Capítulo 2

Grades Computacionais: Conceitos Fundamentais e Casos Concretos

Fabio Kon, Alfredo Goldman

Abstract

Computationally-intensive problems that require large amounts of processing power are increasingly important for advances in science and technology in a wide variety of areas such as Physics, Chemistry, Biology, Engineering, and Finance. Up to the middle of the 1990s, these problems were resolved using large and highly-expensive parallel computers accessible to few research groups. Over the last 10 years, however, it became more efficient and inexpensive to invest in assembling and configuring clusters of inexpensive, commodity personal computers. These PC clusters can offer very large processing powers at a relatively low cost. Recently, the idea of linking multiple, geographically distributed clusters for the establishment of Computational Grids have been receiving more and more attention in both academia and industry. In this chapter, we present the basic concepts behind Grid Computing and describe some of the most important grid middleware systems and the programming models they support. Finally, we give a general overview of some of the most relevant grids currently in use.

Resumo

*Problemas computacionalmente pesados que demandam grande poder de processamento são cada vez mais comuns para o avanço da ciência e da tecnologia em áreas tão variadas quanto Física, Química, Biologia, Engenharias, Finanças, entre outras. Até meados da década de 1990, estes problemas eram resolvidos através de computadores paralelos de grande porte e altíssimo custo. Nos últimos 10 anos, no entanto, tornou-se mais eficiente e barato investir na montagem e configuração de aglomerados (*clusters*) de computadores de baixo custo tais como PCs que, trabalhando em conjunto, oferecem um alto poder de processamento a um custo relativamente baixo. Recentemente, a idéia de interconectar vários aglomerados dispersos geograficamente para a criação de grades computacionais vem ganhando força tanto na academia quanto na indústria. Nesse capítulo, apresentamos os conceitos básicos de Computação em Grade e descrevemos alguns dos principais sistemas de middleware para grades, bem como os modelos de programação que eles comportam. Finalizamos dando uma visão geral de algumas das principais grades hoje em funcionamento.*

2.1. Introdução

A necessidade de alto poder de processamento computacional já é comum há algumas décadas nas ciências exatas e engenharias. Na última década, tal necessidade tornou-se também presença constante nas ciências biológicas e médicas. Mais recentemente, até as ciências humanas têm trabalhado com modelos e simulações que exigem computações sofisticadas e pesadas. Este cenário levou o jornal *The New York Times* a publicar um artigo em 2001 intitulado *All Science is Computer Science* [Johnson 2001], mostrando o quanto o avanço científico e tecnológico da humanidade passou a utilizar os computadores como principal ferramenta para a produção de inovação. Cada avanço obtido, fruto do uso intensivo de computação, leva à necessidade de um poder computacional ainda maior para poder atingir o próximo nível de conhecimento. Cria-se um círculo virtuoso que expande e melhora a qualidade do conhecimento humano e, ao mesmo tempo, testa os limites do que é possível se realizar computacionalmente [Goldchleger 2004].

Até o final da década de 1980, máquinas paralelas de altíssimo custo eram praticamente a única forma de se ter acesso à computação de alto desempenho. Tais equipamentos eram fabricados em quantidades relativamente pequenas (centenas ou milhares de unidades) e eram vendidos apenas para as principais universidades e centros de pesquisa do planeta e para algumas empresas de grande porte que podiam investir alguns milhões de dólares em seu parque computacional.

A partir do início da década de 1990, este cenário começou a mudar quando o custo dos computadores pessoais vendidos no mercado de massa começou a cair rapidamente. O poder computacional destas máquinas cresceu exponencialmente, seguindo a lei de Moore, fazendo surgir a tentação de se interconectar algumas poucas dezenas destas máquinas através de redes locais de alta velocidade para se criar os primeiros aglomerados (*clusters*) de computadores. Soluções específicas de hardware de baixo custo atreladas a configurações específicas de software livre baseadas em Linux, tais como o Beowulf (vide `www.beowulf.org`), tornaram os aglomerados de alto desempenho disponíveis a centenas de instituições e milhares de pesquisadores em todo o mundo.

Em meados da década de 1990, a Internet que já completava mais de 20 anos, passava a disponibilizar conexões estáveis de vários megabits por segundo entre as principais universidades norte-americanas e européias. Veio então a idéia de utilizar estas linhas de comunicação de grande largura de banda para interconectar aglomerados e supercomputadores localizados em diferentes instituições, em pontos geograficamente distantes. Estes sistemas de computação de alto desempenho, distribuídos em redes de escala mundial, tornaram-se uma maneira inovadora de compartilhar recursos computacionais entre várias instituições de forma a maximizar o uso dos recursos e permitir a construção de uma coleção de máquinas ainda mais poderosa.

Esta coleção de aglomerados e supercomputadores geograficamente distantes, mas interconectados, passou a ser chamada de Grade Computacional

(*Computational Grid*) em analogia à rede elétrica (*Power Grid*). A metáfora era baseada no desejo de que, um dia, obter poder computacional para resolver problemas quaisquer seria tão fácil quanto obter eletricidade para ativar um equipamento elétrico. Hoje em dia, é possível viajar com um secador de cabelos a qualquer cidade do Brasil e ativá-lo conectando-o em qualquer tomada, sem o menor esforço. Na verdade, o mesmo pode ser obtido em praticamente qualquer cidade do globo exigindo apenas, em alguns casos, um simples adaptador para a tomada. Com as grades, o mesmo se daria para se obter poder de processamento para executar aplicações computacionalmente pesadas.

Para tornar este sonho possível, a comunidade científica de Computação de Alto Desempenho uniu esforços com a comunidade de Sistemas Distribuídos, criando uma linha de pesquisa que passou a ser chamada de Computação em Grade (*Grid Computing*). Hoje, passados mais de 10 anos da criação da metáfora da rede elétrica, ainda não é tão fácil obter poder de processamento quanto é ligar um secador de cabelos. Por outro lado, já existem uma série de sistemas de middleware, tanto comerciais quanto livres e abertos, que permitem que um profissional graduado em Computação estabeleça uma grade computacional e nela execute aplicações computacionalmente pesadas.

Diversas infra-estruturas de middleware desenvolvidas no decorrer dos últimos 10 anos como, por exemplo, Globus [Foster and Kesselman 1997], Legion [Grimshaw et al. 1997], InteGrade [Goldchleger et al. 2004] e OurGrid [Cirne et al. 2006] já permitem que coleções de máquinas heterogêneas distribuídas em aglomerados fisicamente distantes, mas interconectadas por redes de longa distância como a Internet, trabalhem em conjunto para a resolução de problemas computacionalmente pesados. Os serviços oferecidos por esse middleware permitem a localização das máquinas disponíveis em um determinado instante e auxiliam na execução de programas do usuário de forma altamente paralela em grandes grades contendo de algumas dezenas de máquinas até vários milhares.

O avanço de técnicas de Ciência da Computação em áreas como Otimização, Inteligência Artificial, Reconhecimento de Padrões, Mineração de Dados e Simulação continua levando cada vez mais à criação de algoritmos computacionalmente pesados para a resolução de problemas nas mais diversas áreas da atividade humana. A resolução de problemas computacionalmente difíceis é hoje de extrema importância para ciências como a Física, Química e Biologia e para empreendimentos comerciais relacionados a temas tão variados quanto a prospecção de petróleo, simulações mercadológicas, sistema financeiro, bolsa de valores, indústria de cinema, vídeo e televisão [Bazewicz et al. 2000].

Na próxima seção apresentaremos os principais conceitos básicos relacionados à Computação em Grade. Em seguida na Seção 2.3, apresentaremos alguns middlewares de Grade. Na Seção 2.4, apresentamos algumas Grades em uso atualmente. Finalmente na Seção 2.5, apresentamos alguns dos desafios atuais da área.

2.2. Conceitos básicos

A grande maioria dos problemas relacionados à Computação em Grade já eram estudados há várias décadas pelas comunidades de Sistemas Distribuídos e de Computação de Alto Desempenho. Desde a década de 1980, já haviam esforços para a construção de "Sistemas Operacionais Distribuídos" que teriam como finalidade gerenciar os recursos de uma grande coleção de máquinas interconectadas, possivelmente localizadas em diferentes instituições [Tanenbaum 1995]. Entre os principais sistemas operacionais distribuídos da época, podemos citar o Sprite [Ousterhout et al. 1988] e o NOW (*Network of Workstations*) [Anderson et al. 1995], desenvolvidos pela Universidade da Califórnia em Berkeley, o Amoeba [Tanenbaun et al. 1990], desenvolvido na *Vrije Universiteit* na Holanda, e o Spring [Hamilton and Kougiouris 1993] da Sun Microsystems. Esses sistemas podem ser considerados precursores das grades computacionais modernas e tinham como foco principal o compartilhamento de recursos. Para tanto, boa parte da pesquisa desta época era direcionada para a construção de sistemas de arquivos distribuídos, que ofereciam a possibilidade de compartilhamento de dados e execução remota e migração de processos, que permitiam o balanceamento de carga entre as máquinas da rede.

Já no âmbito da Computação de Alto Desempenho, os esforços se concentravam na criação de algoritmos paralelos [Leighton 1991], na definição de novas arquiteturas para máquinas paralelas [Leighton 1991] e no estudo de algoritmos de escalonamento [Bazewicz et al. 2000] para a otimização do uso dos recursos, principalmente memória, processador e comunicação. As plataformas mais usadas eram máquinas paralelas com dezenas ou centenas de processadores conectados por barramentos de alta velocidade e a pesquisa em software se debruçava sobre sistemas de gerenciamento de memória compartilhada distribuída [Nitzberg and Lo 1991].

A principal diferença entre estes sistemas anteriores e as grades computacionais advém da qualidade dos canais de comunicação entre os nós. Enquanto os supercomputadores paralelos oferecem canais de comunicação de altíssima velocidade, baixíssima latência e alta confiabilidade, as grades computacionais muitas vezes empregam tecnologias do mercado de massas como, por exemplo, gigabit Ethernet para a conexão entre as máquinas de um aglomerado e linhas de transmissão da Internet compartilhada para a conexão entre os aglomerados localizados em diferentes organizações. Estas ligações apresentam velocidade apenas mediana, latência alta e confiabilidade média. Ao contrário dos supercomputadores onde as falhas são muito pouco freqüentes, em uma grade computacional típica, nós entram e saem da grade diariamente; em alguns casos, a todo minuto. Além disso, a rede numa grade computacional normalmente é muito heterogênea: cada aglomerado possui suas próprias características e as ligações entre os aglomerados podem ser de vários tipos e características diferentes.

Devido a essas diferenças, algoritmos paralelos que funcionavam bem em uma máquina paralela dedicada do passado não apresentam necessariamente

um bom desempenho em uma grade da atualidade, onde as latências de comunicação são proporcionalmente muito maiores. É necessário muitas vezes recalibrar-se o algoritmo ou, até mesmo, criar novos algoritmos de forma a realizar mais computação para cada byte de dados transmitido pela rede.

2.2.1. Tipos de grades computacionais

Krauter, Buyya e Maheswaran elaboraram em 2002 uma resenha dos principais sistemas de grades existentes até então [Krauter et al. 2002] e propuseram uma taxonomia identificando diferentes tipos de grades. Para eles, uma **Grade de Computação** (*Computing Grid*) é um sistema de Computação em Grade que visa a integração de recursos computacionais dispersos para prover uma maior capacidade combinada de processamento aos seus usuários.

Já as **Grades Computacionais Oportunistas** (*Opportunistic Grids* ou *Scavenging Grids*) [Goldchleger 2004, Livny et al. 1997] são um subtipo de grades que utiliza os períodos ociosos de estações de trabalho de funcionários de uma organização, laboratórios acadêmicos, máquinas de estudantes, etc. para formar, dinamicamente, uma grade de computação.

Grades de Dados (*Data Grids*) são sistemas cujo objetivo principal é o acesso, pesquisa e processamento de grandes volumes de dados, potencialmente distribuídos em vários repositórios, conectados por uma rede de grande área.

Grades de Serviços (*Service Grids*) oferecem serviços viabilizados pela integração de diversos recursos computacionais como, por exemplo, um ambiente para trabalho colaborativo, ou uma plataforma para aprendizado à distância. Uma variante importante deste tipo de grade são os chamados **Colaboratórios** (*Collaboratories*) [Finholt 2002], laboratórios físicos compartilhados através da Internet de forma que vários pesquisadores e estudantes possam realizar experimentos científicos à distância. Os colaboratórios permitem que laboratórios de alto custo possam ser compartilhados por pessoas localizadas em diferentes cidades, reduzindo os custos e otimizando o uso de recursos escassos.

A tendência, a médio prazo, é que todos estes tipos diferentes de serviços e usos estejam presentes em todas as grades computacionais e não mais será necessário realizar uma diferenciação entre elas. Neste capítulo, utilizaremos o termo grade computacional para nos referir a todos esses tipos de grade.

2.2.2. Principais serviços de uma grade computacional

Sistemas de grades computacionais são hoje em dia implementados como uma camada de middleware que executa sobre sistemas operacionais convencionais. Esse middleware é responsável por esconder os detalhes e particularidades dos diferentes sistemas operacionais sobre os quais ele opera e também por oferecer serviços de alto nível para a execução de aplicações do usuário. A Figura 2.1 apresenta uma arquitetura genérica implementada pela maioria dos sistemas de grades da atualidade [de Camargo et al. 2006b].

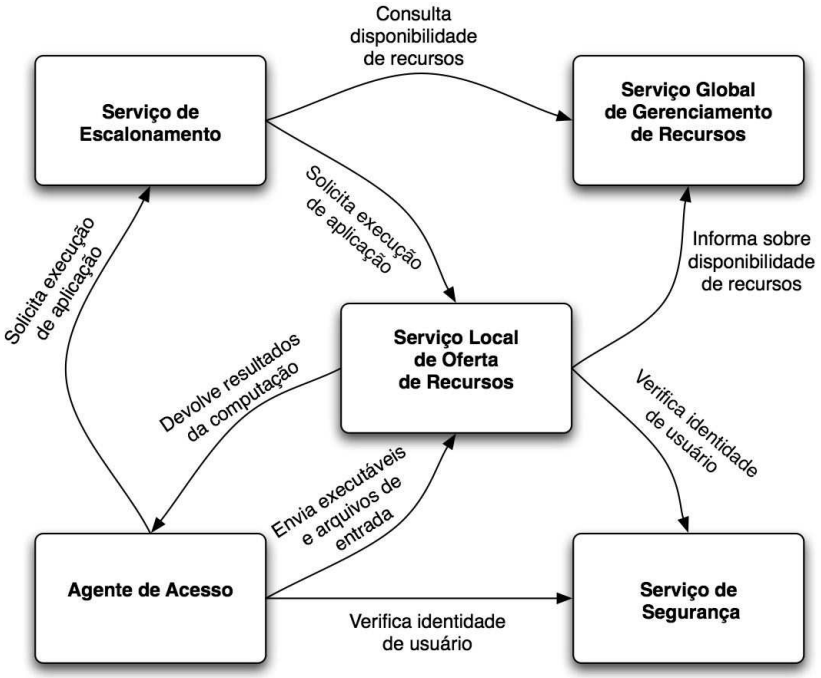

Figura 2.1. Arquitetura genérica de uma grade computacional

O **Agente de Acesso** é o principal ponto de acesso para usuários em sua interação com a grade. Ele é executado em cada nó a partir do qual aplicações serão submetidas para execução na grade. Além de possibilitar a execução de aplicações, ele deve permitir ao usuário que especifique requisitos e parâmetros da execução, deve permitir que o usuário monitore a execução de suas aplicações e deve auxiliar na coleta dos resultados gerados pelas aplicações. Muitos sistemas de grades implementam uma versão Web do agente de acesso permitindo que o usuário interaja com a grade usando um navegador Web qualquer.

O **Serviço Local de Oferta de Recursos** é executado nas máquinas que exportam seus recursos para a grade e é responsável por executar aplicações nos nós da grade. Para tanto, ele precisa obter o código executável da aplicação, iniciar sua execução (normalmente em um novo processo), coletar e apresentar eventuais erros que possam ocorrer durante ou antes da execução da aplicação e devolver os resultados da execução da aplicação para o usuário que a solicitou. Este serviço é também responsável por gerenciar o uso local dos recursos de um determinado nó e por prover, para a grade, informações sobre a disponibilidade de recursos neste nó.

O **Serviço Global de Gerenciamento de Recursos** é responsável por monitorar o estado dos recursos compartilhados em toda a grade e por responder a solicitações de utilização desses recursos. Para tanto, ele auxilia o Serviço de Escalonamento a efetuar o casamento das requisições por recursos solicitadas pelos usuários com as ofertas de recursos. Ele mantém informações dinamicamente atualizadas sobre quais aglomerados e quais nós estão com recursos disponíveis e dados sobre a utilização destes recursos como, por exemplo, utilização do processador, quantidade de memória RAM disponível, quantidade de espaço em disco disponível, etc. Por esse motivo, ele pode também ser chamado de **Serviço de Informações sobre Recursos**. Em alguns casos, esse serviço pode também ser responsável por detectar falhas nos nós da grade e notificar o agente de acesso sobre erros na execução de aplicações presentes nos nós com problemas.

O **Serviço de Escalonamento** determina, para cada aplicação, onde e quando ela irá ser executada. Ele recebe solicitações de execução de aplicações, obtém informações sobre a disponibilidade de recursos consultando o Serviço de Global de Gerenciamento de Recursos e determina a ordem e o local de execução das aplicações.

Finalmente, o **Serviço de Segurança** é responsável por três tarefas principais: (1) proteger os recursos compartilhados de forma que um nó que exporta seus recursos para a grade não sofra ataques executados por aplicações maliciosas, (2) autenticação dos usuários de forma que se saiba quem é responsável por cada aplicação e pela sua execução, permitindo que sejam estabelecidas relações de confiança e que usuários sejam responsabilizados pelos seus atos e (3) fornecer canais seguros de comunicação garantindo a integridade e confiabilidade dos dados. Em ambientes abertos tais como o de SETI@home (`setiathome.berkeley.edu`), por exemplo, é ainda importante prover mecanismos para proteger a aplicação de recursos maliciosos.

A forma como estes serviços são organizados, agrupados e implementados pode variar muito de um middleware de grades para outro. Mas, de forma geral, todos os principais sistemas de grades oferecem essas funcionalidades.

2.2.3. Gerenciamento de Recursos

Assim como a tarefa de um sistema operacional é gerenciar os recursos de um computador, uma das principais tarefas de um middleware de grades é gerenciar os recursos disponíveis na grade. Para atingir esse objetivo, é necessário, antes de mais nada, ter-se o conhecimento de quais recursos estão disponíveis a cada instante, qual o seu tipo e qual a sua capacidade.

Existe uma grande quantidade de recursos que podem ser de interesse para usuários da grade. Os mais comumente associados à computação de alto desempenho são **processador**, que é onde a computação é realizada, e **memória principal**, onde os dados manipulados pela computação são armazenados enquanto a aplicação é executada. Porém, outros recursos também

são de grande valia para a computação de alto desempenho. O **espaço de armazenamento persistente**, também chamado de memória secundária, é onde os dados são salvos de forma a sobreviver a quedas de energia; para tanto normalmente são usados discos rígidos, embora em alguns casos possa-se também usar-se fitas magnéticas, discos ópticos, etc. A **conexão de rede** é outro importante recurso computacional, uma vez que boa parte dos algoritmos de processamento paralelo requer a troca de grande quantidade de dados entre os nós participantes da computação. Dessa forma, quanto maior a qualidade da conexão de rede, melhor o desempenho da computação.

Além dos recursos mais comuns citados anteriormente, em casos específicos, pode ser interessante compartilhar outros tipos de recursos computacionais em uma grade. Placas de processamento de vídeo de alta velocidade, compressores de vídeo em hardware e monitores de vídeo de altíssima qualidade poderiam ser recursos compartilhados em uma grade voltada para o processamento de vídeo, por exemplo. Telescópios eletrônicos, rádio-telescópios e um servidor de armazenamento com capacidade para vários petabytes de dados poderiam ser recursos utilizados por uma grade compartilhada por astrônomos de vários continentes, por exemplo. Sensores de temperatura, pressão e humidade poderiam ser compartilhados por estações de previsão do tempo e do clima. Muito provavelmente, nas próximas décadas, a quantidade de exemplos como estes e a sua ocorrência no mundo da ciência e da tecnologia será cada vez maior e mais freqüente.

Para cada recurso participante de uma grade, é importante conhecer tanto informações estáticas sobre a natureza do recurso quanto informações dinâmicas sobre o seu estado de operação. Para cada processador, é importante conhecer tanto o seu tipo (por exemplo, PowerPC G5 ou Pentium 4) e a sua velocidade (por exemplo, 3.8GHz), que são informações estáticas, quanto a sua carga atual (por exemplo, 93% ocioso no último minuto e 99% ocioso na última hora), que é uma informação dinâmica, i.e., que muda a todo momento.

Ao contrário das máquinas paralelas do passado, boa parte das grades computacionais são heterogêneas, ou seja, cada nó é composto por hardware de diferentes tipos e capacidades. Isso torna a comparação entre a capacidade dos nós muito difícil. Por exemplo, onde uma aplicação será executada mais rapidamente? Em um computador equipado com um PowerPC G5 de 2.0GHz e 90% do processador livre ou em um computador equipado com um Pentium 4 de 3.8GHz com 80% do processador livre? É uma pergunta de difícil resposta que a maioria das grades hoje em dia nem tenta responder precisamente. Uma possibilidade para resolver este problema é tentar encontrar uma medida de desempenho independente de plataforma de hardware, como é feito, por exemplo, pelo programa BogoMips (veja `www.clifton.nl/bogomips.html`); mas, como o próprio nome da ferramenta diz (*bogus* em inglês quer dizer não genuíno), o dado fornecido por ela é apenas um "chute". Além disso, o desempenho do processador também depende da aplicação. Por exemplo, há alguns anos, os processadores AMDs eram mais rápidos do

que os Intel para processamento de texto, mas para aplicações científicas o desempenho era o inverso [1].

Além dos recursos de hardware descritos acima, uma grade pode tratar também de componentes de software como recursos da grade. Por exemplo, um serviço que converte fluxos de vídeo de um formato para outro, pode ser considerado um recurso; um serviço de armazenamento distribuído e replicado de dados pode ser considerado um recurso. As várias versões das bibliotecas do sistema operacional ou dos mecanismos de programação paralela tais como MPI e BSP podem também ser considerados recursos a serem gerenciados pelo middleware da grade.

2.2.4. Monitoramento

Toda grade computacional necessita de um sistema de monitoramento para que a grade possa descobrir, a qualquer instante, quais os recursos computacionais disponíveis, qual o seu tipo e a sua capacidade. Normalmente, isso é realizado fazendo com que o Serviço Local de Oferta de Recursos seja instalado em cada nó da grade para monitorar em intervalos curtos de tempo (por exemplo, a cada 5 segundos) os recursos disponíveis em uma certa máquina. De tempos em tempos (por exemplo, a cada 5 minutos) este serviço envia uma mensagem para o Serviço Global de Gerenciamento de Recursos com informações atualizadas sobre o nó. O Serviço Global então armazena esta informação em seu banco de dados, potencialmente com informações sobre milhares de nós.

Um dos desafios da pesquisa em grades é como implementar o Serviço Global de Gerenciamento de Recursos de forma que ele não se torne um gargalo do sistema e nem um ponto único de falhas. Nas versões mais antigas do sistema Globus, por exemplo, era notório que este serviço não era muito escalável e se tornava rapidamente um grande consumidor de recursos.

As grades mais modernas implementam alguma forma de **Federação de Aglomerados** de forma a distribuir a carga do gerenciamento de informações sobre recursos. Nesse caso, cada aglomerado, normalmente composto por algumas dezenas de máquinas, possui um gerenciador de recursos que recebe informações dos nós de seu aglomerado. O gerenciador desse aglomerado comunica-se com os gerenciadores dos aglomerados vizinhos na federação compartilhando informações sobre os recursos de seu aglomerado. Os protocolos específicos para a federação de aglomerados varia muito de um middleware de grades para outro.

2.2.5. Escalonamento

Uma das principais áreas de pesquisa em grades computacionais é o escalonamento, que em termos gerais corresponde a alocação de tarefas a recursos. No caso de tarefas, há uma grande diversidade: existem desde tarefas

[1] Testes com dados concretos de usos de vários processadores em diversas aplicações podem ser encontrados em `www.pcworld.com`.

diretamente ligadas a processamento, que podem ser processos, *threads*, ou programas até, por exemplo, armazenamento de dados. Em todos esses casos, procura-se uma boa alocação das tarefas aos recursos de forma a otimizar alguma função objetivo previamente determinada.

Para ilustrar a possível dificuldade do problema, apresentamos um exemplo onde, apesar de ser simples encontrar a solução ótima, é um problema difícil (NP-completo). Dados diversos arquivos a serem armazenados, procura-se uma alocação desses às máquinas, de forma que o espaço usado em cada máquina seja igual. O problema no caso de duas máquinas já é bem conhecido e é chamado *partição*. Isto é, dados n, e n inteiros $I = \{i_1, \ldots, i_n\}$, queremos dividir o conjunto I em dois conjuntos I_1 e I_2, $I_1 \bigcup I_2 = I$ e $I_1 \cap I_2 = \emptyset$ tais que:

$$\sum_{i \in I_1} i = \sum_{i \in I_2} i.$$

Logo, se mesmo para problemas simples, a solução pode ser custosa, veremos que para grades computacionais, onde o ambiente é heterogêneo e dinâmico, o melhor não é procurar por soluções exatas mas sim obter boas aproximações.

2.2.5.1. Componentes

A seguir apresentamos a nomenclatura geralmente usada quando se lida com escalonamento em grades. Uma *tarefa* é uma unidade indivisível a ser escalonada, uma *aplicação* é composta por um conjunto de tarefas e aplicações. Um *recurso* é algo capaz de realizar alguma operação, como um processador, um software e linhas de comunicação. Um *sítio* é uma unidade autônoma da grade, composta por recursos. Com esses conceitos, podemos dizer que o *escalonamento de tarefas* é um mapeamento, no tempo, das tarefas para recursos. Além disso, para que o escalonamento seja válido, é importante não ultrapassar as limitações dos recursos, por exemplo, garantir que a cada instante, os arquivos que serão armazenados em um nó não excedam a capacidade de seus discos.

Uma arquitetura geral de escalonamento em grades pode ser representada pelo modelo da Figura 2.2 [Dong and Akl 2006]. Nem todas as grades seguem modelos tão completos e algumas possuem serviços adicionais, mas este modelo é uma abstração bem aceita pelos pesquisadores da área.

Em linhas gerais, o serviço de escalonamento (1) recebe aplicações dos usuários, (2) seleciona os recursos a serem usados pelas aplicações de acordo com os dados do serviço de informações sobre recursos (retângulo *Informações da grade* na figura), e (3) faz o mapeamento entre a aplicação, ou suas tarefas, e os recursos, seguindo algum objetivo pré-definido. O serviço de informação sobre os recursos, acompanha situação da grade, atualizando as informações periodicamente. Existe também um serviço responsável por executar

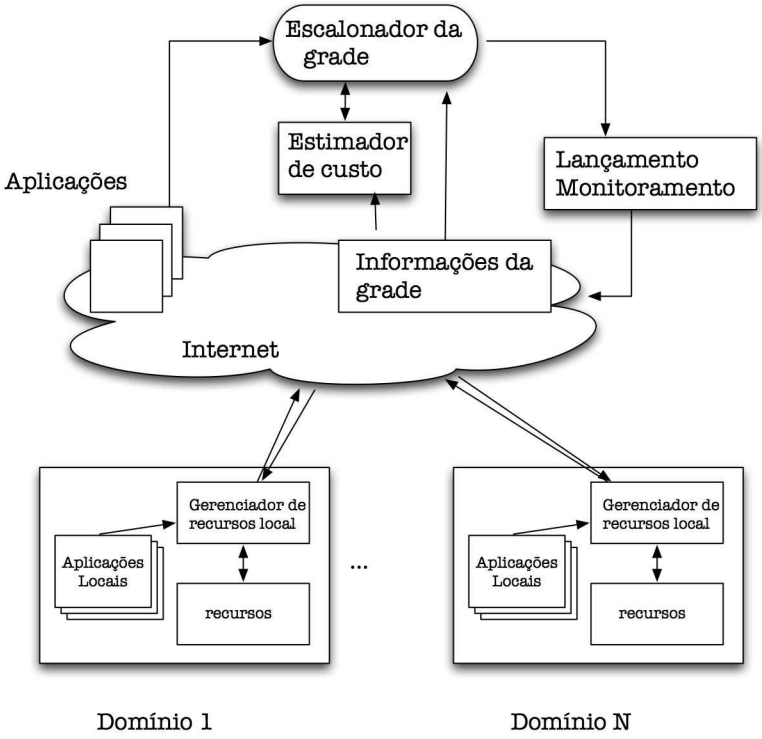

Figura 2.2. Arquitetura genérica de escalonamento

as aplicações (segundo as decisões do escalonador), monitorar sua execução e coletar eventuais dados.

Já o serviço local de oferta de recursos é responsável por duas tarefas principais: o escalonamento local dentro do seu domínio e o envio de mensagens ao serviço de informações sobre recursos. É interessante ressaltar que o escalonamento realizado por esse serviço de escalonamento local ocorre tanto para aplicações locais, do próprio domínio, quanto para aplicações provenientes do serviço de escalonamento da grade. Como parte do serviço local, escalonadores locais, tais como Condor [Thain et al. 2005, Litzkow et al. 1988], PBS/OpenPBS (`www.openpbs.org`), LSF [Zhou 1992], NQS [Albing 1993] e CRONO [Netto and Rose 2003] podem ser usados. A coleta de informações locais pode ser realizada através de ferramentas como o *Network Weather Service* [Rich Wolski 1999], Hawkeye (`www.cs.wisc.edu/condor/hawkeye`) ou Ganglia [Massie et al. 2004].

Entre o serviço de escalonamento e os serviços locais de oferta de recursos se encontra o serviço de lançamento e monitoramento, que é responsável por implementar e acompanhar o escalonamento definido nos sítios da grade.

2.2.5.2. Algoritmos de Escalonamento

Existem dois contextos bem diferentes onde são propostos algoritmos de escalonamento para grades: o estático e o dinâmico. No estático, supõe-se que se tem o conhecimento total das aplicações a serem submetidas antes do seu início. No caso dinâmico, tanto as aplicações podem aparecer a qualquer momento, como as próprias tarefas das aplicações podem não ser conhecidas.

Existem vários algoritmos de escalonamento estático para grades, sendo a grande maioria heurísticas (sem garantia com relação ao melhor possível), e poucos algoritmos de aproximação em alguns casos específicos. Uma lista abrangente de referências pode ser obtida em [Dong and Akl 2006]. A desvantagem dos algoritmos estáticos é a sua pouca flexibilidade fazendo com que vários fatores prejudiquem o seu desempenho, entre eles:

- recursos inicialmente disponíveis podem mudar em um ambiente dinâmico de grade;
- o tempo previsto de uma tarefa pode não corresponder ao esperado e isto pode provocar grandes conseqüências no escalonamento;
- para certas aplicações, a única forma de prever o seu tempo de execução é através de uma execução completa;
- o modelo estático pressupõe um conhecimento total a priori, o que pode não ser possível em diversas situações reais.

No escalonamento dinâmico, a partir de estimativas dos custos das tarefas das aplicações, elas são alocadas a recursos. Mas, nesse caso, como o ambiente é completamente dinâmico, costuma-se permitir a alocação de tarefas no momento de sua criação. Isto é feito através de mecanismos de balanceamento de carga: os recursos menos carregados recebem as tarefas. Este mecanismo de regulação apresenta grandes vantagens em relação à alocação estática, pois permite que o escalonamento se adapte às condições atuais em tempo de execução. Além disto, ele é particularmente interessante em situações onde se deseja usar ao máximo os recursos, ao invés de se priorizar o tempo de término de alguma aplicação.

Além disto o balanceamento de carga pode ser feito a partir das tarefas já alocadas. Nesse caso, o balanceamento pode ser iniciado a partir dos recursos mais carregados em direção aos recursos com menos carga. Outra opção é a busca de trabalho pelos recursos com pouca carga, o que é chamado de roubo de trabalho.

Segundo El-Rewini et al. [El-Rewini et al. 1994], existem quatro formas principais de balanceamento de carga: o uso de filas, onde cada recurso possui

a sua fila local e uma tarefa que chega é alocada na menor fila; o balanceamento onde, periodicamente, é realizado um balanceamento de carga entre os recursos; o balanceamento com custos, onde os possíveis tempos de comunicação entre as tarefas também são considerados e, finalmente, a abordagem híbrida, onde é feito inicialmente um escalonamento estático com as tarefas conhecidas e o dinâmico com as demais.

2.2.5.3. Exemplos de escalonadores

Veremos agora, em linhas gerais, diversos escalonadores para grades. É interessante notar que existem duas abordagens diferentes: o uso de escalonadores de grade (também conhecidos como meta-escalonadores), como visto anteriormente, e uma outra opção na qual escalonadores também cuidam do nível mais baixo, isto é, servem como escalonadores locais. Existem muitos outros que, por limitação de espaço, não abordaremos nesse capítulo, como o SGE (`gridengine.sunsource.net`) e o Maui [Jackson et al. 2001].

O *Sun Grid Engine* (SGE) é um escalonador comercial capaz de gerenciar dezenas de milhares de máquinas. O Maui (`www.supercluster.org/maui`) e seu respectivo gerenciador de recursos Torque são bastante usados na prática; com eles é possível caracterizar facilmente classes de usuários e criar políticas próprias. O Maui é de código aberto, mas o seu sucessor, o Moab, é um produto comercial. Suas principais diferenças são a possibilidade de gerenciar aglomerados privados virtuais, a disponibilidade de ferramentas gráficas de administração e um portal de acesso. Uma outra opção de escalonador é o Catalina (`www.sdsc.edu/catalina`), que é um escalonador para diversos gerenciadores de recursos.

O Globus (vide Seção 2.3.1) possui também os seus meta-escalonadores, isto é, escalonadores responsáveis por coordenar escalonadores locais. No *Globus Toolkit* o meta-escalonador usado é o*Community Scheduler Framework* (CSF), que pode comunicar-se com diversos escalonadores de aglomerados. Para interface para submissão e controle de aplicações é usado o GRAM. Uma outra opção de meta-escalonador é o GridWay (`www.gridway.org`).

A seguir apresentamos com mais detalhes dois escalonadores do segundo tipo discutido acima, ou seja, escalonadores globais que atuam também como escalonadores locais.

SLURM

O *Simple Linux Utility for Resource Management* (SLURM) [Yoo et al. 2003] é um gerenciador de recursos de código aberto projetado para aglomerados heterogêneos com milhares de nós Linux. Ele é portável, escrito em C e possui uma ferramenta de auto-configuração. Ele provê tolerância a falhas e oferece controle de estado das máquinas, gerenciamento de partições, gerenciamento de aplicações, escalonador e módulos para cópia de dados. O sistema é composto por vários módulos, incluindo um *daemon* `slurmd`, executado em cada

nó do sistema e um *daemon* `slurmctld` que é executado no nó de gerenciamento. Além disto, existem outros utilitários que fornecem acesso a execução remota e ao controle de aplicações.

O SLURM possui três funções principais: alocar recursos, de forma exclusiva, ou não, por períodos determinados; fornecer um arcabouço para lançar, executar e monitorar as aplicações nas máquinas; e gerenciar uma fila de trabalhos pendentes no caso de pedidos de uso conflitantes. Intencionalmente, o escalonador é bem simples, do tipo FIFO (*First-In First-Out*), mas ele permite a atribuição de prioridades às aplicações.

Por outro lado, graças a sua simplicidade, o SLURM provê alto desempenho no gerenciamento de aplicações. Para fornecer alguma flexibilidade e personalização, o SLURM aceita *plugins* que podem ser carregados em tempo de execução. Atualmente, o SLURM é usado em cerca de mil sistemas de computação, sendo que alguns deles estão entre os mais rápidos do mundo como o BlueGene/L, o mais rápido do mundo na atualidade (vide `www.top500.org/lists/2007/11/100`).

OAR

O OAR [Capit et al. 2005] é um escalonador de código aberto para aglomerados e grades computacionais. Ele permite a reserva de máquinas de uma grade por usuários que submetem aplicações; logo, a unidade de recurso são as máquinas. Ele é baseado em diversas ferramentas de alto nível, tais como uma base de dados relacional (MySql), a linguagem Perl e uma ferramenta para monitorar e lançar as aplicações.

O casamento (emparelhamento) entre recursos e aplicações é feito através do SQL, que permite atingir os objetivos de buscas bastante específicas com consultas relativamente simples. O monitoramento dos recursos é efetuado por outra ferramenta, o Taktuk (`taktuk.gforge.inria.fr`), que troca informações com o OAR através do banco de dados.

O controle de aplicações é feito através de comandos distintos. Existem comandos para submissão, cancelamento e consulta do estado. Para a submissão, primeiro o OAR verifica se a aplicação é admissível, isto é, se a submissão é válida. Em caso positivo, os dados da aplicação são inseridos na base de dados e é devolvida a resposta “submissão em curso”. A aplicação é então executada assim que seus requisitos puderem ser atendidos.

O escalonamento no OAR é completamente modular e pode ser adaptado às necessidades do administrador da grade. Foram implementadas prioridade com filas de submissão diferentes, casamento de recursos e *backfilling*, que permite que uma aplicação submetida após uma outra seja escalonada se não for atrasar a execução das aplicações há mais tempo na fila. O OAR também permite que os usuários façam reservas caso a aplicação seja admissível. Isto é, se a aplicação pode ser executada no hardware existente, um intervalo de tempo futuro pode ser reservado para sua execução. O OAR está atualmente em produção no Grid 5000 (vide Seção 2.4.1) e já escalonou mais de 5 milhões de aplicações.

As vantagens do OAR são claramente a sua simplicidade e modularidade. Por exemplo, o OAR também tem uma extensão que permite o seu uso em grades oportunistas. Toda a parte relacionada à eventual complexidade de sofisticados algoritmos de escalonamento pode ser inserida à parte, e pode vir a ser um novo módulo do OAR.

2.2.6. Segurança

Com a popularização da Internet no âmbito comercial, governamental, empresarial e até mesmo na vida cotidiana das pessoas, a segurança dos sistemas a ela conectados se tornou um problema de grande importância. Empresas com fortes ligações com a Internet precisam investir vultosas quantias em pessoal, software e equipamento voltados especificamente para a segurança. Até mesmo um cidadão comum, ao usar a Internet em sua casa, deve seguir uma série de recomendações sobre como navegar pela rede, em quais sítios e mensagens de correio eletrônico confiar, de onde baixar software, como configurar sua rede doméstica; caso contrário, em poucos minutos um computador pessoal pode ficar completamente dominado por vírus, *worms*, cavalos de tróia e outras pestes da Internet moderna. Portais de comércio eletrônico e sítios de bancos são alvos típicos do cibercrime organizado que já movimenta bilhões de dólares por ano. Mesmo fazendo uso de profissionais altamente capacitados em segurança, é relativamente comum ler-se na imprensa especializada que importantes sítios de bancos, de megacorporações internacionais ou até mesmo de instalações militares de superpotências são invadidos por *crackers* que visam desde simplesmente ficarem famosos em sua comunidade até causar grandes prejuízos financeiros ou operacionais.

Como até o presente momento a maioria das grades computacionais tem um objetivo acadêmico e de pesquisa científica, não há tanto interesse pessoal dos *crackers* ou interesses financeiros em se invadir tais sistemas. Mesmo assim, é de fundamental importância tratar a questão da segurança com muita seriedade pois as grades computacionais são particularmente vulneráveis a ataques [Pinheiro Jr. and Kon 2005]. A grande maioria dos serviços disponíveis na Web e mesmo na Internet de uma maneira geral, não permite a execução remota de programas fornecidos pelo usuário; normalmente, o usuário interage com o sistema remoto através da interface gráfica do navegador Web ou de uma interface de programação (API). Já no caso das grades computacionais, é essencial que o próprio usuário seja capaz de fornecer o programa que será executado nos nós da grade, o que é uma característica quase que exclusiva das grades. Abre-se portanto uma série de possibilidades de novos ataques; além dos ataques possíveis a outros sistemas, em uma grade um usuário mal-intencionado pode submeter um programa devidamente projetado para roubar informações ou atacar outros nós da Internet; ou então um vírus pode utilizar os próprios mecanismos de escalonamento e distribuição da grade para contagiar todos os seus nós.

O gerenciamento da segurança em uma grade computacional, incluindo o gerenciamento de usuários, permissões e controle de acesso é também mais complicado quando a grade é resultante da aglutinação de máquinas em diferentes domínios administrativos, por exemplo, em diferentes universidades ou corporações. No futuro, a tendência é que as grades sejam utilizadas fortemente também em contextos comerciais e industriais. Portanto, o desenvolvimento da pesquisa científica e o avanço da tecnologia de segurança em grades é de grande importância.

A implementação de sistemas seguros deve ser uma preocupação constante nas equipes de desenvolvimento de sistemas computacionais. Um sistema computacional é seguro se pudermos depender dele e seu software tiver um comportamento esperado [Garfinkel et al. 2003]. Sabemos, no entanto, que criar um sistema computacional totalmente seguro é muito difícil (se não impossível) e devemos definir os limites de risco aceitáveis, respondendo a questões tais como: qual o perímetro da rede a ser considerada? Quais serviços serão disponibilizados externamente? Quais são os controles necessários para o sistema e o nível de segurança pretendido? Desta forma estamos delineando o escopo do nosso problema.

De acordo com Lang e Schreiner, cada sistema computacional tem valores associados aos recursos que devem ser protegidos, alguns tangíveis, outros intangíveis [Lang and Schreiner 2002]. Recursos tangíveis são aqueles aos quais podemos associar valores diretos, ou seja, podemos quantificar um preço por ele (o hardware, por exemplo). Recursos intangíveis (uma informação, por exemplo) são mais difíceis de avaliar pela dificuldade que temos em definir o quanto vale a informação. Lang e Schreiner [Lang and Schreiner 2002] sugerem que devemos quantificar o custo da perda pois é mais apropriado quantificar os impactos negativos do comprometimento do recurso, tais como, custo da troca, danos à reputação, perda de competitividade, etc.

A relação entre o custo da implementação e o custo da perda do recurso nos proporciona o que chamamos de análise de risco. O objetivo da segurança é minimizar o risco. Os custos da implementação de segurança tais como custo com pessoal, hardware, software e tempo gasto não devem ser maior do que os custos associados ao risco potencial de cada possível ataque. É fácil imaginar que os requisitos de segurança necessários para um ambiente bancário devem ser diferentes daqueles esperados para um ambiente educacional. Encontrar um ponto de equilíbrio entre os custos associados à implementação de segurança e o benefício causado por essa implementação é um dos grandes desafios durante o projeto de sistemas de segurança.

Os principais objetivos do Serviço de Segurança de uma grade computacional são:

- **Autenticação** é o processo de estabelecer a validade de uma identidade reivindicada [Ramachandran 2002]. A autenticação é o primeiro passo na segurança de um sistema computacional; junto à confidencialidade e à integridade, consiste num dos pilares da segurança. Em uma grade,

pode ser interessante autenticar os administradores, os provedores de recursos, os provedores do código executável das aplicações e os usuários que solicitam a execução destas aplicações.

- **Irretratabilidade** (*non-repudiation*) consiste em obter provas (ou fortes indícios) de ações realizadas no passado de forma que um indivíduo não possa negar ações que tenha realizado no sistema. Por exemplo, se uma aplicação é responsável por espalhar um vírus pelo sistema ou por utilizar a grade como ponto de partida para ataques a outros sítios da Internet, gostaríamos de ter provas sobre quem foram os usuários que forneceram o código executável da aplicação. Já no caso de uma aplicação ser executada um número exagerado de vezes comprometendo a harmonia no compartilhamento dos recursos, gostaríamos de saber, com segurança, quem é o usuário executando a aplicação excessivamente.
- A **Confidencialidade** impede que os dados sejam lidos ou copiados por usuários que não possuem o direito de fazê-lo. No caso de uma grade compartilhada entre empresas, por exemplo, deve-se garantir que funcionários de uma empresa não tenham acesso aos dados e aplicações de outras empresas.
- A **Integridade de Dados** protege a informação de ser removida ou alterada sem a autorização do dono. Uma grade deve garantir que os dados de saída das aplicações sejam entregues aos usuários sem adulterações.
- **Disponibilidade** é a proteção dos serviços para que eles não sejam degradados ou fiquem indisponíveis sem autorização. Isto implica dados e sistemas prontamente disponíveis e confiáveis. Um usuário deve ter pronto acesso aos resultados de suas computações sempre que deles necessitar. Da mesma forma, sempre que um usuário quiser submeter uma aplicação à grade, o serviço de submissão deve estar disponível e ele deve prontamente executar a aplicação, desde que haja recursos disponíveis. O suporte a disponibilidade pode exigir o uso de outros mecanismos além daqueles normalmente presentes em um serviço de segurança, por exemplo, mecanismos avançados de tolerância a falhas.
- O **Controle** permite que somente usuários conhecidos e que têm direitos de acesso em períodos determinados possam, devidamente, dispor dos recursos disponíveis na grade.
- A **Prevenção** é um dos elementos fundamentais em todo sistema de segurança. Segundo Garfinkel e Spafford [Garfinkel et al. 2003], a segurança de computadores consiste em uma série de soluções técnicas para problemas não técnicos. Podemos gastar grandes quantias de dinheiro, tempo e esforço em sistemas de segurança mas nunca resolveremos problemas como os defeitos desconhecidos nos softwares (*bugs*) ou funcionários maliciosos. Precaver-se em relação a possíveis proble-

mas é uma boa prática de segurança. Devemos "dirigir defensivamente" a fim de evitar desastres de percurso.

- A **Auditoria** é o mecanismo que nos permite descrobrir que ações foram executadas na grade. Ainda não se conhece um sistema de segurança perfeito; sempre é possível que usuários não autorizados possam tentar acessar o sistema, que usuários legítimos tenham efetuado ações erradas ou até mesmo que atos maliciosos possam ser praticados. Nesses casos é necessário determinar o que aconteceu, quando, quem foi o responsável e o que foi afetado pela ação. A auditoria deve ser um registro incorruptível dos principais eventos relacionados à segurança e integridade do sistema. Através da auditoria, podemos nos resguardar das ações dos usuários e até mesmo utilizar mecanismos que impossibilitem que os usuários neguem os seus atos ao utilizar o sistema (vide irretratabilidade).

Para atingir os objetivos acima descritos, o Serviço de Segurança deve utilizar-se de técnicas de Criptografia [Terada 2000] e incorporar os seguintes mecanismos:

- **Canais de comunicação seguros** devem poder ser criados entre os nós da grade de forma que os dados transmitidos não possam ser roubados nem alterados por usuários indevidos. Uma dos meios mais comuns para a implementação desses canais de comunicação é através do protocolo *Secure Socket Layer* (SSL) implementado por bibliotecas como OpenSSH (vide `www.openssh.org`).
- **Armazenamento seguro de dados**: a grade deve garantir que os dados armazenados de forma persistente, por exemplo, em discos magnéticos não poderão ser nem alterados nem lidos por usuários indevidos. Para atingir esse objetivo é necessário controlar, através do sistema operacional dos nós de armazenamento, o acesso aos dados armazenados em disco.
- ***Sandboxing*** é um mecanismo que isola as aplicações do ambiente exterior, como um areião onde crianças brincam isoladas sem ficarem expostas aos perigos do mundo exterior e sem a possibilidade de fazerem muita sujeira ou quebrar algo do mundo exterior. Ao confinar cada aplicação a uma *sandbox*, o middleware da grade pode limitar os recursos computacionais aos quais cada aplicação tem acesso. Por exemplo, é possível determinar que uma dada aplicação não usará mais do que 128MB da memória RAM, não usará mais do que 1GB do disco local e só abrirá canais de comunicação com alguns outros nós específicos da grade e, mesmo assim, apenas usando a API própria do middleware da grade. O *sandbox* barraria então qualquer tentativa de se abrir um soquete TCP/IP, qualquer tentativa de se comunicar com uma máquina não participante da computação em questão e qualquer tentativa de ler arquivos no disco local com exceção dos arquivos que a própria aplicação

criou. Atualmente, uma ferramenta muito utilizada para a implementação de *sandboxes* é a máquina virtual Xen [Barham et al. 2003].

- Uma dos aspectos mais difíceis da segurança em grades é a **proteção das aplicações de ataques por parte de nós maliciosos** da grade. Seria desejável que um computador que fizesse parte da grade não fosse capaz de interceptar ou adulterar dados das aplicações dos usuários da grade. Ainda não existe uma solução adequada e abrangente para evitar que os dados das aplicações sejam copiados pelo nó que a hospeda. Já para garantir que os dados não sejam falseados, normalmente o que se faz é replicar a execução de uma mesma aplicação em vários nós e comparar os resultados [Sarmenta 2002].

2.2.7. Tolerância a Falhas e Checkpointing

No ambiente dinâmico de grades computacionais, onde recursos podem ficar indisponíveis a qualquer momento, um mecanismo de tolerância a falhas é essencial [de Camargo 2007]. No caso de aplicações paralelas que utilizam dezenas de máquinas e cuja execução se estende por muitas horas, a falha de uma única máquina nesse período normalmente faz com que toda a computação já realizada seja perdida. Desse modo, numa grade oportunista, onde máquinas ficam indisponíveis várias vezes num único dia, a execução desse tipo de aplicação sem um mecanismo de tolerância a falhas é inviável.

Além disso, as próprias máquinas que hospedam os serviços essenciais da grade estão sujeitas a falhas, embora com uma freqüência um pouco menor pois normalmente esses componentes fundamentais do middleware da grade são executados em máquinas mais robustas e estáveis. Mesmo assim, é desejável que a queda de um nó gerenciador de um aglomerado não inviabilize o uso de algumas dezenas de nós ou mesmo que quebre os canais de comunicação com outros aglomerados da grade. Para tanto, é necessário que haja alguma forma de replicação dos serviços essenciais da grade e que o sistema seja capaz de chavear de um servidor para outro automaticamente quando necessário.

Em ambos os casos, é necessário um mecanismo para **detecção de falhas** [Hayashibara et al. 2002] de forma que o sistema de gerenciamento da grade seja notificado quando nós falham para que ele possa tomar ações no sentido de reestabelecer os processos que foram prejudicados. Quando um nó de computação falha, é necessário saber quais tarefas de quais aplicações estavam em execução naquele nó e providenciar a sua reexecução em outro nó. Quando um nó contendo serviços de gerenciamento da grade falha, é necessário reiniciar esses serviços em uma outra máquina ou então ativar alguma réplica desse serviço; esse caso pode ser mais complicado pois pode ser necessário notificar uma grande quantidade de clientes de que houve uma mudança no endereço do serviço. A maioria dos sistemas de grade atuais oferece apenas algum suporte limitado a tolerância a falhas e, portanto, esta é uma área ativa de pesquisa e desenvolvimento em grades.

A *recuperação por retrocesso baseada em checkpointing* é um mecanismo que permite reiniciar uma execução interrompida de uma aplicação a partir de um *checkpoint* gerado anteriormente [Elnozahy et al. 2002]. Um *checkpoint*, ou ponto de salvaguarda, contém o estado consistente de uma aplicação em um determinado ponto de sua execução. Um estado é considerado consistente se ele é um estado possível de ser atingido pela aplicação durante alguma execução. *Checkpointing* [Plank et al. 1995] é o processo de gerar um *checkpoint* contendo o estado de uma aplicação. Existem duas abordagens distintas para realizar o *checkpointing* de uma aplicação, no nível de sistema e no nível da aplicação, que diferem no modo como é realizada a captura do estado da aplicação.

Quando tratamos de aplicações paralelas acopladas, isto é, aplicações onde existem trocas de mensagens entre os processos, é necessário um cuidado extra de modo a garantir que os estados gerados sejam consistentes. Os processos desta aplicação paralela podem possuir dependências entre si, de modo que se salvarmos o estado de cada processo isoladamente, o estado global da aplicação pode não ser um estado válido. Diversos protocolos para *checkpointing* de aplicações paralelas foram propostos para garantir a geração de *checkpoints* consistentes [Garcia and Buzato 1999, Elnozahy et al. 2002].

No entanto, protocolos e técnicas para gerar *checkpoints* são apenas uma parte do processo de recuperação por retrocesso. É necessário ainda determinar onde e como armazenar esses *checkpoints*, bem como definir algoritmos para encontrar e eliminar *checkpoints* inúteis. Finalmente, é preciso um mecanismo para detectar falhas de aplicações e reiniciá-las a partir do último *checkpoint* salvo.

Numa aplicação paralela com processos sendo executados em vários nós simultaneamente, são gerados *checkpoints globais* constituídos pelos *checkpoints* produzidos por cada processo da aplicação. Porém, esses *checkpoints* globais não formam necessariamente um estado global consistente. Este problema ocorre devido a uma possível dependência de causalidade entre os processos, produzida devido à troca de mensagens. Desse modo, quando os *checkpoints* dos processos são gerados de maneira arbitrária, é possível que, após uma falha, os processos da aplicação paralela tenham que utilizar *checkpoints* anteriores ao último. No pior caso, a aplicação pode ser forçada a reiniciar a partir de seu estado inicial. Este problema não ocorre com aplicações não-acopladas do tipo saco de tarefas (*bag-of-tasks*, vide Seção 2.2.8), onde a execução de cada processo é independente dos demais.

Diferentes protocolos de *checkpointing* tratam este problema de maneiras distintas. Alguns deixam os processos da aplicação escolherem livremente seus *checkpoints* enquanto outros requerem uma coordenação global desse processo de escolha, de modo a garantir que todos os *checkpoints* gerados formem um estado global consistente.

Dizemos que temos um ***checkpoint* global consistente** quando o estado formado pelos *checkpoints* locais de cada processo mais o canal de comuni-

cação é consistente. Como já vimos, um fator que causa inconsistências no estado global da aplicação são as relações de precedência causais. Dizemos que um *checkpoint* local c de um processo p precede causalmente o *checkpoint* local c' de um processo p' se, e somente se, um evento e, posterior a c em p, precede um evento e', anterior a c' em p'. Em particular, um *checkpoint* global consistente não pode possuir dois *checkpoints* locais c e c' com relação de precedência causal $c \rightarrow c'$, uma vez que c' só poderia ocorrer após a ocorrência de c e não simultaneamente. Na verdade é possível provar uma relação muito mais forte entre *checkpoints* globais consistentes e relações de precedência causal, que diz que um *checkpoint* global é consistente se, e somente se, não existe nenhuma relação de precedência causal entre seus *checkpoints* constituintes. A prova pode ser encontrada em [Wang 1997].

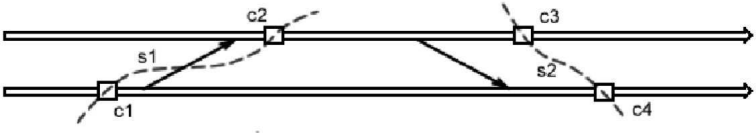

Figura 2.3. Relações de precedência entre *checkpoints*

Na Figura 2.3, que mostra a troca de mensagens entre dois processos distintos, o *checkpoint* $c1$ precede causalmente $c2$, de modo que o *checkpoint* global $s1$ formado por $c1$ e $c2$ não é consistente. Já o *checkpoint* global $s2$ formado por $c3$ e $c4$ é consistente, pois não existem relações de precedência causal.

A **linha de recuperação** corresponde ao *checkpoint* global consistente mais recente que pode ser recuperado. Este é um conceito importante, pois o objetivo da recuperação por retrocesso é reiniciar uma aplicação paralela a partir de sua linha de recuperação. Deve-se notar que esta linha de recuperação não precisa necessariamente corresponder a um estado que ocorreu durante a execução da aplicação, mas a um estado que poderia ter ocorrido. Outra utilização importante da linha de recuperação é na coleta de lixo. Como cada *checkpoint* pode ocupar centenas de megabytes de espaço em disco, é fundamental dispor de um mecanismo para descartar *checkpoints* que não mais são úteis. Uma vez determinada a linha de recuperação, pode-se eliminar todos os *checkpoints* que a precedem.

A dificuldade na determinação da linha de recuperação pode variar dependendo do protocolo de *checkpointing* escolhido. Em alguns casos, o protocolo garante que o último *checkpoint* global é sempre consistente e, nesse caso, a determinação da linha de recuperação é trivial. Já no caso onde os processos gravam seus *checkpoints* de maneira independente, é preciso construir um grafo que captura as dependências entre esses *checkpoints* e percorrê-lo até encontrar a linha de recuperação.

Já a obtenção dos *checkpoints* pode ser feita de duas formas. A primeira, denominada *checkpointing* no nível do sistema, consiste em criá-lo a partir de uma cópia direta do conteúdo do espaço de memória do processo. Esta técnica permite a coleta do *checkpointing* sem nenhuma alteração no código da aplicação. Na verdade, não é nem necessário ter-se acesso ao código-fonte da aplicação, uma vez que o sistema interage diretamente com o processo em execução. A principal desvantagem é que o *checkpoint* produzido é inteiramente dependente da arquitetura de hardware e software utilizada, não permitindo que um *checkpoint* capturado em uma máquina com Linux seja utilizado em uma máquina com Windows, ou que um *checkpoint* capturado em uma máquina com processador Intel de 32 bits seja utilizado em uma máquina com processador Intel de 64 bits ou numa máquina com PowerPC.

Na segunda abordagem, denominada *checkpointing* no nível da aplicação, a aplicação é responsável por fornecer os dados que serão armazenados no *checkpoint*. Esta abordagem possui a desvantagem de exigir alterações no código-fonte da aplicação mas permite a criação de *checkpoints* portáveis, ou seja, que podem ser transportados de uma plataforma para outra [de Camargo 2007, de Camargo et al. 2005].

Finalmente, uma questão importante a ser tratada quando se trabalha com *checkpointing* é o armazenamento dos *checkpoints* de forma eficaz. De pouco adianta armazená-los no próprio nó onde o processo está sendo executado pois quando esse nó cai, os seus *checkpoints* ficam inacessíveis. Portanto, é necessário armazená-los em um repositório central do aglomerado da grade ou então utilizar algum algoritmo mais sofisticado para armazenamento distribuído de *checkpoints*, por exemplo utilizando tabelas de espalhamento distribuídas (*Distributed Hash Tables* ou DHTs). Estudos comparativos mostram que soluções distribuídas, apesar de mais complexas, podem oferecer ganhos substanciais em termos de desempenho no armazenamento e recuperação dos *checkpoints* e também na disponibilidade desse serviço [de Camargo et al. 2006a, de Camargo 2007].

2.2.8. Modelos de Programação

Apesar da grande quantidade de aplicações existentes para grades ligadas ao compartilhamento de espaço de armazenamento, ou até de equipamentos, o recurso mais desejado ainda é o processamento. É importante notar que apesar da analogia inicial entre grades computacionais e redes elétricas (*power grids*), o uso de poder computacional não é simples. Para aproveitar-se dos recursos computacionais de uma grade é necessário desenvolver uma aplicação feita com esse objetivo. Esta aplicação precisa ser escrita segundo algum modelo de programação, que eventualmente expresse o paralelismo, quando for o caso. Um modelo de programação é uma maneira de abstrair a máquina de forma a simplificar a representação da realidade.

Para conseguirmos executar programas paralelos, eles devem estar escritos de forma a permitir a representação do paralelismo. A escolha de um

bom modelo de programação é bem complexa pois, geralmente quanto mais próximos da realidade, mais complexos são os modelos. Por outro lado, modelos excessivamente simplificados não consideram aspectos relevantes da realidade de uma forma suficientemente exata. Logo, a escolha de um modelo é um balanceamento entre a realidade e a simplicidade. Tanto esse tema como a validação de modelos de programação estão fora do escopo desse capítulo.

Apresentamos a seguir uma descrição dos principais modelos utilizados em grades computacionais. Descreveremos em maiores detalhes três modelos: aplicações paramétricas, o MPI e o BSP. Terminamos esta seção com uma descrição sucinta de dois outros modelos (OpenMP e KAAPI) mas ressaltamos que a lista de modelos apresentados está longe de ser exaustiva.

2.2.8.1. Aplicações paramétricas e saco de tarefas

Aplicações paramétricas correspondem a uma classe de aplicações idênticas que são executadas com diferentes parâmetros. O interesse nestas aplicações, além da sua simplicidade, provêm do fato de que boa parte das aplicações paralelas tem esse formato. Aplicações do tipo saco de tarefas (*bag-of-taks*) são ainda mais genéricas, pois correspondem a aplicações onde as tarefas a serem executadas são completamente independentes umas das outras. O seu nome advém da metáfora de um saco cheio de tarefas de onde um escalonador pode ir retirando as tarefas, em qualquer ordem que desejar, e colocá-las para serem executadas em qualquer lugar que suas heurísticas julgarem apropriado.

A simplicidade vem do fato de que a necessidade de comunicação entre tarefas é mínima, pois existem apenas duas comunicações: uma no início, quando são passados os parâmetros iniciais, e uma no término, quando os resultados são transmitidos para serem combinados. Esta simplicidade fez com que esse tipo de aplicação fosse o primeiro a ser efetivamente usado em grades, um ambiente onde podem não existir garantias nem sobre homogeneidade, nem sobre a qualidade da comunicação entre os nós.

A simplicidade desse modelo fornece também algo muito importante para aplicações paralelas: a escalabilidade. Como não há comunicação entre as tarefas durante a computação, aplicações desse tipo podem potencialmente usar uma grande quantidade de máquinas. A paralelização desse tipo de aplicação é tão trivial que elas são também conhecidas como embaraçosamente paralelas (*embarrassingly parallel*). Entre as áreas principais de aplicação desse modelo podemos citar mineração de dados, simulações de Monte Carlo, renderização distribuída, buscas com BLAST em bioinformática e aplicações em física de partículas.

Um ótimo exemplo da escalabilidade desse tipo de aplicação pode ser comprovado através do projeto SETI@home (`setiathome.berkeley.edu`), um projeto de computação voluntária desenvolvido pela Universidade da Califórnia em Berkeley. Nele, voluntários de todo o mundo doam ciclos de computação

em um projeto para a busca de padrões em transmissões recebidas por radiotelescópio. Até a escrita desse capítulo (janeiro de 2008), a potência de cálculo do projeto chega a 387 TeraFLOPS, o que corresponderia à segunda colocação na mais recente lista TOP 500 (`www.top500.org/list/2007/11/100`) dos computadores mais rápidos do mundo. No sítio do projeto podem ser encontrados alguns outros números bem impressionantes. Na esteira do sucesso do SETI@Home, o mesmo grupo criou o middleware BOINC (`boinc.berkeley.edu`) que permite a execução de aplicações do tipo saco de tarefas genéricas.

Um outro projeto interessante ainda nesta linha é o de busca de números primos no formato Mersenne (*The Great Internet Mersenne Prime Search*, `www.mersenne.org`), ou seja, números primos da forma $2^n - 1$. Como no projeto anterior, esse também é baseado em computação voluntária.

Apesar de aplicações paramétricas serem bastante adequadas para grades computacionais, existem aplicações que, quando paralelizadas apresentam padrões mais complexos de comunicação entre as tarefas. Para isto, é necessário algum mecanismo para troca de mensagens entre os nós. Veremos agora algumas formas de fornecer mecanismos para isto.

2.2.8.2. MPI

O *Message Passing Interface* (MPI) é o padrão criado pelo MPI Forum (`www.mpi-forum.org`) que define interfaces, protocolos e especificações para a comunicação entre computadores. Os principais objetivos de MPI são desempenho, escalabilidade e portabilidade. Apesar de ser considerado de baixo nível, isto é, com instruções muito específicas, MPI ainda é o principal padrão para computação paralela. Pode-se inclusive fazer uma analogia com a linguagem Fortran que, apesar de ser antiga e obsoleta em vários aspectos, ainda é muito usada no cálculo científico.

Um outro ambiente distribuído bastante popular na década de 1990, semelhante ao MPI, é o PVM (*Parallel Virtual Machine*). Apesar de ser contemporâneo e de oferecer funcionalidades semelhantes ao MPI, não foi tão utilizado e difundido. Do ponto de vista de pesquisa, esses dois ambientes ainda continuam importantes e são, por exemplo, o assunto principal de conferências científicas internacionais tais como a EuroPVMMPI (`pvmmpi08.ucd.ie`).

Existem duas versões principais de MPI: a versão inicial, com primitivas principalmente para troca de mensagem e ambiente de tamanho fixo, o MPI-1 (que na verdade corresponde à versão 1.1), e a versão mais nova, e muito mais dinâmica, o MPI-2. Entre as características mais importantes da segunda versão podemos citar: criação dinâmica de processos e operações de leitura e escrita em memória remota. Os programas escritos em versões antigas do MPI podem ser usados na versão mais nova. Existem bibliotecas que implementam o padrão MPI para diversas linguagens, tais como C, C++ e Fortran. As implementações mais conhecidas do padrão são MPICH (`www.mcs.anl.gov/research/projects/mpich2`), LAM/MPI (`www.lam-mpi.org`) e, mais re-

centemente, Open MPI (`www.open-mpi.org`), todas estas gratuitas, com código aberto.

Uma importante vantagem competitiva do MPI com relação a outras bibliotecas de comunicação paralela é que existem implementações específicas para diversos tipos de placas de rede de alta velocidade, assim como para computadores paralelos de última geração. É interessante observar que estes fabricantes têm interesse em fornecer implementações para MPI, de forma a popularizar o hardware a ser vendido.

O modelo inicial de comunicação do MPI foi o de troca de mensagens. Nele, um conjunto de processos independentes, com memórias locais, pode trocar informações através do envio de mensagens. Estas trocas são feitas através de primitivas específicas para o envio (`MPI_Send()`) e sua respectiva recepção (`MPI_Recv()`). Isto é, espera-se que exista alguma espécie de sincronização para a troca de mensagens. Veremos, em seguida, que esta sincronização pode ser efetuada de diversas formas.

É interessante ressaltar que os programas escritos em MPI seguem o modelo *Single Program Multiple Data* (SPMD), onde programas idênticos são executados por todos os processos, cada um com os seus dados. Um dos dados locais, que diferencia um processo do outro, é o seu número de identificação. Veremos em breve como usar isto na prática.

Programação MPI

Aconselhamos ao leitor interessado em aprender MPI a obter um dos ambientes e a seguir os tutoriais passo a passo disponíveis nos mesmos. Veremos agora alguns conceitos básicos ligados ao MPI mas, devido ao caráter generalista desse texto, nos limitaremos principalmente à versão MPI-1.

Em todo programa MPI deve ser definido o início e o fim da seção paralela através das chamadas `MPI_Init()` e `MPI_End()`, respectivamente. Observe o exemplo abaixo escrito em C:

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv) {
   int  rc;
   rc = MPI_Init (&argc, &argv);
   if (rc == MPI_SUCCESS) {
     printf ("MPI iniciou corretamente.\n");
   }
   MPI_Finalize ();
   return 0;
}
```

O programa acima pode ser compilado em um ambiente com MPI, através do comando `mpicc`, ou semelhante, que na verdade é um *script* que passa alguns parâmetros importantes ao compilador C. Normalmente, os ambientes de programação MPI oferecem os comandos `mpirun`, ou `mpiexec`, que recebem

como parâmetros, entre outros, o número de processos a serem criados e o programa a ser executado.

É interessante observar que o número de processos não depende do número físico de máquinas e que os processos são criados seguindo a política *round-robin* nas máquinas especificadas. No caso específico do programa acima, se o executarmos em 20 máquinas, teremos 20 mensagens impressas. É interessante notar que apesar dos processos poderem executar em máquinas diferentes, o ambiente MPI faz com que as impressões na saída padrão sejam direcionadas à máquina onde foi iniciado o programa MPI.

O conceito inicial para entendermos um pouco mais do MPI é o "comunicador" (*communicator*), que define um grupo de processos, cada um com um identificador único. Desta forma, existe a possibilidade de um processo enviar uma mensagem para outro. O exemplo a seguir usa o comunicador de uma forma bem simples.

```
int main(int argc, char **argv) {
  int  num_of_processes, rank, rc;
  rc = MPI_Init (&argc, &argv);
  if (rc == MPI_SUCCESS) {
    MPI_Comm_size (MPI_COMM_WORLD, &num_of_processes);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    printf ("Sou o processo %d de %d\n", rank, num_of_processes);
  }
  MPI_Finalize ();
  return 0;
}
```

No exemplo acima, cada processo imprime o seu número individual (`rank`) e o número total de processos que estão sendo executados. Usando o `rank` é possível referenciar outros processos que participam da computação paralela e construir programas que trocam mensagens. Inicialmente, abordamos algumas primitivas ponto a ponto, isto é, entre pares compostos de transmissor e receptor. As mensagens são compostas por duas partes: dados e envelope. Os dados são compostos por tipos escalares ou vetores, e seus respectivos tipos. O MPI oferece vários tipos básicos e também permite a criação de novos tipos. No entanto, apesar desta flexibilidade muitos programadores optam pelo uso de vetores de bytes ao invés de estruturas tipadas, o que nem sempre é uma boa idéia pois torna o código mais sujeito a erros de programação. O envelope contém meta-dados da mensagem, como origem, destino, *tag* (que é um identificador da mensagem) e o comunicador ao qual a mensagem se refere.

```
int main(int argc, char **argv) {
   int numtasks, rank, dest, source, rc, count, tag = 1;
   char inmsg, outmsg; MPI_Status Stat;
   MPI_Init (&argc, &argv);
   ...
```

```
   if (rank == 0) {
       dest = 1;
       outmsg='x';
       printf ("Enviando caractere %c para proc %d\n",
               outmsg, dest);
       rc = MPI_Send (&outmsg, 1, MPI_CHAR, dest,
                      tag, MPI_COMM_WORLD);
   }
   else if (rank == 1) {
      source = 0;
      rc = MPI_Recv (&inmsg, 1, MPI_CHAR, source, tag,
                     MPI_COMM_WORLD, &Stat);
      printf ("Recebi o caractere: %c do proc %d\n",
              inmsg, source);
    }
   MPI_Finalize ();
}
```

No trecho de código acima temos um programa para dois processos, onde o processo com `rank` 0 envia uma mensagem para o processo com `rank` 1 com `tag` 1. Pode-se evitar a recepção com tags usando-se a constante `MPI_ANY_TAG` e, de forma semelhante, aceitar mensagens de qualquer destinatário com a constante `MPI_ANY_SOURCE`. É interessante observar no código acima que, apesar do programa ser único, cada processo executa trechos de código independentes.

O padrão de comunicação acima é o mais simples, pois é síncrono, tanto no envio, como na recepção, isto é, enquanto a mensagem não for efetivamente enviada, o `MPI_Send()` fica bloqueado. O `MPI_Receive()` também fica bloqueado até o recebimento da mensagem. Existem outros tipos de envio e recepção que são não bloqueantes. A esses está associada a instrução `MPI_WAIT()`, que aguarda a finalização de operações de envio ou recepção.

Finalmente, além das trocas de mensagem ponto a ponto, o MPI também oferece diversas primitivas para operações entre um conjunto (comunicador) de processos. Existem desde operações como `MPI_Bcast()` que envia uma mensagem para todos os processos de um grupo e `MPI_Reduce()` que aplica uma operação em dados de todos os processos de um grupo, até primitivas para a troca completa de dados de tamanhos diferentes entre todos os processos `MPI_Alltoallv()`.

Quando o exemplo abaixo é executado com n processos, o processo 0 envia o valor 10 para todos os outros (`MPI_Bcast()`). Os parâmetros correspondem ao endereço inicial dos dados, ao número de dados, ao tipo dos dados, ao iniciador e, finalmente, ao comunicador. Em seguida, cada um calcula o seu próprio `rank` mais um multiplicado pelo valor recebido. Finalmente, a soma de todos os valores calculados é enviada ao processo 0, através do `MPI_Reduce()`, que tem parâmetros semelhantes à difusão e também o tipo de operação `MPI_SUM` e o destinatário do resultado.

```
int main(int argc, char **argv)  {
   int rank, source = 0, dest = 0, rc;
   int value, totalValue;
   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   if (rank == 0)
      value = 10;
   MPI_Bcast (&value, 1, MPI_INT, source, MPI_COMM_WORLD);
   value *= rank + 1;
   MPI_Reduce (&value, &totalValue, 1, MPI_INT, MPI_SUM,
               dest, MPI_COMM_WORLD);
   if (rank == 0)
      printf ("Valor final calculado: %d\n", totalValue);
   MPI_Finalize();
   return 0;
}
```

O MPI-1 também permite a criação de outros comunicadores, de topologias de comunicação específicas, entre outros. Entretanto, a sua versão final é de 1995. Mais recentemente o MPI-2 foi proposto. Veremos algumas de suas funcionalidades a seguir.

Por que o interesse pelo MPI-2?

Na versão 2 do MPI temos algumas primitivas muito interessantes para ambientes dinâmicos, onde o número de máquinas disponíveis pode aumentar com o tempo. O principal deles é o `MPI_COMM_SPAWN()`, através do qual um processo, pode criar em tempo de execução outros processos, possivelmente em máquinas distintas. Além disto, é fácil efetuar a comunicação entre os processos criados e seu criador.

Estão também presentes operações em memória remota (pertencente a um outro processo), que também são conhecidas como *one-side-communications*. Mas, para isto, primeiro é necessário criar-se “janelas” nas quais serão disponibilizadas regiões de memória para a leitura/escrita de dados. Além das primitivas óbvias como `MPI_PUT()` e `MPI_GET()`, existem primitivas que realizam operações coletivas e de sincronização. Em ambientes com suporte a memória compartilhada, esse modelo é bem mais conveniente do que a troca de mensagens.

Também estão disponíveis no MPI-2 extensões de comunicações coletivas do MPI-1, novas formas de criar comunicadores e novas operações; suporte para a criação de novas operações não bloqueantes e para *threads*; e, finalmente, formas de se manipular arquivos em paralelo. É importante ressaltar que o MPI-2 permanece compatível com o MPI inicial.

2.2.8.3. BSP

O *Bulk Synchronous Parallel Model* (BSP) [Hill et al. 1997, McColl 1995, Valiant 1990] é um modelo cujo principal objetivo é fornecer um ambiente que

permita a concepção de algoritmos ao mesmo tempo portáveis e eficazes. O modelo BSP não foi baseado em nenhuma máquina real, mas é adequado às máquinas onde a comunicação é baseada em troca de mensagens.

A principal idéia do modelo é a separação explícita do cálculo (computação) e da comunicação. Os seus conceitos principais são a super-etapa ou super-passo (*super-step*) e a sincronização. Um aplicação é dividida em super-etapas sucessivas e todos os processadores começam uma super-etapa ao mesmo tempo. Entre duas super-etapas sucessivas sempre existe uma etapa de sincronização. Para garantir a troca de mensagens entre os nós, os dados enviados durante uma super-etapa estarão disponíveis nos seus destinos no início da próxima super-etapa, e não antes disso. A Figura 2.4 ilustra o esquema de execução no modelo BSP.

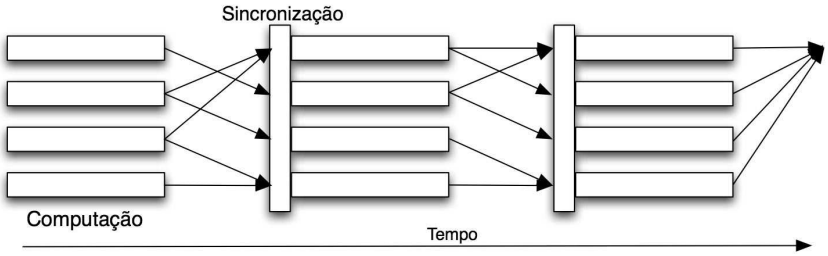

Figura 2.4. Esquema de Execução no modelo BSP

Os três parâmetros utilizados para descrever esse modelo são: p, l e g. Nesse texto, utilizamos as notações propostas em [McColl 1995].

p — número de processadores;

l — custo de uma sincronização global;

g — tempo para transmitir um byte pela rede. Ou seja, $\frac{1}{g}$ é a banda passante.

Para limitar a quantidade de comunicações entre processos, uma máquina no modelo BSP é capaz de efetuar uma $\lceil \frac{l}{g} \rceil$-relação em cada super-etapa. Uma h-relação é uma operação de troca de dados entre processadores, na qual cada processador pode enviar e receber no máximo h palavras.

Uma grande vantagem de usar BSP ao invés de MPI é a possibilidade de se prever o tempo de um algoritmo em um novo suporte de execução; isto pode ser feito estimando o custo máximo de cada super etapa (que corresponde ao máximo do custo de vários programas seqüenciais), calculando o custo da comunicação (que é limitado) e o custo de sincronização. Como conseqüência direta das sincronizações, temos estados globais consistentes, o que facilita a criação de *checkpoints* (vide Seção 2.2.7). Por outro lado, o uso de sincronizações globais faz com que programas BSP não sejam tão facilmente escaláveis no número de processos.

As etapas para obter uma aplicação eficiente no modelo BSP são: balancear a carga de cálculo entre os processadores em cada super-etapa, balancear as comunicações em cada super-etapa (evitar congestionamento) e, finalmente, reduzir o número de super-etapas.

Um outro modelo muito semelhante ao BSP, mas com viés mais teórico é o *Coarse Grained Multicomputers* (CGM) [Dehne et al. 1993]. Nesse modelo simplificado, existem apenas dois parâmetros: o número de processadores p e o tamanho do problema n. Além disto, para que o modelo seja de granularidade grossa, supõe-se que $n/p >> p$. No modelo, também estão definidas algumas restrições como o tamanho da memória local em cada processador, limitada a $O(n/p)$. Além disto, em cada rodada de comunicação é possível apenas efetuar uma h-relação, isto é, em cada rodada, cada processador envia e recebe no máximo $O(n/p)$ dados. Desta forma, o tempo no modelo CGM é dado pelo número de rodadas de comunicação.

É interessante ressaltar que apesar de não considerar diversos detalhes das máquinas, os algoritmos elaborados no modelo CGM, quando implementados, apresentam resultados muito próximos aos previstos [Dehne 1999].

Existem bibliotecas específicas de comunicação no modelo BSP tais como PUB (`www.uni-paderborn.de/~bsp`) e o *Oxford BSP toolset* (`www.bsp-\ -worldwide.org`). Como veremos na Seção 2.3.2, o sistema InteGrade oferece uma implementação compatível com a de Oxford para grades computacionais oportunistas. Além destas, uma interessante iniciativa de Suijlen (`bsponmpi.sourceforge.net`) apresenta uma implementação da BSPlib sobre MPI, o que permite uma grande portabilidade.

Outra característica interessante dos modelos CGM e BSP é a possibilidade de se elaborar o algoritmo buscando o melhor desempenho para, em seguida, implementá-lo usando outros modelos de programação, por exemplo, baseados em passagem de mensagem como o MPI.

Programação BSP

A BSPlib, assim como o MPI-2, provê duas formas de comunicação: a troca de mensagens e o acesso a memória remota. Usando *Distributed Remote Memory Addressing* (DRMA), os processos podem realizar leituras e escritas na memória local ou remota. Com *Bulk Synchronous Message Passing* (BSMP) a comunicação funciona por troca de mensagens. As mensagens são enviadas para a fila de entrada do processo remoto e cada processo pode acessar a sua fila de mensagens locais. Os comandos da BSPlib têm nomes bastante mnemônicos:

bsp_begin (nprocs) — inicia um programa BSP com $nprocs$ processos;

bsp_end() — final da parte paralela;

bsp_pid() — fornece o id do processo;

bsp_nprocs() — fornece o número total de processos;

bsp_sync() — delimitador de super-passos.

Assim como em MPI, para acessos à memória remota é necessário registrar as regiões a serem acessadas. Em DRMA, isto é feito através da instrução `bsp_push_reg()`, que recebe como parâmetros um endereço de uma região de memória e o seu tamanho. Para remover o registro pode-se usar o comando `bsp_pop_reg()`.

Para escrever em memória remota usa-se a instrução `bsp_put()` que recebe como parâmetros o id do processo destino, o endereço inicial dos dados no processo corrente, o endereço no processo destino (que deve estar previamente registrado), um deslocamento (*offset*) e a quantidade de bytes a serem copiados. Graças a esse deslocamento pode-se escrever em qualquer região da memória registrada, e não apenas no seu início. A leitura se faz de forma semelhante com a chamada `bsp_get()`.

Abaixo vemos um exemplo de cálculo de produto escalar, supondo que cada processo contém uma parte contínua dos vetores. Isto é, cada processo deve efetuar o produto escalar local e, em seguida, enviar o resultado para que todos os outros processos possam calcular o produto escalar.

```
#include "BspLib.hpp"
int main (int argc, char **argv) {
  int pid, nprocs = 4;
  double produto, *listaProdutos;  // vetor local a cada processo
  listaProdutos = (double *) malloc (nprocs * sizeof (double));
  bsp_begin (nprocs);
  // registra memória a ser compartilha com demais processos
  bsp_push_reg (&listaProdutos, nprocs * sizeof (double));
  pid = bsp_pid ();
  // cálculo do produto com os pedaços de vetores locais
  prod = calculaProdutoLocal (pid, nprocs);
  // envia os resultados a todos os outros processos
  for (int proc = 0; proc < bsp_nprocs( ); proc++)
    // escreve em proc um valor (double) em listaProdutos usando
    // pid como offset (isto é, o processo i grava na posição i)
    bsp_put (proc, &produto, &listaProdutos, pid,sizeof(double));
  bsp_sync ( ); // final do super-passo
  calculaProdutoEscalarFinal (listaProdutos);
  bsp_end( );
}
```

No exemplo acima, temos a função `calculaProdutoLocal()` que faz o cálculo do produto escalar com dados locais (por exemplo, provenientes de um arquivo) e, em seguida, temos o envio do resultado desse cálculo a todos os processos (inclusive a si mesmo). A soma dos resultados recebidos é feita através da função `calculaProdutoEscalarFinal()`.

No modo de comunicação BSMP, cada processo mantém uma fila de mensagens. Processos remotos colocam mensagens nesta fila. O processo local lê mensagens desta fila. As funções principais são:

- `bsp_send(int pid,void *tag,void *payload,int nbytes)` – envia uma mensagem de tamanho `nbytes` contida em `payload` e com o identificador `tag` para a fila de mensagens do processo `pid`.
- `bsp_set_tagsize(int *nbytes)` – define o tamanho da `tag` como `nbytes`.
- `bsp_get_tag(int *status, void *tag)` – se houver mensagens na fila, devolve o tamanho da próxima mensagem em `status` e o conteúdo da *tag* da mensagem em `tag`.
- `bsp_move(void *payload, int nbytes)` – copia o conteúdo da primeira mensagem da fila em `payload`. A variável `nbytes` representa o tamanho do buffer apontado por `payload`.
- `bsp_qsize(int *packets, int *nbytes)` – devolve o número de mensagens na fila em `packets` e o tamanho total das mensagens em `nbytes`.

Abaixo, repetimos o programa para cálculo de produtos escalares mas agora usando troca de mensagens.

```
#include "BspLib.hpp"
int main(int argc, char **argv) {
  int pid, nprocs = 4;
  double produto;
  double *listaProdutos = (double *) malloc(nprocs *
                                        sizeof(double));
  bsp_begin (nprocs);
  pid = bsp_pid ();
  produto = calculaProdutoLocal (pid, nprocs);
  int tagsize = sizeof(int);
  bsp_set_tagsize( &tagsize );
  // envio do produto escalar local a todos os outros
  for (int proc = 0; proc < bsp_nprocs( ); proc++)
    bsp_send (proc, &pid, &produto, sizeof(double));
  bsp_sync ( );
  for (int proc = 0; proc < bsp_nprocs( ); proc++) {
    int messageSize, source;
    bsp_get_tag (&messageSize, &source)

    bsp_move( &listaProdutos[source], messageSize)
  }
  calculaProdutoEscalarFinal (listaProdutos);
  bsp_end( );
}
```

No exemplo acima, temos duas super-etapas. Na primeira, ocorre o cálculo do produto escalar com os dados locais e o envio a cada um dos outros processos dos dados calculados. Para isto, cada processo efetua a chamada `bsp_send()` `nprocs` vezes. A `tag` é utilizada para identificar o processo

emissor no momento da leitura dos dados da fila local, após a primeira superetapa. Logo, primeiro o comando `bsp_get_tag()` recupera as informações sobre uma das mensagens da fila local e, em seguida, o `bsp_move()` armazena esta mensagem na posição correta `source` do vetor `listaProdutos`.

2.2.8.4. Outros modelos

A seguir veremos descrições sucintas de dois outros modelos, mais modernos, usados para expressar paralelismo. Um deles foi projetado para ambientes com memória compartilhada, uma das tendências dos computadores atuais. O segundo funciona em ambientes dinâmicos, pois fornece mecanismos para o balanceamento de carga.

OpenMP

Open Multi-Processing (OpenMP) é uma API para sistemas com memória compartilhada com suporte a Fortran, C ou C++. A API consiste de um conjunto de diretivas para compilação, bibliotecas e variáveis de ambiente que controlam o comportamento da aplicação em tempo de execução.

O modelo de execução é baseado em `fork-join`, onde um programa começa com uma única linha de execução denominada *thread* principal. Esta é executada seqüencialmente até encontrar o início do trecho paralelo definido pela palavra reservada `Parallel` que é equivalente ao `fork`, seguida por um `End Parallel`, equivalente ao `join`. Nesse ponto, um conjunto de *threads* é criado, sendo que a *thread* principal faz parte desse conjunto. Cada *thread* possui a sua própria área de dados, mas também pode compartilhar dados desde que isso seja especificado no início do trecho paralelo. Não existem instruções explícitas para troca de mensagens, o padrão é ter áreas de memória compartilhadas. Abaixo vemos a estrutura geral de um código OpenMP em C.

```
#include <omp.h>
main () {
   int varp1, varp2, vars1;
   // Código seqüencial
   ...
   // Inicio do trecho paralelo, e escopo das variáveis
  #pragma omp parallel private (varp1, varp2) shared(vars1) {
     // Parte paralela (todas as threads)
     ...
  } // Ponto de sicronizacao com a thread principal
  // Continuação do código seqüencial
  ...
}
```

KAAPI

O *Kernel for Asynchronous and Adaptive Parallel Interface* (KAAPI) é uma biblioteca que permite o desenvolvimento e execução de aplicações distribuídas baseadas em fluxos de trabalho (*workflow*). Esta biblioteca comporta apli-

cações com diferentes graus de granularidade e é direcionada a aglomerados de máquinas multiprocessadas e grades computacionais.

Programas para o KAAPI são escritos utilizando *Athapascan*, a API disponibilizada pela biblioteca. O modelo de programação é baseado em um espaço de endereçamento global e permite descrever as dependências de dados entre os processos da aplicação. Novas tarefas, que são executadas em paralelo com as tarefas atuais, são criadas utilizando a palavra chave `fork`. Uma interface de baixo nível permite a manipulação mais detalhada do fluxo de tarefas.

O escalonamento das tarefas aos processadores alocados é realizado dinamicamente e utiliza um algoritmo do tipo roubo de trabalho (*work-stealing*), isto é, quando um processador está ocioso, ele rouba tarefas de processadores que estão ocupados. O KAAPI fornece tolerância a falhas à execução de aplicações utilizando um protocolo de *checkpointing* denominado TIC (*Theft Induced Checkpointing*). Nesse protocolo, o estado do fluxo de trabalho de um processador é salvo sempre que uma tarefa alocada a esse processador é roubada por outro processo. Além disso, ao invés de armazenar o estado do processo nos *checkpoints*, o KAAPI armazena apenas o estado do fluxo de trabalho atribuído a cada processador, diminuindo assim o tamanho dos *checkpoints* gerados.

Segue abaixo um exemplo de código do KAAPI. Ele lança uma *thread* `main`, que por sua vez lança dez *threads* `print_hello` que imprimem “hello world” e sua identificação.

```
#include <iostream>
#include <athapascan-1.h>
struct print_hello {
  void operator()( a1::Shared_r<int> id )  {
    std::cout << "hello world from "
              << id.read() << " !" << std::endl;
  }
};
struct do_main  {
  void operator()(int argc, char **argv) {
    int numero_de_iteracoes = 10;
    for (int i = 0; i < numero_de_iteracoes; i++) {
      a1::Shared<int> id(i);
      a1::Fork<print_hello>()(id); // cria uma nova tarefa
    }
  }
};
int main(int argc, char **argv) {
  a1::Community com = a1::System::join_community(argc, argv);
  a1::ForkMain<do_main>()(argc, argv); // main fork
                              // executado por um único nó
  com.leave();
  a1::System::terminate();
  return 0;
}
```

2.3. Middleware de grades

Alguns sistemas distribuídos desenvolvidos nas décadas de 1980 e 1990, tais como Condor [Litzkow et al. 1988], Amoeba [Tanenbaun et al. 1990], Legion [Grimshaw et al. 1997] e Globe [van Steen et al. 1999], já poderiam ser classificados como grades computacionais mesmo tendo sido desenvolvidos antes do termo *Grid* ter sido cunhado. Mas o primeiro sistema que se autodefiniu como uma grade computacional foi o Globus, desenvolvido a partir do final da década de 1990. Descrevemos a seguir, em maiores detalhes, três sistemas de middleware para grades computacionais. O primeiro, Globus, é o mais conhecido e utilizado internacionalmente. Os demais, InteGrade e OurGrid, são dois dos principais projetos brasileiros de grades computacionais e apresentam características próprias que os diferem dos demais.

2.3.1. Globus

O Globus [Foster and Kesselman 1997, Foster and Kesselman 2003] é a iniciativa de maior impacto na área de Grades Computacionais e atualmente é utilizado por cientistas de diversas áreas em milhares de projetos de pesquisa em todo o mundo. Suas origens remontam ao I-WAY [FGN+97], uma grade temporária composta por 11 redes de alta velocidade e 17 institutos de pesquisa concebida para funcionar por apenas algumas semanas, às vésperas do congresso *Supercomputing'95*. O sucesso da iniciativa incentivou o desenvolvimento do sistema Globus. O projeto é desenvolvido conjuntamente, principalmente nos Estados Unidos, por um conjunto de universidades e laboratórios de pesquisa dentre os quais se destacam a Universidade de Chicago, Laboratório Nacional de Argonne (ANL), Universidade de Illinois, *National Center for Supercomputing Applications* (NCSA), e a Universidade do Sul da Califórnia.

A versão 1.0 do *Globus Toolkit*, o middleware do sistema globus, foi lançada em 1998 e era utilizada principalmente pelos desenvolvedores do projeto. A versão 2.0, lançada em 2002, já trazia uma série de novas funcionalidades e facilidades de instalação e uso, o que disseminou o uso do Globus por instituições de pesquisa em todo o mundo. A partir da versão 3.0, lançada em 2003, houve uma mudança significativa: o sistema passou a se adequar a uma arquitetura e a um conjunto de interfaces padrão definidas por um comitê de padronização surgido a partir da equipe envolvida com o projeto Globus e seus colaboradores, o *Global Grid Forum*. Esta arquitetura, denominada *Open Grid Services Architecture* (OGSA) vem sendo refinada continuamente desde a sua criação em 2002 e tem o sistema Globus como sua implementação de referência. Em 2006, o *Global Grid Forum* se uniu a uma aliança de empresas interessadas em tecnologia de grades, a *Enterprise Grid Alliance*, para formar o *Open Grid Forum*.

O OGSA utiliza serviços Web (*Web services*) como middleware de baixo nível para a comunicação e os serviços da arquitetura são definidos através de especificações em WSDL (*Web Services Description Language*). A quarta versão do *Globus Toolkit*, chamada de GT4, é a versão mais atual e é distri-

buída como software livre a partir do sítio do projeto: `www.globus.org`. Os principais serviços oferecidos pelo GT4 podem ser divididos em cinco grupos descritos a seguir:

- Os **Serviços de Informação** auxiliam no gerenciamento de vários tipos de informação. O *Monitoring and Discovery Service (MDS)* permite a publicação e consulta de informações sobre a disponibilidade de recursos na grade. O *Trigger* é um serviço que coleta dados de vários recursos e, quando certas regras definidas pelo administrador são validadas, dispara um gatilho que executa ações pré-definidas; por exemplo, pode ser usado para enviar uma mensagem a um usuário quando sua cota em disco está prestes a estourar ou para enviar uma mensagem ao administrador quando a fila de execução de aplicações está longa demais. O *Index* é um serviço que permite concentrar, em um único índice centralizado, um conjunto de informações sobre recursos espalhados em vários locais. Espera-se que cada organização virtual disponibilize um ou mais índices dos recursos disponíveis para aquela organização.
- O **Gerenciamento de Execução** é efetuado pelo componente *Grid Resource Allocation and Management* (GRAM) que é responsável por localizar, submeter, monitorar e cancelar tarefas em execução nos recursos computacionais da grade. Um componente chamado *Dynamic Accounts* permite a um usuário da grade com as devidas permissões criar dinamicamente um espaço de trabalho em um sítio remoto; atualmente isto é implementado criando-se uma nova conta Unix para cada novo espaço de trabalho.
- O **Gerenciamento de Dados** é obtido através de uma coleção de serviços e ferramentas. O GridFTP permite a distribuição de arquivos para os nós da grade. O *Replica Location Service* permite o registro e a recuperação de réplicas no sistema distribuído, fazendo o mapeamento entre nomes lógicos e nomes físicos; seu objetivo é fornecer acesso robusto a dados com alta disponibilidade mesmo em condições adversas em redes de grande área.
- Os serviços de **Segurança** são fortemente baseados em credenciais X.509 para identificação de usuários, serviços e recursos, no protocolo SSL para comunicação segura e na ferramenta OpenSAML `www.opensaml.org` para gerenciamento de autenticação, atributos e autorizações. O serviço conta também com suporte a delegação, o que permite que um usuário delegue alguns de seus direitos de acesso a certos recursos para outros usuários por um período determinado de tempo. Finalmente, o *Community Authorization Service* permite que o administrador de uma organização virtual gerencie coletivamente um grande conjunto de recursos e usuários especificando políticas de segurança e autorizações definidas através do padrão SAML (*Simple Assertion Markup Language*).

- Finalmente, uma série de **bibliotecas de tempo de execução** formam o *Common Runtime*, componentes que auxiliam no desenvolvimento da infra-estrutura da grade e de aplicações a serem nela executadas. A XIO é uma biblioteca extensível em C usada para a entrada e saída de dados; ela provê uma interface única, baseada nas operações `open`, `close`, `read` e `write` que pode ser usada com uma série de tipos diferentes de protocolos de comunicação incluindo TCP, UDP, arquivos locais, HTTP, GSI, GSSAPI_FTP, Telnet e filas. Um conjunto de bibliotecas chamado de *C Common Libraries* provê uma camada de abstração unificada para tipos e estruturas de dados e para chamadas ao sistema operacional e à `libc` que deve ser usada por todas aplicações e serviços do Globus. Por último, um conjunto de bibliotecas é usado para prover suporte aos protocolos de serviços Web usados pelo Globus nas linguagens C, Java e Python.

Atualmente, boa parte da pesquisa em grades é desenvolvida no âmbito do Globus e também um grande número de usuários de grades, tanto científicos quanto empresariais, tem o globus como seu sistema de grades.

2.3.2. InteGrade

O Projeto InteGrade (`www.integrade.org.br`) é uma iniciativa de seis universidades brasileiras (USP, PUC-Rio, UFMS, UFG, UFMA e UPE) e visa desenvolver um middleware inovador que permita a utilização de recursos computacionais ociosos existentes em instituições acadêmicas e empresariais para a execução de aplicações científicas, industriais e multimídia que demandem alto poder computacional. O middleware é baseado em tecnologias avançadas de objetos distribuídos e pretende dar suporte à execução de aplicações paralelas onde haja um nível significativo de comunicação entre os nós (ao contrário de aplicações tipo saco de tarefas onde não há comunicação entre os nós).

O middleware do InteGrade [de Ribamar Braga Pinheiro Junior et al. 2005, Goldchleger et al. 2004] permite a formação de *Grades Computacionais Oportunistas*, i.e., coleções de aglomerados de computadores formados a partir de máquinas já existentes em um grupo de instituições [de Camargo et al. 2006b]. Podem fazer parte da Grade, por exemplo, PCs em um laboratório de alunos, computadores pessoais de secretárias e professores, Macintoshes em um escritório de design gráfico, ou mesmo *clusters* de alto desempenho pertencentes a laboratórios de pesquisa ou grandes empresas. O objetivo do InteGrade é oferecer aos usuários da Grade esse enorme poder computacional sem comprometer a Qualidade de Serviço oferecida aos usuários locais e respectivos donos dos computadores compartilhados, os quais devem sempre ter prioridade.

Esta grade oportunista permitirá uma grande economia de recursos financeiros, de infra-estrutura e ambientais. Ao invés de gastar enormes somas para a aquisição de *clusters* dedicados que ocupam um grande espaço, consomem energia elétrica, produzem ruído e calor e, muitas vezes, passam a maior parte

do tempo ociosos, um middleware como o InteGrade permite a utilização dos recursos computacionais já existentes para a realização de computações úteis às instituições nas quais estão inseridos. Em países menos favorecidos como o Brasil, esta reutilização de recursos já existentes permite que as grades computacionais estejam ao alcance de um número bem maior de grupos de pesquisa e de instituições.

A versão atual do sistema conta com cerca de 30 mil linhas de código-fonte e pretende ser multi-plataforma, incluindo versões para Linux, Windows e MacOS X. Os serviços administrativos que são executados nos nós de gerenciamento da grade são escritos na linguagem Java de forma a oferecer a maior portabilidade possível. Já os componentes que são executadas nas máquinas dos usuários que compartilham seus recursos com a grade são implementados nas linguagens C e Lua de forma a minimizar o consumo de memória e processador a fim de não prejudicar a qualidade de serviço desses usuários. A comunicação entre os nós da grade é feita através do padrão CORBA e os serviços são acessados através de interfaces definidas em IDL (*Interface Definition Language*). Também com o intuito de minimizar o impacto nas máquinas dos usuários, nos nós provedores de recursos utiliza-se para comunicação o OiL (*ORB in Lua*), um ORB CORBA reflexivo e reconfigurável [Maia et al. 2005] que apresenta um pequeno consumo de memória.

A Figura 2.5 apresenta a arquitetura geral do InteGrade com seus principais componentes, que descrevemos a seguir.

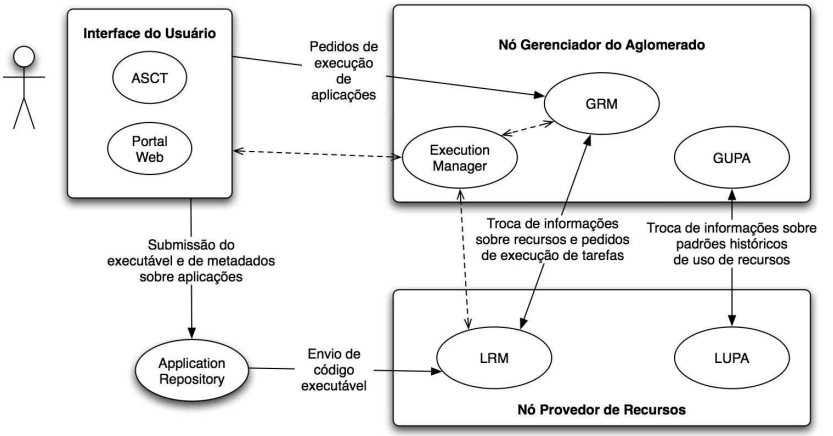

Figura 2.5. Arquitetura geral do InteGrade

- O *Application Submission and Control Tool* (ASCT) é a ferramenta que permite que usuários da grade realizem requisições de execução de apli-

cações, controlem a execução destas aplicações e visualizem seus resultados. Esta ferramenta pode ser executada em qualquer máquina, independente desta compartilhar recursos com a grade ou não. O *Portal* (vide Figura 2.6) permite que usuários da grade realizem, utilizando um portal na Internet, as mesmas tarefas que podem ser realizadas com o ASCT. A vantagem do portal é que usuários podem acessar remotamente o InteGrade a partir de qualquer máquina conectada à Internet utilizando um navegador, sem a necessidade de instalação de um software cliente específico do InteGrade.

- O *Global Resource Manager* (GRM) é o módulo responsável pelo gerenciamento dos recursos de um aglomerado, pela interação com outros aglomerados e pelo escalonamento da execução de aplicações, enquanto o *Local Resource Manager* (LRM) gerencia os recursos de uma única máquina. O *Execution Manager* (EM) é responsável por gerenciar os *checkpoints* das aplicações e acompanhar a execução de todas as tarefas das aplicações em execução em um dos aglomerados da grade. O LRM e GRM informam o EM quando aplicações iniciam e terminam a sua execução e o EM coordena o processo de reinicialização de tarefas que estavam executando em nós que falharam ou se tornaram indisponíveis.

  O GRM mantém uma lista dos LRMs ativos e, ao receber requisições para execução de aplicações, escolhe um LRM que atenda às necessidades da aplicação. Para tal, assim que um LRM é inicializado, ele localiza o GRM de seu aglomerado e se registra, enviando informações sobre características e recursos oferecidos na máquina em que reside, como velocidade do processador, quantidade de memória RAM e sistema operacional. Além disso, o LRM envia periodicamente ao GRM informações contendo a quantidade de recursos disponíveis para utilização em sua máquina. O LRM recebe requisições de execução do GRM e determina se estas requisições podem ser aceitas. Em caso afirmativo, um novo processo é criado para realizar a execução. Caso contrário, o LRM notifica o GRM que a execução não pôde ser realizada. Quando a execução de um processo termina, o LRM notifica o fim desta execução ao *Execution Manager* (EM) que notifica o ASCT ou o portal do usuário. O usuário então pode solicitar, quando desejar, os arquivos de saída de sua aplicação.

- O *Application Repository* (AR) armazena de forma segura os executáveis de aplicações submetidas por usuários para execução na grade. Estes executáveis são enviados ao AR através do ASCT ou do Portal, de modo a serem posteriormente utilizados pelos LRMs, quando sua execução é solicitada por um usuário. O AR pode também armazenar os dados de entrada e de saída das aplicações.

- O *Local Usage Pattern Analyzer* (LUPA) e o *Global Usage Pattern Analyzer* (GUPA) são componentes ainda em desenvolvimento e têm como

objetivo analisar o padrão de utilização das máquinas do aglomerado. O LUPA é executado juntamente com o LRM nos nós provedores de recursos, coletando dados da máquina e analisando seu padrão de uso utilizando técnicas de aprendizado de máquina [Mitchell 1997] e clusterização [Everitt et al. 2001]. Esta informação é disponibilizada ao GRM e, futuramente, permitirá um escalonamento mais eficiente das aplicações da grade.

- Finalmente, dois modelos principais de programação paralela com troca de mensagem entre os nós são oferecidos: BSP e MPI. A biblioteca BSPlib do InteGrade [Goldchleger et al. 2005] permite a execução de aplicações paralelas C/C++ do tipo BSP. Para facilitar o uso de aplicações já existentes, a BSPlib do InteGrade utiliza a mesma API da implementação de Oxford, que é a mais conhecida internacionalmente. Uma versão adaptada ao InteGrade da biblioteca MPICH2 possibilita a execuções de aplicações paralelas do tipo MPI escritas em C/C++, Fortan 77 e Fortran 90. A implementação MPICH2 é uma das mais populares da especificação MPI-2 e é mantida pelo Laboratório Nacional de Argonne dos EUA. Dessa forma, aplicações MPI podem ser executadas no InteGrade sem a necessidade de qualquer modificação no código-fonte ou no seu binário.

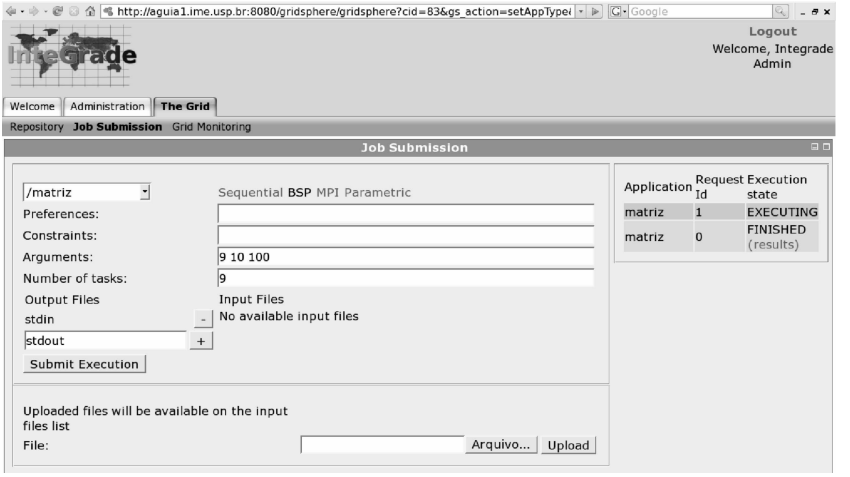

Figura 2.6. Portal do InteGrade para submissão de aplicações

A versão mais recente do InteGrade pode ser obtida como software livre em `www.integrade.org.br` e provê suporte a execução de aplicações paramétricas (saco de tarefas), MPI e BSP. O sistema (1) possibilita a execução de aplicações nas linguagens C, C++, Java e Fortran, (2) contém um serviço de segu-

rança capaz de transmitir todos os dados de forma criptografada e de autenticar usuários e aplicações [de Ribamar Braga Pinheiro Jr. 2008], (3) provê suporte para o armazenamento distribuído de dados [de Camargo and Kon 2007], (4) oferece tolerância a falhas tanto através de uma infra-estrutura de agentes móveis [de Sousa et al. 2006] quanto de *checkpoints* e (5) traz um conjunto de aplicações paralelas acopladas [Alves et al. 2006] que servem de modelo para a criação de novas aplicações paralelas.

2.3.3. OurGrid

OurGrid [Andrade et al. 2003, Cirne et al. 2006] é um dos principais projetos brasileiros na área de Grades Computacionais. Ele é coordenado por pesquisadores da Universidade Federal de Campina Grande e recebeu apoio da empresa HP. O OurGrid oferece um middleware Java para a execução de aplicações seqüenciais e do tipo saco de tarefas (*bag-of-tasks*), ou seja, sem comunicação entre as tarefas.

Uma das principais motivações do projeto foi desenvolver um mecanismo simples para instalação e configuração de grades computacionais dado que as principais infra-estruturas de grade existentes, incluindo o Globus, exigem um grande esforço para a sua instalação e a configuração manual das políticas de compartilhamento entre as várias instituições [Cirne et al. 2003]. Para tanto, desenvolveu-se um mecanismo simples de integração de recursos através de uma rede par-a-par e o compartilhamento desses recursos é baseado no conceito de “rede de favores” que determina que a prioridade de um usuário no uso dos recursos é dada pela razão entre a quantidade de recursos que ele ofereceu à grade e a quantidade de recursos da grade que ele utilizou [de Andrade 2004]. Esta abordagem permite que uma grade seja estabelecida com pouco esforço de configuração e de administração e evita que usuários façam uso de muitos recursos sem compartilhar seus recursos próprios com a grade.

Para prover segurança, o OurGrid utiliza a máquina virtual Xen para proteger os recursos locais de aplicações maliciosas [Barham et al. 2003]. Futuramente, pretende-se também proteger as aplicações de nós maliciosos da grade através de um mecanismo proposto por Sarmenta [Sarmenta 2002] que se baseia em executar réplicas cuidadosamente escolhidas de algumas aplicações em vários nós para detectar possíveis nós sabotadores. Atualmente, OurGrid não oferece suporte a privacidade dos dados e a criptografia dos dados transmitidos por considerar que as aplicações científicas de seus usuários não envolvem confidencialidade.

OurGrid intencionalmente não possui um escalonador global que possua uma visão abrangente da utilização de recursos na grade. Os usuários interagem com a grade através de um agente pessoal chamado MyGrid que é responsável pelo escalonamento das aplicações do usuário e provê um conjunto de abstrações que esconde do usuário a heterogeneidade da grade. Para compensar o fato de que é difícil prever a confiabilidade dos nós de computa-

ção do OurGrid, esse escalonador pessoal utiliza-se da execução replicada da aplicação em mais de um nó. A falta de um escalonador com uma visão abrangente das aplicações em execução no sistema faz com que os escalonadores pessoais compitam entre si e que se desperdice uma certa parcela dos recursos; mas estudos mostram que tal desperdício não necessariamente é grande [Cirne et al. 2006] em alguns contextos. Finalmente, a versão mais recente do escalonador utiliza também o conceito de "afinidade de armazenamento" para executar as tarefas em nós mais próximos de onde os seus dados de entrada estão armazenados, otimizando assim o seu tempo total de execução e a latência percebida pelo usuário [Cirne et al. 2006].

O OurGrid está em produção desde dezembro de 2004 e atualmente reúne dezenas de laboratórios no Brasil e no exterior. Vários pesquisadores de diferentes áreas tais como dinâmica molecular, gerenciamento de recursos hídricos e teste de software utilizam-se do OurGrid para executar suas aplicações. O sistema é distribuído como software livre a partir de `www.ourgrid.org` e qualquer laboratório pode se unir à grade do OurGrid bastando para isso instalar e executar o software.

2.4. Grades em funcionamento

Para dar uma pequena idéia das grades em produção hoje, selecionamos três tipos de grades bem diferentes. Uma acadêmica para pesquisas em Ciência da Computação, em seguida uma grade planetária que também tem como objetivo principal a pesquisa, e finalmente uma grade projetada para oferecer alto desempenho na resolução de aplicações científicas.

Apesar do Globus ser o middleware de grade mais conhecido, e fornecer uma infra-estrutura de alto desempenho, optamos por não apresentar em mais detalhes uma grade baseada nele nesta seção. As razões principais foram a falta de espaço e a facilidade de se encontrar informações sobre o Globus e suas grades na Internet. Entretanto, sugerimos ao leitor interessado uma visita ao sítio da TeraGrid (`www.teragrid.org`) ou da Open Science Grid (`www.opensciencegrid.org`), as duas servem à pesquisa em aplicações científicas e usam o Globus.

2.4.1. Grid 5000

Grid 5000 é uma grade composta por nove aglomerados distribuídos geograficamente pela França e conectados através de uma rede dedicada e rápida. Cada aglomerado tem de 256 a 1000 processadores. A principal finalidade desta plataforma é servir de ambiente de testes para pesquisa em Ciência da Computação. Por ser um ambiente estanque, os experimentos são reprodutíveis, o que é uma grande vantagem para um ambiente de pesquisa. A escolha de fazer a grade isolada da Internet (apenas os portais para submissão de aplicações estão na rede mundial) permite também um ganho de segurança, pois todas as aplicações têm que passar pelos portais.

O ambiente é completamente controlado e existem medidas precisas das bandas passantes de comunicação entre os diversos aglomerados. Além disto,

o ambiente é heterogêneo, pois existem diferentes processadores, como AMD Opteron, Intel Itanium, Inte Xeon e PowerPC. Inclusive, existem três tipos diferentes de redes intra-aglomerados: Myrinet 2G, Myrinet 10G e Infiniband.

Para permitir ainda experimentos completamente abertos, é possível inicializar os computadores do Grid 5000 através de imagens de sistema operacional fornecidas pelos usuários. Logo, é possível configurar completamente também o software que será usado. O ambiente também fornece ferramentas de injeção automática de falhas e uma ferramenta geradora de tráfego em rede. Atualmente, existem cerca de 350 experimentos diferentes sendo executados no ambiente do Grid 5000. O nível de utilização da grade é geralmente superior a 70%. Os desafios atuais do Grid 5000 estão ligados a novos problemas provenientes da interconexão com duas outras grades: uma européia, a DAS 3 (`www.cs.vu.nl/das3`) na Holanda, com 1500 processadores e outra japonesa, a Naregi (`www.naregi.org`) com 100 teraflop/s de capacidade de computação (em 2007).

2.4.2. PlanetLab

O PlanetLab [Chun et al. 2003] é uma iniciativa digna de nota para a conexão de máquinas ao redor do globo. O seu principal objetivo é o desenvolvimento de serviços de rede. Desde 2003, mais de 1000 profissionais usaram o PlanetLab nas suas pesquisas em armazenamento distribuído, mapeamento de rede, redes par-a-par, tabelas de espalhamento distribuídas e processamento de consultas. A grade conta com 840 nós em 416 localidades (03/08).

As conexões entre as máquinas, localizadas na sua maior parte em instituições de pesquisa, é feita através da Internet acadêmica e pública. Todas as máquinas executam o mesmo pacote de software, baseado em Linux, que inclui: mecanismos para reiniciar máquinas e distribuir atualizações de software; ferramentas que monitoram o funcionamento dos nós; atividade de avaliação e controle de parâmetros do sistema; recursos para gerenciar contas de usuários e distribuição de chaves. Este software é chamado de MyPLC e pode ser usado para criar redes privadas semelhantes ao PlanetLab.

Os grupos integrantes da grade podem solicitar a reserva de partes da grade para efetuar experimentos em escala planetária. No momento, existem mais de 600 projetos de pesquisa sendo executados. Além desses experimentos de duração limitada, o PlanetLab também tem experimentos de longo prazo, como por exemplo serviços de monitoramento da rede com o CoMon. Recomendamos uma visita a `comon.cs.princeton.edu` para observar os recursos disponíveis.

Para se conectar a esta grade é necessário seguir uma série de instruções e o processo não é automático. Além disto, existem uma série de exigências de hardware como, por exemplo, no mínimo dois servidores seguindo as especificações fornecidas. No Brasil, os nós do PlanetLab são UFCG, UFMG, USP e RNP em vários estados.

2.4.3. EGEE

O projeto *Enabling Grids for E-science* (EGEE) (`www.eu-egee.org`) é composto por profissionais de mais de 240 instituições de 45 países. Na América Latina o projeto está presente através do EELA (`www.eu-eela.org`) onde o Brasil possui forte atuação. O EGEE foi o sucessor do projeto europeu DataGrid. Seus objetivos principais são construir uma infra-estrutura de grade segura, confiável e robusta e atrair um grande número de usuários científicos e da indústria, fornecendo treinamento e suporte. O projeto inicial terminou em 2006, mas a continuação foi garantida através do projeto EGEE-II.

O EGEE fornece uma infra-estrutura de grade para *e-Science* sempre disponível. O termo *e-Science* é usado para descrever aplicações científicas que precisam de muito poder computacional em ambientes potencialmente distribuídos, ou ainda aplicações científicas que processam uma quantidade de dados muito grande. A grade é composta por 41 mil processadores e espaço em disco de 5 PetaBytes. O sistema operacional usado nas máquinas de grade é o Linux. O middleware usado é o gLite (`glite.web.cern.ch/glite`).

Com tanto poder de processamento disponível, várias aplicações puderam ser executadas nesta enorme infra-estrutura, inicialmente nos campos da Física de Alta Energia e da Biomedicina. Atualmente, as aplicações são das mais variadas áreas como Astrofísica, Química Computacional, Ciências da Terra, Finanças, Fusão, Geofísica, etc. Para que uma estrutura tão grande funcione o projeto é organizado em 10 atividades, em três áreas principais: Rede, Serviços e Pesquisa. Uma descrição completa de cada uma das atividades pode ser encontrada no sítio do projeto.

2.5. A pesquisa atual em grades

Há ainda uma série de problemas em aberto no campo da Computação em Grade o que faz desta área uma excelente oportunidade para trabalhos de pesquisa de Iniciação Científica, Mestrado e Doutorado. Destacamos aqui alguns dos tópicos que merecerão atenção dos pesquisadores nos próximos anos.

Contabilidade e Economia de Grade trata de como controlar a utilização dos recursos computacionais pelos usuários da grade e, eventualmente, cobrar o pagamento pela utilização desses recursos possibilitando a criação de um mercado de compra e venda de poder computacional. Há uma série de abordagens para esse problema sendo pesquisadas atualmente [Buyya et al. 2005, de Andrade 2004].

Grades autônomas consistem em grades computacionais com mecanismos adaptativos da Computação Autônoma de forma a facilitar o seu gerenciamento e a qualidade do serviço oferecido [Parashar and Hariri 2004]. Grades autônomas devem oferecer mecanismos para auto-otimização, auto-proteção, configuração automática e tolerância a falhas automática. Cada um desses tópicos em separado ou todos eles em conjunto são um campo fértil para pesquisa.

Protocolos e algoritmos inteligentes para federação de aglomerados e escalonamento global é também uma das áreas com vários problemas em aberto. Em uma grade de grande porte não é possível que um único nó ou escalonador tenha uma visão total de todo o sistema. Por outro lado, se o gerenciamento dos recursos e o escalonamento de aplicações for baseado apenas em informações locais, pode haver um grande desperdício de recursos. O desafio desta área de pesquisa é obter uma forma inteligente de agrupar os aglomerados da grade em uma federação de forma e criar um conjunto de protocolos e algoritmos de forma a maximizar a utilização dos recursos e a qualidade de serviço oferecida aos usuários da grade.

O **Escalonamento em grades** é uma área onde muito ainda pode ser feito. Uma interessante linha de pesquisa está ligada à cooperação entre grades, onde um escalonador central poderia buscar soluções que fornecessem melhoras para todas as grades participantes. Nesse tipo de problema, é interessante notar que devem ser inseridas regras de conduta para evitar que eventuais participantes se aproveitem da situação apenas compartilhando aplicações e não recursos. As abordagens de resolução podem ser desde uma rede de favores [de Andrade 2004, Cirne et al. 2006] até de teoria de jogos [Rzadca et al. 2007].

O **melhor aproveitamento das novas arquiteturas de hardware** é uma área ainda pouco investigada. As novas arquiteturas multi-processadas, onde cada máquina possui um processador que internamente contém dois, quatro ou mais núcleos é uma área que pode ser explorada. Ainda nesse campo, com o enorme progresso das placas de vídeo, também se estuda como aproveitar o processador das placas gráficas (GPUs) em aplicações computacionalmente pesadas. A dificuldade nesse caso passa a ser o modelo de programação que é bem diferente dos processadores de uso geral.

Referências bibliográficas

[Albing 1993] Albing, C. (1993). Cray NQS: production batch for a distributed computing world. In *Proceedings of the 11th Sun User Group Conference and Exhibition*, pages 302–309, Brookline, MA, USA. Sun User Group, Inc.

[Alves et al. 2006] Alves, C., Cáceres, E., and Song, S. (2006). A BSP/CGM Algorithm for Finding All Maximal Contiguous Subsequences of a Sequence of Numbers. In *Euro-Par '06: Proceedings of the 12th International Euro-Par Conference on Parallel Processing*, pages 831–840. Springer-Verlag.

[Anderson et al. 1995] Anderson, T., Culler, D., Patterson, D., and the NOW Team (1995). A Case for Networks of Workstations: NOW. *IEEE Micro*, 15(1):54–64.

[Andrade et al. 2003] Andrade, N., Cirne, W., Brasileiro, F., and Roisenberg, P. (2003). OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing. In *9th International Workshop on Job Scheduling Strategies for Parallel Processing*. Springer.

[Barham et al. 2003] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the Art of Virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP'2003)*, pages 164–177. ACM Press.

[Bazewicz et al. 2000] Bazewicz, J., Trystram, D., Ecker, K., and Plateau, B. (2000). *Handbook on Parallel and Distributed Processing*. Springer-Verlag.

[Buyya et al. 2005] Buyya, R., Abramson, D., and Venugopal, S. (2005). The Grid Economy. *Proceedings of the IEEE*, 93(3):69–714.

[Capit et al. 2005] Capit, N., Costa, G. D., Georgiou, Y., Huard, G., n, C. M., Mounié, G., Neyron, P., and Richard, O. (2005). A batch scheduler with high level components. In *Cluster computing and Grid 2005 (CCGrid05)*.

[Chun et al. 2003] Chun, B. N., Culler, D. E., Roscoe, T., Bavier, A. C., Peterson, L. L., Wawrzoniak, M., and Bowman, M. (2003). Planetlab: an overlay testbed for broad-coverage services. *Computer Communication Review*, 33(3):3–12.

[Cirne et al. 2006] Cirne, W., Brasileiro, F., Andrade, N., Costa, L., Andrade, A., Novaes, R., and Mowbray, M. (2006). Labs of the world, unite!!! *Journal of Grid Computing*, 4(3):225–246.

[Cirne et al. 2003] Cirne, W., Brasileiro, F., Sauvé, J., Andrade, N., Paranhos, D., Santos-Neto, E., and Medeiros, R. (2003). Grid computing for bag of tasks applications. In *Proceedings of the Third IFIP Conference on E-Commerce, E-Business and E-Goverment*.

[de Andrade 2004] de Andrade, N. F. (2004). Reputação Autônoma como Incentivo à Colaboração no Compartilhamento de Recursos Computacionais. Master's thesis, Pós-Graduação em Informática, Universidade Federal de Campina Grande.

[de Camargo 2007] de Camargo, R. Y. (2007). *Armazenamento Distribuído de Dados e Checkpointing de Aplicações Paralelas em Grades Oportunistas*. PhD thesis, Departamento de Ciência da Computação do IME/USP.

[de Camargo et al. 2006a] de Camargo, R. Y., Cerqueira, R., and Kon, F. (2006a). Strategies for Checkpoint Storage on Opportunistic Grids. *IEEE Distributed Systems Online*, 7(9).

[de Camargo et al. 2006b] de Camargo, R. Y., Goldchleger, A., Carneiro, M. R. F., and Kon, F. (2006b). *Pattern Languages of Program Design 5 (PLoPd5)*, chapter The Grid Architectural Pattern: Leveraging Distributed Processing Capabilities, pages 337–356. Addison-Wesley.

[de Camargo and Kon 2007] de Camargo, R. Y. and Kon, F. (2007). Design and implementation of a middleware for data storage in opportunistic grids. In *CCGrid '07: Proceedings of the 7th IEEE/ACM International Symposium on Cluster Computing and the Grid*. IEEE Computer Society.

[de Camargo et al. 2005] de Camargo, R. Y., Kon, F., and Goldman, A. (2005). Portable checkpointing and communication for BSP applications on dynamic heterogeneous Grid environments. In *SBAC-PAD'05: The 17th International Symposium on Computer Architecture and High Performance Computing*.

[de Ribamar Braga Pinheiro Jr. 2008] de Ribamar Braga Pinheiro Jr., J. (2008). *Xenia: um sistema de segurança para grades computacionais baseado em cadeias de confiança*. PhD thesis, Departamento de Ciência da Computação do IME/USP.

[de Ribamar Braga Pinheiro Junior et al. 2005] de Ribamar Braga Pinheiro Junior, J., de Camargo, R. Y., Goldchleger, A., and Kon, F. (2005). InteGrade: a Tool for Executing Parallel Applications on a Grid for Opportunistic Computing. In *Salão de ferramentas do 23o Simpósio Brasileiro de Redes de Computadores*.

[de Sousa et al. 2006] de Sousa, B. B., da Silva e Silva, F. J., Teixeira, M. M., and Filho, G. C. (2006). Magcat: An agent-based metadata service for data grids. In *AgentGrid2006: 4th International Workshop on Agent Based Grid Computing, IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid'06)*, page 6. IEEE Computer Society.

[Dehne 1999] Dehne, F. (1999). Coarse grained parallel algorithms. *Special Issue of Algorithmica*, 24(3/4):173–176.

[Dehne et al. 1993] Dehne, F., Fabri, A., and Rau-Chaplin, A. (1993). Scalable parallel geometric algorithms for coarse grained multicomputers. In *Proc. ACM 9th Annual Computational Geometry*, pages 298–307.

[Dong and Akl 2006] Dong, F. and Akl, S. G. (2006). Scheduling algorithms for grid computing: State of the art and open problems. Technical Report 2006-504, Queen's University School of Computing.

[El-Rewini et al. 1994] El-Rewini, H., Lewis, T. G., and Ali, H. H. (1994). *Task scheduling in parallel and distributed systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Elnozahy et al. 2002] Elnozahy, M., Alvisi, L., Wang, Y.-M., and Johnson, D. B. (2002). A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408.

[Everitt et al. 2001] Everitt, B., Landau, S., and Leese, M. (2001). *Cluster Analysis*. A Hodder Arnold Publication, 4th edition edition.

[Finholt 2002] Finholt, T. (2002). Collaboratories. *Annual Review of Information Science and Technology (ARIST)*, 36:73–107.

[Foster and Kesselman 1997] Foster, I. and Kesselman, C. (1997). Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11(2):115–118.

[Foster and Kesselman 2003] Foster, I. and Kesselman, C., editors (2003). *The*

Grid 2: Blueprint for a New Computing Infrastructure. Morgan Kaufmann Publishers.

[Garcia and Buzato 1999] Garcia, I. C. and Buzato, L. E. (1999). Progressive construction of consistent global checkpoints. In *ICDCS '99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, page 55. IEEE Computer Society.

[Garfinkel et al. 2003] Garfinkel, S., Spafford, G., and Schwartz, A. (2003). *Practical UNIX and Internet Security*. O'Reilly.

[Goldchleger 2004] Goldchleger, A. (2004). InteGrade: Um Sistema de Middleware para Computação em Grade Oportunista. Master's thesis, Departamento de Ciência da Computação, IME/USP.

[Goldchleger et al. 2005] Goldchleger, A., Goldman, A., Hayashida, U., and Kon, F. (2005). The Implementation of the BSP Parallel Computing Model on the InteGrade Grid Middleware. In *Proceedings of the 3rd ACM/IFIP/USENIX Workshop on Middleware for Grid Computing*, pages 28–29.

[Goldchleger et al. 2004] Goldchleger, A., Kon, F., Goldman, A., Finger, M., and Bezerra, G. C. (2004). InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines. *Concurrency and Computation: Practice and Experience*, 16:449–59.

[Grimshaw et al. 1997] Grimshaw, A., Wulf, W., et al. (1997). The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1).

[Hamilton and Kougiouris 1993] Hamilton, G. and Kougiouris, P. (1993). The Spring Nucleus: A Microkernel for Objects. Technical report, Sun Microsystems, Inc.

[Hayashibara et al. 2002] Hayashibara, N., Cherif, A., and Katayama, T. (2002). Failure detectors for large-scale distributed systems. In *International IEEE Workshop on Self-Repairing and Self-Configurable Distributed Systems (RCDS?2002)*, pages 404–409.

[Hill et al. 1997] Hill, J. M., McColl, W. F., and Skillircon, D. B. (1997). Questions and answers about BSP. *Scientific Programming*, 6(3).

[Jackson et al. 2001] Jackson, D. B., Snell, Q., and Clement, M. J. (2001). Core Algorithms of the Maui Scheduler. In *JSSPP '01: Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*, pages 87–102, London, UK. Springer-Verlag.

[Johnson 2001] Johnson, G. (2001). All science is computer science. The New York Times.

[Krauter et al. 2002] Krauter, K., Buyya, R., and Maheswaran, M. (2002). A Taxonomy and Survey of Grid Resource Management Systems for Distributed Computing. *Software: Practice & Experience*, 32(2):135–164.

[Lang and Schreiner 2002] Lang, U. and Schreiner, R. (2002). *Developing Se-*

cure Distributed Systems With Corba. Artech House.

[Leighton 1991] Leighton, F. (1991). *Introduction to parallel algorithms and architectures: array, trees, hypercubes*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA.

[Litzkow et al. 1988] Litzkow, M., Livny, M., and Mutka, M. (1988). Condor: A Hunter of Idle Workstations. In *8th International Conference on Distributed Computing Systems*, pages 104–111, Washington, D.C., USA. IEEE Computer Society Press.

[Livny et al. 1997] Livny, M., Basney, J., Raman, R., and Tannenbaum, T. (1997). Mechanisms for High Throughput Computing. *SPEEDUP Journal*, 11(1).

[Maia et al. 2005] Maia, R., Cerqueira, R., and Kon, F. (2005). A Middleware for Experimentation on Dynamic Adaptation. In *ACM/IFIP/USENIX 3rd International Workshop on Adaptive and Reflective Middleware*.

[Massie et al. 2004] Massie, M. L., Chun, B. N., and Culler, D. E. (2004). The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(5-6):817–840.

[McColl 1995] McColl, W. (1995). Scalable computing. In van Leeuwen, J., editor, *Computer Science Today: Recent Trends and Developments, 1995*, volume 1000 of *LNCS*, pages 46–61. LNCS 1000, Springer-Verlag.

[Mitchell 1997] Mitchell, T. M. (1997). *Machine Learning*. Computer Science Series. McGraw-Hill.

[Netto and Rose 2003] Netto, M. A. S. and Rose, C. A. F. D. (2003). CRONO: A configurable and easy to maintain resource manager optimized for small and mid-size gnu/linux cluster. In *Proceedings of The 32nd International Conference on Parallel Processing (ICPP'03)*, pages 555–562, Kaohsiung, Taiwan. IEEE Computer Society.

[Nitzberg and Lo 1991] Nitzberg, B. and Lo, V. (1991). Distributed shared memory: a survey of issues and algorithms. *Computer*, 24(8):52–60.

[Ousterhout et al. 1988] Ousterhout, J. K., Cherenson, A. R., Douglis, F., Nelson, M. N., and Welch, B. B. (1988). The Sprite network operating system. *IEEE Computer*.

[Parashar and Hariri 2004] Parashar, M. and Hariri, S. (2004). Autonomic Grid Computing. In *Tutoral at the International Conference on Autonomic Computing (ICAC?04)*.

[Pinheiro Jr. and Kon 2005] Pinheiro Jr., J. B. and Kon, F. (2005). *Minicursos do 20o SBSeg*, chapter Segurança em Grades Computacionais, pages 65–111. SBC.

[Plank et al. 1995] Plank, J. S., amd G. Kingsley, M. B., and Li, K. (1995). Libckpt: Transparent checkpointing under UNIX. In *Proceedings of the USE-*

NIX Winter 1995 Technical Conference, pages 213–323.

[Ramachandran 2002] Ramachandran, J. (2002). *Designing security architecture solutions*. John Wiley New York.

[Rich Wolski 1999] Rich Wolski, Neil T. Spring, J. H. (1999). The network weather service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 15(5-6):757–768.

[Rzadca et al. 2007] Rzadca, K., Trystram, D., and Wierzbicki, A. (2007). Fair game-theoretic resource management in dedicated grids. In *CCGrid '07: Proceedings of the 7th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 343–350.

[Sarmenta 2002] Sarmenta, L. (2002). Sabotage-tolerance mechanisms for volunteer computing systems. *Future Generation Computer Systems*, 18(4):561–572.

[Tanenbaum 1995] Tanenbaum, A. S. (1995). *Distributed Operating Systems*. Prentice-Hall.

[Tanenbaun et al. 1990] Tanenbaun, A. S., van Renesse, R., van Staveren, H., Sharp, G. J., Mullendar, S. J., Jansen, J., and van Rossum, G. (1990). Experiences with the Amoeba Distributed Operating System. *Communications of the ACM*, 33(12):47–63.

[Terada 2000] Terada, R. (2000). *Segurança de Dados - Criptografia em Redes de Computadores*. Editora Edgard Blücher.

[Thain et al. 2005] Thain, D., Tannenbaum, T., and Livny, M. (2005). Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356.

[Valiant 1990] Valiant, L. (1990). A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111.

[van Steen et al. 1999] van Steen, M., Homburg, P., and Tanenbaum, A. S. (1999). Globe: A Wide-Area Distributed System. *IEEE Concurrency*, 7(1):70–78.

[Wang 1997] Wang, Y. M. (1997). Consistent global checkpoints that contain a given set of local checkpoints. *IEEE Transactions on Computers*, 46(4):456–468.

[Yoo et al. 2003] Yoo, A. B., Jette, M. A., and Grondona, M. (2003). SLURM: Simple Linux Utility for Resource Management. In *9th International Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60.

[Zhou 1992] Zhou, S. (1992). LSF: load sharing in large-scale heterogeneous distributed systems. In *Proceedings of the Workshop on Cluster Computing*, Tallahassee, FL. Supercomputing Computations Research Institute, Florida State University.