando, temos duas implementações realizadas, as quais chamamos de AG[21] e MQD[22]. Ambas foram inspiradas em trabalhos descritos na seção anterior 2 de revisão bibliográfica e por isso os detalhes destes ficarão nesta seção, ou mesmo na referência bibliográfica caso sejam necessários maiores detalhes. De uma maneira simples, podemos dizer que o que difere as implementações de AG e MQD aqui realizadas, é a maneira gulosa ou não, de se alocar recursos computacionais disponíveis para os jobs.

No caso do algoritmo de Galstyan aqui implementado, a heurística de alocação é gulosa, porém não é determinística e sim probabilística. Ou seja, em um dado momento quando haverá um evento de escalonamento, com uma probabilidade grande (no caso

[21] de Algoritmo de Galstyan
[22] Multiple Queues with Duplication

de nossa implementação, fixamos este valor em 85%), o recurso escolhido para o job a ser atendido será aquele, dentre os disponíveis, que tiver a maior eficiência. Com uma probabilidade pequena e complementar a anterior, será escolhido aleatoriamente um recurso qualquer dentre os disponíveis. Entenda-se por recurso disponível, não só aquele que possui unidades de computação livres para serem alocadas, como também que estas satisfaçam os requisitos específicos do job a ser alocado, se este possuir algum.

O algoritmo MQD, por sua vez, tenta fazer uma associação entre agrupamentos de jobs e os recursos disponíveis da forma descrita a seguir. Num primeiro momento ou estágio inicial de execução do algoritmo, é feita a ordenação dos jobs existentes e são selecionados para execução os de duração mais curta[23], os quais serão atribuídos aos RMS disponíveis para a alocação, organizados numa lista com uma ordem qualquer definida[24] de maneira intercalada. Esta fase termina quando forem alcançados os limites de jobs por filas[25] dos respectivos RMS, a qual seguirá a segunda e última fase de execução do algoritmo.

[23]é suposto que seja conhecido senão uma previsão do tempo de execução, pelo menos uma maneira de ordená-los pelo seu tempo de execução previsto

[24]no caso da implementação deste trabalho, foi realizado o algoritmo Round-robin clássico para submeter os jobs

[25]no caso, este limite é igual a 20 para todos os RMS. Cabe apenas lembrar um detalhe: devido a lógica do algoritmo MQD, caso este limite considerado seja muito grande, pode ocorrer do total de jobs escalonados na primeira fase desse algoritmo, ter uma participação muito grande dentro do universo total dos jobs escalonados em cada experimento. Para um total de 4 a 6 RMS disponíveis, como foi o caso dos experimentos aqui realizados, temos um intervalo de 80 a 120 jobs escalonados sem que cada RMS estivesse classificado ou ordenado por sua eficiência. Considerando-se um universo de 500 jobs escalonados por experimento, temos um percentual entre 16 a 24 % de jobs que não tiveram sua alocação dirigida pela classificação por eficiência dos RMS. Acreditamos que o ideal é que este percentual não exceda este valor máximo de 24 %.

Algorithm 2: agScheduling

Data:
Conjunto de Jobs não escalonados,
Conjunto de RMS não saturados
Result:
Mapeamento entre RMS e Jobs

```
foreach Job i do
    // primeiro sorteia-se a probabilidade do escalonamento aleatório
    p ← número real no intervalo [0;1];
    if p > 0,85 then
        // com baixa probabilidade a associação é aleatória
        r ← sorteia(RMS);
        associa(r,i);
    else
        // com alta probabilidade a associação é com RMS de maior eficiência
        r ← maxEff(RMS);
        associa(r,i);
    end
end
```

Na segunda fase de execução do algoritmo, é preciso agrupar os jobs restantes para associá-los aos RMS disponíveis. Tendo-se a quantidade de recursos computacionais disponíveis para alocação e a quantidade de jobs ainda não escalonados, obtém-se a razão desta última grandeza pela primeira, a qual podemos chamar de razão de agrupamento, pois esta grandeza servirá de parâmetro para formar o agrupamento de jobs a ser enviado a um RMS específico, dentre os disponíveis. Uma vez conhecida a razão de agrupamento, ordenam-se todos os jobs ainda não escalonados de forma decrescente, de maneira que o primeiro grupo, correspondente aos jobs mais longos dentro da janela da razão de agrupamento são associados ao RMS que possuir a maior eficiência. O segundo grupo é associado ao RMS com a segunda melhor eficiência calculada e assim por diante até que se chegue ao agrupamento de pior eficiência. Uma vez esgotadas as possibilidades de enfileiramento de jobs nos RMS, aguarda-se até a disponibilização de vagas nas filas, refazendo-se o agrupamento, classificação e associação entre RMS e jobs, procurando-se sempre despachar os jobs de maior duração dentro do agrupamento a que ele pertence. Não há duplicação de submissão de um mesmo job como no algoritmo MQD original, isto por conta de uma limitação do ambiente. A lógica interna no GridbusBroker não é compatível com a possibilidade de duplicação de jobs, na versão utilizada.

Algorithm 3: mqdBeginScheduling

```
Data:
Conjunto de Jobs não escalonados,
Conjunto de RMS não saturados
Result:
Mapeamento entre RMS e Jobs
// jobs são ordenados de maneira crescente
Job ← ordenaCrescente(Job);
// procura o próximo RMS disponível numa lista circular
foreach Job i do
    if estaVazio(RMS) then
        // fase inicial está terminada
        return
    else
        r ← proximo(RMS);
        if estaSaturado(r) then
            // RMS saturados são retirados da lista
            remove(r,RMS);
        else
            associa(r,i);
        end
    end
end
```

Tanto numa fase quanto na outra, há o cálculo de eficiência do recurso tal como fora descrito anteriormente. Há apenas uma pequena adaptação a ser explicitada. Quando o algoritmo MQD entra na sua segunda fase de execução, as grandezas média do tempo de execução (*tem*), desvio padrão (*stdtem*) e média do tempo total (*ttm*), não são calculadas como agrupamentos da totalidade dos jobs já finalizados, tal como foi feito até então. Estas grandezas agora são calculadas como médias de cada um dos RMS e não mais da aplicação por completo. Isto foi feito para evitar que recursos que possuíssem um histórico bom de recompensas[26] não fossem penalizados por terem que executar os jobs mais longos, que tenderiam a puxar sua eficiência para baixo.

[26] leia-se, possuíssem eficiência alta e por conta do algoritmo fossem forçados a executar os jobs mais longos

Algorithm 4: mqdMainScheduling

Data:
Conjunto de Jobs não escalonados,
Conjunto de RMS não saturados
Result:
Mapeamento entre RMS e Jobs

```
// jobs e RMS são ordenados de maneira decrescente
Job ← ordenaDecrescente(Job);
RMS ← ordenaPorEficienciaDecrescente(RMS);
// calcula a razão de agrupamento
grp ← tamanho(Job) / tamanho(RMS);
j ← 0;
r ← proximoNaoSaturado(RMS);
foreach Job i do
    if vazio(RMS) then
        // fim dos recursos disponíveis para se escalonar
        return
    else
        // associam-se jobs ao RMS até acabar o agrupamento ou este se saturar
        if associa(r,i) then
            j ← j + 1;
        else
            // saturado o RMS, os jobs pertencentes ao agrupamento serão
            // escalonados apenas em execuções posteriores
            if j < grp then
                j ← j + 1;
            else
                j ← 0;
                remove(r,RMS);
                r ← proximoNaoSaturado(RMS);
            end
        end
    end
end
```

Capítulo 6

Experimentos e Análise dos Resultados

Neste capítulo, apresentaremos a metodologia de realização dos experimentos, os resultados obtidos, assim como serão realizados alguns comentários sobre os resultados obtidos que apontarão para conclusões deste trabalho. Na sessão 6.1 será explicado como foram realizados os experimentos e na sessão seguinte em 6.2, os resultados dos experimentos serão mostrados junto com comentários pertinentes a eles.

6.1 Metodologia dos Experimentos

Feitas as explanações iniciais a respeito do trabalho desenvolvido, voltemos nossa atenção para a parte experimental, a qual nos deve dar subsídio para uma análise do que foi realizado. O objetivo deste trabalho é verificar se as implementações realizadas dos algoritmos acima descritos terão o mesmo comportamento mostrado nos trabalhos realizados com estes, em ambiente simulado pelos próprios autores. Sendo assim, a métrica principal para análise será o *makespan*, ou tempo total de escalonamento, que nada mais é que o tempo total que o GridbusBroker levará para submeter, executar e finalizar todos os jobs para os quais este foi configurado para submeter e executar.

Os experimentos foram realizados tendo a disposição um total de sete RMS. Note que, todos eles são recursos computacionais reais, pertencentes a algum sítio remoto ou local[1], onde uma conta de usuário foi cedida para execução de jobs. Por conta disso, estão todos sujeitos também aos acontecimentos corriqueiros e comuns à RMS reais como, por exemplo, variação no seu desempenho de *throughput*, vazão e tamanho das filas, ocorrência de problemas na execução dos jobs e possivelmente indisponibilidade do sítio, além obviamente de problemas de comunicação e autenticação com a máquina de submissão

[1]no sentido em que está dentro do mesmo domínio de onde o GridbusBroker foi executado

do próprio RMS. Além disso, as contas oferecidas para execução nos sítios não oferecem privilégios de administrador ou superusuário, de maneira que não temos acesso a alteração de configurações do sítio.

Do total de sete RMS disponíveis para execução, nunca foi possível utilizar mais do que seis em um experimento, assim como não foram executados experimentos com menos do que quatro RMS disponíveis para execução. De fato, foram utilizados tantos RMS quantos estivessem disponíveis no momento do experimento, tendo este número variado entre quatro e seis. Como estes estão sujeitos a variações temporais de desempenho, as execuções dos algoritmos foram realizadas por rodadas. Em cada rodada, além dos dois algoritmos AG e MQD é executado também um algoritmo de escalonamento padrão já existente no próprio GridbusBroker chamado de Round-Robin (RR), o qual não realiza nenhum tipo de otimização de escalonamento, pois apenas tenta associar um job na fila a algum recurso computacional disponível. Isto foi feito tanto para se ter um parâmetro de comparação absoluto, como também para efeito de normalização das medidas do tempo total de escalonamento dos algoritmos AG e MQD dentro da rodada. A normalização é feita pela razão de tempo entre o algoritmo RR e o AG ou MQD.

Os sítios disponíveis para submissão de jobs a algum RMS foram:

- 2 clusters do Laboratório de Inteligência Artificial (LabIA) da Coppe/PESC,
- o cluster do Laboratório de Computação Paralela (LCP) da Coppe,
- o cluster do Núcleo de Atendimento em Computação de Alto Desempenho (Nacad) da Coppe,
- o cluster *i*ntegridade do departamento de informática da UFRGS,
- o cluster do Laboratório de Computação Científica (LCC) na UFLA e, por fim,
- o cluster do departamento de Física da UERJ.

Um pequeno detalhe a ressaltar é que no Laboratório de Inteligência Artificial primeiramente citado, haviam no início dos experimentos dois clusters utilizados que posteriormente se fundiram num só. Em um deles, o RMS instalado era o Torque/Maui com 8 processadores e no outro o SGE com 16 processadores. Posteriormente todos fundidos no primeiro RMS. No LCP, o RMS instalado era o SGE; no Nacad, o PBS_PRO; no LCC e

Local	Processadores	RMS
LabIA	24	Torque/Maui
LCP	28	SGE
Nacad	16	PBS_PRO
LCC	44	Torque
UERJ	144	Condor
UFRGS	4	Torque

Tabela 6.1: RMS disponíveis para execução

na UFRGS, apenas Torque sem Maui; e na UERJ, condor. A descrição mais detalhada dos recursos computacionais disponíveis se encontra na tabela 6.1.

Sendo assim, cada experimento supõe uma rodada de execução entre as três implementações de algoritmos. Dentro de cada rodada, a lista de RMS disponíveis para submissão é a mesma, embora esta possa mudar em rodadas futuras, caso algum RMS esteja indisponível ou volte a ficar disponível. Os cenários de execução dos experimentos supõem que o conjunto relevante de parâmetros dos cálculos de eficiência estejam constantes, notadamente o parâmetro de peso (α) e o de aprendizado (l).

Contando-se apenas a variação destes parâmetros, foram produzidos um total de cinco cenários diferentes de experimentos, sendo três referentes a variações do parâmetro α e outros dois referentes a variações do parâmetro l. Nestes, foi executado um total de 500 jobs sem possibilidade de re-escalonamento. Estes servem para determinar empiricamente os valores dos parâmetros em questão com melhor desempenho. Para cada cenário de execução foi realizado um total de 15 experimentos, perfazendo um total de 45 execuções dos algoritmos RR, AG e MQD.

O job a ser executado ([22]) trata-se de uma aplicação que simula a propagação de luz tendo um ponto como fonte em um meio infinito, com dispersão isotrópica. Esta aplicação é intensiva em uso de CPU, mas não requer uso de grandes quantidades de memória, *swap* ou uso de grandes arquivos. Ela foi ligeiramente adaptada do original para que o parâmetro de execução[2] pudesse ser lido da linha de comando. Essa modificação foi necessária para que o job fosse executado no contexto de aplicações de troca de parâmetros[3]. Cada execução da aplicação foi realizada em uma base de dados sempre inicialmente vazia, para

[2]número de fótons

[3]PSA - parameter sweep application

Carga de jobs	Valor mínimo de entrada	Valor máximo de entrada
500	128852	9975258
1000	103421	9991458

Tabela 6.2: Valores de entrada

que, a medida em que esta fosse crescendo, não interferisse no desempenho da própria aplicação.

Os valores de entrada foram tomados aleatoriamente numa distribuição uniforme dentre um intervalo específico e guardados em um arquivo que é dado como entrada ao job. Assim, nas execuções dos três algoritmos dentro da rodada, os valores de entrada dos jobs são sempre os mesmos. Em testes realizados em máquinas do Laboratório de Inteligência Artificial da Coppe, isso significaria execuções do job dentro do intervalo de 3 a 8 minutos[4].

Além dos cenários de execução anteriormente descritos, foram agregados mais quatro outros cenários visando verificar o comportamento dos algoritmos na presença de re-escalonamento e na duplicação do número de jobs associado. Entenda-se por re-escalonamento, o cancelamento da execução de um job em um RMS qualquer, onde este ainda não tenha entrado em execução, após um período pré-determinado de espera na fila do RMS. Posteriormente, a associação do job re-escalonado será feita à um outro RMS ou até ao mesmo ao qual este esteve anteriormente associado, caso o algoritmo de escalonamento vigente assim o decidir. Sempre que um evento de re-escalonamento se fizer necessário, será acrescido 10% ao valor limite de tempo de espera na fila do RMS[5].

Os cenários em questão são a execução dos algoritmos de escalonamento com período de expiração[6] nos valores de 500, 750 e 1000 segundos. Posteriormente, para o quarto cenário de execução, foi tomado o período de expiração como 750 segundos e a carga de jobs foi dobrada de 500 para 1000 jobs. Em todos os quatro cenários de execução aqui descritos, a configuração de parâmetros de cálculo da eficiência dos recursos computacionais será aquela que obtiver o melhor resultado de otimização, dentre as cinco variações destes parâmetros anteriormente citados[7].

[4]configuração básica das máquinas utilizadas: processador P4 2,8 GHz, Memória 1GB, Hard-Disk 80GB, Sistema Operacional Scientific Linux 4.2

[5]este pode ser entendido como um período de expiração da espera do job

[6]timeout

[7]O valor de 750 segundos corresponde à média calculada de tempo de espera em fila até determinado

Por fim, a todos estes experimentos, foi adicionada mais uma métrica para análise, além do *makespan*, que é a métrica principal. Trata-se do balanceamento de carga, pois de um escalonador genérico é esperado também que mantenha a carga de trabalho dividida entre os RMS de maneira a evitar a sobrecarga de um ou alguns deles. Uma das razões também para se analisar o balanceamento de carga é a falta de dados sobre o *makespan* do algoritmo de Galstyan.

No contexto deste trabalho, o balanceamento de carga será medido como a média de ociosidade dos RMS disponíveis para execução, de maneira que, quão maior for esta ociosidade, pior terá sido o balanceamento de carga. A ociosidade do RMS será dada como uma porcentagem do tempo total de escalonamento a qual é o complemento do percentual de atividade do mesmo. O período de atividade do RMS pode ser definido como o espaço de tempo que transcorre entre o início da execução do meta-escalonamento até o momento da finalização do último job que nele será executado. O percentual de atividade é apenas a razão entre período de atividade do RMS e o tempo total de escalonamento. Durante o período de ociosidade do RMS, não apenas suas filas não possuirão mais os jobs pertinentes aos nossos experimentos, como também nenhum deles executará estes jobs até à finalização do algoritmo de escalonamento em questão.

6.2 Resultados e Análise

Recapitulando resumidamente, podemos distinguir os experimentos realizados em três fases distintas: variação dos parâmetros α e l para determinar empiricamente o conjunto destes parâmetros a ser utilizado; desempenho dos algoritmos na presença de re-escalonamento, variando o limite de tempo em fila entre 500, 750 e 1000 segundos; e o aumento em duas vezes a carga de jobs para os algoritmos em questão. Comecemos expondo os resultados da primeira fase dos experimentos.

A análise dos dados assumirá que estes seguem a distribuição normal estatística. Em outras palavras, os erros encontrados na amostra são completamente aleatórios[8], segundo a hipótese. Esta hipótese é tão razoável quanto for possível verificar que a distribuição dos dados coletados não difere muito de uma curva parecida com um sino. Por isso, apresentamos a seguir uma série de três histogramas que possuem dados das execuções

momento dos experimentos. O valor correspondente de desvio padrão desta média é aproximadamente 250 segundos.

[8]também conhecido como ruído branco em estatística

dos algoritmos RR, AG e MQD. Sua função neste trabalho é ilustrar o quão próximo o conjunto de dados medidos está próximo de uma distribuição normal. Dito de outra forma, a hipótese de normalidade dos dados será tão mais irreal o quão distantes estiverem estes histogramas de um sino.

Sobre os dados plotados nos histogramas, estes se referem às 45 execuções dos três algoritmos cada um, nos experimentos sobre reescalonamento. Evitou-se colocar aqui a totalidade de execuções de cada algoritmo de escalonamento pois as variações de parâmetros acabam por particularizar o comportamento dos algoritmos, com exceção apenas do algoritmo RR. Além disso, o número de 45 execuções disponíveis nos testes sobre reescalonamento parece suficiente para visualizar com maior clareza se um histograma desvia claramente do perfil de uma curva de sino. Os histogramas podem ser vistos nas figuras 6.1 para o algoritmo RR, 6.2 para o algoritmo de Galstyan e 6.3 para o algoritmo MQD.

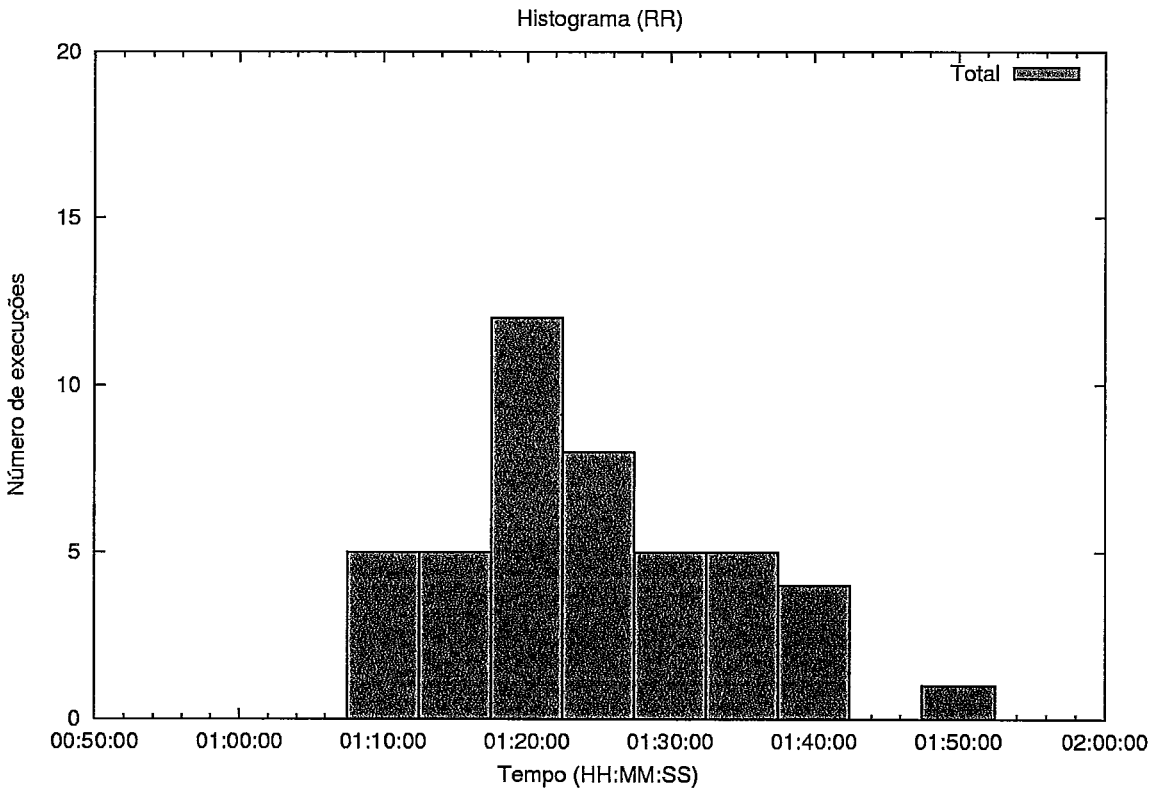


Figura 6.1: Histograma de execuções RR com reescalonamento

Feita essa pequena digressão sobre a hipótese de normalidade dos dados, vamos apresentá-los. A tabela 6.3 mostra as estatísticas absolutas colhidas da execução do primeiro grupo de experimentos, o qual visa verificar qual o conjunto de parâmetros α e l é o mais indicado

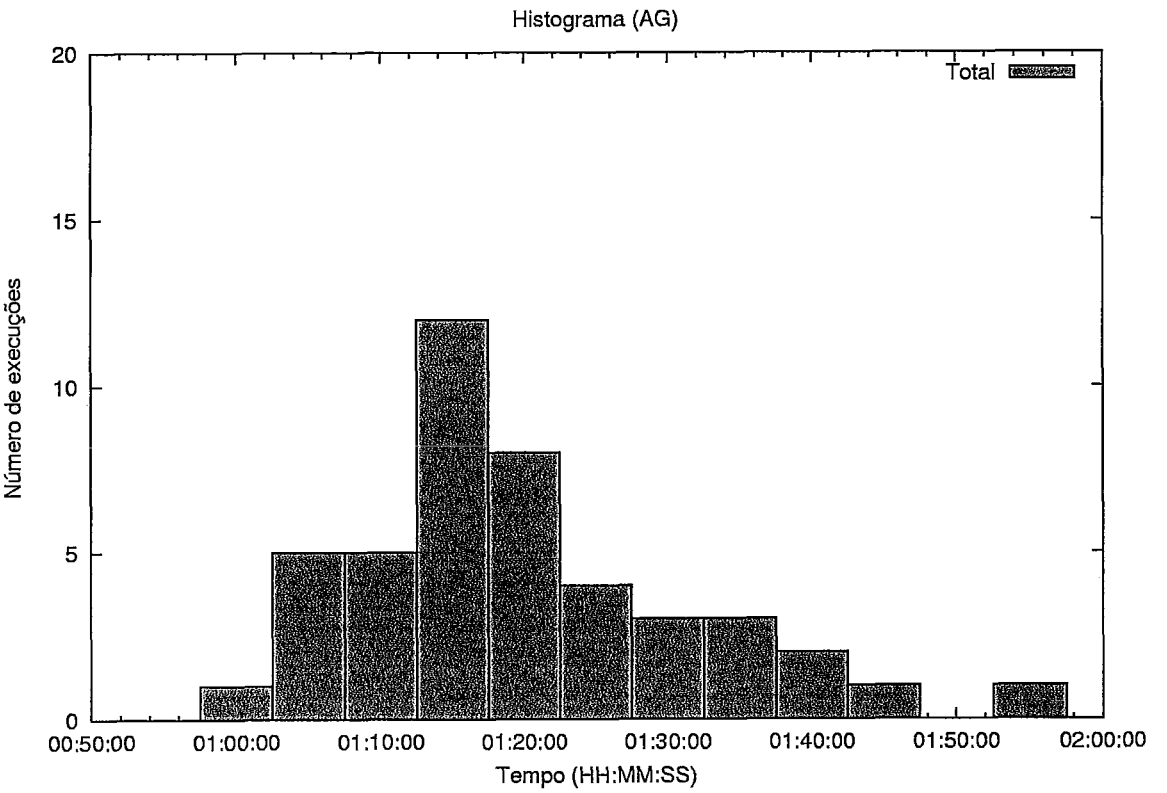


Figura 6.2: Histograma de execuções AG com reescalonamento

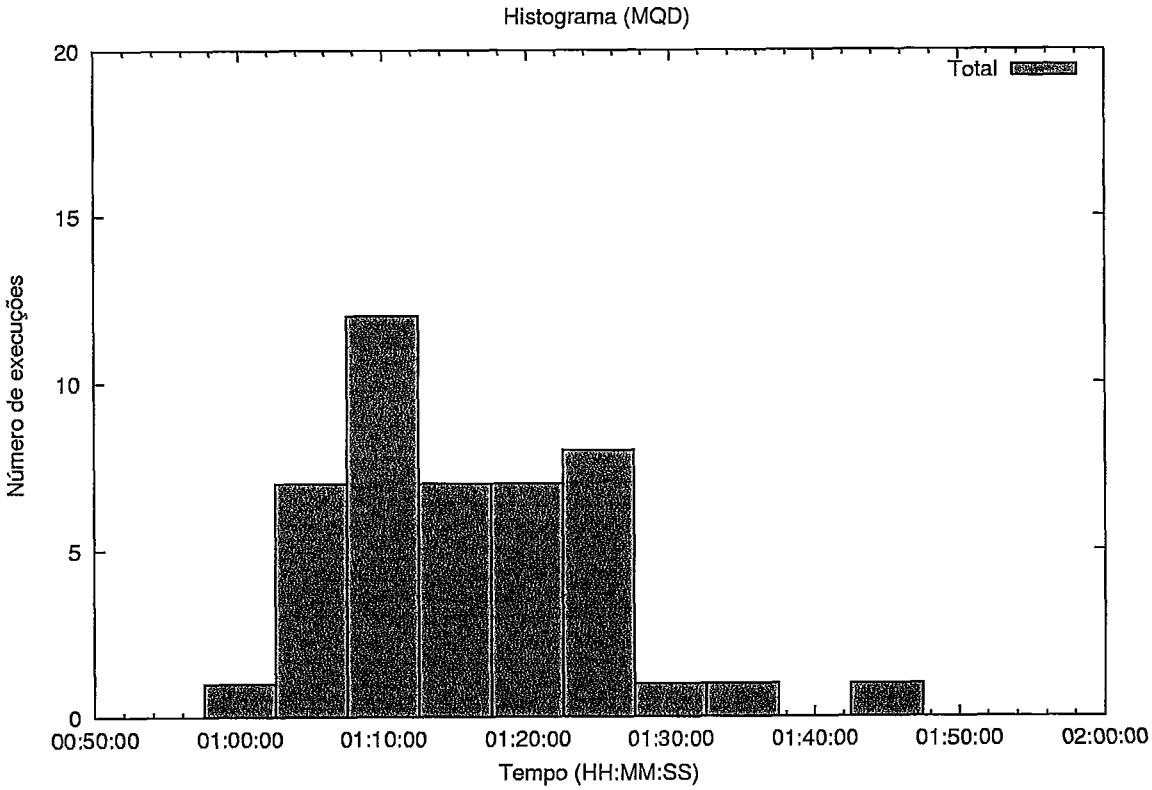


Figura 6.3: Histograma de execuções MQD com reescalonamento

para obter uma otimização do algoritmo. As duas primeiras colunas especificam o ajuste de parâmetros utilizado, lembrando sempre que, para cada linha nas tabelas subsequentes, foram realizados um total de 15 experimentos, de onde foi possível obter as respectivas médias e desvios padrões. Por se tratar de uma tabela com estatísticas absolutas de tempo de execução, os quais foram colhidos num intervalo de tempo relativamente longo, estes dados são apenas ilustrativos do que ocorreu durante a execução, agregando pouco valor à análise dos algoritmos, e por isso não teceremos mais comentários sobre estes.

A análise destes dados será dada pelas demais tabelas apresentadas, mais particularmente 6.4 e 6.5, no que tange à otimização do tempo de execução. A tabela 6.4 mostra a probabilidade de que o tempo de execução médio, medido para os algoritmos de Galstyan e MQD, pertençam a mesma média verificada para o algoritmo RR. Estas probabilidades se referem ao teste estatístico de T-Student entre estas amostras supondo a presença de heterocedasticidade[9]. Em outras palavras, o que esta tabela mostra é a probabilidade de ter havido um ganho ou perda real, para o conjunto de parâmetros utilizados. As probabilidades ali expostas se referem ao teste T-Student, que mede a probabilidade de ambas as amostras possuírem a mesma média. Por isso, os valores dessa tabela devem ser lidos de maneira invertida, pois quão menor for a probabilidade das médias pertencerem à mesma amostra, maior é a certeza de que houve ganho ou perda.

A tabela 6.5 se refere aos dados amostrais de tempo de execução dos algoritmos de Galstyan e MQD, normalizados pelo tempo de execução do algoritmo RR relativo à rodada de execuções do experimento. Sendo mais específico, o que esta última tabela mostra é o ganho ou perda que cada algoritmo teve, em termos de tempo de execução normalizado. Os valores ali presentes tratam-se do complemento para 100% do valor normalizado de tempo de execução. Por conta disso, estes podem ser lidos de maneira direta, sendo valores positivos relacionados a ganhos e negativos relacionados a perdas de desempenho.

Já as tabelas 6.6 e 6.7 também oferecem dados para análise do mesmo grupo de experimentos (variação dos parâmetros α e l), porém de outra métrica: o balanceamento de carga. Lembrando, o balanceamento de carga no contexto deste trabalho trata-se da média da porcentagem de tempo que um determinado recurso deixa de receber requisições de execução, ou seja, que esta não apresenta mais atividade. A tabela 6.6 mostra os valores da métrica de balanceamento de carga, ou ociosidade, para os três algoritmos RR, AG e MQD. A métrica ociosidade apresentada neste trabalho, deve ser vista de maneira inver-

[9]sinônimo para variância heterocedástica, ou seja dispersão heterogênea

Parâmetros		Resultados		
α	l	Algoritmo	Média	Desvio Padrão
0,2	0,3	RR	01:23:43,78	22:41,92
		AG	01:18:37,96	14:51,08
		MQD	01:16:55,34	22:31,17
0,5	0,3	RR	01:26:27,85	21:27,75
		AG	01:19:12,29	19:37,97
		MQD	01:16:15,79	17:16,01
0,5	0,5	RR	01:32:11,67	07:52,76
		AG	01:27:20,81	07:28,03
		MQD	01:32:40,21	19:18,34
0,5	0,7	RR	01:28:32,43	10:59,89
		AG	01:20:04,61	05:08,70
		MQD	01:35:48,85	14:10,65
0,8	0,3	RR	01:35:30,68	16:47,68
		AG	01:30:54,72	16:11,86
		MQD	01:31:04,11	17:39,03

Tabela 6.3: Tempo Total de Execução

tida pois quão maior for este valor mais desbalanceado terá sido a distribuição dos jobs sobre os recursos disponíveis.

Na tabela 6.7, é mostrado o teste T-Student para verificar se as médias de ociosidade apresentadas pelos algoritmos AG e MQD são estatisticamente iguais à média do algoritmo RR. Em outras palavras, esta tabela nos apresenta o grau de certeza para o qual podemos afirmar que houve ganho ou perda de balanceamento de carga para os algoritmos AG e MQD. Da mesma maneira que na tabela 6.4, os valores têm que ser lidos de maneira invertida pois quão menor a probabilidade de as amostras possuírem a mesma média, mais certeza teremos que houve ganho ou perda no balanceamento de carga.

Todas estas tabelas apresentadas até aqui terão uma respectiva tabela com dados referentes ao grupo de experimentos sobre reescalonamento, ou dito de outra maneira, da variação de tempo máximo em fila. Estas serão apresentadas mais a frente quando formos analisar os dados referentes aos experimentos com presença de reescalonamento. Por hora, vamos tecer alguns comentários a respeito dos dados apresentados sobre os experimentos onde houve variação dos parâmetros α e l.

Da tabela 6.4, podemos ver que quando $\alpha = 0,5$ temos os resultados mais significa-

Parâmetros		Resultados	
α	l	AG	MQD
0,2	0,3	23,69%	20,83%
0,5	0,3	14,67%	6,55%
0,5	0,5	4,74%	46,53%
0,5	0,7	0,74%	6,19%
0,8	0,3	22,58%	24,29%

Tabela 6.4: Tempo Total de Execução: comparação entre amostras

tivos em termos de distanciamento entre as amostras para o algoritmo de Galstyan em especial. Para o algoritmo MQD, excetuando-se quando o parâmetro de aprendizado l for diferente de 0,5, o mesmo acontece. A princípio, o resultado mais significativo se dá quando $\alpha = 0,5$ e $l = 0,7$. E o segundo resultado mais significativo ocorre quando $\alpha = 0,5$ e $l = 0,3$. Ocorre que esta tabela não nos oferece se esta significância de diferença entre as amostras é algo bom, no sentido de ter encontrado uma otimização, ou ruim, no caso de ter piorado o desempenho em relação a um algoritmo que não possui otimização nenhuma. Para isto, precisamos verificar o que acontece na tabela 6.5.

Da tabela 6.5, temos que deixa de haver otimização no algoritmo MQD quando o parâmetro de aprendizado l é diferente de 0,3. Com isso, já verificamos que o resultado mais significativo de diferença entre as amostras significa uma certeza de piora de execução para o algoritmo MQD. Para o caso do algoritmo de Galstyan, houve otimização, o que indica que um parâmetro de aprendizado que dê grande importância aos resultados mais recentes, como o utilizado, é o recomendável para um algoritmo de heurística gulosa como o AG. Olhando para o segundo resultado mais significativo de diferença entre as amostras, quando $\alpha = 0,5$ e $l = 0,3$, temos otimização em ambos os algoritmos, sendo que em especial, para o algoritmo MQD foi vista nesta situação a sua maior otimização dentre todas as variações de parâmetros realizadas. Isso indica que um parâmetro de aprendizado que dê maior importância ao histórico de execuções, influencia positivamente um algoritmo que utiliza seus recursos computacionais com base em um ordenamento de desempenho, muito possivelmente porque o histórico de execuções é mais rico em informação de desempenho do que o resultado da última execução.

Note também que as demais execuções quando o parâmetro l é igual a 0,3 também possuem otimizações tanto no algoritmo de Galstyan quanto no MQD. Algumas destas

Parâmetros		Resultados		
α	l	Algoritmo	Média	Desvio Padrão
0,2	0,3	AG	4,65%	7,44%
		MQD	8,33%	7,11%
0,5	0,3	AG	8,36%	8,39%
		MQD	11,62%	6,94%
0,5	0,5	AG	5,10%	5,81%
		MQD	-0,49%	18,93%
0,5	0,7	AG	8,03%	9,86%
		MQD	-8,92%	17,28%
0,8	0,3	AG	4,69%	6,98%
		MQD	4,84%	5,42%

Tabela 6.5: Otimização makespan

porém, quando são confrontadas com os desvios padrões coletados para si, sugerem que há casos onde pode haver piora de desempenho, como de fato foi verificado na amostra. Estes não são muito frequentes, porém podem existir.

O resumo geral desta fase dos experimentos, onde foram tentadas algumas combinações diferentes de parâmetros do algoritmo de aprendizado por reforço, é que o parâmetro α ideal para ambos os algoritmos é aquele que dá importância igual entre as componentes voláteis e não-voláteis do tempo total de execução de um job. Pelo lado do parâmetro de aprendizado l, um valor pequeno tende a deixar mais estável a eficiência durante a execução dos jobs, enquanto que valores maiores tornam a medida de eficiência do recurso mais instável, o que afeta os algoritmos de maneira distinta. De uma maneira geral, podemos dizer que o melhor resultado alcançado foi o que possuía $\alpha = 0,5$ e $l = 0,3$ com alta significância de diferença entre as amostras. Estes parâmetros, para os demais experimentos realizados, estarão fixos nestes valores, pois foram os que obtiveram a melhor otimização conjunta dos algoritmos.

Façamos agora uma análise do balanceamento de carga produzido pelos algoritmos. Da tabela 6.7, temos que o resultado mais significativo entre todos os coletados se deu quando $\alpha = 0,8$, ou seja, quando o cálculo da eficiência do recurso foi mais fortemente influenciado pelo componente volátil do tempo total de execução. Perceba, porém que, da tabela 6.4, esta configuração de parâmetros nos fornece um resultado em termos de *makespan* onde as amostras permanecem mais próximas, ou seja, com menor probabil-

idade de ter ocorrido otimização de tempo de execução. Isso nos induz a crer que um melhor balanceamento de carga nesta configuração foi realizado às custas de ganhos em termos de tempo de execução que deixaram de serem realizados.

Este é um resultado bastante curioso pois contradiz a correlação positiva que a princípio deveria existir entre balanceamento de carga e *makespan*, pois imagina-se que quando o tempo total de execução diminui (ou seja, otimização aumenta), o tempo total de atividade dos RMS durante o escalonamento deveria convergir para valores próximos. De fato, isto é verdade para otimizações de tempo total de execução porém, nem sempre será na situação inversa: quando o *makespan* aumentar. Neste caso, este último pode estar aumentando por outras razões e não necessariamente por conta de um desbalanceamento de carga. Como as medidas de balanceamento de carga privilegiam o que acontece na execução dos últimos jobs remanescentes, esta configuração de parâmetros em questão parece ser útil justamente nesta fase de execução, enquanto que nas demais fases (início e meio) a alocação produzida não ajuda muito na economia de tempo total de execução, muito provavelmente por ter pouca consideração com o tempo de execução do job dentro do RMS (ou da máquina pertencente ao RMS).

Outros dois resultados que demonstram significativa diferença entre as amostras, diz respeito à execução do algoritmo MQD quando $\alpha = 0,5$ e l recebeu os valores 0,3 e 0,7. Resta verificar se esta diferença entre as amostras se refere ou não a uma otimização.

Para isso, temos que verificar na tabela 6.6 a ociosidade auferida. Verificando os valores desta tabela para as entradas referidas anteriormente, temos que só houve piora do balanceamento de carga quando tivemos $l = 0,7$, para o algoritmo MQD. Para os demais valores citados, ambos os algoritmos apresentaram ou otimizações de balanceamento de carga, ainda que de pequena ordem, ou simplesmente permaneceram equivalentes, em se considerando o desvio padrão da amostra. Lembrando que a análise de tempo de execução do algoritmo MQD para este conjunto de parâmetros já havia evidenciado uma piora significativa, tal como podemos verificar agora na métrica de balanceamento de carga, podemos concluir que, para este algoritmo, o ideal é ter uma parâmetro de aprendizado l pequeno.

Isso parece estar relacionado com o impacto que este parâmetro l tem, de ordem direta, sobre as variações sofridas pela medida de eficiência dos recursos computacionais. Valores grandes de l determinam maiores variações da eficiência medida do recurso computacional, o que, se por um lado possibilita captar mais rapidamente variações de desempenho destes recursos em espaços de tempo mais curtos, por outro torna a sua classificação

Parâmetros		Resultados		
α	l	Algoritmo	Média	Desvio Padrão
0,2	0,3	RR	18,65%	10,31%
		AG	17,80%	9,25%
		MQD	16,01%	13,94%
0,5	0,3	RR	15,41%	5,30%
		AG	14,84%	4,58%
		MQD	11,21%	6,41%
0,5	0,5	RR	23,72%	4,61%
		AG	21,42%	5,24%
		MQD	24,05%	11,77%
0,5	0,7	RR	24,73%	4,98%
		AG	23,12%	4,58%
		MQD	32,44%	9,33%
0,8	0,3	RR	15,92%	4,10%
		AG	13,00%	4,80%
		MQD	12,81%	4,17%

Tabela 6.6: Ociosidade

mais instável também. Acreditamos que, muito por conta deste fato, os resultados do algoritmo classificatório (MQD) tenham sido tão significativamente ruins, enquanto que para um algoritmo guloso (AG), se verifique grande significância de otimização.

Sobre o ajuste do parâmetro α para 0,5 como o melhor resultado agregado para ambos os algoritmos, podemos dizer que ambas as componentes de tempo total de execução dos jobs (componente volátil e não-volátil) tem importância similar para a avaliação da eficiência de um recurso computacional. Em um ambiente diferente de um grid, ou seja, sem os complicadores que este ambiente traz para a execução das aplicações, como por exemplo, latência na rede de comunicação, ou tempo em fila de execução, possivelmente a componente não-volátil do tempo de execução total do job teria uma importância maior que as componentes voláteis deste. Ocorre que esta última está presente neste tipo de ambiente, e o ajuste do parâmetro em questão mostra que estes fatores têm grande importância para se tentar minimizar o tempo total de escalonamento dos jobs.

Completada a análise desta fase inicial dos experimentos, onde foram feitos os ajustes de parâmetros do algoritmo de aprendizado por reforço, podemos realizar uma mudança no ambiente de execução para responder a duas possíveis perguntas. Que acontece com

Parâmetros		Resultados	
α	l	AG	MQD
0,2	0,3	40,60%	27,96%
0,5	0,3	37,30%	2,62%
0,5	0,5	10,65%	46,02%
0,5	0,7	17,39%	0,39%
0,8	0,3	4,24%	2,45%

Tabela 6.7: Ociosidade: comparação entre amostras

estes algoritmos na presença de reescalonamento ? E quando dobramos a carga de trabalho, há mudança significativa de desempenho ? Para isso, introduziremos o reescalonamento por tempo máximo em fila como um fator novo para analisar o desempenho dos algoritmos.

Serão analisados três cenários de execução com reescalonamento: com tempo máximo em fila igual a 500, 750 e 1000 segundos. Estes valores se referem à uma aproximação da média de tempo de espera em fila (750 segundos) verificada até então, somada e depois diminuída de um desvio padrão (250 segundos) relativo a esta mesma média. Conjuntamente, será realizado também o aumento de carga de jobs, dobrando-o de 500 para 1000, em um cenário específico de reescalonamento, com tempo máximo em fila de 750 segundos. Lembro aqui que o reescalonamento também estará presente na execução do algoritmo RR, assim como em AG e MQD.

A apresentação das tabelas aqui é similar à que foi feita para o grupo de experimentos sobre variação de parâmetros. Primeiramente, será apresentada a tabela 6.8 que dispõe apenas sobre a média e o desvio padrão do tempo de execução dos algoritmos de escalonamento. Pelo mesmo motivo anteriormente citado para a tabela 6.3, esta é uma tabela apenas ilustrativa e as demais que seguem é que nos fornecerão dados para análise.

A tabela 6.9 nos fornece uma idéia da otimização obtida pelos algoritmos de Galstyan e MQD durante os experimentos com reescalonamento. Os valores ali presentes se referem ao complemento em 100% do tempo de execução normalizado dos algoritmos, e por isso pode ser lida de maneira direta, relacionando otimizações à valores positivos. Junto com esta tabela, deve-se verificar na tabela 6.10 o quão significativamente diferentes são as amostras dos algoritmos de Galstyan e MQD em relação ao algoritmo RR, pois isto serve como medida de certeza sobre a diferença de comportamento entre os algoritmos. Nesta

Parâmetros		Resultados		
Jobs	Tempo em Fila (s)	Algoritmo	Média	Desvio Padrão
500	500	RR	01:21:59,78	08:30,91
		AG	01:16:48,74	17:17,26
		MQD	01:12:05,36	07:50,82
500	750	RR	01:19:46,04	11:50,14
		AG	01:16:24,47	09:03,81
		MQD	01:12:04,76	09:38,06
1000	750	RR	02:12:30,26	12:11,14
		AG	02:06:40,72	09:37,55
		MQD	02:06:07,04	12:52,83
500	1000	RR	01:23:50,28	07:40,72
		AG	01:19:11,02	08:00,07
		MQD	01:15:48,43	10:00,93

Tabela 6.8: Reescalonamento - Tempo Total de Execução

última, trata-se de uma probabilidade calculada pela distribuição T-Student sobre se as médias calculadas pertencem à mesma amostra. Por isso, seus valores devem ser lidos de maneira inversa, uma vez que pequenos valores desta probabilidade significam um grau de certeza maior de que ambas as amostras tiveram comportamentos diferentes.

Sendo assim, da tabela 6.10 temos que o algoritmo MQD obteve um resultado melhor que o algoritmo de Galstyan, pois foi significativamente diferente do algoritmo RR, enquanto que o algoritmo de Galstyan teve maior probabilidade de pertencer a mesma média do algoritmo RR. Percebemos também que o tempo máximo de espera em fila menor possível (500 segundos) obteve os melhores resultados tanto em otimização de tempo de execução do algoritmo quanto no teste de significância da diferença entre as amostras. Ou seja, quando o reescalonamento induz a um maior número de eventos de escolha de recursos computacionais, os algoritmos de Galstyan e MQD revelam possuir melhores critérios de escolha, em especial o MQD. Ainda sobre o comportamento dos algoritmos em relação ao seu tempo de execução, podemos verificar que estes não se alteraram quando foi dobrada a carga de jobs, permanecendo semelhantes os ganhos de tempo de execução, ainda que o algoritmo de Galstyan mostre uma pequena melhora na execução de 1000 jobs.

Sobre o balanceamento de carga nessa outra fase dos experimentos, o padrão acima descrito se repete. As tabelas 6.11 e 6.12 nos mostram respectivamente os valores da

Parâmetros		Resultados		
α	1	Algoritmo	Média	Desvio Padrão
500	500	AG	6,72%	15,18%
		MQD	11,56%	10,15%
500	750	AG	1,86%	6,16%
		MQD	6,98%	4,85%
1000	750	AG	4,02%	7,09%
		MQD	4,75%	5,66%
500	1000	AG	5,46%	5,49%
		MQD	9,53%	8,75%

Tabela 6.9: Reescalonamento - Otimização makespan

Parâmetros		Resultados	
Jobs	Tempo em Fila (s)	AG	MQD
500	500	15,48%	0,13%
500	750	23,16%	8,24%
1000	750	7,90%	8,70%
500	1000	21,99%	0,52%

Tabela 6.10: Reescalonamento - Tempo Total de Execução: comparação entre amostras

Parâmetros		Resultados		
Jobs	Tempo em Fila (s)	Algoritmo	Média	Desvio Padrão
500	500	RR	33,74%	11,15%
		AG	12,71%	6,00%
		MQD	11,35%	8,06%
500	750	RR	12,08%	2,80%
		AG	9,48%	4,08%
		MQD	6,44%	2,09%
1000	750	RR	9,79%	10,41%
		AG	7,38%	2,69%
		MQD	3,86%	2,00%
500	1000	RR	14,41%	11,35%
		AG	8,74%	4,69%
		MQD	5,79%	5,79%

Tabela 6.11: Reescalonamento - Ociosidade

métrica de ociosidade para os experimentos e da semelhança entre as amostras colhidas em relação ao algoritmo RR. Da tabela 6.12, temos novamente que o algoritmo MQD mostra significativa diferença em relação ao algoritmo RR. Já o algoritmo de Galstyan tem comportamento semelhante apenas quando o tempo máximo em fila é de 500 segundos, enquanto que nas demais ocasiões possui uma probabilidade de cerca de 20% de pertencer a mesma amostra que das execuções de RR. A otimização obtida é fornecida pela tabela 6.11, a qual apresenta uma otimização razoavelmente grande de ociosidade quando o tempo máximo de execução em fila é de 500 segundos. Para os demais valores há também otimização de balanceamento de carga, mas de uma ordem menor. Novamente, quando foi dobrada a carga de jobs para o escalonador a otimização se mantém, havendo até uma pequena melhora pois os índices de ociosidade têm ligeira queda.

Aqui podemos verificar um comportamento interessante. A otimização em ambos os algoritmos, quando o tempo de espera em fila foi especificado em 750 segundos, embora seja percentualmente pequena, esta é bastante significativa, como podemos ver na tabela 6.10. Para o algoritmo MQD em particular, com tempo de espera em fila iguais a 500 e 750 segundos, os valores desta tabela mostram que a mudança de comportamento sobre balanceamento de carga efetivamente ocorreu.

Terminada a mostra das estatísticas colhidas referentes aos algoritmos utilizados, resta apenas comentar a tabela 6.13, que fornece a média e o desvio padrão no número de

Parâmetros		Resultados	
Jobs	Tempo em Fila (s)	AG	MQD
500	500	0,00%	0,00%
500	750	18,29%	0,24%
1000	750	19,96%	2,34%
500	1000	22,08%	11,84%

Tabela 6.12: Reescalonamento - Ociosidade: comparação entre amostras

reescalonamentos realizados por cada algoritmo. Estes são aqui mostrados para se averiguar o perfil dos escalonadores. Da tabela, temos que não há diferença significativa para o número de reescalonamentos quando o tempo máximo em fila é igual ou superior a 750 segundos. Quando este é igual a 500 segundos verifica-se que tanto o algoritmo de Galstyan quanto o MQD apresentam números maiores de reescalonamentos feitos que o algoritmo RR.

Isto muito se deve ao comportamento dos algoritmos na escolha de seus recursos computacionais. No algoritmo RR, os recursos computacionais são escolhidos de maneira aleatória, sendo que a chance de um job ser enviado a um recurso computacional é igual entre todos. Nos demais algoritmos, ocorre uma escolha não aleatória para os jobs, o que torna o reescalonamento em muitos momentos menos eficaz pois a probabilidade de um job reescalonado ser novamente escalonado para o mesmo recurso computacional aumenta. Por isso, o aumento significativo no número de reescalonamentos. Outro ponto interessante a se notar foi que houve pouca ou quase nenhuma variação no número de reescalonamentos realizado quando a carga de jobs dobrou. Este fato é curioso, pois a princípio seria de se supor que o aumento do número de jobs enviados aos RMS aumentaria o número de reescalonamentos. Também não possui relação explícita com os algoritmos, até porque foi um fenômeno verificado igualmente entre os três algoritmos. Provavelmente, deve estar relacionado a alguma mudança ocorrida no ambiente computacional durante a execução destes testes.

Parâmetros		Resultados		
Jobs	Tempo em Fila (s)	Algoritmo	Média	Desvio Padrão
500	500	RR	79,8	21,91
		AG	94,47	17,88
		MQD	104	16,59
500	750	RR	56,6	16,18
		AG	54,73	12,86
		MQD	50,07	17,7
1000	750	RR	57,4	19,7
		AG	61,2	20,58
		MQD	54,8	22,28
500	1000	RR	21,47	16,03
		AG	19,67	13,67
		MQD	22,4	14,81

Tabela 6.13: Reescalonamento - Total Reescalonado

Capítulo 7

Conclusões e Trabalhos Futuros

Aqui seguem descritas as principais conclusões a que este trabalho se reporta. Uma primeira conclusão bastante clara é que em ambos os casos, seja na execução do algoritmo de escalonamento de Galstyan ou na execução do MQD, houve ganho tanto em termos de otimização no tempo total de execução quanto no próprio balanceamento de carga do sistema. Obviamente que este ganho depende do ajuste de parâmetros do algoritmo que calcula a eficiência do recurso computacional. É importante mencionar também que ganhos ocorridos em termos de *makespan* tendem a se reproduzir como ganhos de balanceamento de carga também.

De fato, há apenas um caso onde claramente houve uma piora no desempenho do algoritmo de escalonamento que foi quando utilizamos $l = 0,7$ para o algoritmo MQD. Por outro lado, houve claramente uma otimização quando $\alpha = 0,5$ e $l = 0,3$ no algoritmo MQD. Esta diga-se de passagem foi até ligeiramente superior a esperada, pois o trabalho onde foram realizadas as simulações com o algoritmo MQD dão notícia de um ganho da ordem de 3 a 10% de economia de tempo total de execução, segundo [18]. Possivelmente, esta ordem de ganho obtida nas simulações estaria correlacionada a um conjunto de valores dos parâmetros α e l não executados durante nossos experimentos.

Ambos os algoritmos também podem obter otimizações no balanceamento de carga do sistema caso seja utilizado o reescalonamento por tempo máximo de espera em fila. Este, por sinal, não parece ter um efeito negativo no desempenho de ambos os algoritmos, mantendo-se a otimização de ambos os algoritmos ainda que possivelmente numa escala menor. Não há dados para comparação de experimentos teóricos com balanceamento de carga no algoritmo MQD e por isso pouco pode ser dito sobre este em termos de validação de seu desempenho.

Sobre as simulações com o algoritmo de Galstyan, os dados tanto sobre balanceamento de carga quanto sobre tempo médio de espera foram de ganhos bastante significativos, em torno de 40%. Nos experimentos realizados, os que mais próximos estiveram deste desempenho foram os experimentos com $\alpha = 0,8$, ou na presença de re-escalonamento, que parece ter o efeito de dispersar mais o comportamento entre os algoritmos RR e AG ou MQD. O fato de os RMS nos experimentos terem um limite de até 20 jobs ativos em cada um, pode ter sido uma restrição que tenha efeito inibir o desempenho do balanceamento de carga do algoritmo de Galstyan. Isto porque não há relato de tal tipo de restrição na simulação realizada. E no caso dos experimentos, este limite máximo frequentemente foi atingido e com isso, era necessário esperar a finalização de alguns jobs de um RMS para se poder voltar a submeter jobs a este.

Este trabalho obviamente não esgota a possibilidades de pesquisa na área, considerando esta os algoritmos de escalonamento de Galstyan e MQD. Cito aqui portanto alguns trabalhos possíveis como continuação deste:

- executar outros tipos de aplicação com outros perfis que sugiram comportamento pouco previsível ou probabilístico de execução.
- executar outras aplicações em que a quantidade de dados transferido seja relevante, ou seja, em que os arquivos de entrada ou saída sejam grandes o suficiente para realizarem atrasos significativos.
- investigar como outras heurísticas de escalonamento frequentemente citadas como min-min, max-min ou suffrage por exemplo, poderiam fazer uso da medição ou ranqueamento realizado pela eficiência medida dos recursos computacionais.
- investigar se o tempo gasto em transferência de arquivos poderia ser considerado um componente não-volátil do tempo de execução do job, tanto em ambientes de rotas estáticas quanto dinâmicas.
- executar experimentos onde a execução de metaescalonadores seja sobreposta, fazendo estes concorrerem em uso de um mesmo conjunto de recursos computacionais, para verificar se a submissão consegue ser coordenada implicitamente, ou seja, sem comunicação explícita, tal como foi o caso do artigo original do AG, conforme [14].

- aumentar a escala de execução dos jobs ao realizar testes de execução com carga de jobs acima de 1000.
- investigar como incluir a duplicação de jobs, tal como feito no algoritmo MQD original, assim como investigar o impacto desta em termos de *makespan* e balanceamento de carga, tanto para o algoritmo MQD, quanto para o algoritmo AG.

Bibliografia

[1] BOERES, C., FONSECA, A. A., DE AMORIM MENDES, H., *et al.*, "An EasyGrid portal for scheduling system-aware applications on computational Grids", *Concurrency and Computation: Practice and Experience*, v. 18, n. 6, pp. 553–566, 2006.

[2] BOSE, A., WICKMAN, B., WOOD, C., "MARS: A Metascheduler for Distributed Resources in Campus Grids", *grid*, v. 00, pp. 110–118, 2004.

[3] BUYYA, R., ABRAMSON, D., GIDDY, J., "Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid", *hpc*, v. 01, p. 283, 2000.

[4] CASANOVA, H., "Simgrid: A toolkit for the simulation of application scheduling", In: *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, (Washington, DC, USA), p. 430, IEEE Computer Society, 2001.

[5] CASANOVA, H., OBERTELLI, G., BERMAN, F., *et al.*, "The apples parameter sweep template: user-level middleware for the grid", In: *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, (Washington, DC, USA), p. 60, IEEE Computer Society, 2000.

[6] CASANOVA, H., ZAGORODNOV, D., BERMAN, F., *et al.*, "Heuristics for scheduling parameter sweep applications in grid environments", In: *HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop*, (Washington, DC, USA), p. 349, IEEE Computer Society, 2000.

[7] CHULIANG WENG, M. L., LU, X., "An Online Scheduling Algorithm for Assigning Jobs in the Computational Grid", *IEICE Transactions on Information and Systems*, v. E89-D, pp. 597–604, Feb 2006.

[8] CIRNE, W., BRASILEIRO, F., COSTA, L., *et al.*, "Scheduling in bag-of-task grids: The pauá case", In: *SBAC-PAD '04: Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'04)*, (Washington, DC, USA), pp. 124–131, IEEE Computer Society, 2004.

[9] CLAUS, C., BOUTILIER, C., "The dynamics of reinforcement learning in cooperative multiagent systems", In: *AAAI '98/IAAI '98: Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, (Menlo Park, CA, USA), pp. 746–752, American Association for Artificial Intelligence, 1998.

[10] DONG, F., AKL, S., "Scheduling algorithms for grid computing: State of the art and open problems", Tech. Rep. 2006-504, School of Computing, Queen's University, Jan 2005. Technical Report.

[11] E. HUEDO, R. M., LLORENTE, I., "The GridWay Framework for Adaptive Scheduling and Execution on Grids", *Scalable Computing - Practice and Experience*, v. 6, pp. 1–8, Sep 2005.

[12] EL-GHAZAWI, T., GAJ, K., ALEXANDINIS, N., *et al.*, "Conceptual comparative study of job management systems", tech. rep., George Mason University, Fev 2001. Technical Report.

[13] FOSTER, I., KESSELMAN, C., TUECKE, S., "The Anatomy of the Grid: Enabling Scalable Virtual Organizations", *The International Journal of High Performance Computing Applications*, v. 15, pp. 200–222, Fall 2001.

[14] GALSTYAN, A., CZAJKOWSKI, K., LERMAN, K., "Resource allocation in the grid using reinforcement learning", In: *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, (Washington, DC, USA), pp. 1314–1315, IEEE Computer Society, 2004.

[15] GRIDWISETECH, "Grid brokers and metaschedulers: Market overview". http://www.gridwisetech.pl/resources/reports/report_2006_grid_brokers.pdf, Fevereiro 2006.

[16] KAELBLING, L. P., LITTMAN, M. L., MOORE, A. P., "Reinforcement Learning: A Survey", *Journal of Artificial Intelligence Research*, v. 4, pp. 237–285, 1996.

[17] KRAUTER, K., BUYYA, R., MAHESWARAN, M., "A taxonomy and survey of grid resource management systems for distributed computing", *Software Practice and Experience*, v. 32, n. 2, pp. 135–164, 2002.

[18] LEE, Y. C., ZOMAYA, A. Y., "A Grid Scheduling Algorithm for Bag-of-Tasks Applications Using Multiple Queues with Duplication", *icis-comsar*, v. 0, pp. 5–10, 2006.

[19] NADIMINTI, K., VENUGOPAL, S., GIBBINS, H., *et al.*, "The gridbus grid service broker and scheduler (2.4.4) user guide". http://www.gridbus.org/broker/2.4.4/manualv2.4.4.pdf, Agosto 2007.

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

[21] NéMETH, Z. N., SUNDERAM, V., "A formal framework for defining grid systems", In: *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, (Washington, DC, USA), p. 202, IEEE Computer Society, 2002.

[22] PRAHL, S., "Tiny monte carlo". http://omlc.ogi.edu/software/mc/tiny_mc.c, Agosto 2007.

[23] SUTTON, R., BARTO, A., *Reinforcement Learning: An Introduction*. Cambridge, MA, MIT Press, 1998.

[24] TEAM, T. G., "The globus alliance". http://www.globus.org/, Agosto 2007.

[25] VENUGOPAL, S., BUYYA, R., WINTON, L., "A grid service broker for scheduling distributed data-oriented applications on global grids", In: *MGC '04: Proceedings of the 2nd workshop on Middleware for grid computing*, (New York, NY, USA), pp. 75–80, ACM Press, 2004.

[26] VENUGOPAL, S., BUYYA, R., WINTON, L., "A Grid service broker for scheduling e-Science applications on global data Grids: Research Articles", *Concurr. Comput. : Pract. Exper.*, v. 18, n. 6, pp. 685–699, 2006.

[27] WATKINS, C. J. C. H., DAYAN, P., "Technical Note: Q-Learning", *Mach. Learn.*, v. 8, n. 3-4, pp. 279–292, 1992.

[28] WENG, C., LU, X., "Heuristic scheduling for bag-of-tasks applications in combination with QoS in the computational grid", *Future Gener. Comput. Syst.*, v. 21, n. 2, pp. 271–280, 2005.