UMA FERRAMENTA PARA GERENCIAMENTO AUTOMÁTICO DE TAREFAS EM UM AMBIENTE DE GRADE

José Afonso Lajas Sanches

TESE 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. Inês de Castro Dutra, Ph.D.

Prof. Vítor Manuel de Morais Santos Costa, Ph.D.

Prof. Felipe Maia Galvão França, Ph.D.

Prof. Maria Clicia Stelling de Castro, D.Sc.

Prof. Mario Vaz da Silva Filho, D.Sc.

RIO DE JANEIRO, RJ - BRASIL

MARÇO DE 2004

SANCHES, JOSÉ AFONSO LAJAS

Uma ferramenta para gerenciamento automático de tarefas em um ambiente de grade [Rio de Janeiro] 2004

X, 57 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2004)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1 - Computação em grade (*Grid Computing*)

2 - Gerenciamento automático de tarefas

I. COPPE/UFRJ II. Título (série)

À minha família, amigos e namorada

Agradecimentos

Agradeço a Deus Pai pela sua misericórdia e força, a Nosso Senhor Jesus Cristo pela sua palavra e sacrifício e ao Divino Espírito Santo pela sua unção - este trabalho é a eles consagrado, e sem estas graças não poderia ser realizado.

Agradeço a Nossa Senhora por rogar sempre por nós, seus filhos adotivos.

Agradeço a meus imensamente queridos pais, José e Beatriz, sem o amor, carinho e apoio dos quais eu não teria chegado a lugar nenhum, e a meus irmãos, Heloisa e Ricardo. Vocês são as pessoas mais importantes da minha vida!

Agradeço à minha querida Julia, pelo seu amor e compreensão - e por ser o que é.

Agradeço a meus orientadores, Inês e Vítor, por sua paciência, dedicação e incentivo constante. Vocês são pessoas muito especiais e uma fonte de inspiração eterna.

Agradeço ao meu orientador de Projeto Final de graduação, Alexandre Lucas, por ter me incentivado a cursar o mestrado; aos professores da Universidade Católica de Petrópolis José Carlos Tavares, José Augusto Cunha e José Helayël pelas cartas de recomendação para a Coppe.

Agradeço a meus amigos, companheiros de luta (e festa) na Coppe: Ana Paula, Anderson, André, Cláudio, Daniele, Eduardo, Luciana Campos, Luciana Itida, Luiz Otávio, Luís Rodrigo, Marluce, Newton, Patrícia Kayser, Paula, Ricardo, Rodrigo, Sergio e Tatiana - vocês foram, e são, fabulosos! A meus amigos que, de uma forma ou de outra, me deram força durante este trabalho: Carmen, Cristiano, Flávia, Francisco de Assis, Fábio, Gustavus, Marcelo, Patrícia, Renata e Victor; ao Marco da padaria pelo suporte e fornecimento de hardware. Muito obrigado pela existência de vocês.

Agradeço por último, porém com a mesma importância, à equipe de apoio e professores da Coppe/Sistemas, pelo bom trabalho e conhecimento ministrado.

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

UMA FERRAMENTA PARA GERENCIAMENTO DE TAREFAS EM UM AMBIENTE DE GRADE

José Afonso Lajas Sanches

Março/2004

Orientadores: Inês de Castro Dutra
Vítor Manuel de Morais Santos Costa
Programa: Engenharia de Sistemas e Computação

Certas atividades computacionais para a resolução de problemas podem ser intensivas, devido ao número de tarefas a serem executadas e dados a serem analisados pelas mesmas. Então, o paralelismo precisa ser usado, para um desempenho razoável e mesmo para tornar a solução daqueles problemas viável. Um ambiente em grade, devido ao seu potencial, se apresenta como uma boa alternativa.

Entretanto, fatores como geração de saídas erradas ou interrupção de tarefas em execução podem ocorrer, e torna-se inviável para o usuário lidar com eles quando há muitas tarefas em execução.

Neste trabalho, resolvemos parte destes problemas oferecendo ao usuário uma ferramenta para gerenciamento automático de tarefas em um ambiente de grade (*Grid*).

Nossos resultados mostraram que o gerenciamento automático de um grande número de tarefas em ambientes de grade é importante e permite ao usuário se concentrar em outras tarefas, tais como preparação e análise de dados, deixando o gerenciamento dos experimentos a cargo do sistema. Também procuramos analisar alguns pontos a serem desenvolvidos neste sentido de automatizar o gerenciamento de tarefas.

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

A TOOL FOR MANAGING THE EXECUTION OF JOBS IN A GRID ENVIRONMENT

José Afonso Lajas Sanches

March/2004

Advisors: Inês de Castro Dutra

Vítor Manuel de Morais Santos Costa

Department: Computing and Systems Engineering

The aim of this work was to construct and evaluate a tool for managing the execution of jobs in a grid environment.

Some computational activities can be very intensive, because they may spread a huge number of jobs to execute, and that may generate a huge amount of data to be analyzed. To reach a reasonable performance and even be able to solve some problems, parallelism must be used. A grid environment is known to be a good alternative, due to its potential.

However, some factors like the production of corrupted data or task interruption, due to some failure in the system, can occur, and it is very inconvenient to handle them manually, specially in the context of a huge number of jobs.

Our results showed that the automatic managing of a huge number of tasks in grid environments is worth, avoiding any kind of user intervention. We also tried to show some points to be still developed towards task managing automatization.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Este trabalho teve por objetivo a construção e avaliação de uma ferramenta para gerenciamento automático de tarefas em um ambiente de grade (*Grid*).

Atividades computacionais para a resolução de problemas em determinadas áreas do conhecimento - como experimentos em Física ou Inteligência Artificial - podem exigir grande quantidade de trabalho, devido ao elevado número de tarefas a serem executadas e à grande quantidade de dados a serem analisados. Exemplos destas atividades são aplicações em Aprendizado de Máquina (*Machine Learning*), que freqüentemente precisam repetir experimentos sobre subconjuntos de dados distintos. Se o tamanho do problema for aumentado, o tamanho deste conjunto de dados crescerá rapidamente, gerando um número massivo de experimentos, e de tarefas computacionais ou *jobs* - freqüentemente com altos tempos de execução - para resolvê-los.

Então, qual seria o resultado se tentássemos executar estas tarefas em um ambiente monoprocessado? Naturalmente, experimentos com estas características não estariam prontos antes de alguns meses ou mesmo anos, tornando inviável a sua realização. Portanto, o *paralelismo* precisa ser empregado nestes casos.

Para lidar com eles, um *ambiente de grade* se apresenta como uma boa alternativa, por possuir grande poder computacional. Ian Foster e Carl Kesselman definem uma grade como "uma infraestrutura de hardware e software que fornece acesso consistente, fidedigno, amplo e barato a recursos computacionais de topo de linha (*high-end*)"[17].

Ambientes de grade introduzem novos desafios de *software* e *hardware*, pois os recursos numa grade estão fora dos recursos da rede local. Logo, os recursos podem estar sujeitos a tarifação, autorização ou velocidade de *links* de comunicação. Este

fator, aliado à quantidade elevada de tarefas disparadas pelas aplicações que rodam em ambientes de grade, podem provocar perda de tarefas. Além disso, devido à heterogeneidade própria dos ambientes de grade, as tarefas podem sofrer uma interrupção repentina de sua execução ou geração de saídas erradas. Uma vez que as tarefas são numerosas, torna-se extremamente difícil e demorado lidar manualmente com estes incidentes, através da resubmissão manual de todas as tarefas corrompidas.

Alguns trabalhos prévios tentam resolver um outro problema, que é o das *dependências* entre as tarefas, assim como o Chimera [18], ou o Condor DAGman [31], que foram delineados no sentido de gerenciar essas dependências de forma automática.

No entanto, os escalonadores de recursos para grades não possuem suporte para a resubmissão de tarefas corrompidas, deixando o serviço para o usuário. A observação deste fato levou Dutra *et al* [13] a construírem um protótipo, que consistia basicamente em: (1) um programa que preparava automaticamente dados de entrada para uso das tarefas a serem executadas, e (2) um *daemon* verificador de jobs falhos.

Este protótipo serviu como base para a construção desta ferramenta, cujas principais características são:

- Uma interface Web para a interação do usuário com o ambiente Condor de execução, e acompanhamento da execução das tarefas;
- Preparação dos dados de entrada para uso das tarefas, que serão a seguir submetidas a um ambiente de escalonamento numa grade, e para isto foi escolhido o escalonador Condor [22, 31];
- Monitoramento de tarefas, com a verificação das tarefas falhas e sua resubmissão com base em erros acontecidos e em tempo de execução excessivo;
- Detecção automática de dependências entre jobs.

A ferramenta foi construída no contexto de experimentos em Aprendizado de Máquina, descritos em mais detalhes na Seção 4.2.

Com relação a este texto, sua estrutura está dividida em seis capítulos. Este primeiro capítulo tem a função de fornecer uma visão geral do trabalho, seu objetivo e os fatores que o motivaram. O segundo capítulo é dedicado aos conceitos básicos e aspectos gerais da computação em Grade.

O terceiro capítulo trata de descrever as aplicações para grades utilizadas como fundamento deste trabalho. No quarto capítulo, as origens, o propósito e a implementação física de nossa ferramenta são apresentados e detalhados. O quinto capítulo descreve algumas análises e séries de experimentos realizadas para verificar a eficácia da ferramenta e do sistema escalonador usado.

Por fim, o sexto capítulo disserta sobre as conclusões obtidas a partir da realização deste trabalho, e os trabalhos futuros passíveis de serem pesquisados.

Capítulo 2

Computação em Grade (*Grid Computing*)

2.1 Filosofia, Motivações e Definição

Antes de abordarmos em detalhes o conceito e potencialidades da *Computação em Grade*, é conveniente voltarmos um pouco na história. Onde estão as origens de sua filosofia?

Ian Foster e Carl Kesselman [16], estabelecem um paralelo entre o desenvolvimento da computação hoje e o desenvolvimento das redes de distribuição de eletricidade por volta de 1910. Nesta época, já era possível a geração de energia elétrica em escala considerável, de modo que foram construídos vários dispositivos que funcionavam alimentados pela nova energia. Porém, como torná-la democratizada, se nem todos poderiam dispor de um gerador ou coisa parecida?

Era necessário que o novo recurso fosse acessível a todos, e isto foi conseguido com a criação das *malhas ou grades de energia elétrica* (*Eletric Power Grids*) e as tecnologias de transmissão e distribuição a elas associadas. Na opinião dos autores, as malhas de energia foram a verdadeira revolução na história da humanidade - e não o advento da eletricidade por si mesmo. Afinal, disponibilizando energia barata, "limpa" e abundante, elas tornaram possível não somente o desenvolvimento de outros dispositivos (tais como eletrodomésticos), como também o surgimento de um sem-número de indústrias, deflagrando transformações econômicas e sociais sem precedentes.

Então, por que não expandir os poderes computacionais a muitos, de forma barata, abundante e "limpa", como acontece com a energia elétrica?

Foi exatamente isto que projetistas americanos tiveram em mente, na segunda

metade dos anos 60. Em 1965, o MIT, o Bell Labs e a General Electric se decidiram por trabalhar em um *computador utilitário*, isto é, um *mainframe* de alto poder computacional (o GE-645) que pudesse suportar o atendimento a centenas de usuários simultâneos. Com inspiração no modelo de malhas de energia elétrica, foi projetado um sistema operacional para exercer o controle sobre tal máquina, que foi batizado de MULTICS [7].

Infelizmente, o projeto MULTICS redundou num fracasso comercial. Os parâmetros tomados como base no projeto mudavam constantemente com a evolução da tecnologia e das técnicas de integração de circuitos, segundo [34]. O resultado foi que em 1969 o MULTICS tinha se transformado em um sistema imenso e cheio de labirintos, que mal podia suportar um punhado de usuários.

Um fator decisivo para o posterior nascimento das grades foi a popularizacão, a partir da segunda metade dos anos 80, dos *sistemas distribuídos*, que são definidos como sistemas com múltiplos processadores onde cada processador compõe uma unidade independente, com recursos próprios, e essas unidades são ligadas por redes de interconexão. Os avanços tecnológicos que permitiram a criação deste tipo de ambiente multiprocessado foram dois, de acordo com [34]:

1. o desenvolvimento de *hardware* com velocidade cada vez maior e cada vez mais barato; e

2. o advento das redes locais de alta velocidade (LANs).

O produto destes avanços foi a possibilidade de conseguir grande poder computacional - através da reunião de várias máquinas - a um preço razoável, o que seria inviável alguns anos antes. As redes de alta velocidade viabilizaram a comunicação rápida entre máquinas, pois uma comunicação lenta é prejudicial ao processamento compartilhado entre elas.

A Seção 2.1.1 fornece mais detalhes sobre os sistemas distribuídos, comparando-os em termos de estrutura com as grades computacionais.

Não obstante o barateamento contínuo dos componentes de computadores e criação de estruturas de rede cada vez mais eficientes, nos dias de hoje ainda são poucas as pessoas - físicas ou jurídicas - que dispõem de recursos computacionais em larga escala, num quadro bastante semelhante ao que se via para a energia elétrica em 1910.

De forma análoga às grades de energia elétrica, Foster e Kesselman definem uma Grade Computacional (*Computational Grid*) como sendo "uma infraestrutura de hardware e software que fornece acesso consistente, fidedigno, amplo e barato a recursos computacionais de topo de linha (*high-end*)"[17].

Em seguida, detalhemos um pouco mais os termos-chave que compõem a definição de Foster e Kesselman:

Infraestrutura Uma grade computacional está diretamente relacionada, acima de tudo, com uma reunião em grande quantidade de recursos (ciclos de CPU, dados, sensores...), que necessitam de hardware suficiente para conseguir as interconexões necessárias, e de software competente para controlar e monitorar o conjunto de recursos.

Fidedigno Os usuários precisam ter a garantia de que conseguirão em suas aplicações níveis consideráveis ou mesmo altos de desempenho, ao usarem uma grade. Na falta desta garantia, será um desperdício executar e até construir dada aplicação.

Consistente Faz-se necessário o uso de serviços padronizados, acessados por interfaces padronizadas e que operem de acordo com parâmetros padronizados, à feição da energia elétrica. A construção de aplicações e um uso amplo serão impraticáveis caso não se cumpram tais parâmetros.

Amplo Esta qualidade se refere a poder contar com serviços que estão sempre disponíveis, não importa o ambiente no qual se esteja. Não é que os serviços se encontrem em todos os lugares, ou sejam acessados de qualquer lugar; a idéia é a de possuir acesso universal *dentro* dos confins de um único ambiente de grade.

Barato Finalmente, um acesso barato aos recursos de uma grade deve ser fornecido. Isto é essencial, se considerarmos que as pessoas que trabalham com grades computacionais desejam torná-la largamente aceita, e utilizada.

Um dos projetos mais considerados na área de computação em grade é o projeto Globus [2, 15], que conduz pesquisa e desenvolvimento para criar tecnologias fundamentais para grades. Este projeto é mencionado com mais detalhes na Subseção 2.3.2.1.

2.2 Grades x Sistemas distribuídos tradicionais

Segundo Németh e Sunderam [25], as grades são consideradas descendentes mais poderosos e sofisticados dos *sistemas distribuídos* comuns.

Um *sistema distribuído* pode ser definido como um ambiente multiprocessado onde cada processador compõe uma unidade (*nó*) independente, com uma memória privativa, relógio (clock) e outros recursos (por exemplo, dispositivos de I/O) próprios. Cada um destes nós pode ser um PC, uma estação de trabalho (*work-station*) ou até mesmo um supercomputador. A separação física entre os *nós* é normalmente bem nítida, e eles se comunicam por meio de redes de interconexão. Por esta razão, os sistemas distribuídos são denominados *fracamente acoplados*, em contrapartida aos sistemas onde vários processadores compartilham memória e possuem relógio global (ou *fortemente acoplados*)[27].

Silberschatz *et al* [27] identifica quatro razões básicas para o uso de computação distribuída:

- *Compartilhamento de Recursos* - Uma vez que os nós num ambiente distribuído estão conectados um ao outro, um usuário locado em um nó A poderá se utilizar de um recurso não encontrado localmente em A - por exemplo, uma impressora pertencente ao nó B.

- *Aumento de velocidade ou Speedup* - Caso uma determinada aplicação possa ser dividida em várias partes menores, então estas partes poderão ser distribuídas pelos nós de um ambiente distribuído, permitindo a execução destas partes em paralelo.

- *Confiança* - Se algum nó de um sistema distribuído falhar, os nós restantes (no caso de serem máquinas de propósito geral) poderão continuar operando sem maiores problemas. Contudo, se o sistema for composto por máquinas que cuidam de funções importantes - tais como o sistema de arquivos - uma falha em um nó poderá comprometer o seu funcionamento como um todo.

- *Comunicação* - Os usuários em nós diferentes de um sistema distribuído têm a possibilidade de trocar informações, graças às redes de comunicação. Para isto, lança-se mão do mecanismo de *passagem de mensagens*, que permite a execução de funções em um nível "mais alto", tais como transferência de

arquivos, login, correio eletrônico e chamadas remotas a procedimento (RPC ou *Remote Procedure Call*).

Tanto nas grades como em sistemas distribuídos, os programas ativos - isto é, os processos - precisam utilizar-se de recursos computacionais (processador, memória e periféricos) pertencentes a nós computacionais. Quando estas necessidades de recursos forem satisfeitas, um processo terá condições de ser executado. Sob este aspecto apenas, a diferença entre grades e sistemas distribuídos praticamente não existe.

Entretanto, a *Computação em Grade* inova pela maneira com que disponibiliza os recursos existentes - isto é, ao modo como uma *máquina virtual* é estabelecida a partir destes recursos, de acordo com [25].

Em sistemas distribuídos tradicionais, assumimos a existência de um conjunto (*pool*) de nós computacionais, os quais são disponibilizados aos usuários ativos.

O acesso ao *pool* virtual é realizado por alguma espécie de autenticação a cada nó - tecnicamente, também é possível o acesso a um sistema inteiro através de, digamos, *login* a um único nó. Normalmente, se um determinado usuário ganhar o acesso a um nó A, a este mesmo usuário será permitido o uso de todos os recursos vinculados àquele nó A. O mesmo é feito com todos os outros nós; se, porventura, um processo necessita de recursos em dois nós para ser executado (um programa paralelo, talvez), estes serão escolhidos no *pool* (digamos, A e B) com base, é claro, nas necessidades do processo. Este mapeamento poderá ser feito manualmente (pelo próprio usuário), pelo próprio processo ou por algum programa *dispatcher*. Contudo, para que a *máquina virtual* seja estabelecida é necessário que o usuário que disparou o processo possua permissões para uso de A e também de B. Assim que o usuário se autentica, o processo é mapeado aos nós A e B, formando a partir deles (e todos os seus recursos) a citada *máquina virtual*. Então, o processo poderá iniciar sua execução.

Uma outra particularidade de um sistema distribuído é a de um usuário poder ter conhecimento prévio das capacidades e características dos nós, tais como o sistema operacional que um nó usa, sua arquitetura e a velocidade do seu processador, entre outras informações. Finalmente, o *pool* é formado, comumente, por nós pertencentes a um mesmo domínio, e por causa desta característica, a quantidade de nós em um *pool* normalmente não vai além da casa das centenas; a quantidade típica costuma estar na faixa entre 10 e 100 nós [25].

A máquina virtual é estabelecida de maneira diferente nas grades. Existem algumas outras descrições a respeito delas que vale a pena mencionar aqui, com o objetivo de entendermos melhor do que se trata a máquina virtual. Por exemplo:

- "um ambiente largamente expandido que consiste transparentemente de estações de trabalho, computadores pessoais, mecanismos de renderização gráfica, supercomputadores e dispositivos não-tradicionais: por exemplo, TVs, torradeiras, etc."[20],

ou mesmo,

- "uma coleção de recursos geograficamente separados (pessoas, computadores, instrumentos, bancos de dados) conectados por uma rede de alta velocidade distinta por uma camada de software, com freqüência chamada de *middleware*, a qual transforma uma coleção de recursos independentes em uma máquina virtual simples e coerente"[21].

Enfim, é possível concluir que o *pool* virtual em uma grade não é direcionado a nós computacionais - máquinas completas - mas sim a cada um dos *recursos* disponíveis; estes poderão ser não somente ciclos de CPU e memória, mas também meios de armazemento, dispositivos de entrada e saída (áudio, por exemplo), dados e até mesmo programas executáveis, no caso de o cliente da grade não dispor destes localmente.

Com os ambientes em grade, desvinculam-se os *recursos* dos *nós*. A um usuário será possível ter acesso a recursos das máquinas, sem que para isso tenha de autenticar-se a uma delas (ou a muitas). O que existirá é uma *única* autenticação para entrar no *pool* virtual contendo recursos. Uma vez autenticado, o usuário possuirá direitos de uso sobre um recurso - digamos, uma impressora em um nó A - mas não sobre *todo* o nó A, ou melhor, não sobre *todos* os seus recursos, como se faz num sistema distribuído comum.

Somente a partir da autenticação do usuário tenta-se mapear seus processos a recursos. No caso de um processo requerer, digamos, duas CPUs com memória de 100 Mbytes e uma impressora, ele será mapeado - de forma automática - para nós que tenham posse de tais recursos. Após esta fase, a máquina virtual será ativada, para a execução do processo.

A Figura 2.1 ilustra, de forma sucinta, os conceitos de um sistema distribuído e uma grade. Os nós são representados por retângulos, cada um com os seus recursos representados por círculos. No *nível físico*, os nós e seus recursos são vistos de forma bruta, tanto pelas grades como pelos sistemas distribuídos. Um *pool* virtual é visto nos sistemas distribuídos como um grupo de máquinas, ao passo que nas grades um *pool* é formado pelos recursos em separado. A partir deste *pool*, uma aplicação forma uma *máquina virtual*; nos sistemas distribuídos, a formação da máquina virtual é dada pela seleção de máquinas que possuam os recursos de que uma aplicação precisa. Já nas grades, a máquina virtual é formada efetivamente por esses recursos, sem que a aplicação tenha acesso a máquinas inteiras.

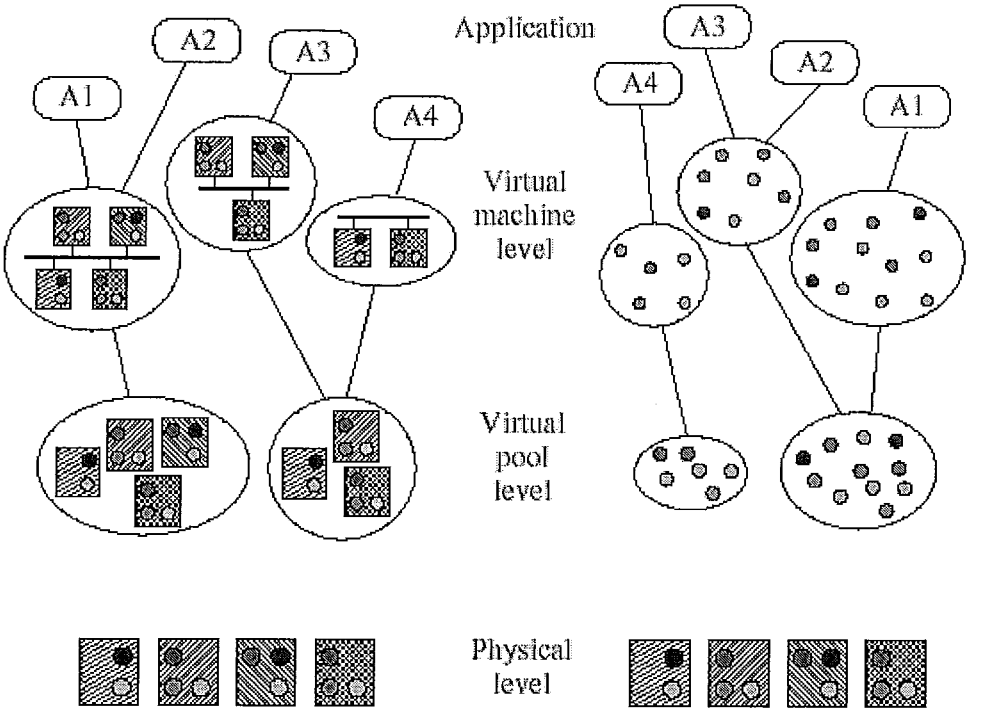


Figura 2.1: Representação lógica de um sistema distribuído (à esquerda) e de uma grade (à direita). Os *nós* são representados por retângulos e os *recursos*, por círculos [25].

Uma vez que não é necessária a autenticação em cada um dos nós, torna-se fácil o *compartilhamento de recursos em larga escala*, característica essencial das grades. Estes são capazes de abranger máquinas de vários domínios geograficamente distantes, aumentando dessa forma a variedade e quantidade de recursos disponíveis.

Todos estes motivos fazem com que um usuário (e seus processos) faça proveito da

grade de forma transparente. Isto é, não conhece *a priori* - ou conhece muito pouco - sobre qual máquina detém qual recurso, ou quantos recursos estão vinculados a qual máquina, entre outros pontos. Para o usuário, é irrelevante saber mais detalhes sobre os nós que lhe fornecem os recursos.

Em suma, os ambientes distribuídos convencionais são baseados na *propriedade* dos recursos, ao passo que as grades são direcionadas ao *compartilhamento* de recursos [25]. Ambos os ambientes foram projetados para suportar processamento de alto desempenho. Entretanto, as grades oferecem mais poder computacional.

2.3 A Evolução

A evolução das grades computacionais é descrita por De Roure *et al* em [9], através da identificação de três gerações.

2.3.1 A Primeira Geração

As primeiras iniciativas tomadas em direção às grades ocorreram no fim dos anos 80, com os projetos de ligação de *sites* de supercomputação. Esta prática ficou conhecida na época como *metacomputação*, termo que foi bastante popularizado graças ao artigo de Smarr e Catlett [28] e tornou-se a palavra-chave dos *sistemas de primeira geração*. O grande objetivo da metacomputação, em poucas palavras, era fornecer recursos computacionais para aplicações que precisavam ter um alto desempenho.

Dentre os projetos significativos desenvolvidos nesta primeira geração, vale a pena fornecer uma breve descrição do I-WAY [11], um ambiente que tinha como objetivo principal integrar centros de supercomputação ao longo dos Estados Unidos. Recursos como conjuntos de dados, poderosos computadores e ambientes virtuais residiam em dezessete *sites* diferentes pelo país, e eram conectados por dez redes, diferentes entre si no que se refere à largura de banda, ao protocolo utilizado e às tecnologias de roteamento, entre outras características.

Em cada um dos dezessete *sites* participantes, uma estação de trabalho UNIX rodava um servidor de ponto-de-presença (I-POP), que agia como um *gateway* para o ambiente. Cada I-POP dispunha de mecanismos que permitiam autenticação, reserva de recursos, criação de processos e funções de comunicação uniformes ao I-WAY. Havia também um escalonador de recursos denominado CRB (*Compu-*

tational Resource Broker), um para cada servidor I-POP. Sua implementação foi estruturada em termos de se ter um único CRB central e, em cada um dos *sites*, um CRB local. Se certo usuário tivesse de rodar algumas tarefas (digamos, em uma máquina no *site* A), ele faria o pedido ao CRB central. Este, por sua vez, entraria em contato com os demais *sites* (B, C...) através das CRBs locais, que associariam as tarefas aos recursos, conforme a disponibilidade destes.

O projeto I-WAY apresentou algumas limitações. Além de lhe ter faltado escalabilidade - isto é, de funcionar especificamente para interconexões rápidas e recursos poderosos, pois as aplicações precisavam disto - o I-WAY continha uma série de características que pareceriam inapropriadas nos dias de hoje. A instalação de uma plataforma I-POP facilitava a configuração de serviços no I-WAY de uma maneira uniforme, porém isto significava que cada um dos *sites* teria de ser especialmente configurado para participar do I-WAY. De mais a mais, a plataforma e servidor I-POP criaram um, dentre vários, pontos de falha no projeto do I-WAY. Em outras palavras, a falha de um I-POP significava que um *site* inteiro estaria fora do ar no ambiente I-WAY.

Contudo, o I-WAY foi inovador, bem-sucedido, e um dos sistemas que pavimentou o caminho para outros projetos de computação em grade que viriam a seguir. Boa parte do desenvolvimento de software do I-WAY foi aproveitado para o projeto Globus [15].

2.3.2 A Segunda Geração

Enquanto os primeiros esforços em direção às grades almejavam simplesmente reunir alguns centros de supercomputação, para execução de aplicações que exigiam alto desempenho, hoje em dia a situação é um pouco diferente. Foster e Kesselman [16] apresentam uma visão das grades como entidades mais abrangentes.

Atualmente, as grades são capazes de juntar bem mais do que um conjunto de centros especializados. Fatores como o desenvolvimento de redes de alta velocidade e a adoção de padrões permitiram visualizar uma grade de uma maneira mais ampla. Uma grade é, pois, considerada hoje uma infraestrutura distribuída em escala global, podendo suportar várias aplicações que peçam poder computacional ou dados em grande quantidade.

Com esta nova formatação para as grades, três pontos principais têm de ser enfrentados:

1. *heterogeneidade* - existe um grande número de recursos de natureza distinta, que se espalham por muitos domínios administrativos e perfazem uma expansão global;

2. *escalabilidade* - o número de recursos contidos numa grade pode chegar à casa dos milhões, e as aplicações que precisarem de recursos geograficamente dispersos deverão ser tolerantes à latência de acesso;

3. *adaptabilidade* - há que se lidar com o fato de um recurso deixar de funcionar numa grade, e se isto acontecer, as aplicações deverão mudar o seu comportamento dinamicamente - afinal, como dizem de Roure *et al.*, "uma falha de um recurso é a regra, não a exceção".

Para lidar com a questão de um ambiente de grade heterogêneo, o conceito de *middleware* foi adaptado. Numa grade, o middleware é usado com o objetivo de esconder sua natureza heterogênea, e fornecer aos usuários - e claro, a suas aplicações - interfaces convenientes aos serviços disponíveis. Estabelecer e usar *padrões* também é uma forma bastante eficiente de lidar com a heterogeneidade; afinal de contas, os sistemas usam padrões e APIs (*Application Programming Interfaces*) variadas, e isto requer que os serviços e aplicações sejam modificados para a extensa gama de plataformas que se pode encontrar numa grade.

Em meio às tecnologias de núcleo (*core tecnologies*) da *segunda geração*, vale a pena destacar um projeto que foi (e é) bastante considerado: o Globus.

2.3.2.1 O Globus

O Globus [15] habilita aplicações a lidar com recursos computacionais distribuídos e heterogêneos como se fossem uma mera máquina virtual, por meio de uma infraestrutura de software. Um elemento central do Globus é o Globus Toolkit, que define os serviços básicos e capacidades necessárias para se construir uma grade computacional. O Globus Toolkit consiste em uma variedade de componentes que implementam funcionalidades básicas, tais como segurança, alocação de recursos, gerência de recursos e comunicação.

Os módulos do Globus Toolkit [15] são os seguintes:

1. *Procura de recursos e alocação* - Este componente fornece mecanismos para expressar as necessidades da aplicação por recursos, para identificar recursos

que combinem com estas necessidades, e para escalonar recursos para as aplicações, visto que eles já foram encontrados.

2. *Comunicações* - Este componente fornece mecanismos básicos de comunicação, que por sua vez devem permitir a implementação eficiente de uma gama de métodos de comunicação - incluindo passagem de mensagens, chamada remota a procedimento (RPC) e *multicast*.

3. *Serviço de informação sobre recursos* - Este componente fornece um mecanismo uniforme para obter informação em tempo real sobre a estrutura e estado de cada recurso.

4. *Interface de autenticação* - Este componente fornece mecanismos básicos de autenticação que podem ser usados para validar a identidade tanto de usuários como de recursos.

5. *Criação de processos* - Este componente é usado para iniciar o processamento em um recurso, uma vez que ele já foi encontrado e alocado.

6. *Acesso a dados* - Este componente é responsável por fornecer acesso de alta velocidade a formas de armazenamento permanente, como arquivos.

Juntos, os módulos do Globus Toolkit podem definir uma *máquina virtual meta-computacional*. A definição desta máquina virtual simplifica o desenvolvimento de aplicações e melhora a portabilidade, permitindo que os programadores pensem em coleções de recursos heterogêneas e geograficamente distribuídas como entidades unificadas.

O Globus é uma evolução do projeto I-WAY, mencionado na Subseção 2.3.1. Em vez de suportar apenas aplicações de alto desempenho, a sua ênfase voltou-se na direção de serviços mais abrangentes, que pudessem oferecer suporte a *organizações virtuais*, que são definidas em [16] como um conjunto de indivíduos e/ou instituições que são definidas por regras de compartilhamento de recursos. Estas regras dizem respeito à definição clara e cuidadosa do *que* será compartilhado, *quem* terá permissão para compartilhar e sob *quais* condições o compartilhamento acontecerá.

2.3.3 A Terceira Geração

Os sistemas de segunda geração tornaram possível a obtenção de computação em larga escala. À medida em que soluções vão sendo descobertas na tecnologia em grades, outros aspectos foram aparecendo. Para que se consiga criar uma nova aplicação em grades, é necessário que haja uma maneira mais flexível - leia-se automática - de se reunir recursos (componentes e informação) em torno deste objetivo.

Existe nos sistemas de *terceira geração* um forte senso de automação. Com o surgimento de novas aplicações, mais carentes de recursos (p.ex. videoconferência), cada vez torna-se mais difícil para os seres humanos lidar diretamente com a heterogeneidade e tamanho de uma grade, delegando esta tarefa a processos dentro dos sistemas, o que leva à autonomia. Esta autonomia sugere, é claro, que eventos como falhas em recursos possam ser resolvidos pelo sistema sem que haja intervenção manual.

Por apresentarem equivalências com as capacidades de auto-organização e cura dos sistemas biológicos, usamos doravante o termo "autonômico" para nos referirmos a este tipo de sistemas em grade. No que concerne a [5], um sistema autonômico possui oito propriedades:

1. precisa de um conhecimento detalhado de seus componentes e estados;
2. deve configurar e re-configurar a si mesmo dinamicamente;
3. procura otimizar seu comportamento para atingir seu objetivo;
4. é capaz de se recuperar de falhas;
5. protege-se contra ataques;
6. conhece seu ambiente;
7. implementa padrões abertos;
8. e por fim, otimiza o uso de recursos.

Estas características começam a ser observadas nos sistemas de terceira geração em desenvolvimento.

A terceira geração representa uma visão mais abrangente da computação em grade, e proverá a estrutura futura para a *e-Science* - um termo que está relacionado com as requisições para se construir uma nova ciência.

Por volta de 2001, *arquiteturas de grade* eram descritas em muitos projetos, sendo que a mais famosa delas foi o modelo em camadas proposto por [16]. Nessa época, o modelo de *serviços Web* estava ganhando em popularidade, apresentando padrões *service-oriented.* Além disso, a computação baseada em agentes já era consideravelmente desenvolvida, visto que *agentes de software* podem ser vistos como produtores, consumidores ou mesmo intermediadores de serviços. Enfim, a *computação em grade*, *agentes de software* e *serviços Web* são três tecnologias que se combinam em um paradigma orientado a serviços, que provê a flexibilidade necessária para a *e-Science* - e para as grades de terceira geração.

2.4 Arquitetura Proposta para uma Grade

É pertinente levarmos em consideração que estabelecer compartilhamento entre OVs (*organizações virtuais*, definidas no item 2.3.2.1) requer uma nova tecnologia, e não pura e simplesmente uma rede de larga escala.

A *arquitetura de Grade* proposta por Foster, Kesselman e Tuecke [16] procura abordar o problema da *interoperabilidade.* A interoperabilidade é uma característica que precisa estar presente numa grade, pelo fato de múltiplas plataformas, linguagens e ambientes de programação estarem presentes nas relações de compartilhamento. Este termo, em um ambiente de rede, significa a existência de *protocolos* em comum, de onde concluímos que a arquitetura de Grade é essencialmente uma arquitetura de protocolos. Por sua vez, estes definem os mecanismos básicos pelos quais os usuários (e recursos) negociam, estabelecem, gerenciam e exploram relações de compartilhamento.

Esta arquitetura proposta é organizada em camadas, à feição do esquema apresentado na Figura 2.2. Estas camadas são:

1. *Matéria-prima* (*Fabric*) - A camada de Matéria-prima representa os recursos concretos cujo acesso compartilhado é mediado pelos protocolos que estão acima dela. Neste nível, são implementadas as operações locais, e específicas, que se pode fazer em cada um dos recursos disponíveis, com base nas operações de compartilhamento definidas nos níveis superiores;

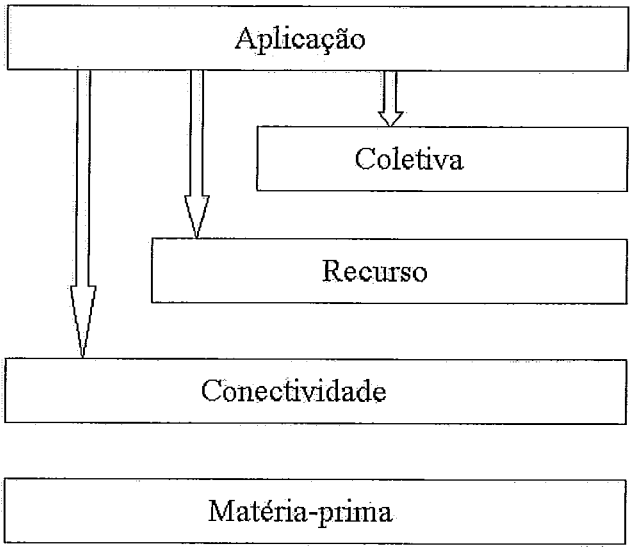


Figura 2.2: Uma arquitetura de Grade proposta por [16].

2. *Conectividade* - Esta camada define protocolos de núcleo (*core protocols*) de comunicação e autenticação, apoiando as transações na rede que são específicas de uma grade. Os protocolos de comunicação tornam possível a troca de dados entre os recursos da camada inferior, isto é, a Matéria-prima. Já os protocolos de autenticação, a seu tempo, são construídos sobre os serviços de comunicação, de forma a fornecer mecanismos seguros para identificar usuários e recursos;

3. *Recurso* - A camada de Recurso constrói, sobre a camada de Conectividade, protocolos de comunicação e autenticação para definir outros protocolos que realizam ações sobre recursos individuais, como: negociação segura, iniciação, monitoramento, controle, operações sobre contas, etc. Implementações destes protocolos fazem chamadas às funções da Matéria-prima, para acessar e controlar recursos concretos;

4. *Coletiva* - Nesta camada seguinte, há protocolos que não estão concentrados nas interações com um único recurso (como na camada anterior), mas sim associados a *coleções* de recursos, numa maneira mais global. Dada esta razão, a camada recebeu o nome de Coletiva; e

5. *Aplicação* - Esta é a camada final, à qual foi dado o nome de Aplicação, compreendendo as aplicações de usuários que operam nos ambientes de OVs.

2.5 Aplicações da Computação em Grade

Foster e Kesselman [16] identificaram, por meio de alguns experimentos, cinco classes majoritárias de aplicações onde as grades serão (e são) largamente utilizadas. As classes citadas aparecem na Tabela 2.1.

2.5.1 Supercomputação distribuída

As aplicações de supercomputação distribuída, como por exemplo o DIS (*Distributed Interactive Simulation*), são grandes problemas que precisam de recursos computacionais em larga escala, que podem ser fornecidos pelos sistemas em grade.

O DIS é uma técnica usada para treinamento e planejamento em ambiente militar. Situações nas quais o DIS é aplicado podem envolver centenas de milhares de entidades, sendo que cada uma delas apresenta comportamentos potencialmente complexos. Segundo Foster e Kesselman, mesmo os sistemas distribuídos maiores conseguem suportar no máximo situações englobando 20.000 entidades, rodando o DIS. Pesquisadores no Instituto de Tecnologia da Califórnia mostraram como muitos supercomputadores podem ser reunidos para que o DIS possa analisar situações muito mais amplas e complexas.

Tratando-se de uma arquitetura de grade para supercomputação distribuída, há alguns desafios que devem ser encarados. Entre eles, destacam-se a necessidade de co-escalonar os recursos que são escassos e caros, escalar os protocolos e algoritmos para centenas de milhares de nós e conseguir níveis altos de aproveitamento em sistemas heterogêneos.

2.5.2 Computação de Alta Vazão (*High-throughput*)

Na computação de alta vazão, uma grade pode ser utilizada para escalonamento de um grande número de tarefas fracamente acopladas ou mesmo independentes entre si (o que normalmente pode ser chamado de *bag-of-tasks*). Exemplos deste tipo de tarefa são as aplicações de Aprendizado de Máquina (*Machine Learning*), que são computacionalmente intensivas, e que geram um número elevado de tarefas neste

estilo. Neste caso, o objetivo será de aproveitar ciclos de CPU ociosos em estações de trabalho.

Um bom exemplo dentre os sistemas que realizam este tipo de trabalho é o Condor [22], escalonador desenvolvido pela Universidade de Wisconsin. O Condor é apresentado com mais detalhes na Seção 3.1.1.

Categoria	Características
Supercomputação Distribuída	Problemas excessivamente grandes, que precisam de muitas CPU's, memória...
Computação de Alta Vazão	Agrega muitos recursos ociosos com o objetivo de aumentar a vazão
Computação Conforme a Necessidade	Recursos remotos integrados com alguma computação local
Com Intensividade de Dados	Novas informações extraídas de bancos de dados extensos (ou de *muitos* bancos de dados)
Colaborativa	Suporte de trabalho colaborativo envolvendo vários participantes

Tabela 2.1: As cinco classes majoritárias das aplicações que usam grades

2.5.3 Computação Conforme a Necessidade ("On-demand")

Uma grade é uma excelente opção quando se precisa de recursos que não se encontram localmente, tais como repositórios de dados, sensores especializados, *software*, e por aí afora. Por exemplo, não vale a pena possuir um *cluster* de PCs se raramente há atividades de processamento pesado; em vez disto, poderemos adquirir temporariamente tal recurso via uma grade computacional.

Dentre os sistemas que exercem este tipo de computação estão o NEOS [8] e o NetSolve [4], resolvedores (*solvers*) que permitem aos usuários acoplarem software remoto e recursos a suas aplicações.

Os desafios que se deve enfrentar em computação *on-demand* são derivados da natureza dinâmica das requisições de recursos, e das populações extremamente

largas de usuários e recursos. Entre estes desafios podemos citar: configuração, posicionamento de recursos e escalonamento, entre outros.

2.5.4 Com Intensividade de Dados

O foco nestas aplicações é sintetizar informações novas, a partir de dados que são mantidos em repositórios geograficamente distribuídos, bancos de dados e bibliotecas digitais. Esta ação é freqüentemente custosa em termos de comunicação e de processamento.

Alguns experimentos de física de alta energia gerarão, no futuro, vários terabytes de dados por dia (por volta de 1 pentabyte por ano). Para que um evento possa ser detectado, uma larga fração destes dados deverá ser analisada. Os físicos que precisarão deles estão distribuídos pelo mundo, e portanto os sistemas onde os dados estão armazenados também o estarão.

Entre os desafios encontrados nas aplicações *data-intensive* estão o escalonamento e configuração de volumosos fluxos de dados por múltiplos níveis de hierarquia.

2.5.5 Computação Colaborativa

A princípio, as aplicações colaborativas estão relacionadas com possibilitar e melhorar as interações humanas, e são normalmente estruturadas como um espaço virtual compartilhado. Muitas aplicações colaborativas almejam habilitar o uso compartilhado de recursos como arquivos de dados ou simulações; neste caso, elas também têm características dos outros tipos de aplicações já descritas.

Podemos citar como exemplo o sistema BoilerMaker, desenvolvido no Argonne National Laboratory, que permite a múltiplos usuários colaborarem no projeto de sistemas controladores de emissão em incineradores industriais [12].

A partir da perspectiva de uma grade, os desafios consistem, principalmente, em requisições de tempo real impostas pelas capacidades humanas de percepção e na riqueza de interações que podem ocorrer nesta classe de aplicações.

Capítulo 3

Fundamentos para o Trabalho

Este capítulo descreve as ferramentas para grades efetivamente utilizadas como fundamento deste trabalho, dissertando sucintamente sobre os gêneros em que elas se classificam e identifica limitações que motivam o trabalho apresentado no Capítulo 4.

3.1 Sistemas Escalonadores de tarefas em Grades

A função deste gênero de ferramentas é associar tarefas (*jobs*) ativas de usuários a *recursos* disponíveis em um ambiente de grade, de forma transparente ao usuário.

Foram selecionados aqui alguns sistemas escalonadores que gozam de relativa popularidade no meio acadêmico ou no meio comercial. Estes sistemas, assim como grande parte dos escalonadores de tarefas em grades, foram inicialmente desenvolvidos para plataformas distribuídas locais. Os sistemas abordados foram o Condor [22, 31], o PBS [32], o SGE [14] e o LSF [6], que estão descritos a seguir.

- O Condor é um pacote de software para executar tarefas (*jobs*) em lotes (*batch*). O sistema suporta uma gama de plataformas diferentes, tais como sistemas operacionais do tipo UNIX (por exemplo, Linux e Solaris) e Windows NT. Um mesmo *pool* pode incluir máquinas de múltiplas plataformas. O Condor é desenhado para aproveitar equipamento que esteja ocioso, respeitando o direito dos proprietários sobre suas máquinas.

- O PBS (*Portable Batch System*) [32], é um sistema de processamento em *batch* e gerenciamento de carga, desenvolvido originalmente pela NASA. Este sistema suporta desde *clusters* até supercomputadores. O módulo escalonador de tarefas do PBS permite que o administrador de sistema defina quais tipos

de recursos (e quanto de cada recurso) poderá ser usado por cada tarefa. Este módulo possui total conhecimento sobre as tarefas ativas e uso dos recursos do sistema, além de permitir que os *sites* em uma rede estabeleçam suas próprias políticas de escalonamento para a execução de tarefas. O PBS mune o usuário de uma interface gráfica para submissão, rastreamento (*tracking*) de tarefas e propósitos administrativos.

- O SGE (*Sun Grid Engine*) [14] é baseado no software desenvolvido pela Genias, conhecido como Codine/GRM. Este sistema possui uma *área de retenção* e várias *filas* que abastecem as tarefas com serviços. No SGE, um usuário submete uma tarefa, e declara um perfil de *requisitos* para a mesma. No momento em que uma das filas estiver preparada para receber uma tarefa nova, o SGE determina que tarefas combinam com aquela fila, e então despacha a tarefa com a maior prioridade ou maior tempo de espera na área de retenção; o SGE tentará iniciar novos *jobs* na fila que combine melhor com cada um deles, ou mesmo numa fila mais vazia.

- O LSF (*Load Sharing Facility*) é outro exemplo de sistema comercial da Platform Computing Corporation [6]. O LSF dispõe de um compartilhamento de carga distribuído, além de programas para enfileiramento em lote (*batch queuing*) que gerenciam e analisam os recursos e o nível de carga em uma rede composta por computadores heterogêneos.

Por ser uma das bases deste trabalho, o Condor está descrito em detalhes na próxima seção. Das ferramentas que foram apresentadas, o Condor é considerada a mais adequada para os ambientes de grade, devido a suas características de escalonamento *oportunista* - ou seja, o aproveitamento de ciclos de CPU que estejam ociosos.

3.1.1 O Condor

O *Condor High Throughput Computing System* [22, 31] que faz parte do Projeto Condor, da Universidade de Wisconsin-Madison, é um sistema para escalonamento de *jobs* e recursos computacionais em larga escala. Este sistema fornece, entre outros itens, políticas de escalonamento de tarefas, monitoramento de recursos e um esquema de prioridades.

A arquitetura e mecanismos do Condor permitem o seu bom desempenho em duas áreas: a *computação de alta vazão* (*high-throughput*) e a *computação oportunista*. Estas áreas estão fortemente ligadas; é possível obter grande poder computacional utilizável por processos ativos, se houver meios de aproveitar ciclos de CPU ociosos em máquinas - ainda que a ociosidade não seja *total*.

Alguns dos mecanismos especiais incluídos no Condor são os seguintes [35]:

- *ClassAds* - Os *ClassAds* são semelhantes a anúncios de jornal; cada uma das máquinas em um *pool* informa seus atributos (tais como memória RAM, tipo de CPU, sistema operacional, carga de processamento e outras propriedades estáticas e dinâmicas) através de um *ClassAd*. Da mesma maneira, quando se submete uma tarefa (*job*), um usuário especifica os requisitos dos quais precisa para a sua execução - por meio de um *ClassAd*. Por exemplo, é possível que um usuário queira que seu *job* seja executado somente em máquinas com mais de 128 Mbytes de RAM, e o Condor tentará verificar se existem máquinas que atendam este requisito, verificando seus *ClassAds*. O Condor faz o papel de um *matchmaker*, combinando tarefas com máquinas.

- *Checkpoint de jobs e migração* - Se uma máquina em um *pool* ficar indisponível de uma hora para outra, algum *job* que estava sendo executado nela poderá sofrer um *checkpoint* - isto é, guardar o ponto onde o *job* parou de ser executado - e migrar para outra máquina disponível, a fim de continuar a execução a partir do *checkpoint*. Esta é uma poderosa forma de tolerância a falhas, e preserva o tempo de computação já consumido pelo *job*.

- *Chamadas de sistema remotas* - Embora execute tarefas em máquinas remotas, o Condor pode preservar o ambiente de execução local, graças às *chamadas de sistema remotas*. Ou seja, se um *job* estiver para ser executado em uma máquina remota, seu usuário não tem necessidade de, digamos, transferir previamente à referida máquina dados utilizados pelo *job*. O programa comportar-se-á como se estivesse sendo executado localmente.

Lançando mão destes mecanismos, o Condor é capaz de gerenciar os ciclos de processamento desperdiçados em máquinas não-utilizadas. Por exemplo, ele pode ser configurado de modo a permitir a execução de *jobs* em uma máquina somente se a CPU e teclado da mesma estiverem ociosos. Então, se um *job* está executando em

uma máquina e um usuário volta e começa a utilizá-la, o Condor poderá se utilizar do *checkpoint*, e então migrar esta tarefa para uma outra máquina que possua condições de recebê-la. Naturalmente, não será necessária nenhuma transferência de dados.

A primeira versão do Condor foi instalada no Departamento de Ciências da Computação da Universidade de Wisconsin-Madison, em 1987, e se tornou, ao longo dos anos, uma ferramenta essencial para os pesquisadores de todo este centro acadêmico.

3.1.1.1 *Matchmaking*

O mecanismo de *Matchmaking* usado pelo Condor nasceu da necessidade de facilitar as funções de *escalonamento* (gerência de um recurso pelo seu proprietário, alocando-o a tarefas) e *planejamento* (tentativa de aquisição de recursos por usuários, com base em certas métricas). O escalonamento é uma tarefa corrente nos ambientes distribuídos, com um algoritmo centralizado - afinal, os recursos num sistema distribuído têm um proprietário único (uma organização, ou centro de pesquisa). Visto que numa grade os recursos têm múltiplos proprietários, que não compartilham um mesmo algoritmo de escalonamento, é preciso que haja uma fase de planejamento [35].

O *Matchmaking* é efetuado em quatro passos, cuja representação está disposta na Figura 3.1. No primeiro passo (1), agentes (representando usuários, ou suas tarefas) e recursos (representando proprietários) anunciam suas características e requisições através dos já mencionados *ClassAds*. No segundo passo (2), um processo *matchmaker* verifica os *ClassAds* e forma pares de agentes com recursos, que satisfaçam as preferências e características de um e outro. No terceiro passo (3), o *matchmaker* notifica as partes da combinação feita, e então a sua intermediação termina. No quarto e final passo (4) a *reivindicação*, o agente e o recurso estabelecem contato, e cooperam para a execução de um *job*.

Um *ClassAd* pode ser melhor visto como um conjunto de expressões unicamente identificadas, usando um modelo de dados semi-estruturado, de modo que nenhum esquema específico é solicitado pelo *matchmaker*. Cada expressão é chamada de *atributo*, e cada atributo tem um *nome* e um *valor*. Dois atributos significativos para o Condor são **Requirements** e **Rank**. O primeiro indica necessidades, enquanto o último define preferências de execução. Exemplos de *ClassAds*, para uma tarefa e para uma máquina, são mostrados na Figura 3.2.

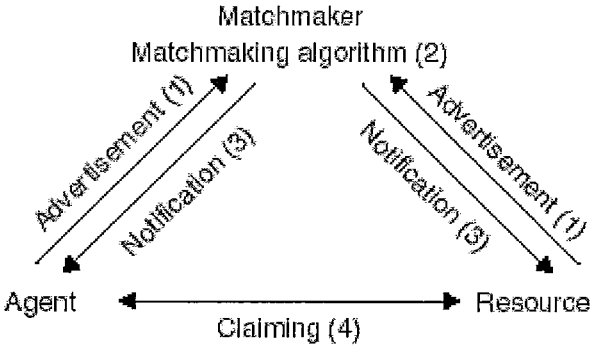


Figura 3.1: O mecanismo de *matchmaking* [35].

Job ClassAd	Machine ClassAd
[	[
MyType = "Job"	MyType = "Machine"
TargetType = "Machine"	TargetType = "Job"
Requirements =	Machine = "nostos.cs.wisc.edu"
((other.Arch=="INTEL"&&	Requirements =
other.OpSys=="LINUX")	(LoadAvg <= 0.300000) &&
&& other.Disk > my.DiskUsage)	(KeyboardIdle > (15 * 60))
Rank = (Memory * 10000) + KFlops	Rank = other.Department==self.Department
Cmd = "/home/tannenba/bin/sim-exe"	Arch = "INTEL"
Department = "CompSci"	OpSys = "LINUX"
Owner = "tannenba"	Disk = 3076076
DiskUsage = 6000	]
]	

Figura 3.2: ClassAds no ambiente Condor [35].

Note que a tarefa precisa ser executada em uma máquina com arquitetura INTEL e um sistema operacional LINUX, e esta necessidade é correspondida pela máquina, formando desse modo uma combinação.

As preferências e necessidades de cada *job* - ou conjunto deles - são definidas pelo usuário em um *arquivo de submissão*, que serve como fonte para a montagem de um *ClassAd.* A página do sistema Condor [31] fornece um tutorial detalhado a respeito de como submeter tarefas para execução no Condor.

3.1.1.2 Ambientes de Execução no Condor

O Condor possui vários ambientes de execução de tarefas, chamados de *universos.* Esta é uma informação que o usuário deve incluir em seu arquivo de submissão. Os universos mais comuns são o universo *standard* e o universo *vanilla.*

O universo *standard* permite que uma tarefa ativa se utilize dos mecanismos

de chamadas de sistema remotas e fornece as condições necessárias para o uso do *checkpoint*, migrando uma tarefa parcialmente terminada para outra máquina, se necessário. Para que o ambiente *standard* possa ser usado, é preciso re-ligar o programa com as bibliotecas do Condor.

Já o universo *vanilla* provê uma forma de executar tarefas que não podem ser re-ligadas. Embora a maioria dos programas possa ser preparada para o universo *standard*, existem exceções como os *shell scripts* dos ambientes tipo Unix. Infelizmente, as tarefas no universo *vanilla* não podem lançar mão de *checkpoints* ou chamadas de sistema remotas. Quando uma máquina deixa de ser ociosa (por exemplo, um usuário volta de seu almoço e começa a operá-la), o Condor tem duas opções a respeito do que fazer com os *jobs*: interromper sua execução até que a máquina fique de novo livre, ou recomeçar sua execução em outra máquina disponível. O acesso a dados de entrada ou saída de um *job* só poderá ser feito se eles forem explicitamente transferidos, ou se o Condor for configurado para trabalhar num sistema de arquivos compartilhado.

Quanto aos demais universos, eles são brevemente descritos a seguir:

- **PVM** - O universo PVM (*Parallel Virtual Machine*) permite que os programas escritos para uma interface PVM sejam executados dentro do ambiente oportunista do Condor.

- **MPI** - Este universo permite o funcionamento de programas que usem MPI (*Message Passing Interface*) no Condor.

- **Globus** - O universo Globus pretende atender a usuários que querem disparar jobs para o ambiente Globus [15] a partir do Condor, fornecendo para tal uma interface padrão. Cada *job* é traduzido para a linguagem RSL do Globus (*Resource Specification Language*), e na seqüência é submetido ao Globus por meio do protocolo GRAM [23].

- **Java** - Um programa submetido ao universo Java poderá ser executado em qualquer máquina contendo uma JVM (Java Virtual Machine).

- **Scheduler** - No universo Scheduler, um *job* pode ser submetido e executado em condições diferentes, como a de não esperar para ser combinado com uma máquina. Em vez disso, ele será executado na mesma máquina onde foi submetido, e nunca será interrompido, ainda que a máquina esteja ocupada.

Mais detalhes sobre o funcionamento destes tipos de universos são encontrados em [31].

3.2 Gerenciamento Automático de Tarefas

Algumas ferramentas para ambientes de grade procuram tornar menos árdua a vida do usuário que precisa fazer experimentos excessivamente grandes (pesquisas em Inteligência Artificial, Física, Astronomia e Biologia, entre outras), disparando assim uma grande quantidade de tarefas.

A *resubmissão* de tarefas é um ponto relevante a ser analisado numa situação destas. Se um determinado usuário submeter 10.000 tarefas a um ambiente de grade, várias destas tarefas poderão falhar - por exemplo, por falta de memória, falhas na rede, no sistema operacional ou no gerenciador de recursos - fazendo com que o usuário tenha que verificar quais falharam e resubmetê-las. Em nosso caso, ainda é possível que ocorram problemas na aplicação ou no *central-manager* de um *pool* do Condor (máquina central que é responsável pelas funções vitais do Condor, tais como *Matchmaking* e *Checkpointing*). Infelizmente, se o usuário tiver de fazer isto manualmente, será um trabalho inviável, ainda que a porcentagem das tarefas falhas seja pequena - se ela for de 5% neste caso, serão 500 tarefas a submeter, o que é um número considerável.

Um outro ponto delicado concernente a um grande número de tarefas é a *análise de dependências* entre elas. Vamos supor que 10.000 tarefas sejam disparadas em um experimento. Essas tarefas são divididas em dois grupos A e B, que têm 5.000 tarefas cada um. Cada tarefa Bn ($1 \leq n \leq 5000$), para poder começar a executar, precisa de um arquivo de saída gerado por uma correspondente An, como mostrado na Figura 3.3.

Mais uma vez, é fato que lidar manualmente com este fator é difícil e demorado; como as tarefas são muitas, torna-se inviável que o usuário procure exercer o controle do término de tal tarefa para poder iniciar uma outra.

Com relação a estes dois pontos, a situação se agrava quando é constatado que o usuário é leigo em computação, ou pelo menos no uso de ambientes de grade.

Objetivando evitar o infortúnio causado pelo controle de tarefas manual, esforços foram gastos no sentido de gerenciá-las automaticamente. Mencionemos alguns exemplos:

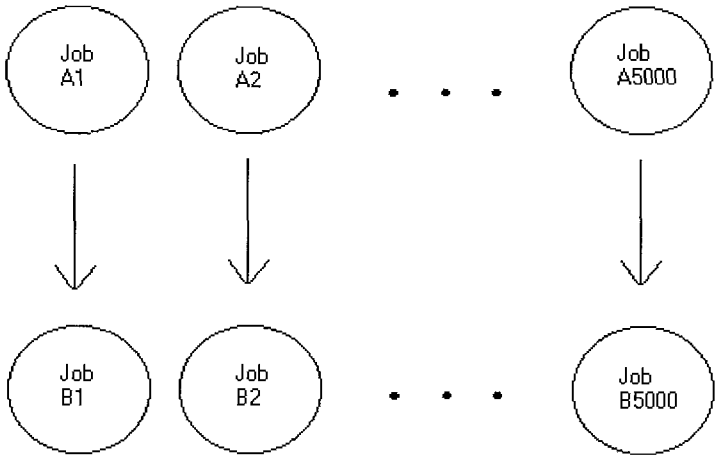


Figura 3.3: Um exemplo simples de dependências entre *jobs* de um mesmo experimento.

- O Chimera [18], da Universidade de Chicago, é um sistema que trata de gerenciar dados que não são obtidos por meio de experimentos, mas sim *derivados* de outros dados (chamados de *dados virtuais*), sendo que a linguagem para acesso a eles pelas aplicações se denomina VDL (*Virtual Data Language*). O Chimera possui um *catálogo de dados virtuais* para representar procedimentos para derivação de dados e dados derivados. Este catálogo é combinado com um interpretador VDL, que traduz pedidos do usuário em construção e pesquisa (*queries*) de entradas em bancos de dados. Entre outras funções, a linguagem VDL suporta a identificação de dependências entre derivações (representações de tarefas em exeçução), através da análise de seus arquivos de entrada e saída. A partir desta análise, o Chimera pode traduzir estas dependências entre tarefas num DAG (grafo acíclico direcionado, ou *Directed Acyclic Graph*), e com a ajuda do planejador Pegasus [10], pode a partir do DAG gerar uma entrada para o Condor DAGman [31].

- O Condor DAGman [31] é um meta-escalonador para o Condor que gerencia dependências entre tarefas. O Condor pode encontrar máquinas para a execução de programas, porém não escalona tarefas com base em dependências. O DAGman submete tarefas ao Condor numa ordem representada por um DAG, que pode ser usado para representar um conjunto de programas onde a entrada, saída ou execução de um ou mais programas é dependente de um ou

mais programas.

- Dutra *et al* [13] apresentam um sistema específico para lidar automaticamente com grandes experimentos. Este sistema é o protótipo que deu início a este trabalho, e suas características estão melhor explicadas no Capítulo 4.

Uma vez que o trabalho trata também de análise de dependências entre os *jobs*, a linguagem do Condor DAGman foi escolhida para ser usada nessa parte. A escolha se deve ao fato desta linguagem estar inserida no ambiente Condor, e ser compreendida diretamente por ele. Na próxima seção o funcionamento do DAGman será descrito em mais detalhes.

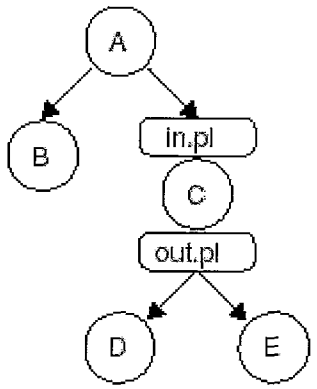
3.2.1 O Condor DAGman

O Condor DAGMan (*Directed Acyclic Graph Manager*) é um meta-escalonador para o Condor, que cuida de gerenciar dependências que possam vir a existir entre *jobs*. Um grafo acíclico direcionado pode ser usado para representar um conjunto de programas nos quais a entrada, saída ou execução de um ou mais programas é dependente do término de um ou mais programas. Os programas são nós (vértices) no grafo, e as arestas (arcos) identificam as dependências. O DAGMan submete os *jobs* ao Condor de acordo com a ordem representada por um DAG.

Cada nó (programa) no DAG precisa de seu próprio arquivo de submissão, onde se especificam os requisitos do *job*. O DAG é definido pelo conteúdo de um arquivo de entrada, onde se definem as dependências entre os *jobs*.

Então, de que forma seria um arquivo de entrada para o DAGman? A Figura 3.4 demonstra a linguagem por ele aceita. Uma entrada JOB, por exemplo, associa um nome abstrato (A) com um arquivo de submissão (a.condor). Por sua vez, uma entrada PARENT-CHILD estabelece uma relação entre vários jobs (pelo menos entre dois). Neste caso, os *jobs* B e C não poderão entrar em execução até que A tenha terminado, enquanto os *jobs* D e E não poderão entrar em execução até que C tenha terminado. Tarefas que sejam independentes entre si poderão executar em qualquer ordem, naturalmente.

O *job* C está associado com dois programas de *script* PRE e POST, por meio da entrada SCRIPT. O comando PRE identifica um programa que deverá rodar antes de um *job* entrar em execução, enquanto o comando POST identifica um programa que deve rodar após a execução de um *job*. Os programas PRE são comumente usados




```
JOB A a.condor
JOB B b.condor
JOB C c. condor
JOB D d.condor
JOB E e.condor
PARENT A CHILD B C
PARENT C CHILD D E
SCRIPT PRE C in.pl
SCRIPT POST C out.pl
RETRY C 3
```

Figura 3.4: Um DAG e seu arquivo de entrada [35].

para preparar o ambiente de execução, transferindo ou descompactando arquivos, e os POST podem ser usados para avaliar a saída da tarefa. É conveniente ressaltar que estes programas PRE e POST não são considerados como *jobs* pelo DAGman, sendo executados na máquina de submissão.

Por fim, a entrada RETRY permite ao usuário repetir a execução de um *job* no caso de erros como executáveis corrompidos ou sistemas de arquivos desmontados, que por vezes não são detectados pelos sistemas. No caso do *job* C, se o mesmo terminar com sucesso mas cometer algum erro não-convencional (por exemplo, deixar de produzir um arquivo de saída, o que será detectado por POST), o erro será detectado como se tivesse sido descoberto em C, e o Condor DAGman re-submeterá o *job*.

É possível haver *jobs* falhos devido à natureza do sistema distribuído, a problemas na rede, ou mesmo em recursos requisitados. Neste caso, o *job* perderá o contato com o Condor, e o DAGman não ficará a par desta perda.

3.2.2 Limitações do conjunto Condor/Condor DAGman

Como já havia sido colocado, os sistemas escalonadores/gerenciadores de tarefas não dispõem da parte de re-submissão, e com o Condor não é diferente. Até o momento, não há nada criado para a re-submissão de tarefas totalmente independentes entre si, quando não se faz necessário aplicar o DAGman.

Quando se utiliza do Condor DAGman, o usuário é capaz de controlar possíveis dependências entre as tarefas utilizando uma linguagem especial. Além disso, novas tentativas de submissão de tarefas são possíveis automaticamente com o comando RETRY, dependendo do tipo de erro que ocorra com a tarefa. No entanto, se

o número de tarefas do usuário estiver na casa das centenas ou acima, torna-se inviável identificar cada tarefa e estipular cada dependência escrevendo um *script* na linguagem do DAGman. Esta situação se agravará se o usuário não tiver muita familiaridade com ambientes multiprocessados.

Ainda discutindo sobre o DAGman, é também importante notar que, caso um *job* produza um erro tal como um arquivo de saída inexistente ou corrompido (exemplificado na Seção 3.2.1), o DAGman não o reconhecerá, a não ser que o usuário explicitamente forneça um arquivo POST para verificar as saídas do *job*.

Finalmente, não foi verificado nenhum limite de tempo para um *job* permanecer na fila de tarefas ativas no ambiente Condor; isto poderia ser útil no caso do *job* ficar paralisado (*deadlock*), por conta de algum imprevisto, tal como um cálculo mal feito. Nesse caso, estabelecer-se-ia um limite de tempo que, se ultrapassado, ocasionaria a re-submissão do *job*.

Capítulo 4

A Ferramenta

Neste quarto capítulo, as origens, o propósito e implementação física de nossa ferramenta serão apresentados e detalhados.

4.1 Apresentação da Ferramenta

Propomos uma ferramenta para a realização de experimentos com grande quantidade de *jobs* por meio de um diagrama, disposto na Figura 4.1. Esta ferramenta pode ser organizada em três partes principais:

- *Interface Web* - Esta parte cuida da interação do usuário com a ferramenta, permitindo um acesso facilitado do mesmo ao ambiente de grade e a obtenção, via HTML (*HyperText Markup Language*), de informações sobre o andamento da execução dos *jobs*.

- *Submissão de Tarefas* - A parte de submissão de tarefas é responsável pela preparação dos dados de entrada para uso dos *jobs*, que serão a seguir submetidos ao ambiente Condor.

- *Monitoramento de Tarefas* - Finalmente, a parte de monitoramento de tarefas se refere às partes de coleta e organização de informações sobre os *jobs* e verificação do andamento de suas execuções, efetuando a re-submissão deles caso isto se torne conveniente.

Além destes módulos, foi efetuada também a integração com o Condor DAGman, com o objetivo de manipular as dependências entre *jobs* no ambiente Condor.

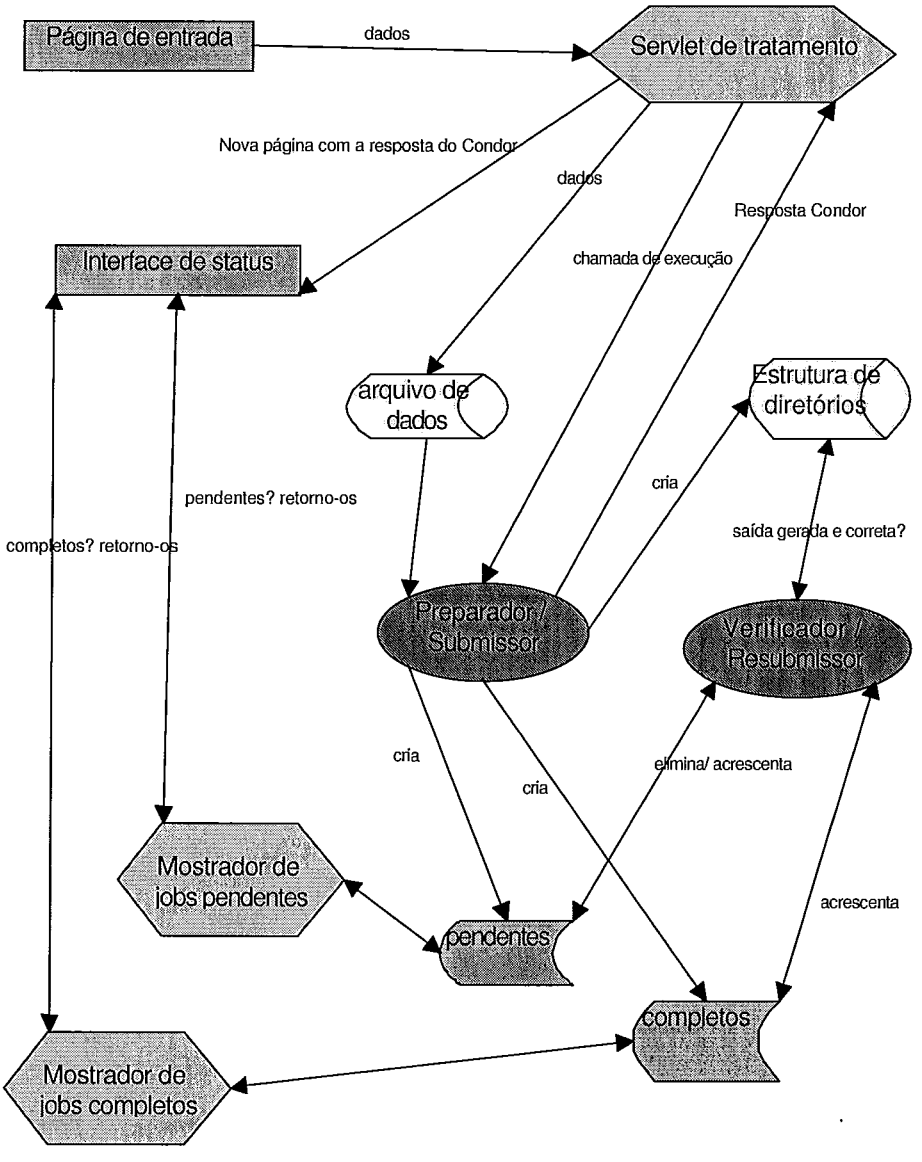


Figura 4.1: Diagrama da nova ferramenta. As páginas Web são representadas por retângulos, estruturas de arquivos por cilindros, *servlets* por hexágonos, programas internos por formas ovais e bancos de dados por setas largas.

4.2 Experimentos com Aprendizado de Máquina

A aplicação central com a qual trabalhamos em nossa ferramenta é o *Aprendizado de Máquina* (*Machine Learning*). De acordo com Tom Mitchell [24] o Aprendizado de Máquina é o estudo de algoritmos computacionais que melhoram seu desempenho

automaticamente conforme ganham experiência na realização de alguma tarefa, como jogar xadrez, por exemplo.

O objetivo de um sistema *aprendiz* é melhorar (e inicialmente construir) sua capacidade de resolução de problemas, executando determinada tarefa com base numa experiência de *treinamento* e uma medida de desempenho - que, no caso do xadrez, poderia ser o número de jogos vencidos. O aprendiz acumula conhecimento sobre *como* melhorar a realização de uma tarefa.

Os sistemas de Aprendizado de Máquina têm sido bem-sucedidos em tarefas de *data mining*, como aplicações na área de medicina. O Aprendizado de Máquina pode assumir um papel importante em analisar bancos de dados médicos para suporte a decisões. Por exemplo, um aprendiz pode encontrar modelos que ajudem a identificar complicações em operações de alto risco.

Muitas vezes, o mecanismo dos sistemas *aprendizes* se traduz em examinar modelos "candidatos" - árvores de decisão, redes neurais ou cláusulas lógicas - para encontrar bons modelos que caracterizem um conjunto de dados positivo para um determinado problema.

O ato de explorar este conjunto de modelos candidatos (conjunto também denominado *espaço de procura*) é muitas vezes dispendioso do ponto de vista computacional. Mais ainda, uma análise completa pode exigir a execução de um número massivo de experimentos - que podem gerar grande quantidade de *jobs* - e a análise de uma grande quantidade de dados. Naturalmente, se tais experimentos fossem rodados em seqüência, poderiam levar meses ou mesmo anos para produzirem resultados. Então, um ambiente de *paralelismo* é necessário para que se obtenha bom desempenho nestes experimentos. Na verdade, às vezes um ambiente de paralelismo é a *única* maneira que existe para levar a cabo experimentos com estas características [13].

Este trabalho servirá de apoio para uma série de experimentos em aprendizado de máquina, que serão brevemente descritos abaixo.

Tradicionalmente, algoritmos de aprendizado de máquina dividem o conjunto de dados disponível em um *training set* e um *test set.* Estes algoritmos usam os dados no *training set*, extraindo um modelo que é avaliado com o *test set.* Nos experimentos mencionados, é usada a técnica de *cross-validation* [30], que consiste em dividir o *training set* em vários subconjuntos, alocados em k pastas. Então os dados são treinados k vezes, cada vez em k-1 pastas. O subconjunto que não é

utilizado a cada vez é usado para avaliar a precisão do modelo. Além do valor de k, o método para se dividir o *training set* também pode ser escolhido (usualmente entre os métodos *block*, *round-robin* e *random*).

Dentro de cada uma das k pastas, pode-se aplicar a técnica de *cross-validation* mais uma vez, criando l subpastas. Isto auxilia na realização do *tuning*, isto é, a tentar achar o melhor conjunto de parâmetros para cada pasta.

Com freqüência, os sistemas de aprendizado de máquina produzem *classificadores*, que são programas que classificam uma entrada de acordo com um modelo. Existem classificadores que combinam as predições de vários outros classificadores, produzindo uma predição simples (*ensembles*). Há vários métodos de geração de *ensembles*, entre os quais estão o *bagging* [3], o *boosting* [19] e o método de *different seeds*.

Esta seção teve por objetivo descrever, de forma sucinta, o formato de aplicações onde a ferramenta, produto desta tese, é aplicada. Vejamos os aspectos principais de seu funcionamento logo a seguir.

4.3 O Funcionamento

Apresentamos nesta seção, numa ordem lógica, os eventos que caracterizam o funcionamento de nossa ferramenta no contexto de aprendizado de máquina.

4.3.1 Entrada de parâmetros pelo usuário

Para que a submissão de *jobs* ao ambiente Condor se concretize, o usuário precisará fornecer alguns parâmetros por meio de campos em uma página HTML, denominada *Página de entrada* e representada na Figura 4.2. A maioria corresponde às aplicações de aprendizado de máquina já mencionadas na Seção 4.2. Logo a seguir, é dada uma relação desses parâmetros:

1. `Diretório de trabalho` - É o diretório-raiz para a montagem da estrutura de diretórios contendo os dados preparados, para serem utilizados pelos *jobs*;

2. `Nome da aplicação` - É o nome da aplicação que gerou o conjunto de dados (*dataset*) inicial, dados que são posteriormente preparados;

3. `Diretório da aplicação` - Caminho para se encontrar o conjunto de dados inicial;

4. `Número de pastas`;

5. `Número de subpastas` para o *tuning*;

6. `Número de iterações`;

7. `Tipo de experimento`;

8. `Lógica utilizada` - Tipo de aprendizagem usado (teorias clausais ou cláusulas);

9. `Forma de divisão`;

10. `Sistema aprendiz` - É o sistema aprendiz que será usado. No momento, este trabalho lida somente com o sistema Aleph [29].

11. `Sistema operacional` - Este é um parâmetro especial para o sistema Yap [33] indicando qual será a versão do mesmo a ser usada, visto que a linguagem Prolog é aplicada na fase de preparação de dados;

12. `Parâmetros do sistema Aleph` - Informações de entrada para o Aleph;

13. *`Accuracies`* (precisões) e `tamanhos de cláusula` - Graus de precisão para se fazer o *tuning*, e tamanho dos modelos a serem examinados para cada grau.

Após o fornecimento de parâmetros, o usuário clicará um botão na parte inferior da página, para enviá-los a uma determinada máquina pertencente a um *pool* do Condor, que denominamos *máquina de entrada*. Esta máquina de entrada contém alguns *Servlets* Java [26], sendo que um deles, o *Servlet de tratamento*, é responsável por:

1. Receber os parâmetros fornecidos pelo usuário através da página HTML e escrevê-los em um arquivo-texto, o *arquivo de dados*;

2. Depois da criação do arquivo-texto, chamar o programa *Preparador / Submissor* que possui algumas funções, explicadas na próxima subseção;

3. Quando o programa anterior termina, retorna uma saída a este servlet (uma mensagem do Condor), que a trata e passa ao usuário também em formato HTML, pela *Interface de status*.

Além dos Servlets, a máquina de entrada contém toda a parte estrutural da nossa ferramenta, conforme a representação da Figura 4.1.

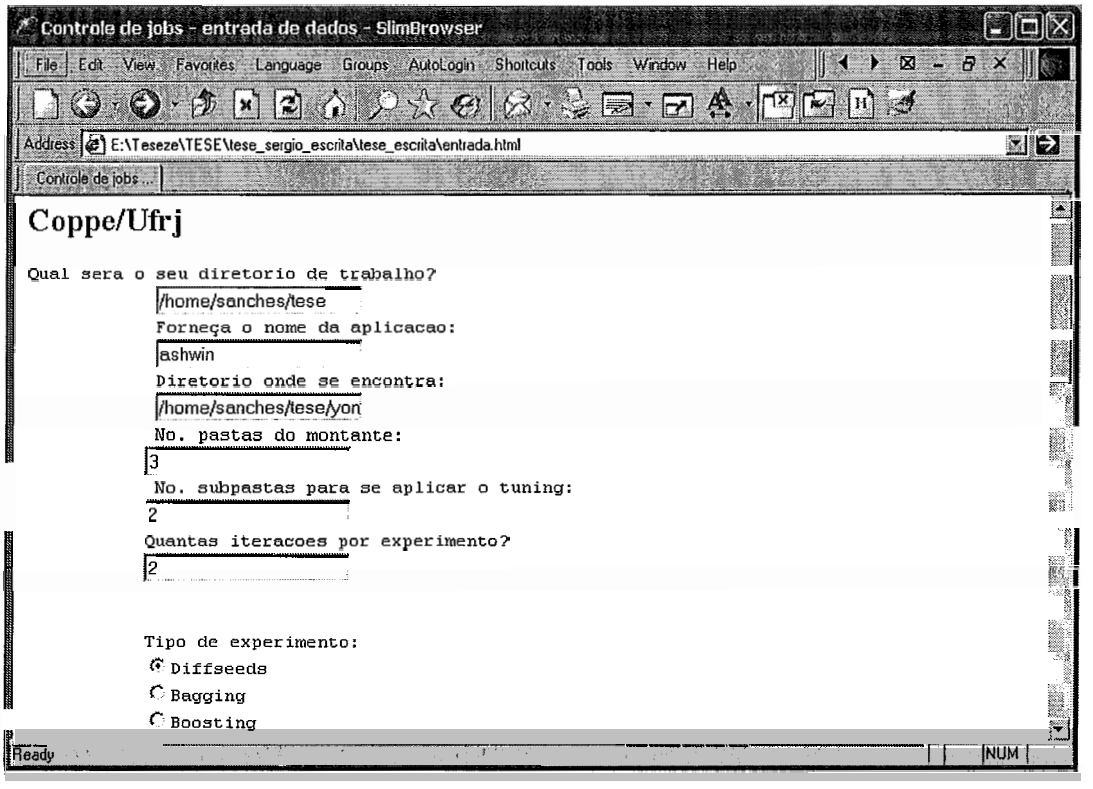


Figura 4.2: A página HTML para os parâmetros do usuário

4.3.2 Criação da estrutura de diretórios e submissão de tarefas

Assim que é chamado pelo servlet, o *Preparador / Submissor* inicia o seu trabalho. Uma *Estrutura de diretórios* é criada em primeiro lugar, com base em alguns dos parâmetros fornecidos na *Página de entrada.* A estrutura básica é formada de acordo com a Figura 4.3, onde k é o número de pastas, l é o número de subpastas e Acc (*accuracy*) é uma precisão dada pelo usuário.

Nesta *Estrutura de diretórios*, os dados da aplicação escolhida pelo usuário são copiados para o diretório "fontes" onde é aplicado, posteriormente, o *cross-validation*, de acordo com a técnica de divisão (*block*, *round-robin* e *random*) e o número de pastas escolhidos pelo usuário. A seguir, preparam-se os dados para o *tuning* de acordo com o número de subpastas, usando a mesma técnica de divisão. O *tuning* é feito com três graus de precisão diferentes, criando-se diretórios novos para cada um deles em cada subpasta. Este número de graus de precisão poderá ser transformado em mais um parâmetro para escolha do usuário.

Agora, consideremos que o usuário forneceu as seguintes informações: 3 pastas, 2 subpastas e 2 iterações por experimento, com os três graus de precisão 0.7, 0.9 e

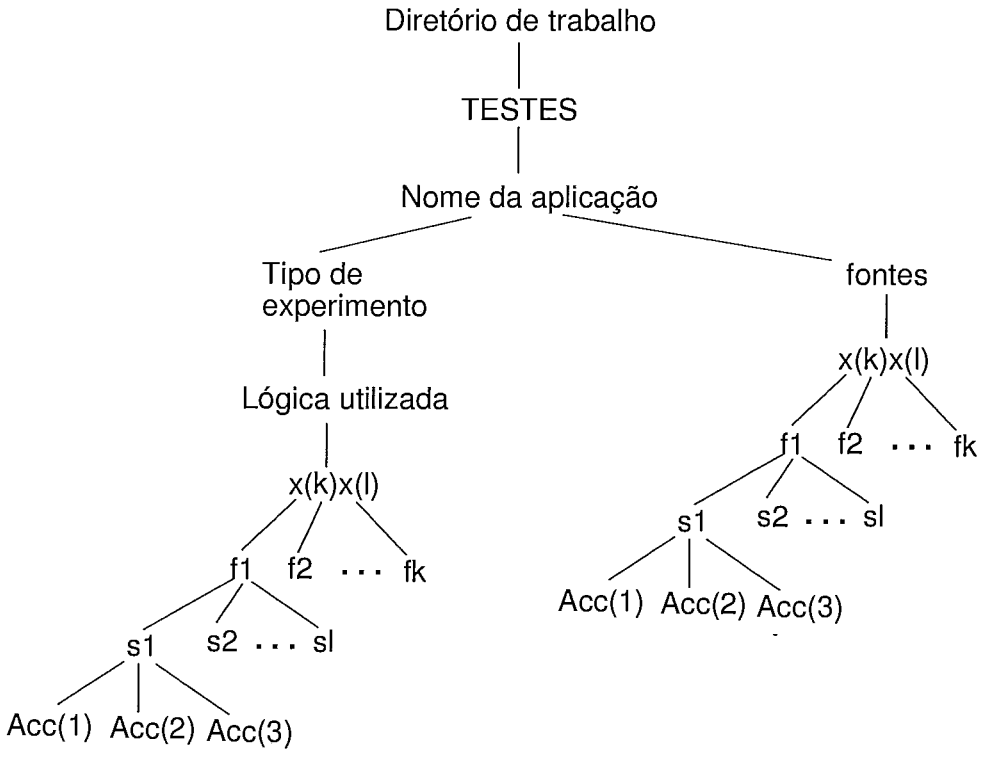


Figura 4.3: A estrutura de diretórios

1.0. Neste caso, o total de *jobs* submetidos adiante é igual a (3*2*2*3)= 36. Uma breve descrição das técnicas de *tuning* e *cross-validation* é dada na Seção 4.2.

Concluída a divisão e a preparação para o *cross-validation* e o *tuning*, os dados preparados são copiados para a estrutura de diretórios nomeada com o tipo de experimento que o usuário escolheu. A seguir, o *Preparador / Submissor* cria os arquivos-fonte (condor.in) para cada tarefa. Estes arquivos contém consultas que são executadas num sistema Prolog.

Finalmente, o *Preparador / Submissor* submete a quantidade de tarefas estipulada ao *pool* do Condor, formado por várias máquinas. Após a submissão, o *Preparador / Submissor* cria um banco de dados MySQL `jobscondor` contendo duas tabelas: a tabela de `pendentes` (c) e a tabela de `completos` (d).

Cada upla na tabela de `pendentes` contém informações sobre um *job* que foi submetido ao Condor, porém ainda não terminado. Já uma upla na tabela de `completos` guarda informações sobre um *job* que já tenha terminado com sucesso. Ao ser criada a tabela de `pendentes`, esta possui uplas relativas a todos os *jobs* submetidos, enquanto a tabela de `completos`(d) está vazia.

A Figura 4.4 mostra os campos das tabelas de `pendentes` e de `completos`. Nas

duas tabelas, os campos **jobid**, **data**, **hora**, **arqsaida** e **vezes-sub** significam, respectivamente, o número identificador do *job* no Condor, a data e a hora de sua submissão, o nome do arquivo de saída cuja sintaxe deve ser verificada (caminho completo) e um controle sobre o número de vezes em que o *job* foi re-submetido até então.

Na tabela de `pendentes`, há dois campos extras: **condorin**, que indica o diretório onde se encontram os arquivos condor.in de cada *job*, e **id-interno**, uma identificação interna para cada *job* a ter sido submetido. Estes campos serão de grande importância quando for necessária a re-submissão de um *job*.

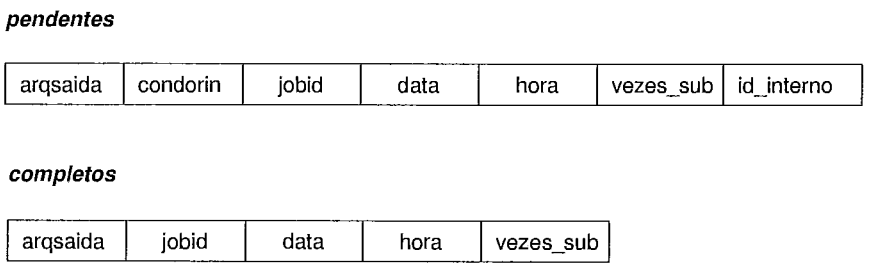


Figura 4.4: Os campos das tabelas de *jobs* pendentes e *jobs* completos

O *Preparador / Submissor* foi escrito totalmente em linguagem C. Quando se faz a divisão de dados, esse programa faz chamadas a *scripts* em outras linguagens (para as técnicas de *random* e *round-robin*, em AWK [1] e para a técnica de *block*, em Prolog). Para a criação do banco de dados MySQL, foi usada uma API (*Application Programming Interface*) que possibilita a integração da linguagem C com essa ferramenta.

Ao fim da execução do *Preparador / Submissor*, o mesmo retorna uma saída do Condor para o *Servlet de tratamento*, que por sua vez a repassa ao usuário por meio de uma página HTML, que chamamos de *Interface de status*. Este nome se deve ao fato desta página possuir dois botões para que o usuário acompanhe o andamento dos *jobs* submetidos, como mostra a Figura 4.5.

4.3.3 Monitoramento das tarefas

Para monitorar a execução dos *jobs* recém-submetidos, um *daemon*, denominado *Verificador / Resubmissor*, é colocado em ação. Suas tarefas consistem em:

1. Re-submeter *jobs* falhos ou terminados sem produzir as saídas esperadas;

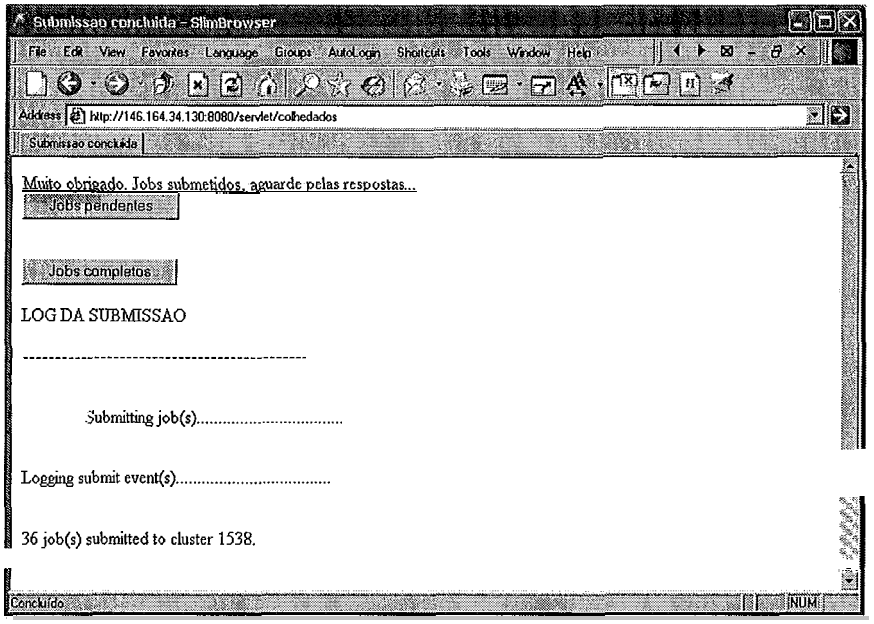


Figura 4.5: Uma nova página HTML para acompanhar o andamento dos *jobs*

2. Atualizar as tabelas do banco de dados `pendentes` e `completos`, à medida em que os *jobs* são terminados corretamente.

A vida do *Verificador / Resubmissor* consiste em, de tempos em tempos, extrair as informações dos *jobs* na tabela `pendentes` verificar o término destes *jobs* no ambiente Condor, a integridade de suas saídas - gravadas na estrutura de diretórios - e re-submetê-los ou incluir uma upla com as informações do *job* na tabela `completos`, em caso de término bem-sucedido.

Vejamos os passos de funcionamento do *Verificador / Resubmissor*, em forma de algoritmo;

Enquanto `pendentes` não for vazia
    `conjuntouplas` := todas as uplas da tabela `pendentes`
    Enquanto `conjuntouplas` não chegar no fim
        extrair **upla** de `conjuntouplas`
        Se **upla.jobid** não estiver na fila de jobs do condor
            Então apagar **upla** de `pendentes`
            Se **upla.arqsaida** existe
                Então verificar sintaxe de **upla.arqsaida**
                Se sintaxe = OK
                    Então incluir **upla** na tabela `completos`
                fim-Se
                Senão *resubmeter-job*(**upla.condorin**, **upla.vezes-sub**, **upla.id-interno**)
                    incluir **nova upla** na tabela `pendentes`

```
                fim-Senão
            fim-Se
            Senão resubmeter-job(upla.condorin, upla.vezes-sub, upla.id-interno)
                incluir nova upla na tabela pendentes
                fim-Senão
        fim-Se
        Senão
            tempo-na-fila := data-atual:hora-atual - upla.data:upla.hora
            Se tempo-na-fila >= tempo-limite
                Então remove o job do Condor, usando upla.jobid
                resubmeter-job(upla.condorin, upla.vezes-sub, upla.id-interno)
                apagar upla de pendentes
                incluir nova upla na tabela pendentes
            fim-Se
        fim-Senão
    fim-Enquanto
    esperar alguns minutos
fim-Enquanto
```

A função *resubmeter-job* merece uma análise mais completa. Ela recebe como parâmetros os valores **upla.condorin**, **upla.vezes-sub** e **upla.id-interno**, relativos ao *job* que, pelos motivos expostos no algoritmo anterior (inexistência de saída, saída corrompida ou término do tempo-limite na fila) será resubmetido.

Usando o valor **upla.condorin**, é possível encontrar o diretório onde estão os arquivos condor.in.* para os jobs. Um valor **upla.id-interno** determina *qual* desses arquivos condor.in.* é o arquivo de entrada correspondente ao *job* que falhou. Por exemplo, se o valor **upla.id-interno** for igual a 7, então o arquivo de entrada do *job* tem nome condor.in.7. Este arquivo servirá como entrada para a nova submissão.

Quando a nova submissão de um *job* era feita, verificaram-se falhas esporádicas no *central-manager* do Condor, de forma que o contato com ele era perdido e a submissão não podia ser feita. Devido a isto, foi inserido um tratamento para este erro na função *resubmeter-job*, pelo qual se repete a tentativa de submissão até que o *central-manager* esteja ativo novamente.

Após a resubmissão do *job* ao Condor, uma nova upla é criada e incluída na tabela `pendentes`, com a exclusão da antiga. O *job* resubmetido possui um novo **jobid** e novas **data** e **hora** de submissão. Contudo, mantém os valores **condorin**, **id-interno** e **arqsaida** da upla excluída.

O valor **vezes-sub** é incrementado em relação à upla antiga; permite-se que este número chegue a 3 vezes, que é o limite de resubmissões de *jobs* estipulado. No

entanto, este número também poderá ser implementado como um parâmetro a ser fornecido pelo usuário.

Como mostrado no algoritmo, se o *job* terminar com sucesso e seu arquivo de saída estiver consistente, uma nova upla é criada na tabela `completos` com as suas informações.

O *Verificador / Resubmissor* também foi construído em linguagem C, usando a API de ligação ao MySQL já mencionada.

Enquanto o *daemon* trabalha nas verificações, re-submissões e atualizações das tabelas no banco de dados, a *Interface de status* serve como um informativo ao usuário que deseja saber do andamento dos *jobs*, de acordo com a Figura 4.5.

O primeiro botão da página (onde se lê "Jobs Pendentes"), ao ser pressionado, **acionará mais um Servlet Java.** O *Mostrador de jobs pendentes*, por mérito de uma API e de um *driver* JDBC (*Java DataBase Connectivity*), faz a leitura de todas as uplas existentes na tabela `pendentes` e retorna valores de cada uma dessas uplas (como **jobid**, **data** e **hora**) ao usuário em formato HTML. Por sua vez, o segundo botão da página ("Jobs Completos") aciona um Servlet Java semelhante, o *Mostrador de jobs completos* que faz a leitura da totalidade das uplas na tabela `completos`. A Figura 4.6 mostra o resultado de uma pesquisa a respeito dos *jobs* ainda na fila do Condor.

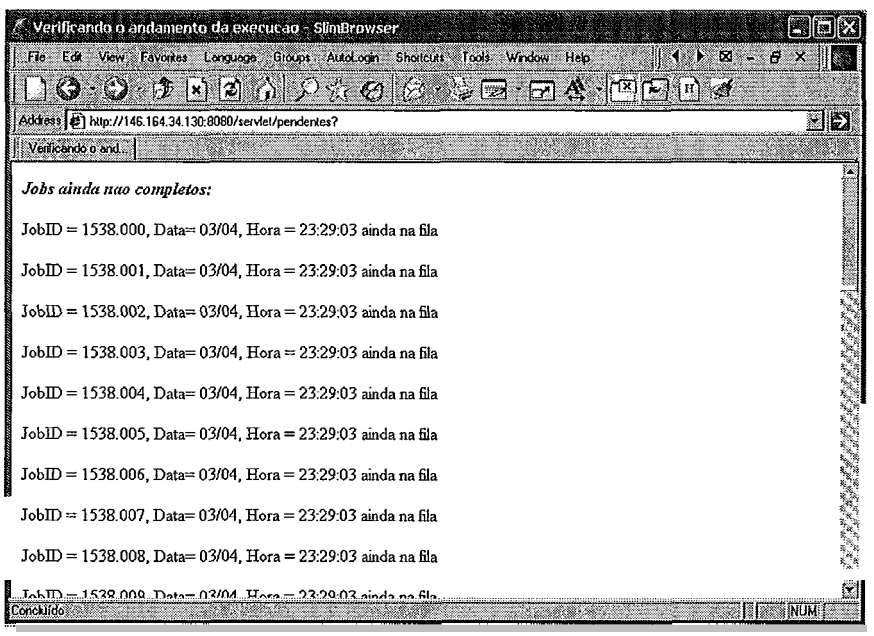


Figura 4.6: Resposta da ferramenta à pesquisa sobre *jobs* pendentes

4.3.4 Controle de dependências

O Condor DAGman foi integrado a este trabalho com o objetivo de mostrar que as dependências entre os *jobs* podem ser controladas de forma automática, sem que o usuário precise escrever nenhum tipo de arquivo de submissão. Uma nova versão contendo o mecanismo do Condor DAGman foi criada, e um exemplo simples é descrito no próximo capítulo, que trata dos experimentos realizados.

4.4 Comentários

O protótipo criado por Dutra *et al*, a base para esta ferramenta, consistiu em um programa que criava uma estrutura de diretórios semelhante à descrita na Subseção 4.3.2 e em um *daemon* verificador de *jobs* falhos. O protótipo não possuía ainda a parte de submissão automática de *jobs*, e além disso, a verificação de erros tinha de ser executada manualmente. O programa construtor da estrutura foi escrito na linguagem *Shell Script* do sistema Linux, e o *daemon* foi criado na linguagem AWK [1].

Este protótipo foi testado num ambiente Condor na Universidade de Wisconsin-Madison, apresentando resultados satisfatórios [13]. Deste protótipo, foram reaproveitados os *scripts* para divisão citados anteriormente, na Subseção 4.3.2.

A ferramenta apresentada neste capítulo possui pontos a serem melhorados. Um deles é o fato da ferramenta estar limitada ainda ao escopo das aplicações de Aprendizado de Máquina, sendo que no futuro desejamos dar a ela um caráter mais geral, desenvolvendo-a para mais tipos de aplicações. Pretende-se, também, melhorar a interface HTML, com relação aos seguintes pontos:

1. Possibilitar que o usuário informe o grafo de dependências que traduza a natureza dos seus experimentos, possivelmente através de uma interface gráfica. Este é um ponto importante a ser trabalhado, embora tenhamos a consciência de que definir grafos de dependências para experimentos com um número massivo de tarefas não é trivial;

2. Da mesma maneira, fazer com que o usuário possa informar que sistema escalonador deseja utilizar (por ora, só usamos o Condor), com base em dados de comportamento - assim como o tempo médio de execução de *jobs*

ou tempos médios nas filas de escalonamento - que revelarão os desempenhos dos escalonadores disponíveis num *pool* de máquinas;

3. Retornar ao usuário os arquivos de saída produzidos pelos experimentos, e também os arquivos de entrada (condor.in), no caso de tarefas que gerem erros constantes, ultrapassando o número de resubmissões;

4. Cuidar das autenticações ao *pool* de máquinas, construindo uma página preliminar à *Página de entrada*, onde o usuário será autenticado por meio de um *login* e uma senha.

O objetivo de todas estas medidas será fornecer, cada vez mais, transparência e confortabilidade para o usuário ao lidar com um ambiente de grade.

Os motivos da escolha da linguagem C para escrever o construtor da estrutura e o *daemon* foram a sua portabilidade e robustez. Já o sistema de banco de dados MySQL foi adotado para que se obtivesse praticidade no manuseio de informações, visto que os sistemas de bancos de dados provêem mecanismos de seleção, inserção e exclusão de dados convenientes ao usuário. De mais a mais, APIs de integração com as linguagens C e Java tornam o MySQL ideal para este ambiente.

No Capítulo 5, apresentamos os experimentos realizados num *pool* Condor com o uso da ferramenta proposta.

Capítulo 5

Resultados Experimentais

Este capítulo tem por objetivo verificar a eficácia da ferramenta apresentada, bem como analisar o comportamento do escalonador Condor na alocação dos jobs a recursos (máquinas).

5.1 O Ambiente

Para a extração destes resultados, utilizamos um *cluster* de oito máquinas, cada qual com: um processador Pentium III cuja velocidade é 1.5 Ghz, uma memória RAM com capacidade de 1.2 Gbytes e um disco rígido com capacidade de 8 Gbytes. Somente uma máquina deste *cluster* possui contato direto com a Web e a rede interna da Coppe/Sistemas, através de um número IP externo. Quanto às restantes, cada uma delas possui um número de identificação interno, sem contato direto com o meio exterior.

Neste *cluster*, um *pool* Condor foi instalado e configurado, sendo que a versão de instalação deste escalonador utilizada foi a 6.4.7.

A máquina que possui um IP externo é a máquina que contém toda a estrutura da ferramenta, ou seja, a *máquina de entrada*, termo explicado na Seção 4.1. Esta máquina também cumpre a função de *central-manager* do *pool* Condor, definida na Seção 3.2. Todas as máquinas, neste caso, são configuradas como máquinas capazes de submeter tarefas e executá-las no *pool*. No âmbito da ferramenta, porém, a *máquina de entrada* é a única responsável por submeter tarefas.

5.2 Medidas de desempenho

Com o propósito de analisar a eficácia de nossa ferramenta no monitoramento de jobs falhos, algumas submissões de jobs foram feitas a este *pool*, com a devida preparação dos dados para execução e a monitoração por parte do *daemon*. Destas submissões, foram extraídas algumas medidas - em especial, o número de jobs que falharam, o tempo médio de alocação de *jobs* a máquinas no *pool* e o tempo médio de término dos *jobs*.

As Tabelas 5.1, 5.2, 5.3 e 5.4 mostram as características das séries de experimentos realizadas e algumas estatísticas de execução. Os experimentos que realizamos foram aprendizagens de características de componentes carcinogênicos em ratos.

Tabela 5.1: Os resultados da primeira série de experimentos, com tamanho de cláusula igual a 4

Categoria	**Valores**
Total geral de *jobs*	108
Jobs terminados	108
Jobs terminados sem re-submissões	106
Jobs terminados com uma re-submissão	2
Jobs terminados com duas re-submissões	0
Jobs terminados com três re-submissões	0
Jobs não terminados	0
Média de tempo de alocação de *jobs*	51 min, 15 seg
Média de tempo de término de *jobs*	57 min, 9 seg

Tabela 5.2: Os resultados da segunda série de experimentos, com tamanho de cláusula igual a 4

Categoria	**Valores**
Total geral de *jobs*	216
Jobs terminados	216
Jobs terminados sem re-submissões	212
Jobs terminados com uma re-submissão	4
Jobs terminados com duas re-submissões	0
Jobs terminados com três re-submissões	0
Jobs não terminados	0
Média de tempo de alocação de *jobs*	1 h, 26 min, 58 seg
Média de tempo de término de *jobs*	1 h, 32 min, 11 seg

Tabela 5.3: Os resultados da terceira série de experimentos com tamanho de cláusula igual a 4

Categoria	**Valores**
Total geral de *jobs*	252
Jobs terminados	252
Jobs terminados sem re-submissões	251
Jobs terminados com uma re-submissão	1
Jobs terminados com duas re-submissões	0
Jobs terminados com três re-submissões	0
Jobs não terminados	0
Média de tempo de alocação de *jobs*	1 h, 48 min, 58 seg
Média de tempo de término de *jobs*	1 h, 52 min, 35 seg

Tabela 5.4: Os resultados da quarta série de experimentos, com tamanho de cláusula igual a 5

Categoria	**Valores**
Total geral de *jobs*	126
Jobs terminados	123
Jobs terminados sem re-submissões	97
Jobs terminados com uma re-submissão	20
Jobs terminados com duas re-submissões	5
Jobs terminados com três re-submissões	1
Jobs não terminados	3
Média de tempo de alocação de *jobs*	5 h , 30 min , 14 segs
Média de tempo de término de *jobs*	6 h , 17 min , 30 segs

O *tempo de alocação* de um *job* é o tempo decorrido entre a sua submissão ao ambiente Condor e a alocação de um recurso (máquina) ao *job*, enquanto o *tempo de término* é aqui definido como o tempo decorrido entre a submissão de um job e o tempo em que o mesmo termina sua execução.

A primeira série de experimentos dispara 108 tarefas, com tamanho de cláusula igual a 4, ao passo que a segunda e a terceira disparam 216 tarefas e 252 tarefas respectivamente, com igual tamanho de cláusula. O tempo de execução de um *job* nos experimentos de Aprendizado de Máquina é diretamente influenciado pelo tamanho da cláusula - este tempo aumenta de forma exponencial conforme esse tamanho cresce, e isto é claramente notado na quarta série de experimentos. Apesar de disparar poucos *jobs* (126), a quarta série mostra tempos médios de alocação e término de *jobs* três vezes maiores que os da terceira, que tem o dobro de *jobs*. Esta

quarta série de experimentos utiliza tamanho de cláusula igual a 5.

Segundo as Tabelas 5.1, 5.2 e 5.3, houve *jobs* falhos nas três primeiras séries (2 na primeira, 4 na segunda e 1 na terceira), o que significou detectá-los e resubmetê-los, sem nenhum tipo de intervenção por parte do usuário. Isto significa que não houve a necessidade de percorrer centenas de resultados para saber se alguma saída deixou de ser gerada ou se alguma saída gerada estava corrompida. Nenhum destes *jobs* falhos foi resubmetido por término de limite de tempo (que neste caso foi estabelecido como 1 dia inteiro na fila) ou foi resubmetido mais de uma vez.

Na série de experimentos com tamanho de cláusula igual a 5, a necessidade de automatização se faz ainda mais presente. Uma quantidade de 20,6% das tarefas apresentaram alguma espécie de erro. Pelo que foi observado, estes erros resultaram em falhas nos arquivos de saída das tarefas. Além disso, 2,4% das tarefas não puderam terminar devido ao número-limite de re-submissões (3) ter sido ultrapassado.

Esta série de experimentos mostra que as tarefas com tamanho de cláusula maior (e tarefas grandes em qualquer contexto) são mais sujeitas a erros. Uma vez que o Yap e o Condor não informam códigos de erro para as tarefas que falharam, não foi possível informar ao usuário a causa dos erros. Porém, ao melhorarmos o sistema, pretendemos reportar os arquivos de entrada (condor.in) destes *jobs* ao usuário para que este possa investigar essa causa com mais detalhes.

5.2.1 Testes com o Condor DAGman

O Condor DAGman foi incluído neste trabalho com o objetivo de detectar as dependências ocorridas entre aplicações. Uma vez que as tarefas de Aprendizado de Máquina testadas são independentes entre si, associamos a cada uma delas uma tarefa inócua (que produz uma mensagem de "Hello World") formando uma relação de dependência como aquela disposta anteriormente, na Figura 3.3. Em outras palavras, cada uma das tarefas inócuas é dependente de uma tarefa de *Machine Learning*. O serviço de construir estas dependências é automaticamente feito por uma versão de nossa ferramenta, que cria um arquivo na sintaxe do DAGman e o chama logo a seguir. Ao usuário basta informar o nome do executável a partir do qual são disparados os *jobs* dependentes.

Em nossos testes, foi comprovado que todas as tarefas inócuas só iniciavam sua execução depois que as suas tarefas-pais de *Machine Learning* terminavam. Optou-se

por usar a instrução RETRY com um limite de três vezes para re-submissão de um nó, para o caso de haver algum erro de sistema (como um desmonte de sistema de arquivos). Porém, para as saídas inexistentes ou com sintaxe errada, a estrutura da ferramenta já previa um cuidado especial; de sorte que não foi embutida na ferramenta nenhuma chamada a algum *script* POST (entrada definida na seção 3.2.1) para verificar a saída dos *jobs*.

5.2.2 Análise comportamental do escalonador Condor

Além de testar a ferramenta, julgamos conveniente fazer uma análise do sistema escalonador usado nas séries de experimentos, para que pudéssemos estar cientes de limitações ao fazer o escalonamento das tarefas para o *pool*.

Para tal, a terceira série de experimentos, representada na Tabela 5.3, foi tomada como base, tendo gerado 252 tarefas com tamanho de cláusula 4. A partir desta série, foi construído um gráfico, que está representado na Figura 5.1. Afora pela máquina de entrada, que estava sendo usada por um usuário na maior parte do tempo, e por poucos acessos rápidos em três máquinas internas, o nosso *pool* esteve praticamente todo dedicado à série de experimentos, com um pico de uso de 7 máquinas ao mesmo tempo.

O eixo horizontal (x) deste gráfico traduz o universo do tempo *total* de execução, em segundos, do conjunto de 252 tarefas, que vai de zero (o momento em que foram disparadas) até 12.327 (o tempo em que a última tarefa termina), ou 3 horas, 29 minutos e 50 segundos. Por sua vez, o eixo vertical (y) representa a totalidade de tarefas, identificadas por números que vão de 0 a 252.

O *tempo de execução* de uma determinada tarefa é conhecido por uma reta horizontal, ligando dois pontos no gráfico:

(número da tarefa, tempo alocação a uma máquina no Condor) e *(número da tarefa, tempo de término da execução).*

Quando um *job* sofre um *checkpoint*, sendo transferido para execução em outra máquina, divide-se o tempo de execução em duas retas. A reta para a primeira parte é formada desde o ponto onde o *job* é alocado até o ponto onde o *checkpoint* ocorre. Já a segunda reta é formada desde o ponto em que o *job* é alocado pela segunda vez até o ponto em que ocorre o seu término. Claramente, um job poderá sofrer

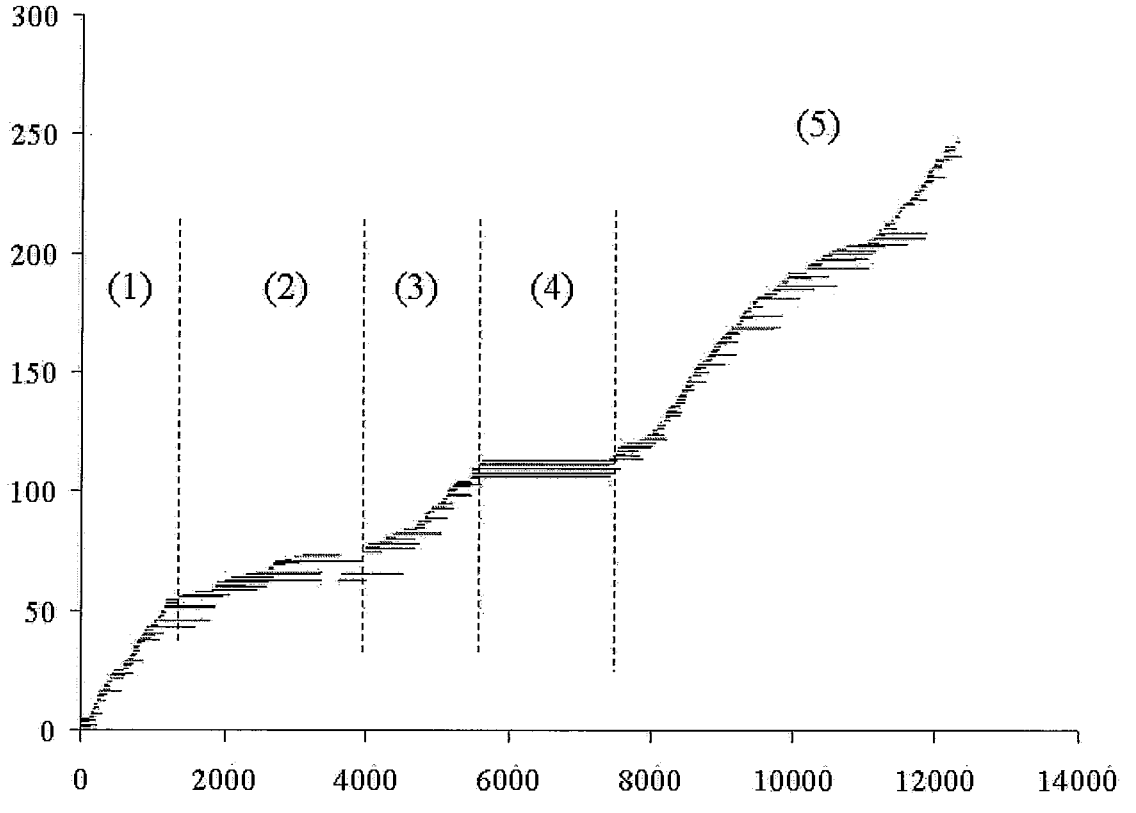


Figura 5.1: Representação gráfica das execuções no Condor de todas as tarefas de uma série de experimentos. Uma linha horizontal equivale ao tempo de execução de uma tarefa (em segundos), e um valor no eixo vertical equivale ao número da tarefa.

mais de um *checkpoint* durante sua execução, caso que não aconteceu nesta série de experimentos.

Levando em conta este panorama, e estando o nosso *cluster* praticamente todo dedicado aos experimentos - apenas a máquina de entrada estava sendo usada durante alguns períodos - apresentamos logo a seguir uma descrição dos eventos relevantes observados na Figura 5.1.

Tabela 5.5: Taxas de alocação de tarefas por trecho

Trecho	**Taxas de escalonamento (1 tarefa/no.segundos)**
(1)	24,67
(2)	115,07
(3)	40,22
(4)	—
(5)	36,87

Nos primeiros 1.382 segundos, representados pelo trecho (1), a taxa média de escalonamento foi de uma tarefa disparada a cada 24,67 segundos. Veja os valores dispostos na Tabela 5.5. Já no trecho (2), houve uma queda significativa nesta taxa (1 tarefa a cada 115,07 segundos), com o aumento do tempo de execução das tarefas. Observando o trecho (3), a taxa média volta a aumentar, para o valor de uma tarefa a cada 40,22 segundos.

Um fenômeno curioso foi observado na faixa de tempo (4). Neste intervalo, nenhuma alocação foi feita, e as tarefas já alocadas tiveram um tempo de execução excessivamente longo. Analisando mais profundamente, foi constatado que cada uma destas tarefas saiu da máquina onde estava e foi alocada em outra máquina, em "saltos", sem que fosse feito um *checkpoint*. As ocorrências destes "saltos" não constam no arquivo de *log* que o Condor cria para relatar os eventos que ocorrem na execução de tarefas de um usuário.

No último trecho, a taxa volta a se estabilizar com relação ao início (1 tarefa a cada 36,87 segundos), sendo que ela volta a ter uma ligeira diminuição entre 9.489 e 11.031 segundos, novamente com o aumento do tempo de execução das tarefas.

Uma conclusão que pode ser tirada deste estudo é que o escalonador Condor pode apresentar irregularidades em fazer com que determinada tarefa, uma vez alocada, execute sem interrupções; é sabido que, se uma máquina ficar ocupada repentinamente (algum usuário abrindo um terminal, por exemplo), isto pode aumentar o tempo de execução de uma tarefa, que poderia receber a ação de um *checkpoint* ou parar e continuar a execução na mesma máquina pouco depois. Contudo, as irregularidades se apresentaram mesmo nas máquinas que estavam desocupadas nos trechos (2), (4) e (5). É importante frisar que, durante o trecho (4), nem mesmo a máquina de entrada estava ocupada.

Com estes incidentes, o desempenho dos experimentos pode ser comprometido. Uma tarefa como as que submetemos, com tamanho de cláusula igual a 4, pode levar por volta de 3 minutos para terminar (180 segundos), executada sozinha. Se executarmos as 252 tarefas disparadas pelos experimentos com uma máquina apenas, teremos o tempo total de: 252*180 = 45.360 segundos.

Por outro lado, o tempo total nesta série de experimentos com o Condor, com 252 tarefas num pico de 7 máquinas, é de 12.327 segundos. Portanto, o *speedup* obtido pelo ambiente de grade com o Condor é igual a: 45.360/12.327 = 3,67. Este valor mostra que os benefícios do ambiente multiprocessado podem não estar sendo bem

aproveitados, o que pode justificar a possibilidade de o usuário escolher o escalonador a ser utilizado, dependendo da situação apresentada.

5.3 Síntese e discussão

Os resultados obtidos provaram a eficácia da ferramenta na gerência automática de *jobs* em execução, e apontaram possíveis deficiências no *software* de escalonamento Condor.

Com relação ao uso do DAGman, é necessário incluir ainda outros tipos de dependências para serem manipulados automaticamente. Após isto ter sido feito, uma possibilidade é fazer com que o usuário possa fornecer informações sobre o grafo de dependência mais adequado às suas tarefas, via uma interface HTML. Naturalmente, existe a consciência de que isto não é uma ação trivial quando se trata de um número massivo de tarefas.

Escolher o tipo de sistema escalonador de tarefas a ser usado é uma das possibilidades para a nossa ferramenta. Com base em análises como a última, pode-se orientar o usuário para escolher o sistema escalonador para submeter seus *jobs* - é claro que, dependendo da natureza do ambiente de execução, um escalonador pode levar vantagem sobre outro. A idéia é deixar o usuário ciente destes fatos, para que possa extrair o melhor dos ambientes de grade.

Capítulo 6

Conclusões e Trabalhos Futuros

Neste texto, apresentamos uma ferramenta para permitir a realização de grande quantidade de experimentos num ambiente de grade. O nosso trabalho baseia-se nas idéias originalmente propostas por Dutra *et al* [13], e inclui as seguintes contribuições:

- integração de todos os componentes do sistema numa única ferramenta flexível; e
- gestão totalmente automática de tarefas, incluindo uma recuperação automática de erros.

Nossos resultados mostraram a real ocorrência de erros em grandes séries de experimentos, como as mencionadas anteriormente, e a eficácia desta ferramenta em resolvê-los.

Ainda existem muitos pontos a serem trabalhados nesta ferramenta e em trabalhos semelhantes, no sentido de se ter gerenciamento automático de tarefas de uma forma mais ampla. Por exemplo, uma possibilidade de melhora surge se ao usuário for dada a oportunidade de escolher entre vários escalonadores diferentes, com base em pré-análises como as realizadas no Capítulo 5 - atualmente, esta ferramenta trabalha com apenas um escalonador de tarefas, que é o Condor.

No momento, estamos a lidar apenas com as aplicações de Aprendizado de Máquina descritas. É relevante a expansão desta idéia para mais tipos de aplicações. Um dos trabalhos a serem desenvolvidos é, portanto, transformar a ferramenta, de maneira a torná-la de propósito geral.

É importante também desenvolver as capacidades do Condor DAGman na ferramenta. Atualmente, a versão contendo o DAGman trata apenas de um tipo

de dependências (no estilo da Figura 3.3, denominado *fortemente acoplado*), pois nosso objetivo se ateve a verificar a manipulação automática delas entre as tarefas ativas. Uma das idéias surgidas consiste em fazer com que o usuário escolha, através da interface Web, qual é o tipo de dependência que corresponderá ao seu tipo de problema.

A computação em grade ainda é uma área bastante nova, com um enorme potencial, em que existe muito a ser descoberto; cremos ser possível apontar uma série de possibilidades não pensadas para esta ferramenta, e mesmo transformá-la em algo maior. Porém, acreditamos ter dado um passo importante no sentido de automatizar o gerenciamento de tarefas, tornando o uso dos ambientes de grade mais conveniente. Desta forma, é provável que as grades possam, no futuro, servir satisfatoriamente a outros campos que não possuam afinidade nenhuma com ambientes computacionais, como a profissionais de ciências humanas.

Referências Bibliográficas

[1] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *The AWK programming language.* Addison-Wesley Longman Publishing Co., Inc., 1987.

[2] The Globus Alliance. http://www.globus.org. página web.

[3] L. Breiman. Bagging predictors. *Machine Learning*, 2(24):123–140, 1996.

[4] Henri Casanova and Jack Dongarra. Netsolve: A network server for solving computational science problems. Technical Report CS-95-313, University of Tennessee, Novembro 1995.

[5] IBM Autonomic Computing. http://www.research.ibm.com/autonomic. página web.

[6] Platform Computing. http://www.platform.com/products/overview.html. página web.

[7] F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the multics system. *AFIPS Joint Computer Conference*, pages 619–628, 1965.

[8] Joseph Czyzyk, Michael P. Mesnier, and Jorge J. Moré. The network-enabled optimization system (neos) server. Preprint MCS-P615-0996, Argonne National Laboratory, Argonne, Illinois, 1996.

[9] D. de Roure, M. A. Baker, N. R. Jennings, and N. R. Shadbolt. The evolution of the grid. 2002.

[10] Ewa Deelman, James Blythe, Yolanda Gil, and Carl Kesselman. Pegasus: Planning for execution in grids. GriPhyn Technical Report 20, University of Southern California, Novembro 2002.

[11] T. DeFanti, I. Foster, M. Papka, R. Stevens, and T. Kuhfuss. Overview of the i-way: Wide area visual supercomputing. *International Journal of Supercomputing Applications*, 10:123–130, 1996.

[12] D. Diachin, L. Freitag, D. Heath, J. Herzog, W. Michels, and P. Plassmann. Remote engineering tools for the design of pollution control systems for commercial boilers. *International Journal of Supercomputer Applications*, 10(2):208–218, 1996.

[13] I. Dutra, D. Page, V. Santos Costa, J. Shavlik, and M. Waddell. Toward automatic management of embarrassingly parallel applications. *Proceedings of International Conference on Parallel and Distributed Computing (Euro-Par)*, 2003.

[14] Sun Grid Engine. http://www.sun.com/software/gridware/. página web.

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

[16] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*, chapter Computational Grids, pages 15–51. Morgan Kaufmann Publishers, Inc., 1999.

[17] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid. *International Journal of Supercomputer Applications*, 15(3):472–602, 2001.

[18] I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. *14th Conference on Scientific and Statistical Database Management*, 2002.

[19] Y. Freund and R. Shapire. Experiments with a new boosting algorithm. In *Proceedings of the 14th National Conference on Artificial Intelligence*, pages 148–156. Morgan Kaufmann, 1996.

[20] A. S. Grimshaw and W. A. Wulf. Legion - a view from 50.000 feet. In *Fifth IEEE International Symposium on High Performance Distributed Computing, IEEE Computer Society Press, Los Alamitos, California*, Agosto 1996.

[21] G. Lindahl, A. Grinshaw, A. Ferrari, and K. Holcomb. Metacomputing: What's in it for me. 2002.

[22] M. J. Lizkow, M. Livny, and M. W. Mutska. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, Junho 1988.

[23] Globus Resource Allocation Management. http://www.unix-globus.org/developer/resource-management.html. página web.

[24] T. Mitchell. *Machine Learning.* McGraw-Hill, 1997.

[25] Z. Németh and V. Sunderam. A comparison of conventional distributed computing environments and computational grids. *International Conference on Computational Science (ICCS)*, 2:729–738, 2002.

[26] Java Servlet. http://www.java.sun.com. página web.

[27] A. Silberschatz and P. Galvin. *Operating System Concepts*, chapter 1, pages 14–20. John Wiley e Sons, Inc., 1999.

[28] L. Smarr and C. Catlett. Metacomputing. *Communications of the ACM*, 1(35):44–52, 1992.

[29] A. Srinivasan. *The Aleph Manual.* Oxford, 2001.

[30] M. Stone. Cross-validatory choice and assessment of statistical predictions. *Journal of The Royal Statistical Society B*, (36):111–147, 1974.

[31] Condor High-Throughput Computing System. http://www.cs.wisc.edu/condor. página web.

[32] Portable Batch System. http://www.openpbs.org. página web.

[33] The Yap Prolog System. http://www.ncc.up.pt/ vsc/yap/. página web.

[34] Andrew S. Tanembaum. *Sistemas Operacionais Modernos*, chapter 1, page 9. Prentice-Hall do Brasil, Ltda, 1995.

[35] D. Thain, T. Tannembaum, and M. Livny. Condor and the grid. *White Paper*, 2002.