

Fundação Universidade Federal do Rio Grande

Engenharia de Computação

Projeto de Graduação

*XgridApp - Execution Environment for
Grid Applications*

Jean Paulo Sandri Orengo

Orientadores:

Prof. Dr. Cláudio Fernando Resin Geyer

Prof. Dr. Nelson Lopes Duarte Filho

Rio Grande

2004

“Idéias não são metais que se fundem mas
sim matérias que se congregam.”
Rui Barbosa

Este trabalho é dedicado ao meu filho Bruno,
à minha amada companheira Jutami
e às nossas famílias.

Agradecimentos

Ao longo deste trabalho obtive contribuições de várias pessoas. Citá-las uma a uma seria demasiadamente extenso e enfadonho, porém, os principais contribuintes devem ser lembrados.

Agradeço aos meus orientadores Nelson Lopes e Cláudio Geyer pelo apoio técnico/teórico, por proporem o tema e por guiarem-me ao longo do seu desenvolvimento. Agradeço a Luciano da Silva pelo apoio técnico na implementação do protótipo e pelo apoio teórico decisivo na concepção do modelo. Não posso esquecer também de Gerson Leiria que me ajudou a configurar o laboratório para realizar os testes finais, de Rodrigo Real que foi muito prestativo, ajudando-me tanto com detalhes teóricos quanto práticos e de Mauro Real que contribuiu com o estudo de caso.

Além do apoio técnico, não teria atingido os objetivos deste trabalho sem um apoio emocional. Tal apoio veio de minha companheira Jutami Cassol, da minha família e de meus amigos Marcelo Linder, Cássio Carvalho e Gustavo Sabin. Por fim, agradeço meu filho Bruno pela paciência que teve, esperando-me durante cinco anos para que eu pudesse adquirir o grau de engenheiro.

Sumário

Lista de Figuras	vii
Lista de Tabelas	ix
Lista de Abreviações	x
Resumo	xi
Abstract	xii
1 Introdução	1
1.1 Tema	1
1.2 Motivação	1
1.3 Objetivos	2
1.4 Contribuição do autor	2
1.5 Estrutura do texto	3
2 Conceituando <i>Grids</i>	5
2.1 Características e benefícios do uso de <i>Grids</i>	6
2.1.1 Explorar recursos subutilizados	6
2.1.2 Processamento Paralelo	6
2.1.3 Organizações Virtuais	7
2.1.4 Confiabilidade	8
2.1.5 Gerenciamento da infraestrutura de TI	8
2.2 Aspectos desfavoráveis no uso de <i>Grids</i>	9
2.3 Plataformas de execução	10
2.3.1 Multiprocessadores simétricos - SMP	10
2.3.2 Máquinas maciçamente paralelas - MPP	11
2.3.3 Máquinas com memória compartilhada distribuída - DSM	11
2.3.4 Redes de Estações de Trabalho - NOW	11

2.3.5	Máquinas Agregadas - COW	12
2.3.6	Grades Computacionais - <i>Grids</i>	13
2.4	Principais aspectos da Arquitetura de um <i>Grid</i>	13
2.4.1	Balanceamento de carga	14
2.4.2	Modelos de programação	15
2.5	Uma palavra sobre medidas de desempenho	16
2.5.1	<i>SpeedUp</i>	16
2.5.2	Eficiência	16
2.5.3	Lei de Hamdahl	16
2.6	Exemplos de <i>grids</i>	17
3	Exemplos de infraestruturas para <i>Grids</i>	18
3.1	Globus	18
3.1.1	Segurança e autenticação	18
3.1.2	Alocação e localização de recursos	19
3.1.3	Comunicação	20
3.1.4	Transferência de dados	21
3.1.5	Informação sobre os recursos	21
3.1.6	Estado atual	21
3.2	MyGrid	21
3.2.1	Arquitetura MyGrid	22
3.3	Condor	23
3.3.1	<i>Flock of Condors</i>	24
3.3.2	Condor-G	24
4	Modelo	25
4.1	Arquitetura	25
4.1.1	Camada de Programação	26
4.1.2	Camada de <i>RunTime</i>	28
4.1.3	Camada de Gerenciamento	32
5	XgridApp - Implementação	35
5.1	Linguagem de Programação	35
5.2	Características da Implementação	36

5.2.1	Camada de Programação	37
5.2.2	Camada de <i>Runtime</i>	38
5.2.3	Camada de Gerenciamento	39
5.3	Estudo de caso: Análise de seções retangulares de concreto	39
5.3.1	Execução em ambiente homogêneo	41
6	Considerações finais	49
6.1	Trabalhos Futuros	50
	Referências	51

Lista de Figuras

1.1	Arquitetura de <i>software</i> ISAM	3
2.1	A visão de um <i>grid</i> pelo usuário	5
2.2	Decomposição de uma tarefa em subtarefas	7
2.3	Duplicação de <i>jobs</i>	8
2.4	Gerenciamento de recursos de TI	9
2.5	Arquitetura de uma máquina SMP	10
2.6	Arquitetura de uma máquina MPP	11
2.7	Arquitetura de uma máquina NOW	12
2.8	Intragrid	13
2.9	Intergrid	14
3.1	Arquitetura Globus ToolKit	19
3.2	Exemplo de emprego de Globus-GRAM	20
3.3	Arquitetura MyGrid	22
3.4	Principais componentes de uma comunidade <i>Condor</i>	24
4.1	Camadas da Arquitetura XgridApp	26
4.2	Pseudocódigo de uma <i>task</i>	27
4.3	Hierarquia de tarefas	28
4.4	Diagrama de estados de uma <i>task</i>	29
4.5	Passos de um roubo	30
4.6	Diagrama de estados de uma <i>task eater</i>	32
4.7	Exemplo de algoritmo de histórico	34
4.8	Arquitetura XgridApp estendida	34
5.1	Código da classe Task	37
5.2	Código da classe TimeGoalCriteria	38
5.3	Pseudocódigo de uma <i>Task Eater</i>	39
5.4	Funcionamento da terceira camada	40
5.5	Variabilidade na resposta de estruturas	40

5.6	Fluxograma geral para análise probabilística de estruturas	41
5.7	Contribuição no total de trabalho - 2 máquinas	44
5.8	Contribuição no total de trabalho - 4 máquinas	45
5.9	Contribuição no total de trabalho - 8 máquinas	45
5.10	Contribuição com e sem <i>time goal</i> - 2 máquinas	45
5.11	Contribuição com e sem <i>time goal</i> - 4 máquinas	46
5.12	Contribuição com e sem <i>time goal</i> - 8 máquinas	46
5.13	Tarefas computadas - 2 máquinas	46
5.14	Tarefas computadas - 4 máquinas	47
5.15	Tarefas computadas - 8 máquinas	47
5.16	<i>SpeedUp</i> das execuções em máquinas homogêneas	47

Lista de Tabelas

3.1	Serviços Globus	18
5.1	Execução em 2 máquinas usando <i>Time Goal</i>	42
5.2	Execução em 2 máquinas sem usar <i>Time Goal</i>	42
5.3	Execução em 4 máquinas usando <i>Time Goal</i>	42
5.4	Execução em 4 máquinas sem usar <i>Time Goal</i>	43
5.5	Execução em 8 máquinas usando <i>Time Goal</i>	43
5.6	Execução em 8 máquinas sem usar <i>Time Goal</i>	43
5.7	SpeedUp usando <i>TimeGoal</i>	44
5.8	SpeedUp sem usar <i>TimeGoal</i>	44

Lista de Abreviações

ATM	<i>Asynchronous Transfer Mode</i>
COW	<i>Cluster of WorkStations</i>
CPU	Unidade Central de Processamento
DSM	<i>Distributed Shared Memory</i>
EXEHDA	<i>Execution Environment for High Distributed Applications</i>
FTP	<i>File Transfer Protocol</i>
GASS	<i>Global Access to Secondary Storage</i>
GRAM	<i>Globus Resource Allocation Manager</i>
GSI	<i>Globus Security Infrastructure</i>
HTTP	<i>Hypertext Transfer Protocol</i>
ISAM	<i>Infraestrutura de Suporte às Aplicações Móveis</i>
JVM	<i>Java Virtual Machine</i>
MDS	<i>Metacomputing Directory Service</i>
MIMD	<i>Multiple Instructions - Multiple Data</i>
MPP	<i>Massively Parallel Processors</i>
NOW	<i>Network of WorkStations</i>
PDA	<i>Personal Digital Assistant</i>
RAID	Agrupamento Redundante de Discos Independentes
RMI	<i>Remote Method Invocation</i>
RPC	<i>Remote Procedure Call</i>
SIMD	<i>Single Instructions - Multiple Data</i>
SMP	<i>Symmetric Multiprocessors</i>
TCP	<i>Transmission Control Protocol</i>
TI	Tecnologia da Informação
WQR	<i>Work Queue with Replication</i>
XgridApp	<i>Execution Environment for grid Applications</i>

Resumo

Este trabalho concentra seu estudo na definição de um *framework* que facilite a construção de aplicações em grade e gerencie sua execução de forma eficiente. Para a descrição da computação é utilizado o conceito de tarefas preguiçosas. A distribuição da carga é garantida usando um mecanismo do tipo *work-stealing*.

O *framework* está organizado em três camadas: programação, *runtime* e gerência de recursos. A camada de programação permite ao desenvolvedor definir seu problema usando conceitos abstratos como tarefa. A distribuição da computação entre os nodos é efetuada pela camada de *runtime*, que realiza o escalonamento segundo a abordagem *work-stealing*. Já a terceira camada gerencia os recursos do sistema.

Palavras-chave: *grid computing*, escalonamento, modelo de programação paralela

Abstract

This work focuses on a definition of a framework that facilitates the development of grid applications and manager efficiently its execution. For the computation description is used the concept of lazy tasks. The load distribution is warranted using a work-stealing like mechanism.

The framework is organized into tree layers: programming, runtime and resources manager. The programming layer allows the designer to define his problem using abstract concepts as task. The runtime layer is responsible for the load distribution based on an approach called work-stealing. Finally, the third layer manages the system resources.

Keywords: grid computing, scheduling, parallel programming model

1 *Introdução*

1.1 Tema

Este trabalho é dedicado ao estudo de novas abstrações para programação e escalonamento de aplicações voltadas para *grids* computacionais ¹. Para isso é proposto um modelo de ambiente de execução para aplicações em *grids* (XgridApp).

1.2 Motivação

Com o advento da internet, usuários de todo o mundo puderam desfrutar de uma infraestrutura de comunicação e compartilhamento de informações. Um passo maior nesta evolução nos leva a *Grid Computing*, possibilitando não só o compartilhamento de informações mas também de recursos de *hardware* e *software*.

A idéia de compartilhar recursos e adquiri-los a medida do necessário foi historicamente proposta em 1965 pelos projetistas do sistema operacional MULTICS [TAN 2003]. De forma semelhante a qual alguns autores definem *grids*, MULTICS foi projetado para que o acesso a recursos computacionais fosse feito como os serviços de água e eletricidade: o cliente obtém os recursos sob demanda e paga pelo que consome.

Apesar de MULTICS poder ser visto como ancestral dos *grids* computacionais, seu ancestral mais próximo é “metacomputing”, termo recente, da década de 90, que descreve projetos que visavam interligar centros de supercomputadores nos Estados Unidos. O termo *grid* foi de fato criado no *workshop* chamado “Building a Computational Grid” que ocorreu em 1997.

O termo *Grid Computing* é empregado para designar a utilização de uma máquina virtual composta de vários recursos heterogêneos distribuídos geograficamente. Virtual porque, para os usuários, é como se existisse apenas uma única máquina de grande poder e recursos; heterogênea pois é construída a partir de máquinas de diferentes arquiteturas e redes de interconexão; e distribuída geograficamente porque é constituída de elementos dispersos pelo globo.

Entre os atrativos de *grids* computacionais estão a alta capacidade de poder computacional disponível, o uso do processamento ocioso dos computadores, a agregação de novos recursos a medida em que se tornam necessários e a criação de organizações virtuais

¹Alguns autores empregam o termo *grades* computacionais na tradução de *grid computing* para a língua portuguesa.

voltadas ao compartilhamento de recursos.

Este novo paradigma traz à luz uma série de desafios a serem alcançados. Porém, são expressivos os benefícios que nos traz tanto no campo da ciência, permitindo acelerar o processamento de simulações e cálculos complexos, entre outros, como no campo dos negócios aumentando a produtividade e flexibilidade das empresas [IBM 04].

Para poder desfrutar dos benefícios desta nova tecnologia é indispensável a utilização de ferramentas que sejam flexíveis, atendendo às necessidades das mais variadas aplicações, e de fácil utilização por parte dos usuários e programadores, escondendo destes detalhes específicos sobre a máquina virtual.

Torna-se necessário portanto, o emprego de um *middleware*, disponibilizando ao programador um paradigma de programação simples e eficiente, e um ambiente de execução que gerencia os recursos.

1.3 Objetivos

O objetivo geral deste trabalho é propor um modelo de ambiente de execução para aplicações em *grids*. Destacam-se como objetivos específicos:

- caracterizar as frentes de pesquisa relativas à proposição do XgridApp, relacionando as mesmas com o estado da arte;
- diminuir o tempo necessário ao processamento de aplicações empregando algoritmos paralelos de distribuídos;
- facilitar a tarefa de desenvolver aplicações voltadas para *grids*, através de bibliotecas de programação;
- disponibilizar ferramentas de *software* capazes de gerenciar a execução de aplicações em *grids*;
- implementar e testar um protótipo do modelo proposto;
- fornecer subsídios para a elaboração de relatórios, artigos e trabalhos futuros, relacionados com o tema pesquisado.

1.4 Contribuição do autor

O presente trabalho está inserido no projeto ISAM [ISA 04], mais precisamente no sub projeto EXEHDA [EXE 04]. XgridApp emprega os serviços ISAM/EXEHDA em sua arquitetura, facilitando seu desenvolvimento e agregando funcionalidades ao protótipo criado. Em contrapartida, o modelo proposto será anexado ao ISAM, constituindo uma segunda opção frente ao modelo mestre-escravo já em operação.

O tema do projeto ISAM (Infraestrutura de Suporte às Aplicações Móveis Distribuídas) é a criação de uma infraestrutura de suporte necessária para a implementação de aplicações

móveis distribuídas com comportamento adaptativo em um ambiente da *Pervasive Computing*. A Computação Pervasiva (*Pervasive Computing*) integra as premissas da computação em grade e da computação móvel, contemplando aplicações com novas funcionalidades. Computação Pervasiva é a proposta de um novo paradigma computacional, que permite ao usuário o acesso ao seu ambiente computacional a partir de qualquer lugar. O usuário poderá utilizar equipamentos com diferentes perfis de hardware, os quais poderão ter suporte a operação móvel ou não.

O projeto de pesquisa ISAM propõe um *middleware* voltado para o gerenciamento de recursos em redes heterogêneas, com suporte a mobilidade de *software* e *hardware* e adaptação dinâmica. A estratégia consiste de um ambiente integrado: (i) que oferece um paradigma de programação direcionado aos objetivos e seu respectivo ambiente de execução; e (ii) que gerencia o processo de adaptação através de um modelo colaborativo multinível, no qual tanto o ambiente de execução como a aplicação, participam das decisões adaptativas.

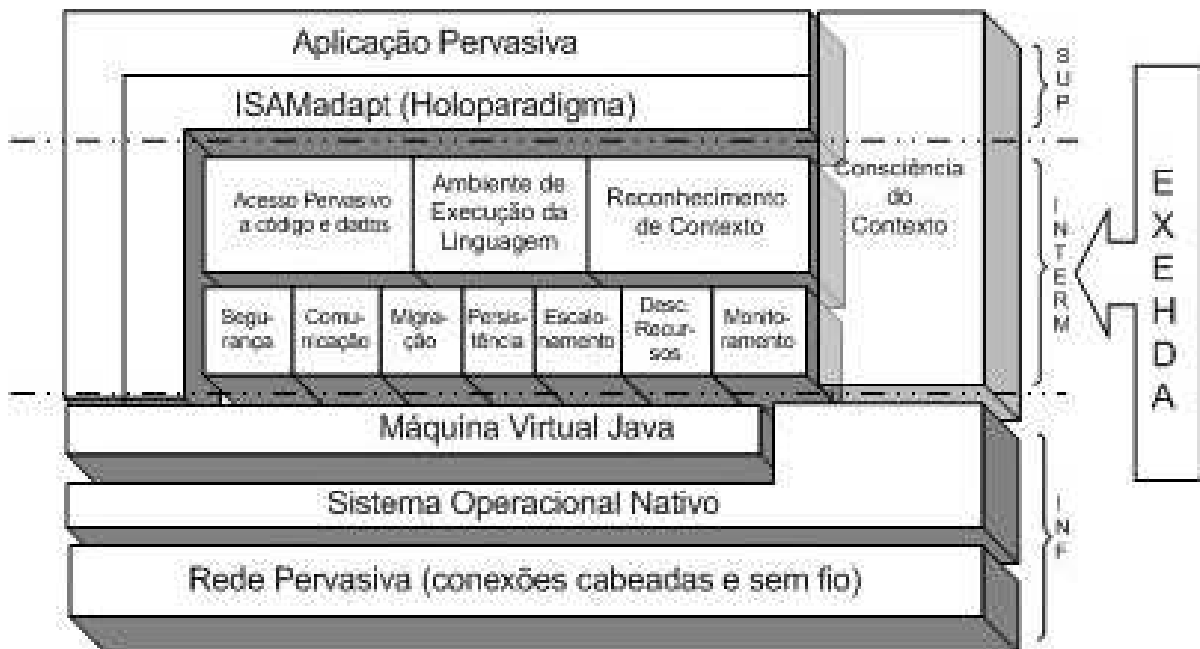


Figura 1.1: Arquitetura de *software* ISAM

A figura 1.1 ilustra as camadas do *middleware* ISAM. A camada superior (SUP) da arquitetura é composta pela aplicação móvel distribuída. Na segunda camada (INTERM) residem os serviços de suporte à execução das aplicações. É nesse nível que XgridApp será inserido. Já a camada inferior (INF) é composta pelas tecnologias empregadas nos sistemas distribuídos existentes, tais como sistemas operacionais nativos e a máquina virtual Java.

1.5 Estrutura do texto

O trabalho está dividido em seis capítulos. No capítulo introdutório (capítulo corrente), é apresentado os objetivos e a motivação para a realização deste trabalho. No

capítulo 2 é apresentado uma revisão sobre os conceitos que cercam a tecnologia *grid* assim como alguns exemplos atuais de sua utilização. O capítulo 3 se destina a descrever algumas infraestruturas voltadas para *grids* que serviram de inspiração para este trabalho. O modelo de ambiente de execução, as idéias principais e seus conceitos são apresentados no capítulo 4. No capítulo 5 é descrita a implementação de um protótipo, relacionando-o com o modelo e justificando as decisões de projeto. Neste capítulo são apresentados também um estudo de caso e os resultados obtidos. As considerações finais assim como propostas de trabalhos futuros são discutidos no capítulo 6.

2 *Conceituando Grids*

O termo *Grid Computing* surgiu em meados da década de 90, afim de denominar uma infraestrutura de computação distribuída para uso da ciência e engenharia avançadas [TUE 2001]. Mas afinal, o que são *Grids*?

Pode-se pensar em *grids* como sendo uma máquina virtual composta de vários recursos heterogêneos distribuídos geograficamente. Virtual uma vez que, para os usuários, é como se existisse apenas uma única máquina de grande poder e recursos; heterogênea pois é construída a partir de máquinas de diferentes arquiteturas e redes de interconexão; e distribuída geograficamente porque é constituída de elementos dispersos pelo globo (fig. 2.1).

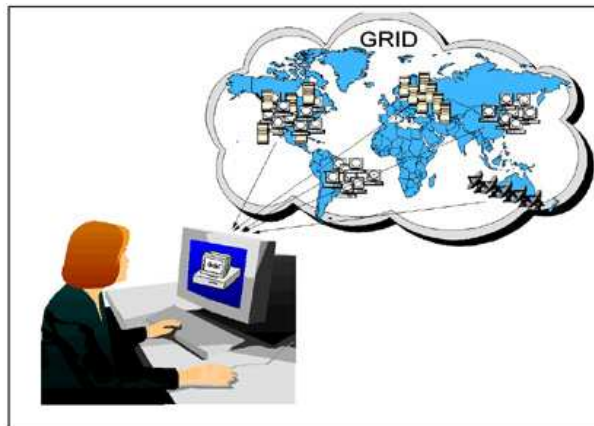


Figura 2.1: A visão de um *grid* pelo usuário

Outra maneira de ver um *grid* é imaginá-lo como uma rede elétrica: a rede elétrica disponibiliza aos seus clientes energia sob demanda escondendo detalhes sobre a origem da energia, a complexidade da malha de transmissão e distribuição. De forma semelhante, um *grid* computacional é uma rede na qual o usuário se conecta para obter recursos computacionais. Para o indivíduo, basta ligar o seu computador para usufruir de uma imensa quantidade de recursos, como se o seu computador fosse na verdade um gigantesco *mainframe* dotado de diversos periféricos.

Apesar desse novo campo da ciência ter nascido da comunidade de processamento de alto desempenho, e por isso sua caracterização como sendo uma infraestrutura para processamento distribuído, seu emprego no meio corporativo tem ganhado força [IBM 04]. Neste ambiente, pode-se dizer que um *grid* possibilita a criação de organizações virtuais com o objetivo de compartilhar recursos e realizar operações sob demanda.

O intuito dessa seção não é definir o termo *grid*, mas sim conceituá-lo. Existem diversos tipos distintos de *grids*, variando desde o número de recursos disponíveis até o seu propósito. Porém, de um modo geral, todos possuem as seguintes características [CIR 2003]:

- Heterogeneidade: composto de recursos heterogêneos, não somente computadores com arquiteturas distintas mas também demais periféricos e recursos como impressoras, telescópios, entre outros
- Alta dispersão geográfica: os recursos podem estar dispersos pelo globo
- Compartilhamento: não é dedicado a uma aplicação específica
- Múltiplos domínios administrativos: por ser formado por recursos de diversas organizações, possui uma variedade de domínios administrativos
- Controle distribuído: devido a sua natureza complexa, não é possível gerenciá-lo de forma centralizada

2.1 Características e benefícios do uso de *Grids*

Com o intuito de melhor conceituar *grids*, nesta seção explorar-se-ão seus benefícios assim como seus possíveis usos.

2.1.1 Explorar recursos subutilizados

A grande maioria das organizações possui um grande potencial de processamento ocioso. Calcula-se que máquinas *desktop* estejam ocupadas somente 5% do tempo [BER 2002] [LIV 2003], mainframes ficam ociosos 40% do seu tempo e servidores trabalham menos que 10% de um dia típico [IBM 04]. *Grids* computacionais oferecem uma infraestrutura para explorar esses recursos subutilizados, aumentando a eficiência do seu uso.

Uma maneira de tirar proveito desse potencial é escalonar tarefas, já em execução, migrando-as para nodos ¹ ociosos caso a máquina hospedeira esteja em um pico anormal de atividade. Assim, se ocorrerem picos de atividade na organização, recursos são alocados sob demanda. Por exemplo, um servidor pode ter uma aplicação sendo executada em *background* e, ao receber uma grande quantidade de pedidos de clientes, pode migrar esta aplicação para outra máquina, de forma transparente ao usuário.

2.1.2 Processamento Paralelo

Uma das características mais atrativas de *grids* é seu potencial para execução de aplicações paralelas. Aplicações escritas para serem executadas em máquinas paralelas são tipicamente decompostas em diversas tarefas que realizam parte do problema. Cada

¹Na terminologia adotada neste texto, nodo, nó e máquina são sinônimos.

uma dessas tarefas pode ser executada em um nodo do *grid* e o resultado final pode ser obtido em uma fração do tempo seqüencial.

A escalabilidade, tamanho e dispersão de um *grid* favorecem aplicações com baixa comunicação entre tarefas e grande processamento sobre os dados.

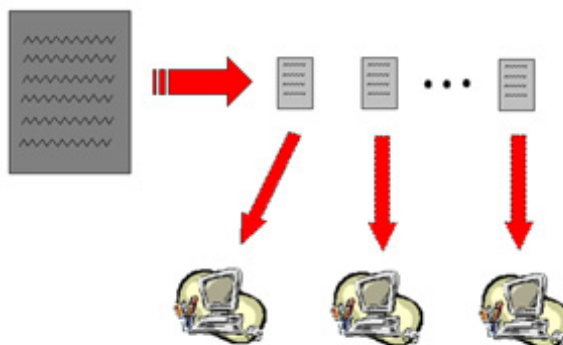


Figura 2.2: Decomposição de uma tarefa em subtarefas para serem executadas em diferentes máquinas

2.1.3 Organizações Virtuais

A função de um *grid* vai muito além do uso eficiente de recursos e o aumento do *throughput* de aplicações. Ele pode servir de base para a criação de organizações virtuais com a finalidade de compartilhar recursos entre seus participantes.

Desde o advento da internet, o compartilhamento de arquivos tem sido amplamente praticado por indivíduos e organizações. Com a tecnologia *grid* o termo compartilhar atinge um patamar maior, referindo-se não somente a arquivos mas também a equipamentos digitais como impressoras, DVD's, etc, a ciclos de CPU, espaços de armazenamento, e qualquer outro recurso passível de ser "virtualizado". Por exemplo, *Data Grids* podem expandir a capacidade de armazenamento de uma máquina, além de servirem como *backup* das informações vitais de uma instituição, bastando para isso, duplicar os dados.

Não há limite para o que se pode partilhar. Certas organizações possuem equipamentos especiais, *softwares* com licenças dispendiosas e serviços próprios. Pode-se imaginar um cenário onde apenas algumas máquinas tenham instalado determinados *softwares*. No momento em que uma aplicação rodando em outra máquina necessite deste componente, ela pode migrar para aquela que o possua. Essa hipótese não se restringe somente a componentes de *software*. Uma aplicação pode necessitar de um telescópio e, da mesma maneira, ela pode migrar através do *grid* para ser alojada no equipamento correto. Um motor de busca pode ser dividido sobre o *grid* sendo alocado em nodos que possuam conexões distintas com a internet, ampliando significativamente a largura de banda efetivamente empregada.

Tão importante quanto compartilhar é fazê-lo de forma transparente ao usuário, respeitando as regras e políticas de acesso de cada instituição. Este é o grande objetivo dessa fabulosa tecnologia, mas também é um dos seus maiores desafios.

2.1.4 Confiabilidade

Atualmente sistemas confiáveis são construídos sobre *hardware* redundante. Tecnologias RAID [STA 2003] são empregadas para atingir armazenamento de dados de forma confiável, processadores duplicados garantem substituição em caso de falha sem ter que desligar o equipamento, sistemas de refrigeração e abastecimento de energia são reforçados por mais de uma fonte, *nobreaks* garantem o funcionamento mesmo em falhas na distribuição de energia. Todos esses equipamentos constroem sistemas confiáveis, contudo, devido ao uso de componentes duplicados e circuitos lógicos mais sofisticados, essa organização tem alto custo de aquisição.

Sistemas confiáveis podem ser construídos baseados em tecnologias de *software* ao invés de *hardware*, com um custo relativamente baixo [BER 2002]. Os nodos de um *grid* não são fortemente acoplados, e numa eventual falha, outros podem dar continuidade às operações sem o conhecimento do usuário, bastando para isso que o gerenciador do sistema re-submeta as tarefas. Em sistemas de tempo real, múltiplas cópias de importantes tarefas são postas em execução sobre diversas máquinas. Os resultados de ambas podem ser testados em busca de qualquer tipo de inconsistência, como interrupção do funcionamento da máquina, dados corrompidos ou falsificados. A figura 2.3 demonstra esta possibilidade.

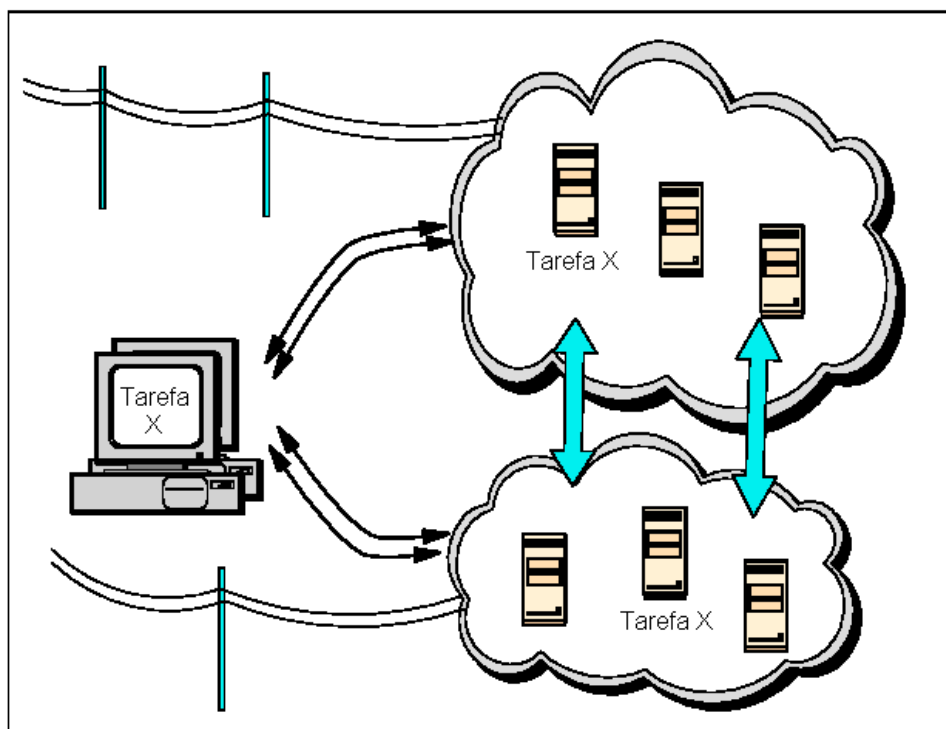


Figura 2.3: A figura ilustra a duplicação da tarefa X em diversas máquinas para garantir confiabilidade de sua execução e de seus resultados

2.1.5 Gerenciamento da infraestrutura de TI

Uma consequência da “virtualização” dos recursos é a facilidade que os gerentes possuem para gerir a infraestrutura de TI das organizações. Como mostra a figura 2.4, fica

mais fácil visualizar a capacidade e utilização dos recursos computacionais.

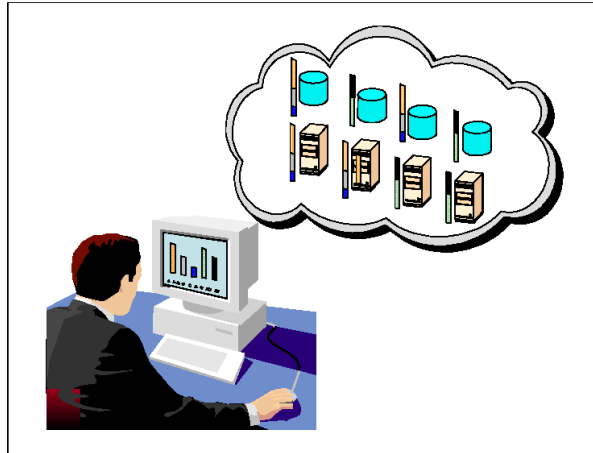


Figura 2.4: O gerente vê os recursos de TI e sua utilização, podendo modificar as políticas de utilização dos mesmos

Da maneira como hoje é organizado, cada projeto ou departamento de uma instituição é responsável pelos seus recursos computacionais. Esse arranjo pode ser oneroso e pouco eficiente devido ao sub aproveitamento do *hardware* disponível. Unidades podem compartilhar sua infraestrutura de TI, tornando desnecessário o investimento em um número maior de recursos. Gerentes têm em suas mãos meios de modificar a política de utilização dos recursos visando o melhor aproveitamento dos mesmos.

A troca de equipamentos (*upgrade*) da instituição também se beneficia dessa tecnologia. Projetos não precisam parar por causa da manutenção de suas máquinas visto que suas tarefas são alocadas nas demais máquinas.

2.2 Aspectos desfavoráveis no uso de *Grids*

Existem muitos fatores que devem ser levados em conta antes de implantar um *grid*.

Nem todas as aplicações são paralelizáveis, ou passíveis de serem realocadas dinamicamente sem perda de consistência. Ferramentas para transformar algoritmos sequenciais em paralelos ainda estão num estado prematuro. Entretanto, já estão disponíveis diversas ferramentas para ajudar no desenvolvimento de aplicações paralelas para *grids*. A tarefa de ajustar programas para explorar os atributos de um *grid* é, de modo geral, árdua e freqüentemente requer ótimos programadores.

Adicionalmente, a configuração de um *grid* pode afetar a performance, confiabilidade e segurança da organização [BER 2002].

2.3 Plataformas de execução

Uma aplicação paralela é composta por várias tarefas que são executadas em diversos processadores. Os processadores (e suas arquiteturas) usados por um programa constituem a plataforma de execução da aplicação.

Plataformas de execução de aplicações distinguem-se por uma série de aspectos, sendo os mais relevantes: conectividade, heterogeneidade, compartilhamento e escala. Os canais de comunicação que interligam os processadores compõem a conectividade da plataforma. A heterogeneidade trata das diferentes arquiteturas e velocidades dos processadores usados. Já o compartilhamento discorre sobre a possibilidade dos recursos serem compartilhados por mais de uma aplicação ao mesmo tempo. Escala estabelece quantos processadores a arquitetura poderá ter.

É fundamental conhecer as diferentes plataformas, uma vez que cada programa paralelo possui requisitos que serão melhor atendidos por uma determinada plataforma. As principais classes de plataformas são: SMPs, MPPs, DMPs, NOWs, COWs e *Grids*. Apesar de *grids* constituírem uma classe a parte, eles podem agregar qualquer outra classe em sua formação. As seções seguintes explorarão cada uma delas.

2.3.1 Multiprocessadores simétricos - SMP

SMPs (*Symmetric Multiprocessors*) são máquinas constituídas de vários processadores que compartilham a mesma memória e o mesmo barramento (fig. 2.5). O fortíssimo acoplamento dos processadores garante uma excelente conectividade. Todavia, por compartilharem o mesmo barramento, não tem grande escalabilidade, uma vez que a disputa por acesso a memória degrada o desempenho a medida em que mais processadores são adicionados.

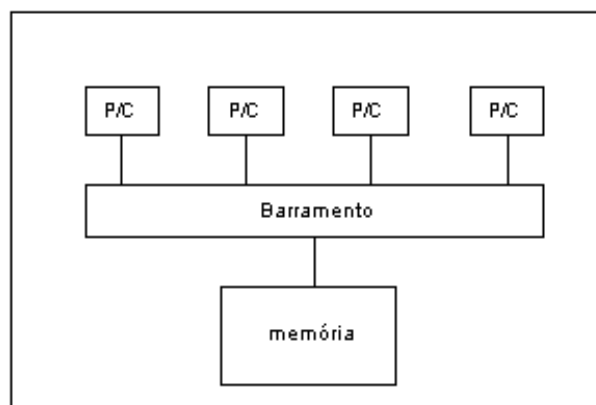


Figura 2.5: Arquitetura de uma máquina SMP

Pode-se citar como exemplos, o IBM R50, SGI Power Challenge, SUN Ultra Enterprise 10000, HP/ Convex e DEC Alpha Server 8400.

2.3.2 Máquinas maciçamente paralelas - MPP

Máquinas maciçamente paralelas, ou MPPs (*Massively Parallel Processors*), são compostas por milhares de pares processador/memória, cada qual com seu endereçamento de memória próprio, conectados por redes dedicadas de baixa latência.

O seu uso é controlado por um escalonador que gerencia partições. Cada requisição é alocada a uma determinada partição, não ocorrendo, portanto, compartilhamento de recursos entre as aplicações, como ilustra a figura 2.6.

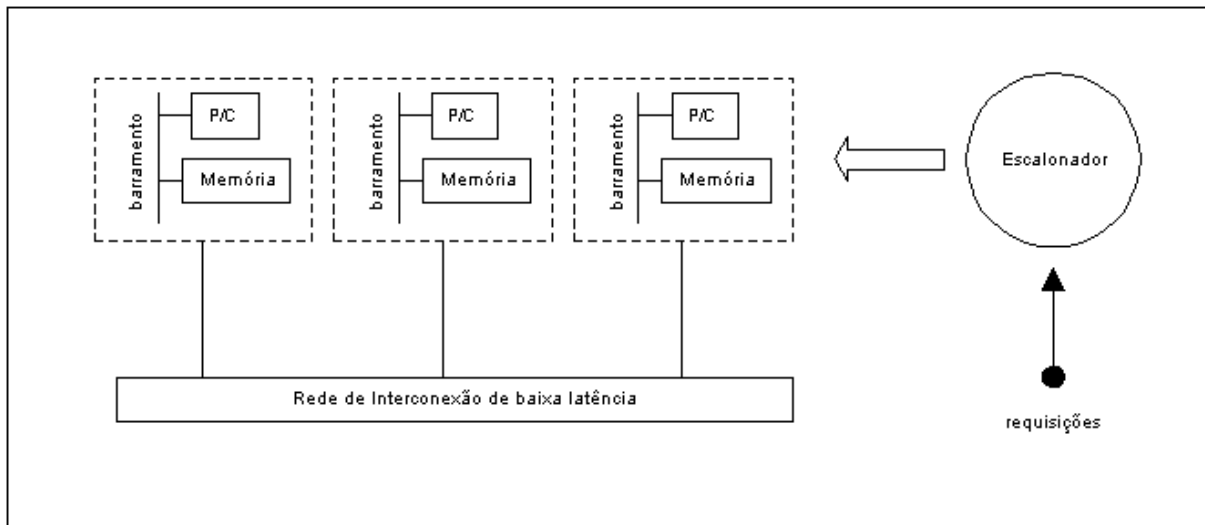


Figura 2.6: Arquitetura de uma máquina MPP

São exemplos de MPPs o Intel Paragon, Connection Machine Cm-5 e IBM SP2.

2.3.3 Máquinas com memória compartilhada distribuída - DSM

A principal diferença entre MPPs e DSMs (*Distributed Shared Memory*), é que nestes, apesar da memória ser distribuída, existe somente um espaço de endereçamento. Técnicas de coerência de *cache* devem ser providas por *hardware* ou *software* [NAV 2004].

Cray T3D, estações de trabalho rodando TreadMarks e Stanford DASH são exemplos de DSM.

2.3.4 Redes de Estações de Trabalho - NOW

NOWs (*Network of WorkStations*) são sistemas constituídos de várias estações de trabalho completas ligadas por tecnologias de redes tradicionais como Ethernet e ATM. Na verdade, uma rede local serve de plataforma para execuções paralelas.

Essa arquitetura tem como atrativo seu baixo custo, entretanto, se caracteriza também por baixo desempenho, associado principalmente ao uso de interconexões não otimizadas para aplicações paralelas.

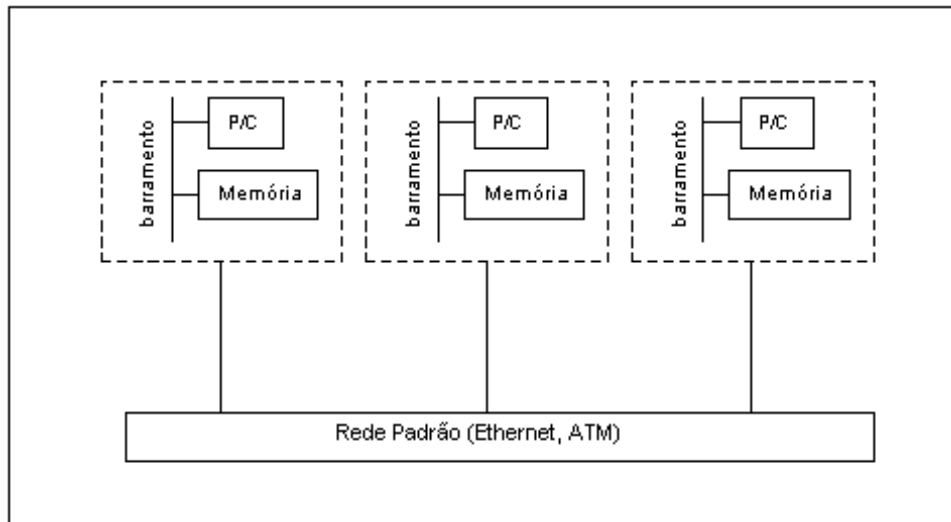


Figura 2.7: Arquitetura de uma máquina NOW

Como exemplo do uso de NOWs pode-se tomar os estudos de caso descritos na seção 5.3.1.

2.3.5 Máquinas Agregadas - COW

Como evolução natural das NOWs, com a finalidade de explorar totalmente seu potencial para execução de aplicações paralelas, surge as COWs (*Cluster of WorkStations*). Da mesma forma que em NOWs, máquinas agregadas são construídas com estações de trabalho, mas são projetadas visando a execução de programas paralelos. De fato, as estações não possuem teclado, mouse ou monitor, o sistema operacional é otimizado, tendo vários serviços desabilitados. Até mesmo as camadas de rede podem ser simplificadas uma vez que as necessidades de comunicação de aplicações paralelas diferem das redes locais.

A primeira vista, MPPs e COWs parecem versar sobre as mesmas arquiteturas, porém, alguns atributos às diferem de forma sutil:

- As redes que conectam MPPs são tipicamente mais rápidas que as redes de COWs
- O adaptador de rede de um COW é fracamente acoplado ao processador, pois é ligado ao barramento externo da CPU em um nível hierárquico menor. Já em MPPs, o adaptador de rede é ligado no mesmo barramento, tendo um ganho significativo de performance
- Um sistema operacional completo, porém adaptado, é instalado em cada nó de um *cluster* enquanto em nós MPP, reside apenas o núcleo do sistema operacional

São exemplos de COW o iCluster do HP Labs de Grenoble com rede FastEthernet, o agregado Amazônia do CPAD-PUCRS com rede Myrinet e a máquina didática “Branca-de-neve e os sete anões” instalada na FURG, com rede padrão Ethernet.

2.3.6 Grades Computacionais - *Grids*

Por fim, no topo da evolução de plataformas NOWs e COWs estão os *grids* computacionais. *Grids* são mais heterogêneos e têm maior distribuição que os demais, são constituídos de processadores e também equipamentos digitais como microscópio, robôs, entre outros. Além disso, os recursos podem estar sob tutela de diferentes entidades, possuindo políticas de acesso e permissão ortogonais.

Existem *grids* de todos os tamanhos, desde poucas máquinas de um departamento até inúmeros equipamentos de organizações espalhadas pelo globo.

A figura 2.8 mostra um *grid* formado por máquinas e recursos adicionais de um departamento, conectados por uma rede local. A gerência desta configuração é mais simples, pois não necessita de políticas de uso e segurança muito sofisticados. Um escalonador é indicado para criar prioridades no uso de certos recursos. Essa organização é conhecida como *IntraGrid*.

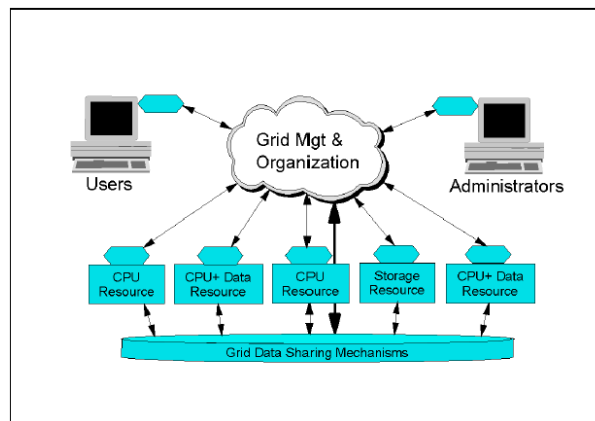


Figura 2.8: Um *grid* construído em um pequeno departamento

Seguindo a progressão, ao ser expandido por diversos departamentos em diversas cidades, o *grid* passa a necessitar de políticas de uso mais elaboradas, ditando quais recursos são acessíveis a quem e em quais horários. Restrições de segurança também são impostas sobre dados e recursos sensíveis. Um *grid* como este recebe o status de *InterGrid* (ver fig 2.9).

2.4 Principais aspectos da Arquitetura de um *Grid*

Na concepção da infraestrutura de um *grid* muitas questões devem ser resolvidas. Questões como modelo de programação e balanceamento de carga são comuns em todas as plataformas, todavia, *grids* possuem necessidades adicionais como tolerância à falhas, segurança e fatores econômicos.

Em um ambiente tão disperso e heterogêneo, falhas ocorrem com frequência: um nodo pode ser desligado por falha na fonte de energia, dados podem ser corrompidos ou perdidos durante as comunicações, uma tarefa pode ser perdida em seu movimento migratório,

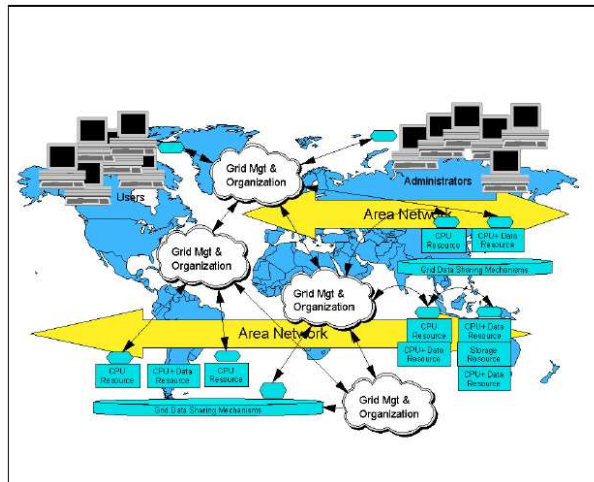


Figura 2.9: Um *grid* formado por recursos de diversos departamentos espalhados pelo mundo

enfim, existem muitas possibilidades catastróficas. Uma infraestrutura completa deve prover mecanismos para detectar e tratar todos estes tipos de falha de forma transparente ao usuário.

A autenticação e as permissões de acesso tem que ser consideradas devido à existência de múltiplos domínios administrativos. Garantir um mecanismo uniforme de acesso e permissão a cada recurso exige técnicas e metodologias novas.

Para pôr em prática um *grid* fatores econômicos devem ser levados em conta. Por exemplo, a criação de um *grid* entre duas ou mais instituições exige regras de compartilhamento que equilibrem a contribuição de cada uma e restitua financeiramente aquela que for prejudicada. Uma proposta para solucionar este problema é criar economias *grid*. A idéia central é criar uma moeda virtual que é usada para comprar e vender o uso de recursos [CIR 2003].

O foco deste trabalho reside em desenvolver um escalonador e um modelo de programação para aplicações em grade. Para isso, os aspectos descritos acima não serão considerados.

2.4.1 Balanceamento de carga

Em um *grid*, alguém tem que escolher quais recursos serão utilizados por uma aplicação e quais das suas tarefas serão alocadas em cada recurso. Esse papel está sob responsabilidade do escalonador.

Cada computador tem um escalonador próprio, o sistema operacional, que recebe solicitações de vários processos e, com base em seu amplo conhecimento, decide qual pedido será atendido. Esse escalonador é chamado de escalonador de recurso.

Não é possível construir um escalonador de recurso para um *grid* que consiga monitorar satisfatoriamente recursos tão heterogêneos e de tamanha escala. Neste cenário, os escalonadores de aplicação é que decidem quais recursos uma tarefa irá usufruir. Escalon-

adores de aplicação não controlam os recursos que usam, eles obtêm acesso a tais recursos via solicitações aos escalonadores de recursos.

Para tomar as melhores decisões, escalonadores de aplicações devem conhecer detalhes das aplicações que escalonam e dados sobre o estado dos recursos. Por isso, eles são projetados visando uma aplicação ou classe de aplicações. É importante frisar que um escalonador bom é aquele que atinge um equilíbrio entre o uso dos recursos, mantendo-os sempre trabalhando. Se o escalonador for ruim, alguns recursos ficarão sobrecarregados enquanto outros ficarão ociosos.

2.4.2 Modelos de programação

As aplicações que pretendem tirar proveito de um *grid* devem ser escritas tendo em mente questões sobre sincronização, falhas, transferência de dados, decomposição em subtarefas, etc. É imprescindível que novas abstrações sejam criadas para facilitar o desenvolvimento dessas aplicações.

Em [CAV 2004] é apresentada uma distinção entre os modelos de programação baseado no elemento delimitador do paralelismo:

Paralelismo de tarefa Neste modelo o programador cria uma série de tarefas que irão operar sobre conjuntos distintos de dados. O programador foca seus esforços na identificação das atividades concorrentes e sua distribuição. Pode-se fazer uma analogia entre este modelo e a classe MIMD (múltiplas instruções - múltiplos dados) da classificação de Flynn².

Paralelismo de dados Este modelo é caracterizado pela execução paralela de uma mesma atividade sobre diferentes partes de uma mesma coleção de dados. O programador constrói uma única tarefa que passa a executar sobre dados distintos. Nota-se a semelhança com a classe SIMD (único fluxo de instrução, múltiplos dados).

Uma vez escolhido o modelo com base no seu tipo de paralelismo, deve-se decidir sobre qual paradigma de programação utilizar para explicitar sincronizações e transferência de dados. Na literatura os paradigmas mais citados são: troca de mensagens, chamadas remotas a métodos(RMI) e a procedimentos(RPC).

No paradigma de troca de mensagens, a sincronização e transferência de dados são dadas pelo envio e recebimento de mensagens. Exemplos desse paradigma são as bibliotecas MPI e PVM. Já em RMI e RPC, existem respectivamente, métodos e procedimentos que podem ser invocados remotamente. Isso dá ao programador um nível de abstração maior, sendo mais indicado que a troca de mensagens. A linguagem de programação Java [SIK 2003] possui mecanismos para criação de métodos remotos [SES 04].

Cabe aqui colocar uma última consideração sobre tipos de aplicações. As aplicações podem ser classificadas quanto ao nível de granulosidade. O atributo granulosidade versa sobre a razão entre o tempo necessário ao cálculo de uma operação e os custos envolvidos

²Esta classificação é freqüentemente associada à descrição de arquiteturas de computadores, contudo, também se aplica a modelos de programação.

nas trocas de dados entre as tarefas que compõem a aplicação. Um programa que envolve grandes quantidades de operações sobre os dados e pouca comunicação entre as tarefas é classificado como uma aplicação de granulosidade grossa. No lado oposto, um programa que necessite de grandes volumes de trocas de dados para realizar pequenos cálculos é dito de granulosidade fina. As aplicações de granulosidade média e grossa são as que melhor exploram os benefícios de um *grid*.

2.5 Uma palavra sobre medidas de desempenho

Ao se desenvolver aplicações para rodarem em *grids* espera-se ter uma forma de medir a melhora de desempenho. Essa medida serve também para verificar a qualidade de uma infraestrutura *grid*.

2.5.1 *SpeedUp*

O *SpeedUp*, que é calculado a partir da expressão 2.1, quantifica a melhora de desempenho de uma execução paralela em relação a uma execução seqüencial. Na expressão, S_n representa o valor de *speedup* para uma máquina de n processadores, T_1 representa o tempo de execução com 1 processador e T_n o tempo de execução para n processadores.

$$S_n = \frac{T_1}{T_n} \quad (2.1)$$

2.5.2 Eficiência

A eficiência define o nível médio de utilização dos processadores em um sistema paralelo. A eficiência pode ser mensurada pela equação 2.2.

$$E_n = \frac{S_n}{n} \quad (2.2)$$

2.5.3 Lei de Hamdahl

Anunciada por Gene Hamdahl, esta lei define o *speedup* máximo que pode ser obtido por n processadores. Sendo σ a proporção do processamento obrigatoriamente seqüencial e, $\sigma - 1$ a proporção que pode ser paralelizada, o *speedup* máximo que pode ser atingido empregando n processadores é [REA 2002]:

$$S_n = \lim_{n \rightarrow \infty} \frac{1}{\sigma + \frac{1-\sigma}{n}} = \frac{1}{\sigma} \quad (2.3)$$

2.6 Exemplos de *grids*

Apesar dessa área da ciência da computação ter pouco mais de dez anos, é possível citar alguns exemplos de emprego da mesma tanto em ambientes científicos e acadêmicos, como em ambientes corporativos:

- TeraGrid - TeraGrid almeja ser a maior e mais rápida infraestrutura distribuída para pesquisas científicas de todo o mundo. Seus recursos somam 20 teraflops de poder computacional espalhados por diversos centros de pesquisa em todo o mundo [TER 04].
- CERN - A organização europeia de pesquisa nuclear [CER 04] usará a tecnologia *grid* para processar os dados gerados pelo seu acelerador de partículas. É esperado que os recursos de milhares de instituições ao redor do mundo, ligados por *switches* 10Gigabit Ethernet, consigam processar os mais de 10 milhões de Gigabytes de dados gerados por ano.
- Seti@Home - Seti (*Search for Extraterrestrial Intelligence*) é um esforço científico para determinar se existe vida extraterrestre. Os dados coletados pelos seus imensos radares são analisados por mais de cinco milhões de computadores. Para colaborar com o projeto, os interessados só precisam obter uma cópia da proteção de tela seti@home. Quando o computador não estiver sendo usado, ele recebe via internet uma porção dos dados, analisa e envia os resultados para um servidor da instituição [SET 04].
- C.O.R.E. Digital Picture - C.O.R.E. é uma grande empresa de animação gráfica. Seu *grid* é formado por 70 máquinas Linux com processadores duais e 21 Silicon Graphics. Um dos processadores das máquinas Linux é de uso exclusivo de seu dono e o outro fica disponível para o *grid*. Para aumentar a produtividade, as tarefas de cada funcionário são distribuídas entre os processadores duais e também nos demais processadores caso estejam ociosos [LIV 2003].
- IBM Boeblingen Lab - IBM conta com um *grid* formado por três *clusters*, distribuídos em diferentes departamentos, usado para realizar simulações dos protótipos de processadores zSeries [IBM 04].

3 Exemplos de infraestruturas para Grids

Para tornar a tecnologia *grid* viável, surgiram diversos esforços tanto acadêmicos (Condor, MyGrid, Globus) quanto comerciais (Entropia, distributed.net). Esse capítulo destina-se a descrever os principais aspectos de alguns desses sistemas, que de alguma forma contribuíram para a concepção desse trabalho.

3.1 Globus

O objetivo do projeto Globus é criar uma infraestrutura para servir de base na construção de *metacomputers*¹. Para tanto, foi desenvolvido o Globus ToolKit, um conjunto de primitivas (módulos) de baixo nível para serem utilizados na geração de serviços de alto nível, com o intuito de facilitar a computação em *grid* [KES 96]. Essa percepção é ilustrada pela figura 3.1.

Globus ToolKit é formado por um conjunto de módulos. Cada módulo tem um propósito bem delineado e define uma interface. A tabela 3.1 sumariza os principais módulos.

Serviço	Funcionalidade
GSI	Segurança e autenticação
GRAM	Submissão e controle de tarefas
Nexus	Comunicação entre tarefas
MDS	Informações sobre os recursos
GASS	Transferência de arquivos
GridFTP	Transferência de arquivos

Tabela 3.1: Serviços Globus

3.1.1 Segurança e autenticação

Em um ambiente formado por diversos domínios administrativos, a tarefa de autenticar um usuário e estabelecer as operações que têm acesso, não é trivial. O módulo GSI

¹*Metacomputer* é um computador virtual construído dinamicamente com recursos heterogêneos distribuídos geograficamente, ou seja, é um sinônimo de *grid*

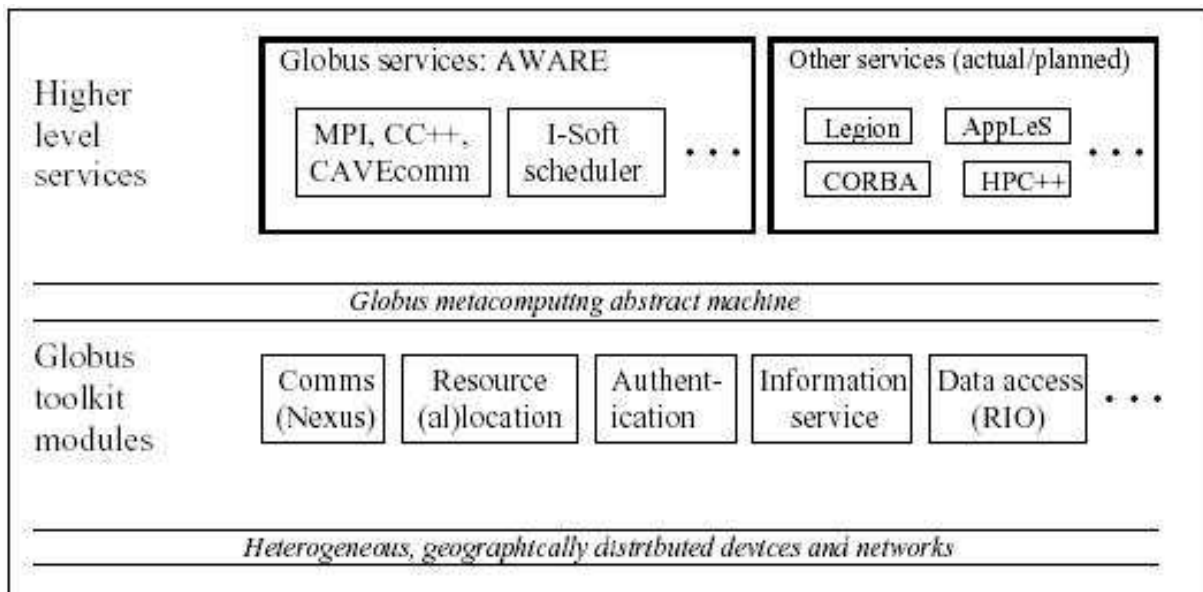


Figura 3.1: Arquitetura Globus ToolKit, os módulos servem para a criação de serviços de alto nível como bibliotecas de programação, escalonadores, etc

(*Globus Security Infrastructure*) trata deste problema.

Ao invés do usuário se identificar em cada domínio administrativo, o serviço GSI permite que somente uma autenticação seja necessária e válida para todo o *grid*. Uma vez reconhecido pelo GSI, o usuário tem livre acesso aos demais serviços Globus.

As restrições sobre as operações que o usuário pode ou não realizar são de competência do administrador local. Isso é alcançado mapeando a identidade Globus para um usuário local.

3.1.2 Alocação e localização de recursos

Como apresentado na seção 2.4.1, *grids* não possuem escalonadores de recurso e sim escalonadores de aplicação. O usuário utiliza um escalonador de aplicação que escolhe os recursos e submete as tarefas para os escalonadores de recursos.

O universo de diferentes escalonadores de recursos gerou a necessidade de tratá-los de forma uniforme, através de primitivas independentes do escalonador. Os serviços da interface GRAM (*Globus Resource Allocation Manager*) permitem submissões à escalonadores de recursos descrevendo-as em uma linguagem universal.

GRAM também provê mecanismos para expressar os requerimentos de recursos, para identificá-los e escaloná-los após sua descoberta. Possui ainda serviços para monitorar o desempenho e controlar os recursos individualmente. A figura 3.2 apresenta um exemplo do emprego de GRAM. No exemplo pretende-se realizar uma simulação interativa em 100.000 entidades. Inicialmente, o escalonador de aplicação especificamente desenvolvido para a simulação, repassa os requisitos para o GRAM que refina o pedido especificando-o em termos de ciclos de CPU, memória e latência de comunicação. Depois disso, é local-

izado os melhores recursos que atendam aos requisitos, e finalmente é feita a submissão das tarefas.

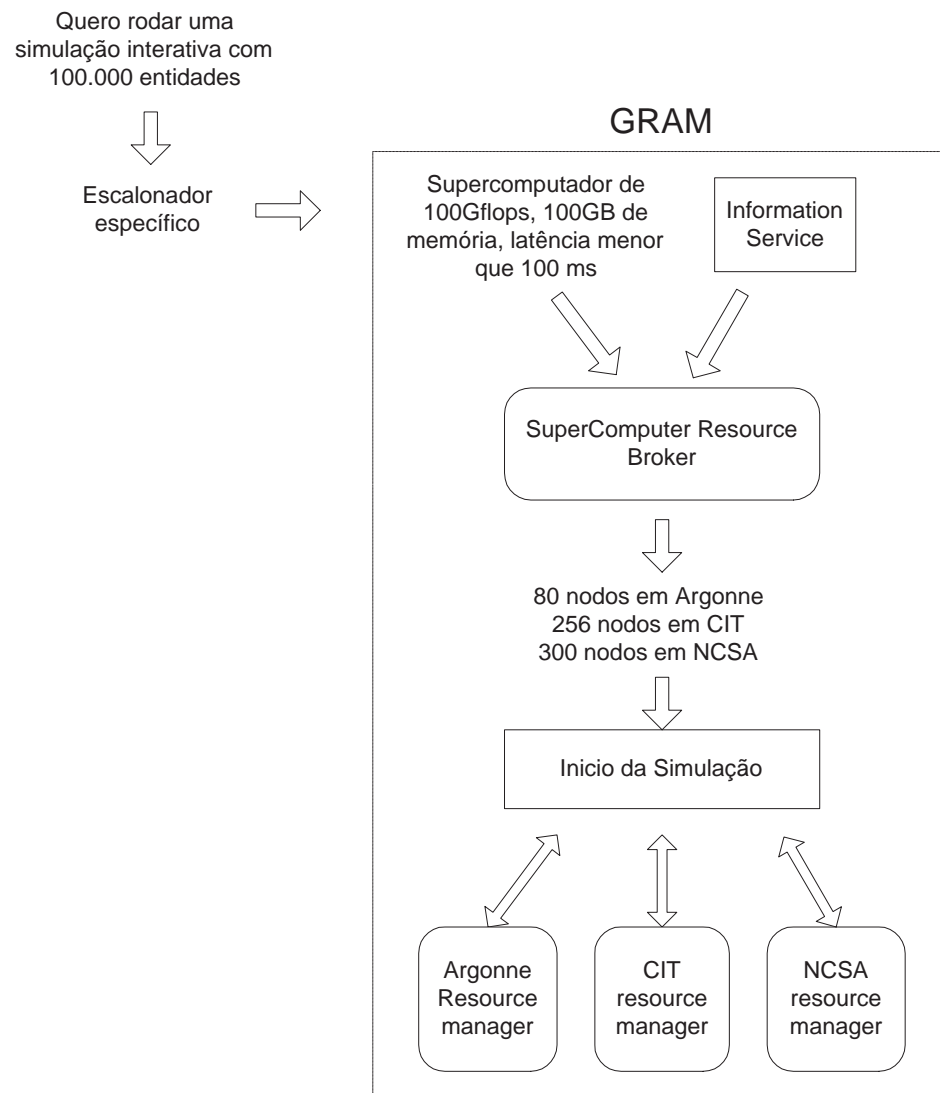


Figura 3.2: Exemplo de emprego de Globus-GRAM

3.1.3 Comunicação

O módulo Nexus tem a responsabilidade de prover mecanismos para a comunicação entre tarefas que sejam genéricos o suficiente para se adaptar a melhor tecnologia de comunicação, porém não tenham perda de performance nas diferentes arquiteturas em que possa vir a executar uma aplicação.

Nexus fornece uma interface de baixo nível, que suporta apenas uma única operação de comunicação, a RSR (*Asynchronous Remote Service Request*), que é semelhante a uma invocação remota de procedimento. Apesar de ser muito primitiva essa operação se adapta a arquitetura vigente, escolhendo a melhor forma de comunicação (memória compartilhada, *sockets* TCP, etc).

Por ser de baixo nível, Nexus é usado para desenvolver ferramentas e mecanismos de comunicação com maior nível de abstração tais como troca de mensagens, RPC, RMI, *multicast*, entre outros. Um bom exemplo é o MPI-G, biblioteca MPI para aplicações *grids* construída sobre o Nexus, herdando seus benefícios.

3.1.4 Transferência de dados

Freqüentemente, aplicações necessitam acessar dados de arquivos armazenados em memória secundária. Em Globus existem dois módulos para transferência de dados: GASS (*Global Access to Secondary Storage*) e GridFTP.

GASS foi o sistema de acesso remoto a arquivos inicialmente proposto. Ele permite acesso a dados de forma otimizada e com boa performance. Entretanto, devido à existência de servidores já consolidados (FTP), os administradores preferiam não ter que instalar outro servidor. Com isso em mente, foi criado o GridFTP, uma extensão ao protocolo FTP. Agora, mesmo que a extensão GridFTP ou GASS não estejam instalados em um dado servidor, os dados ainda podem ser acessados por FTP sem, contudo, obter os benefícios da implementação Globus.

3.1.5 Informação sobre os recursos

A gerência e correta execução de um *grid* dependem de informações atualizadas sobre o estado dos recursos que o compõe. MDS (*Metacomputing Directory Service*) permite obter informações sobre a velocidade do processador, tipos de interface de rede, espaço de memória disponível, latência de comunicação, largura da banda de rede, número de nodos de uma máquina paralela, etc.

3.1.6 Estado atual

Na versão atualmente sendo desenvolvida, os serviços Globus serão baseados em *Web Services* [TUE 2002] [TUE 04], num esforço para tornar mais amplo o campo de atuação de *grids*, buscando atingir não só o processamento de alto desempenho mas também o processamento comercial pesado [CIR 2003].

3.2 MyGrid

MyGrid foi concebido para ser um sistema simples, completo e seguro com o intuito de aumentar o número de usuários de *grids* [CIR 04]. Sua arquitetura permite a execução de aplicações *Bag of Tasks*, ou seja, aplicações cujas tarefas são independentes e podem ser executadas em qualquer ordem.

Uma aplicação MyGrid é composta de três partes ou subtarefas: inicial, *grid* e final, que são executadas nessa ordem. A subtarefa inicial é tipicamente usada para transferir os dados de entrada para as máquinas do *grid*. As subtarefas *grid* são os programas que re-

alizam a computação em si. Após todas as subtarefas *grid* terminarem suas computações, a sub tarefa final recolhe os resultados obtidos (salvos em arquivos).

3.2.1 Arquitetura MyGrid

Na terminologia MyGrid, a máquina que controla a execução da aplicação, i.e., a máquina que distribui os dados de entrada e coleta os resultados, é chamada de máquina base. As demais máquinas que executam as tarefas são conhecidas como máquinas de *grid*.

Para se juntarem ao *grid*, as máquinas devem suportar alguns serviços essenciais, tais como: execução remota, transferência de arquivos da máquina de *grid* para a máquina base e da máquina base para a máquina de *grid*. Esse conjunto de serviços formam o *Grid Machine Abstraction* (fig. 3.3).

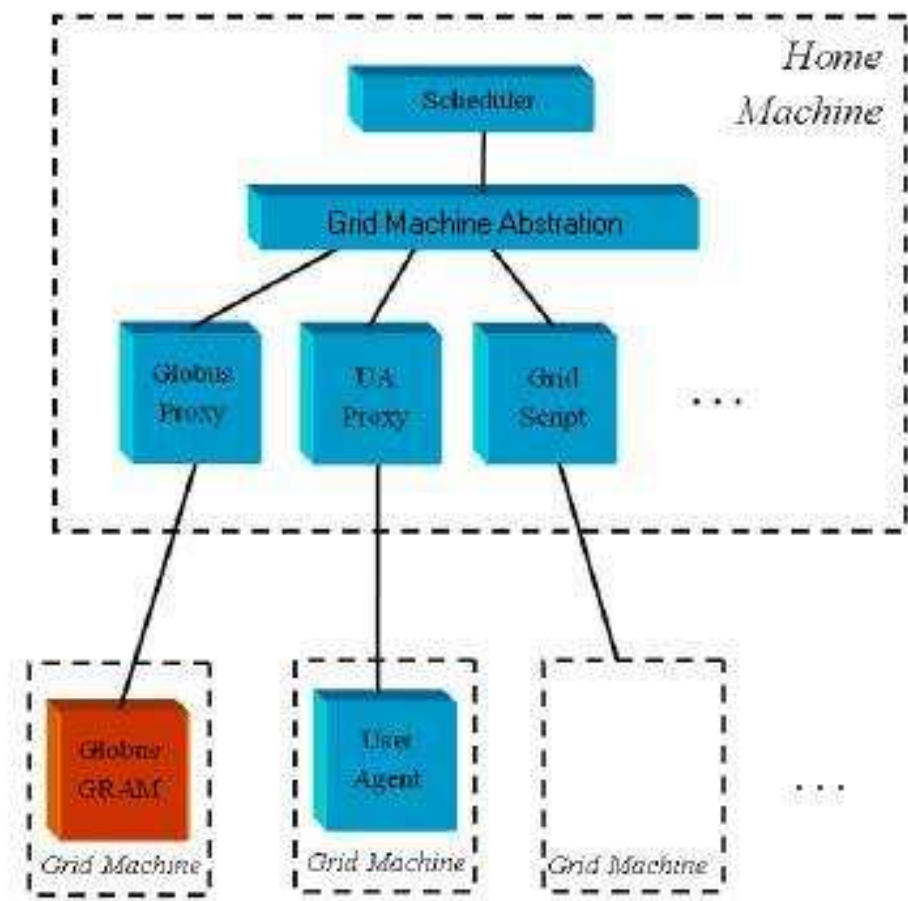


Figura 3.3: Arquitetura MyGrid

O *Grid Machine Abstraction* pode ser implementado de várias formas. O módulo *Grid Script* possibilita o acesso a máquinas nunca antes utilizadas, necessitando para isso somente a escrita de três *scripts* que descrevem como se dará o acesso. Caso o nodo possa ser acessado via serviços Globus, emprega-se o módulo Globus Proxy. Alternativamente, pode-se usar o módulo *User Agent*, que permite o acesso através dos serviços básicos e

ainda serviços mais especializados.

Um dos componentes da arquitetura de maior importância é o *scheduler*. O *scheduler* recebe a lista das tarefas a serem executadas, seleciona as máquinas que às executarão e submete e monitora as execuções. Este escalonador foi concebido para obter boa performance sem necessitar de previsões sobre o estado dos recursos do *grid*. Para isso emprega o algoritmo WQR (*Work Queue with Replication*).

WQR usa replicação de tarefas para compensar a falta de informações sobre o ambiente. Seu funcionamento é bastante simples. Existe uma fila de tarefas para serem alocadas, e sempre que um processador estiver disponível, receberá uma tarefa. Após todas as tarefas terem sido alocadas, elas serão replicadas em outros nodos. Uma vez obtido os resultados de uma das réplicas, as demais serão abortadas.

Segundo [SAU 2003], a performance do WQR é bastante próxima de algoritmos conhecidos como Sufferage e o Dynamic FPLTF que dependem de monitoramento e previsão (dos recursos) bastante precisos. Porém, consome mais ciclos de CPU desperdiçando, em média, até 40% dos mesmos.

3.3 Condor

Condor foi originalmente proposto para fornecer grande poder computacional a médio e longo prazo utilizando *clusters* e processadores ociosos em rede [LIV 2002]. Ao longo dos anos sofreu modificações até chegar ao estado atual, onde serve de infraestrutura para *grids* [LIV 2003].

A arquitetura Condor pode ser explicada descrevendo cada um dos seus elementos principais: *Agent*, *Resource* e *MatchMaker*. O usuário submete seus *jobs* ao *agent*, informando detalhadamente quais tipos de máquinas quer utilizar, ditando o número de CPUs, arquitetura, memória disponível, latência da rede, etc. Por exemplo, o usuário pode solicitar que somente processadores Intel, com 128MB de memória sejam usados. Cabe ao *agent* repassar os requisitos para o *MatchMaker*.

O dono de cada nodo especifica as restrições de uso de sua máquina. Exemplificando, o dono pode restringir o uso de seu equipamento aos momentos em que ele estiver ocioso, proibir a execução de *jobs* provenientes de determinado funcionário, etc. Essas restrições são passadas para o componente *resource* que, logo após, envia esses dados ao *MatchMaker* (fig 3.4).

O escalonador do sistema é o *MatchMaker*. Ele possui as informações sobre os recursos disponíveis e as tarefas que querem ser alocadas. Com base nesse conhecimento, o *MatchMaker* encontra o par tarefa/recurso mais compatível e os apresenta. Fica a cargo do *agent* entrar em contato com o *resource* escolhido para verificar se os dados ainda são válidos. Caso fique confirmada a compatibilidade, a tarefa entra em execução.

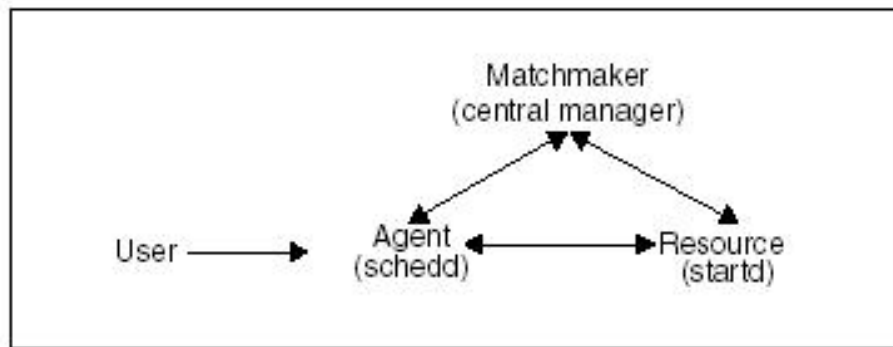


Figura 3.4: Principais componentes de uma comunidade *Condor*

3.3.1 *Flock of Condors*

Apesar da elegância de sua implementação, Condor não permitia que um usuário participasse de mais de uma comunidade (*Condor Pool*). Seu uso era restrito ao domínio de um único *MatchMaker*.

Com a incorporação de um novo componente à arquitetura, o chamado *Gateway*, ficou possível integrar várias comunidades. Após um acordo de cooperação entre duas comunidades, cada uma instala um *gateway* em seu *condor pool*. Dentro de seu domínio, um *gateway* é visto como uma máquina normal, porém, externamente ela representa todos os recursos e tarefas internas. É através dos *gateways* que as tarefas migram entre os *pools* e os recursos são vistos por ambas as comunidades.

A grande vantagem do *Flock of Condors* é que sua implementação é completamente transparente ao usuário. Firmar o acordo de cooperação entre duas entidades é um processo longo mas, uma vez que tenha sido afirmado, os usuários de ambas as entidades podem usufruir do acordo sem terem que ajustar suas aplicações.

3.3.2 Condor-G

Flock of Condors ainda assim tinha um grande empecilho. Usuários e recursos que não pertencessem a nenhum *pool* não poderiam tomar proveito de Condor, inviabilizando seu uso em *grids*. Objetivando adaptar Condor para execuções em grade foi projetado Condor-G. Condor-G emprega Globus-GRAM para auxiliar na captação dos recursos.

O Condor-G *agent* submete ao GRAM os *daemons* Condor que serão executados em cada recurso escolhido pelo mesmo. Todos esses recursos formarão o *Condor Pool* do usuário. Quando entram em execução, os *daemons* notificam o *MatchMaker* iniciado pelo usuário e exportam suas características. Nesse momento os *jobs* são enviados para o *agent* e os melhores casamentos tarefa/recurso são encontrados.

Condor-G é um exemplo de como Globus é poderoso e útil para construção de novas infraestruturas *grid*. O uso conjunto dos protocolos Globus e dos *daemons* Condor permitiu que o sistema Condor fosse confortavelmente estendido para plataformas *grid*.

4 *Modelo*

Este trabalho focaliza dois pontos de fundamental importância na computação em grade: novas abstrações para a programação e distribuição de carga. Como será visto na seção 4.1.1, para a definição da computação é usado o conceito de tarefa. Uma tarefa pode ser vista como uma *thread* local que pode migrar para outro nodo do *grid*. Vários outros trabalhos tem utilizado múltiplas *threads* como ferramenta de programação, por exemplo, [LEI 94] e [ROC 98], porém, o diferencial deste trabalho é a utilização de tarefas preguiçosas, de forma semelhante à definida em [REV 2004].

Tarefas preguiçosas são como *threads* locais, contudo, elas somente são ativadas em função dos recursos de que necessitam. Enquanto não houver uma máquina disponível para executar uma tarefa preguiçosa, as informações necessárias à sua execução permanecem em uma lista do escalonador. Desta forma, uma tarefa que não entrou em execução, não consome tempo de CPU da máquina hospedeira e ocupa pouco espaço em memória, em relação à uma *thread* em execução.

A distribuição das tarefas entre os nodos do *grid* é feito segundo uma abordagem *work-stealing*. O mecanismo de *work-stealing* foi empregado com sucesso para escalonamento em *clusters* na implementação de Athapascan [ATH 04] e foi também proposto para ser usado no escalonamento em redes de *workstations* [BLU 94] e implementado em Cilk [CIL 04]. Neste trabalho busca-se usar tal mecanismo para escalonamento distribuído de tarefas em um *grid* computacional.

A idéia por trás do *work-stealing* é bastante simples: quando um nodo fica ocioso, i.e., sem tarefas para executar, ele rouba tarefas da lista de tarefas de outro nodo (a vítima). Esta abordagem é oposta ao *work-sharing*, no qual, no momento da criação de uma nova tarefa, o escalonador busca migrá-las para outra máquina.

4.1 *Arquitetura*

Para atender o objetivo de facilitar a programação e, ao mesmo tempo, gerenciar a execução de aplicações em grade, este modelo foi subdividido em três camadas. Cada camada possui uma responsabilidade única. A divisão proporciona uma visão organizada dos problemas que devem ser resolvidos para atingir os propósitos deste trabalho. As camadas mencionadas compreendem:

- 1º Camada (Camada de programação): Tem por objetivo disponibilizar ao programador um modelo de programação simples, através de bibliotecas de linguagem. O programador poderá descrever o problema usando conceitos abstratos como tarefas.

Estas tarefas possuem a peculiaridade de poderem ser criadas sem serem ativadas, ficando em uma fila de espera na segunda camada, e poderão ser ativadas (ou não) pelo escalonador em função da disponibilidade dos recursos de que necessitam.

- 2º Camada (Camada de *runtime*) : Tem por objetivo principal alocar as tarefas em nodos disponíveis, ou seja, é o escalonador do sistema. Realiza seu trabalho com base nas informações passadas pela terceira camada sobre os recursos existentes. É empregado uma política de escalonamento do tipo *work-stealing*.
- 3º Camada (Camada de gerência de recursos): Seu objetivo é gerenciar os recursos, mantendo informações sobre os nodos que estão participando do *grid*. É responsável por disponibilizar estas informações para a segunda camada.

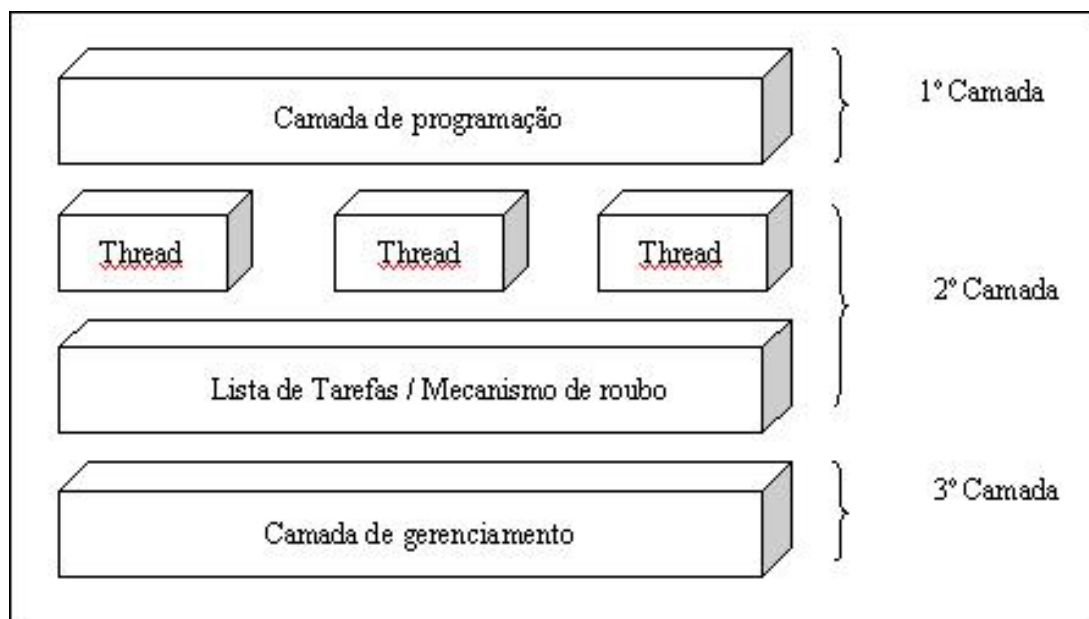


Figura 4.1: Camadas da Arquitetura XgridApp

Nas seções seguintes serão discutidos os pontos principais a cerca de cada camada do modelo proposto.

4.1.1 Camada de Programação

O programador define seu problema usando o conceito de *Task* (ou tarefa). Uma *task* pode ser vista como uma *thread* local que executa parte da computação.

Ao ser iniciada, uma *task* recebe seus dados de entrada e cria uma *task* filho, que nada mais é que uma cópia da mesma, porém sem trabalho. As *tasks* filho são tarefas preguiçosas (*Lazy Tasks*), ou seja, elas podem ou não serem ativadas, dependendo da disponibilidade dos recursos de que necessitam. Enquanto não forem ativadas, as informações necessárias a sua execução ficam em uma fila de espera do escalonador local. Desta forma, não desperdiçam ciclos de CPU e despendem pouca memória para armazenar informações de controle, como a identificação da sua *task* pai, entre outros. *Lazy tasks*

são ativadas quando ocorre um roubo, i.e., quando uma outra máquina do *grid* que estava ociosa rouba uma *lazy task* da lista deste nodo.

Não é possível determinar quando e se ocorrerá um roubo. Se uma *task* repartisse o trabalho entre seus filhos antes deles serem ativados e ficasse esperando pelos resultados, haveria um provável desperdício de tempo e memória. Para melhor explorar os recursos disponíveis, é necessário que a *task* permaneça trabalhando e somente quando um de seus filhos for ativado, é que deve repartir o trabalho que resta ser feito. A tarefa fica então em um laço: enquanto existir trabalho a fazer, ela adquire uma pequena porção de tamanho fixo e executa (ver pseudocódigo abaixo). Caso sua *task* filho seja ativada, parte do trabalho será delegado a ela. O tamanho deste depende da aplicação mas usualmente será metade do trabalho que resta a ser feito pelo pai. Ao término de seu trabalho a tarefa pai espera pelos resultados de seus filhos.

```

Adquira uma parte do trabalho
Enquanto existir trabalho a ser feito
    Faça execute uma porção do trabalho
    Adquira uma parte do trabalho
Fim Enquanto

```

Figura 4.2: Pseudocódigo de uma *task*

É importante que o leitor consiga discernir corretamente tarefa de trabalho. Tarefa é a descrição de **como fazer** o trabalho(algoritmo). Já o trabalho especifica **o que** deve ser feito (uma ou mais variáveis). Por exemplo, para ordenar um vetor é preciso ter um algoritmo de ordenação (tarefa) e um vetor para ordenar (trabalho).

Durante a execução, a criação e roubo de *lazy tasks* pode ser visto como uma árvore. Como mostra a figura 4.3, o modelo de tarefas é hierárquico, em cada nível existe pelo menos uma *task* pai e zero ou mais filhos. O número de filhos ativos dependerá exclusivamente do número de roubos pois uma mesma *lazy task* pode ser roubada inúmeras vezes. Entretanto, em cada roubo receberá partes diferentes do trabalho.

Essa abordagem restringe a classe de problemas que podem ser resolvidos. Os problemas mais indicados são aqueles em que os dados podem ser enumerados e o algoritmo usa a técnica de divisão e conquista. Simulações de Monte Carlo e algoritmos de ordenação de dados como o MergeSort são exemplos de problemas resolvíveis usando *tasks*.

Para compreender o funcionamento de uma *task* é preciso conhecer seus estados e transições. A figura 4.4 mostra o diagrama de estado de uma *task*. Toda *task* (com exceção da raiz) é inicialmente uma *lazy task* e está portanto **Desativada**. Ao ser roubada, ela tem que receber trabalho de seu pai, ficando no estado conhecido como **Iniciada**. Neste estado ela também cria uma *task* filho que vai para a lista local do nodo ladrão. Após ter sido iniciada a *task* começa a calcular uma pequena porção do trabalho. Esse comportamento se repete até que todo o trabalho tenha sido calculado, a *task* permanece no estado **Calculando**. Se uma outra máquina ficou ociosa e roubou uma *lazy task*, a *task* tem que dividir o trabalho que resta a ser feito e doar uma parte para seu filho (estado **Dividindo trabalho**). Conseqüentemente, caso um filho termine de calcular sua

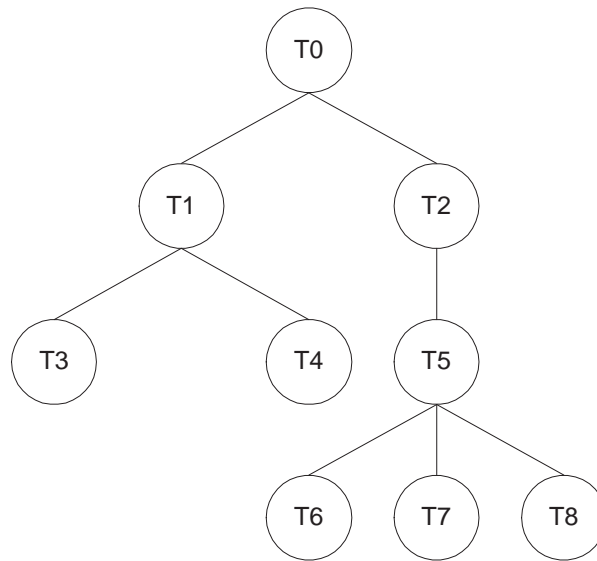


Figura 4.3: Hierarquia de tarefas durante a execução

parte, ele envia ao seu pai seus resultados que devem ser armazenados e eventualmente processados no estado **Recebendo resultados**. Com a partilha do trabalho, pode ser que a *task* tenha terminado seu processamento antes mesmo de receber os resultados de todos os seus filhos ativos, ela deve então esperar que todos eles terminem (estado **Esperando resultados dos filhos**). Uma vez recebidos todos os resultados, deve-se combiná-los (estado **Combinando resultados**) para obter a solução final e finalmente publicá-los.

O estado **Publicando resultados** tem diferentes comportamentos dependendo de qual é a *task*. Caso seja a raiz, nesse estado devem ser mostrados os resultados ao usuário, seja no monitor ou em outro dispositivo. Contudo, se a *task* for um filho, ela deve enviar seus resultados parciais para seu pai e assim sucessivamente, até que todo o trabalho seja computado.

Apesar de tolerância à falhas não pertencer ao escopo deste trabalho, cabe aqui uma consideração interessante. Cada *task* possui uma lista de seus filhos ativos e dos respectivos trabalhos. Essa informação pode ser usada para criar um mecanismo de tolerância à falhas bastante simples. O programador usuário pode especificar um tempo limite para que os resultados de um filho sejam calculados e enviados para o pai. Caso esse tempo seja ultrapassado, provavelmente devido a uma falha, o mesmo trabalho será entregue a outra *lazy task* em um próximo roubo. Tal mecanismo é simples e garante um nível de tolerância razoável, entretanto, exige que o programador conheça muito bem o comportamento de sua aplicação, cabendo a ele implementar toda essa funcionalidade.

4.1.2 Camada de *RunTime*

A camada de *runtime* é de fato o escalonador do sistema. O modelo emprega um escalonador de dois níveis: o nível mais alto faz a distribuição da carga entre os nodos usando um mecanismo de *work-stealing*; já o nível inferior busca diminuir os *overheads*

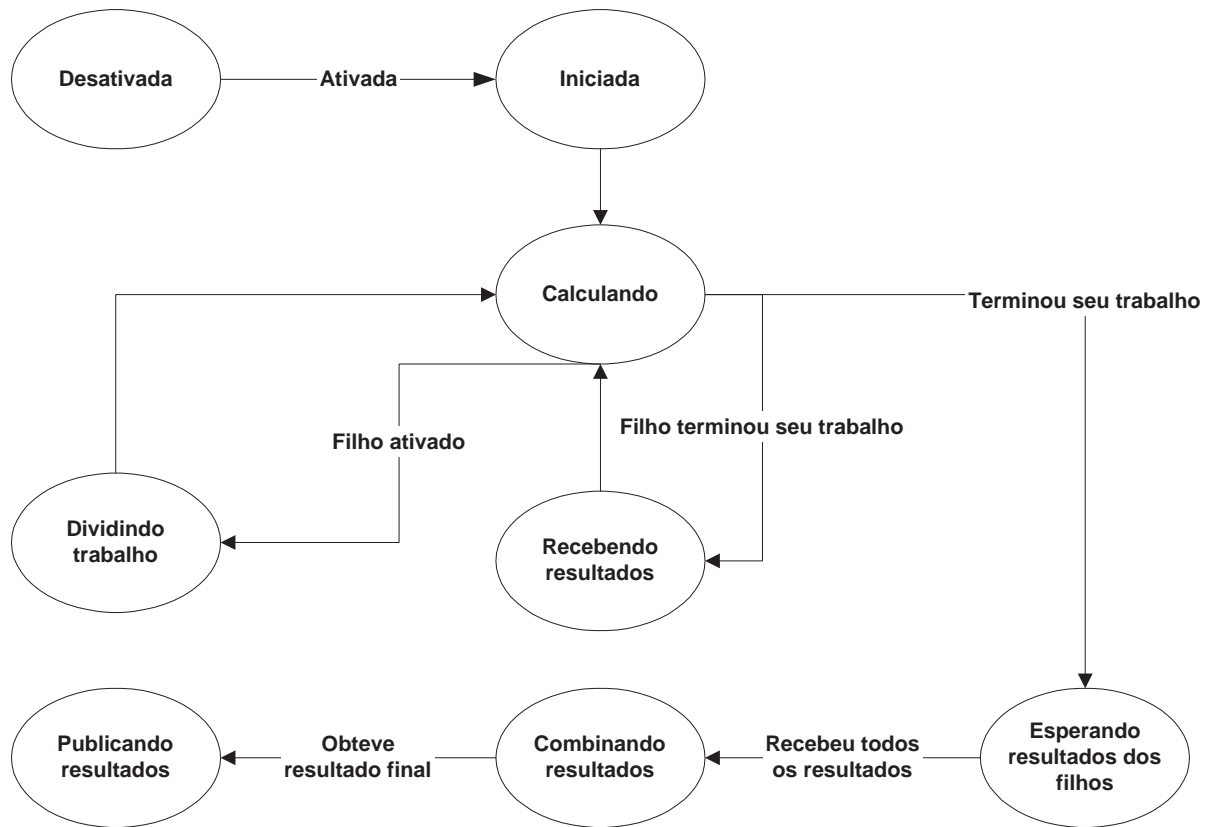


Figura 4.4: Diagrama de estados de uma *task*

de comunicação e melhor explorar o potencial das arquiteturas SMP usando uma técnica de *multithreading* baseada em [GAU 99].

Cada nodo do sistema possui uma lista de *lazy tasks* ainda não ativadas. Quando termina o processamento de uma *task*, o nodo busca uma nova em sua lista local. Caso a lista esteja vazia, um procedimento de roubo é acionado: é escolhido um nodo vítima que terá parte de sua lista roubada. O mecanismo de *work-stealing* é portanto, acionado quando um nodo do sistema está ocioso.

A interação entre o nodo ladrão e o nodo vítima acontece mediante dois agentes: *Thief Agent* e o *Victim Agent*. *Thief Agent* escolhe uma vítima de acordo com as informações da 3º camada e em seguida, entra em contato com o *Victim Agent* da máquina alvo. *Victim Agent* verifica se possui alguma *Lazy Task* em sua lista. Caso exista uma, antes de encaminhá-la ao *Thief Agent*, é feito o pedido de trabalho para seu pai, que se encontra na vítima. Desta forma, não é necessário mais uma comunicação entre os nodos para adquirir trabalho, a *task* roubada já o possui. Tendo recebido a nova *task*, o nodo cria automaticamente a *lazy task* filho. A figura 4.5 descreve todos os passos envolvidos em um roubo.

A escolha da vítima pode ser efetuada de várias formas. A mais simplória seria escolher aleatoriamente um dentre os nodos conhecidos. Por outro lado, a escolha pode ser fundamentada em informações sobre cada máquina, tais como, latência da rede, distância entre nodos, número de vezes que uma ação de roubo sobre determinado nodo teve sucesso, etc. Aquele algoritmo é bastante simples de implementar e faz sua escolha muito rapi-

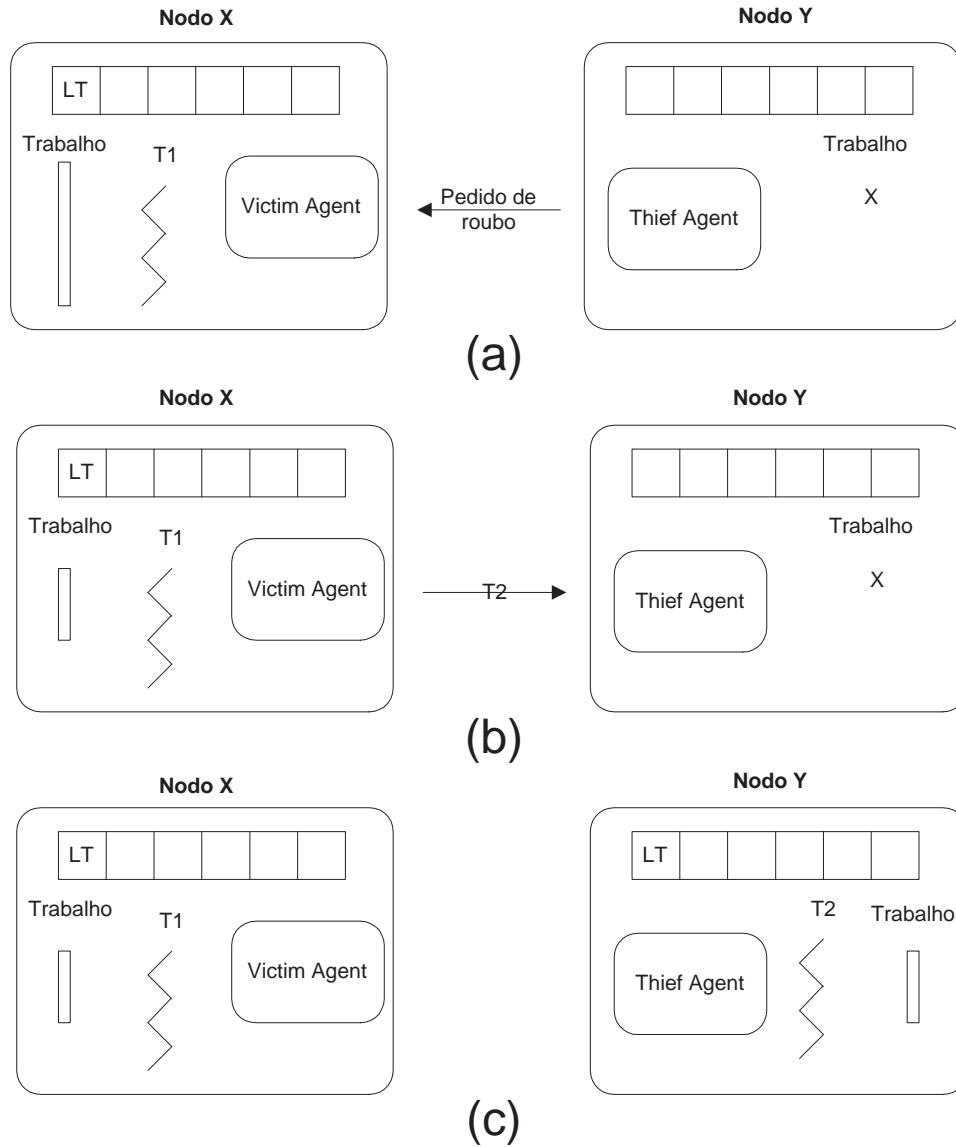


Figura 4.5: No passo (a) o nodo X está executando sua *task* T1 e o nodo Y está ocioso. Ele então dispara o mecanismo de roubo, fazendo com que o seu *Thief Agent* entre em contato com o *Victim Agent* do nodo X. Ao processar o pedido de roubo no passo (b), o *Victim Agent* percebe que existe uma *lazy task* (LT) em sua lista. É criada a *task* T2 com os dados da LT e parte do trabalho de T1 é passado para T2. Finalizando a operação de roubo no passo (c), T2 recebe um fluxo de execução no nodo Y e cria sua própria *lazy task*. A LT do nodo X permanece na lista, podendo ser roubada novamente, mas receberá outra parte do trabalho

damente, contudo, pode errar com maior frequência que este. Manter informações atualizadas sobre cada recurso pode ser muito custoso em termos de espaço e processamento e mesmo assim a heurística escolhida pode não ser adequada. Além disso, a decisão leva mais tempo para ser tomada. Todavia, uma boa heurística e informações corretas podem levar a uma taxa de erro significativamente baixa.

Em cada nodo participante da execução distribuída, um escalonador de baixo nível fica responsável por criar um grupo de *threads* batizadas de *Task Eaters*. Cada uma dessas *threads* executa um laço que, a cada iteração, busca e executa uma *task* da lista gerenciada pelo *Task Manager* daquele nodo. Se a lista estiver vazia, a *task eater* dispara uma operação de roubo de trabalho (*stealing*) em relação a outro nodo do sistema.

Todas as operações de uma *task eater* podem ser sumarizadas no diagrama de estados da figura 4.6. Ao entrar em execução, uma *task eater* procura adquirir uma *task* da lista local, ação simbolizada pelo estado **Adquirindo tarefa**. Estando a lista vazia, é acionado o mecanismo de roubo e a *task eater* permanece esperando (estado **Acionando roubo**). O roubo pode não ter sucesso e, portanto, deve-se tentá-lo repetidas vezes. Se o alvo foi corretamente escolhido, terá retornado uma nova *task* que deve ser executada (estado **Executando tarefa**). Mesmo que todas as tentativas de roubo tenham fracassado, deve-se impor um limite às mesmas, pois seu fracasso pode estar relacionado ao término da aplicação. Neste caso, a *task eater* entra no estado **Abortado** e é eliminada.

A existência de várias *task eaters* permite que sejam utilizados todos os processadores de máquinas SMP, extraindo ao máximo seu potencial. Além disso, seu uso pode suavizar perdas de processamento devido a espera dos filhos de uma *task*. No momento em que uma *task* passa para o estado Esperando resultados dos filhos, permanecendo bloqueada, desperdiça ciclos de CPU ¹. Neste caso, uma outra *task eater* pode ser instanciada. Ao entrar em execução a nova *thread* acionará o mecanismo de roubo, receberá uma nova *task* que passará a ocupar os ciclos ora perdidos.

Com a técnica *work-stealing* garante-se que máquina nenhuma ficará sobrecarregada e tão pouco alguma ficará ociosa. Mas uma questão ainda fica pendente: como definir o tamanho do trabalho em um ambiente composto de recursos heterogêneos? Como dito na seção anterior, poder-se-ia entregar sempre metade do trabalho que falta ser computado. O problema dessa abordagem é que sendo o ladrão menos eficiente, o tempo de execução passa a ser limitado pelo seu desempenho e ele acaba atrasando o resultado final. Deve-se usar uma técnica que seja sensível à mudança dinâmica de desempenho de cada nodo. A técnica aqui apresentada é chamada *Time Goal* e foi inspirada nos trabalhos do projeto ISAM [SIL 2003].

É definido um tempo ideal (*time goal*) para que seja calculado uma pequena porção do trabalho ². Ao término da execução de uma *task* a sua *task eater* calcula o tamanho do próximo trabalho levando em conta o tempo ideal e o realmente despendido (pode ser a média de cada tempo ou o seu último valor). Em um futuro roubo, a nova *task* receberá uma porção de trabalho grande, caso tenha calculado as porções em menor tempo que o *time goal* ou então receberá um trabalho menor, caso tenha levado mais tempo que o ideal em cada sub computação. Desta maneira atinge-se uma granulosidade ajustável em

¹Nota-se que ao entrar no estado de espera por resultados, a máquina passa a ficar ociosa

²Uma *task* fica em um laço executando pequenas porções do trabalho de tamanho fixo(rever seção 4.1.1)

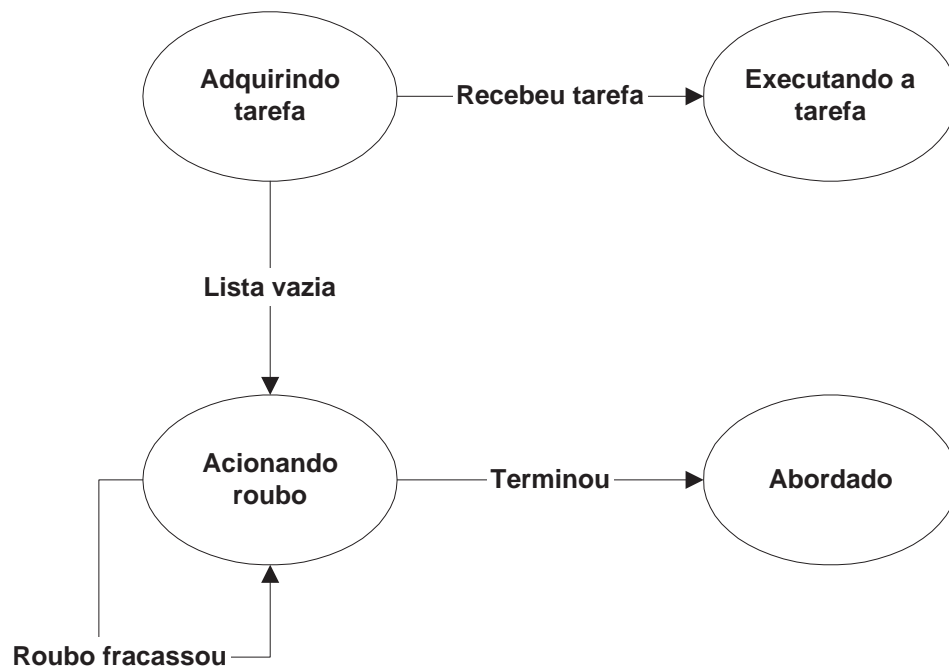


Figura 4.6: Diagrama de estados de uma *task eater*

relação ao desempenho variável de cada máquina.

O texto acima descreve o comportamento no caso geral, onde o nodo ladrão já tenha computado pelo menos uma *task*. No primeiro roubo de uma máquina, ela não possui registro do tempo de execução de uma porção do trabalho. Sem ter essa informação o alvo envia ao ladrão uma *task* que tem como trabalho apenas uma única porção. Após finalizar sua execução, a *task eater* possui o tempo despendido e pode calcular o tamanho correto do próximo trabalho.

A técnica de *time goal* é na verdade pertencente a 1º camada uma vez que somente o programador conhece suficientemente sua aplicação ao ponto de definir um tempo ideal de execução de cada porção de seu problema e a forma de calcular o tamanho do novo trabalho.

4.1.3 Camada de Gerenciamento

Em uma execução de determinada aplicação, deve-se ter informações sobre todos os recursos existentes. Na camada de gerenciamento é colhida toda a informação sobre os nodos do *grid*.

Os nodos ociosos que passam a contribuir com o *grid* devem publicar seus endereços em uma entidade responsável pelo controle dos recursos. Essa entidade recebe o nome de *Node Manager*. Dependendo da política de escolha da vítima, o *node manager* irá reter maior ou menor número de informações.

Em uma política de escolha aleatória, a única informação necessária é o endereço de todos os nodos. Já em uma política que emprega heurísticas, o volume de conhecimento

deve ser maior. Características como distância entre os nodos, latência da rede, se o nodo é um grande fornecedor de trabalho, são possíveis informações coletadas. Essas características podem variar com o tempo e portanto, deve-se atualizar os dados do *Node Manager* constantemente.

O *node manager* pode ser distribuído ou centralizado. Na implementação centralizada existe somente um único *node manager* para todo o *grid*, contendo informações sobre todos os recursos. Seu acesso é inevitável e imprescindível em casos de roubo, entrada de um nodo ao sistema ou atualização de dados. Todos esses acessos concorrentes degradam o desempenho geral do *grid*. O *node manager* passa a ser um gargalo do sistema e por isso, a versão centralizada não é escalável, servindo tão somente para *grids* pequenos como os *intragrids*.

Inúmeras técnicas de sistemas distribuídos podem ser empregados para implementar um *node manager* distribuído. Uma abordagem possível é manter em cada nodo um histórico das últimas vítimas de roubo. Cada histórico pode ser visto como um *node manager* local. Ao ser acionado, o agente de roubo (*Thief Agent*) analisa seu histórico e encontra uma vítima dentre os nodos conhecidos. Caso a vítima não tenha *tasks* para serem roubadas, ela encaminha a mensagem de roubo para outro nodo que ela conhece. Quando este nodo responde, envia uma *task* e também seu endereço. Agora o nodo inicial tem trabalho a realizar e também conhece mais um nodo do sistema. Desta forma, cada nodo tem a visão de apenas parte do *grid*.

Dois casos especiais devem ser tratados: quando é a primeira vez que o nodo é iniciado e quando ele não consegue se comunicar com nenhum nodo do seu histórico. Uma alternativa seria manter servidores de endereços de nodos, computadores que mantêm nomes de alguns nodos e são chamados de *Central Node Manager*. Podem existir vários *Central Node Manager* distribuídos pelo *grid*. Logo que é adicionado ao sistema, o nodo publica seu endereço no servidor que conhece. Se não conseguir comunicação com as vítimas de seu histórico, ou é a primeira vez que é iniciado, o nodo pode pedir ao servidor de endereços uma lista de possíveis vítimas.

A dinâmica de funcionamento do algoritmo de históricos é enfatizada pela figura 4.7. O nodo N1 encontra-se ocioso e dispara um roubo. Este nodo conhece somente os nodos N2, N4 e N5. Sua política de escolha decide pelo nodo N2 (a). Porém N2 também não tem tarefas, e ele passa a mensagem para N3 (b). O nodo N3 possui trabalho, ele envia para N1 uma tarefa e seu endereço (c). Agora N1 conhece mais um nodo do *grid* (d).

Por fim, a gravura 4.8 representa cada uma das camadas descritas nas seções anteriores, explicitando todos os seus elementos estruturais.

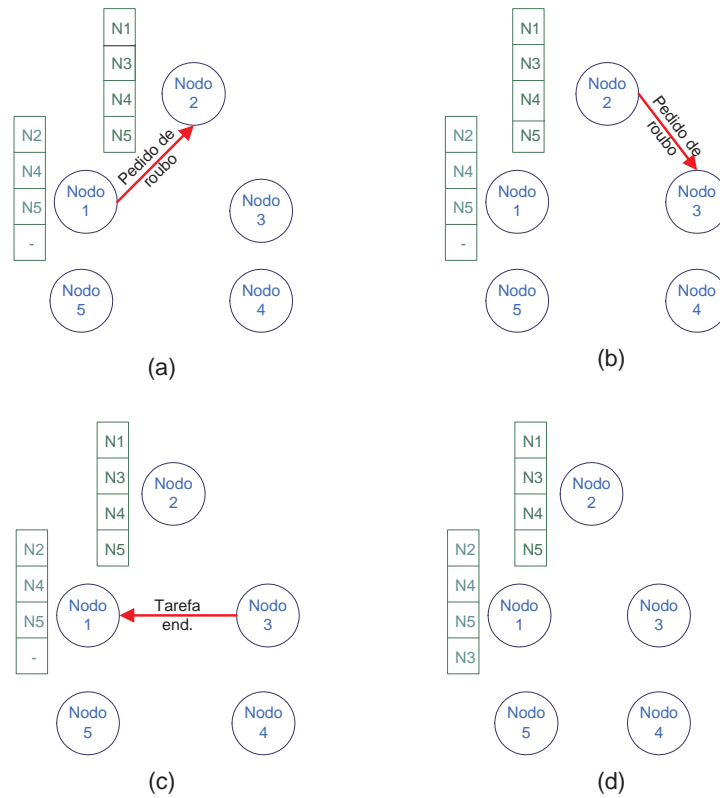


Figura 4.7: Dinâmica de execução de um algoritmo baseado em histórico

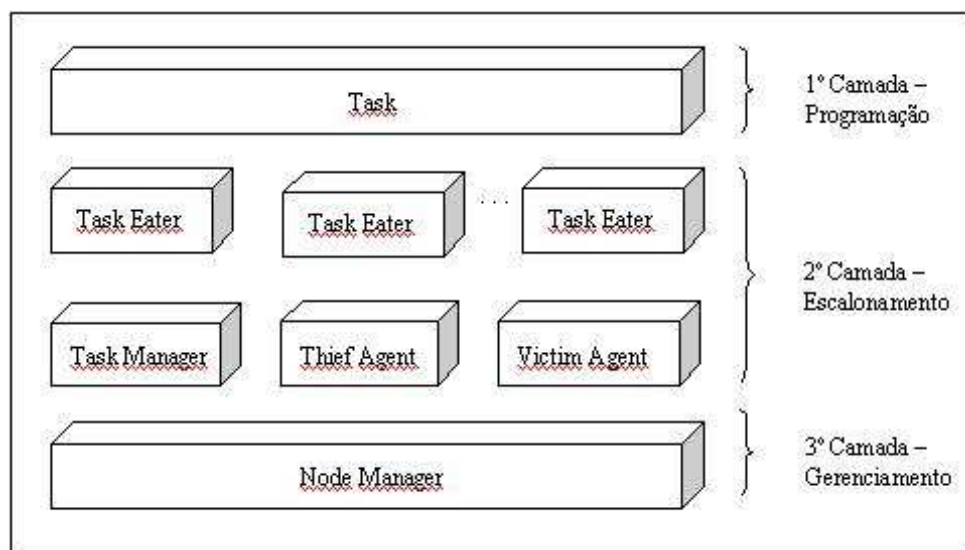


Figura 4.8: Camadas da arquitetura XgridApp, estendidas

5 *XgridApp* - Implementação

Tendo em mãos um modelo bem delineado, resta ainda implementar um protótipo objetivando demonstrar a qualidade do mesmo. Neste capítulo será descrita uma implementação do modelo proposto, assim como um estudo de caso.

5.1 Linguagem de Programação

A primeira decisão que deve ser tomada à cerca de uma implementação é a linguagem de programação empregada. Neste trabalho utilizou-se a linguagem Java. Os principais aspectos que tornam Java uma escolha oportuna como base para a construção de um protótipo de infraestrutura *grid* são:

- Portabilidade: Sem duvida é a característica mais importante no que tange sistemas heterogêneos. Ao compilar um código Java, ao invés de se obter um código nativo para a máquina destino, obtém-se um código intermediário chamado de bytecode. O bytecode é interpretado pela máquina virtual Java (JVM). Desta forma, a responsabilidade de manter a portabilidade da aplicação é transferida para a JVM que deve ter uma implementação para cada plataforma. Atualmente existem implementações de JVMs para plataformas como Solaris, Windows, MacOS, além de implementações desenhadas para sistemas embarcados, PDAs, celulares, entre outros.
- Carga dinâmica de código: Java permite que os códigos das classes sejam carregados dinamicamente tanto a partir de arquivos locais como de um servidor HTTP.
- Segurança: A JVM executa as aplicações em um ambiente controlado que evita acessos indevidos à memória. Além disso, possui gerentes de segurança (*Security Manager*) para controlar o acesso aos recursos nativos.
- Suporte a concorrência e sincronização: Java possui em sua API a classe *Thread* a qual serve para disparar linhas de execução concorrentes dentro da aplicação. O controle a seções críticas e sincronizações é feito por abstrações conhecidas como monitores [SIL 2003a].
- Produtividade no desenvolvimento de *software*: Java possui uma série de benefícios para a produção de *software*, destacando-se: (i) o gerenciamento automático de memória, a sua característica de ser uma linguagem fortemente tipada e o tratamento estruturado de exceções, tornam a programação menos suscetível a erros;

- (ii) o uso de componentes plugáveis permite o reuso de *software*, técnica muito difundida atualmente; (iii) o modelo de programação orientado à objetos favorece o encapsulamento de dados e componentes.
- **Objetos Distribuídos:** outro aspecto oportuno de Java, no tocante à computação distribuída, é a existência de um modelo de computação baseado em objetos distribuídos, denominado RMI. No modelo RMI, os objetos Java passam a estar habilitados a receberem invocações de métodos advindas de outros nodos além do que executa o dado objeto, ou seja, existe o suporte à execução de invocações remotas de método.

5.2 Características da Implementação

O modelo apresentado no capítulo 4 deixa em aberto uma série de questões como a política de escolha do nodo vítima, gerenciamento dos nodos de forma centralizada ou distribuída, o próprio paradigma de programação utilizado, entre outros. Nesta seção será definida a arquitetura do protótipo e serão descritos todos os aspectos que a permeiam.

Acrescenta-se à arquitetura mostrada na figura 4.8, uma quarta camada, que representa o *middleware* ISAM. A integração com os módulos do ISAM tem um objetivo colaborativo: XgridApp emprega os serviços ISAM/EXEHDA para auxiliar no gerenciamento da execução e, em contrapartida, o modelo *task/work-stealing* é usado no ISAM como uma segunda opção frente ao modelo mestre-escravo já em operação.

As primitivas para escalonamento de objetos [SIL 2003b] possibilitam disparar aplicações XgridApp a partir de uma única máquina. A máquina base escolhe o número de nodos que deseja, as características dos mesmos, como memória disponível, nível de uso do processador, latência de rede, etc. Em vista desses requisitos, os serviços do EXEHDA selecionam alguns dos nodos que estão compondo o *grid* naquele momento. EXEHDA passa a ser empregado de forma semelhante ao Globus-GRAM, selecionando os nodos adequados e migrando o código XgridApp e as classes das tarefas do usuário. Nota-se que não é necessário ter os módulos do XgridApp instalados nas máquinas da grade, apenas é necessário que o *middleware* ISAM esteja presente.

Uma vez que o código tenha migrado para os nodos constituintes do *grid*, XgridApp passa a gerenciar a execução da aplicação. Os mecanismos de *work-stealing* e gerenciamento dos nodos pertencentes a aplicação tomam conta do fluxo de execução. Durante esse período, os serviços de monitoramento dos nodos [YAM 2004] permitem manter dados atualizados sobre cada máquina. Essa informação pode ser usada na escolha dos nodos vítimas.

No protótipo atual a integração ainda não foi realizada, desta forma, o disparo da aplicação e a escolha dos nodos são efetuados manualmente pelo usuário. Como a escolha das vítimas é feita de forma randômica, não é necessário o uso de informações adicionais. Todavia, o protótipo está bem modulado e foi desenhado para facilitar a futura integração.

5.2.1 Camada de Programação

No protótipo, a classe *Task* implementa as funcionalidades da abstração de mesmo nome. O programador deve estender a classe *Task*, sobrescrevendo os métodos abstratos adequando-os aos seus propósitos.

O código da figura 5.1 ilustra todos os métodos abstratos da classe *Task*. Inicialmente, o método **initTask** deve ser programado para receber os parâmetros iniciais e o trabalho total destinado à tarefa. Este método é chamado somente no momento em que a *Task* for ativada ¹, pois, estando inativa, não possui um fluxo de execução e não contém trabalho. O método **exec** é responsável por computar cada sub parte do trabalho. Este método é chamado automaticamente repetidas vezes, até que todo o trabalho residente no nodo tenha sido efetuado. Seu parâmetro de entrada é um objeto que contém uma porção do trabalho. O método **getWork** retorna trabalho ao chamador, que pode ser o próprio objeto ou um de seus filhos. Caso o pedido seja proveniente de um filho, o método *getWork* tipicamente retornará metade do trabalho que resta ser feito. Já se o pedido for do próprio objeto, o método retornará uma pequena porção fixa de trabalho.

Toda a vez que uma *Task* filho terminar sua execução, ela deve enviar ao seu pai seus resultados. Para isso, ela invoca o método remoto **sendResults** de seu pai. O código do método *sendResults* deve ser construído para receber, armazenar e, eventualmente tratar, os resultados parciais dos filhos. Ao final de sua computação, a *Task* deve combinar os resultados calculados com os resultados obtidos pelos seus filhos. No método **merge**, os dados armazenados pelo método *sendResults* devem ser combinados com os resultados alcançados na *Task* corrente.

```
abstract class Task{

    abstract public void exec(Object param);

    abstract public void initTask (Object param);

    abstract public Object getWork (Who who);

    abstract public void sendResults (Object result, long who);

    abstract public void merge ();

}
```

Figura 5.1: Código da classe Task

O uso da técnica de *time goal* é facultativo. Se for da preferência do programador, ele pode usá-lo através da classe *TimeGoalCriteria*. Para tanto, deve-se estender a classe *TimeGoalCriteria* e definir como será calculado o novo valor de tamanho da tarefa, levando em consideração o tempo ideal (*time goal*) e o realmente gasto (*elapsedTime*) no cálculo de cada porção do trabalho. Esse método deve ser chamado internamente pelo método *getWork* da *Task*.

¹Uma *task* é ativada quando for roubada por outro nodo.

```

abstract class TimeGoalCriteria {

    public abstract Object guessNewWorkSize
        (Object param, long elapsedTime, long timeGoal);

}

```

Figura 5.2: Código da classe TimeGoalCriteria

O programador fica responsável apenas por sobrescrever os métodos listados acima. A gerência da ativação de *lazy tasks*, o controle do número de filhos ativos, a espera pelos resultados dos filhos, o laço de repetição das chamadas ao método *exec* e demais tarefas necessárias para o correto funcionamento da aplicação são tratadas pela segunda camada, mais precisamente, pelas *Task Eaters* (ver seção 5.2.2). O desenvolvedor do aplicativo deve apenas ter em mente o que representa cada método e sua seqüência de execução que começa por *initTask*, passa por *exec* repetidas vezes (enquanto houver trabalho) e finalmente é chamado o método *merge*. Esse último possui duplo comportamento: caso a *task* seja filho de alguém, este método deve chamar o método *sendResults* da *task* pai, mas caso esta seja a *task* raiz, este método deve mostrar os resultados ao usuário.

5.2.2 Camada de *Runtime*

Uma máquina participante do *grid* possui pelo menos uma *Task Eater* que é responsável por gerenciar e executar as tarefas. Cada *Task Eater* permanece em um laço buscando tarefas da lista local ou disparando o mecanismo de roubo.

A figura 5.3 mostra os passos executados por uma *Task Eater*. O primeiro passo é adquirir uma tarefa para ser executada. Em seguida uma cópia da mesma, porém sem trabalho e sem fluxo de execução, é salva na lista local para um futuro roubo. Ao longo dos passos 3 a 8, é computado cada pequena porção de trabalho. A cada invocação do método *exec* é armazenado o tempo decorrido para ser usado como medida de desempenho da máquina em um próximo roubo. No nono passo é efetuada a sincronização entre a tarefa e seus filhos. Por fim o método *merge* é chamado e a tarefa termina sua contribuição.

Enquanto fica neste laço, a *task* pode receber pedidos de trabalho advindas de algum filho ou receber resultados parciais de outro. Esses pedidos são interceptados pela *Task Eater* que registra o número de filhos ativos e de resultados recebidos. Desta forma, o programador não precisa lidar com a gerência dos filhos nem com o mecanismo de sincronização necessário para esperar os resultados.

Ao executar o primeiro passo mostrado na figura 5.3, a *Task Eater* pode se deparar com a lista de tarefas vazia. Ela então dispara um roubo. O roubo é de responsabilidade do *Thief Agent* que escolhe aleatoriamente uma das possíveis vítimas existentes e solicita ao respectivo *Victim Agent* uma tarefa com trabalho.

No nodo vítima, o *Victim Agent* pede ao *Task Manager* uma tarefa contendo trabalho. Cabe ao *Task Manager* encontrar uma tarefa e solicitar trabalho ao seu pai (que está

1. Adquira uma *Task* T
2. Cria uma cópia de T e coloque na lista de *Lazy Tasks* local
3. Chame o método `getWork` de T
4. **Enquanto** existir trabalho **Faça**
 5. Chame o método `exec` de T
 6. Armazena tempo gasto
 7. Chame o método `getWork` de T
8. **Fim Enquanto**
9. Retira da lista a tarefa preguiçosa referente a T
10. Espere resultados dos filhos de T
11. Chame o método `merge` de T
12. Volte para o passo 1

Figura 5.3: Pseudocódigo de uma *Task Eater*

executando neste nodo), através do método *getWork*.

Durante a espera por resultados a máquina fica ociosa. Para evitar o desperdício de poder computacional em caso de espera de resultados, é criada automaticamente uma nova *task eater* que passa a executar uma nova *task* roubada de outro nodo.

5.2.3 Camada de Gerenciamento

No protótipo do XgridApp foi implementado um *Node Manager* centralizado devido a sua simplicidade. Uma peculiaridade desta abordagem é que a escolha da vítima passa a ser computada no *Node Manager*. Como afirmado na seção anterior a escolha é randômica.

O seu funcionamento está ilustrado na figura 5.4. No exemplo, o *grid* é formado de apenas duas máquinas (computador 1 e computador 2) que no início da execução publicam no *Node Manager* seus endereços ². Ao longo da execução, o computador 2 fica ocioso e solicita ao *Node Manager* que lhe indique uma vítima (passo (b)). No passo (c), o computador 2 recebe a indicação para roubar do computador 1, e em seguida, ele o faz.

5.3 Estudo de caso: Análise de seções retangulares de concreto

A análise de estruturas de concreto armado ainda é feita com base em valores determinísticos adotados para as propriedades dos materiais e para as dimensões da estrutura. Porém, sempre existe alguma incerteza sobre que valor as propriedades mecânicas dos materiais irão assumir na estrutura, e sobre quais serão as suas características geométricas finais, após a construção. A resposta da estrutura a um determinado carregamento é uma função de várias variáveis aleatórias, que afetam o seu desempenho.

²Devido a implementação dos agentes ser feita em Java RMI, ao invés do endereço de cada nodo é passado um “ponteiro remoto” para os objetos distribuídos.

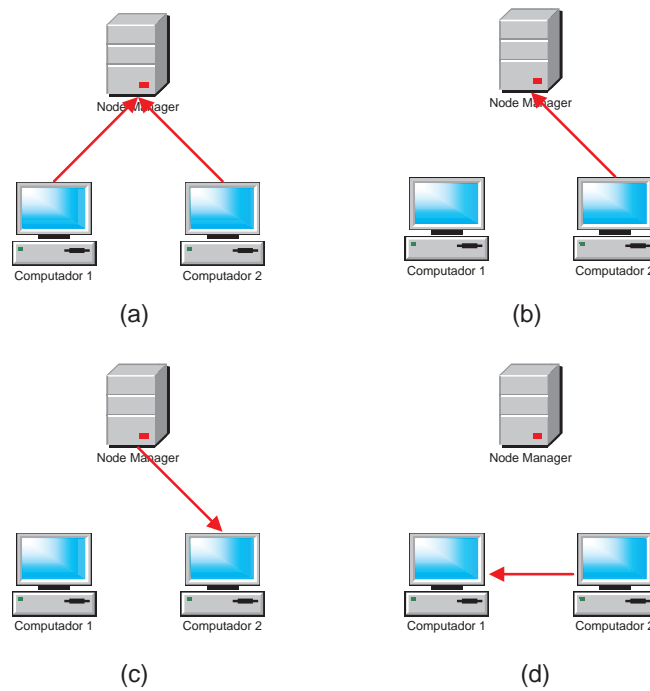


Figura 5.4: Funcionamento da terceira camada

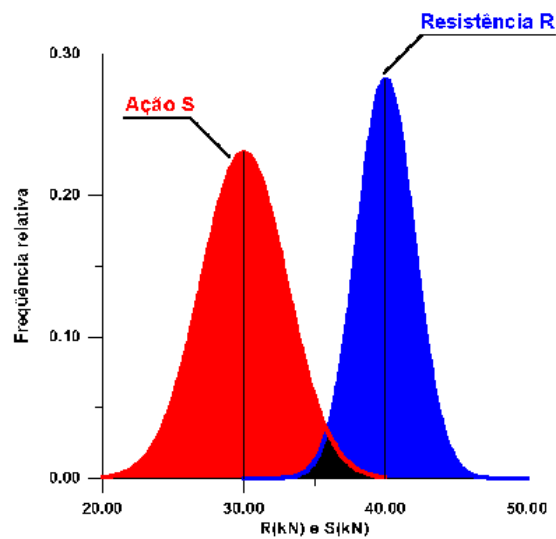


Figura 5.5: A resposta da estrutura (resistência) ao seu carregamento(ação) varia de acordo com a variabilidade das propriedades mecânicas e geométricas da estrutura

A análise probabilística de seções retangulares de concreto armado permite calcular a probabilidade de falha de uma viga ou pilar. Tal cálculo é baseado em simulações usando-se o método de Monte Carlo. O fluxograma da figura 5.6 descreve um algoritmo geral para análise probabilística de estruturas. Cada iteração da estrutura de repetição é independente uma da outra e portanto, o algoritmo é trivialmente paralelizável.



Figura 5.6: Fluxograma geral para análise probabilística de estruturas

Na implementação paralela do algoritmo, o método *getwork* foi programado para dividir o total de iterações em porções de 500 iterações:

$$\{\{1, 2, \dots, 500\}, \{501, 502, \dots, 1000\}, \dots, \{89999500, \dots, 9 \times 10^7\}\}$$

A cada chamada ao método *exec*, a *task* obtém uma dessas porções. O cálculo do *time goal* é feito sobre o tempo despendido para computar uma porção. Na ativação de um filho, ele receberá metade das porções que restam ser calculadas ou então uma porcentagem dessa metade, dependendo do uso ou não do *time goal*.

Nas execuções onde se empregou a técnica de *time goal*, a primeira vez em que um nodo rouba uma tarefa, ele a recebe com apenas uma porção de iterações. Após computá-la, o nodo terá registrado o tempo gasto e assim, em um futuro roubo, poderá receber uma porcentagem da metade, dependendo de seu desempenho. Se atingir o tempo ideal, receberá exatamente a metade do que resta ser feito, porém, se levar mais tempo na execução de uma porção, receberá menos da metade do trabalho.

A seção retangular de concreto armado usado como exemplo nos testes têm as seguintes dimensões: 20 x 40 cm. Foram executadas 9×10^7 iterações.

5.3.1 Execução em ambiente homogêneo

A execução em ambiente homogêneo contou com computadores Athlon XP 2.0 GHz, com 256 MB de memória RAM interligados por um *switch* de 100 Mbs. O sistema

operacional instalado em ambas é Linux. As máquinas são pertencentes ao laboratório de engenharia de computação da FURG.

Ao todo foram efetuadas execuções com 1, 2, 4 e 8 máquinas. Os resultados de cada execução são mostrados nos gráficos das figuras 5.7 a 5.16 e nas tabelas 5.1 a 5.8. Uma análise das tabelas revela que a taxa de erro na escolha da vítima, i.e., quando a vítima escolhida não possui trabalho para ser roubado, ficou entre 4 e 9 %, aproximadamente. Isso mostra que uma implementação simples, com uma política de escolha aleatória, não é adequada mesmo em ambientes com número reduzido de nodos. Em um *grid* real, composto de vários nodos, a taxa de erro tende a ser ainda maior.

Máquina	Contribuição (%)	n^o Tasks	n^o Task Eaters
micro 22 (raiz)	52,283	10	2
micro 23	47,716	2	1
total	100	12	3
tempo	10m44.244s		
taxa de erro(roubo)	0		

Tabela 5.1: Execução em 2 máquinas usando *Time Goal*

Máquina	Contribuição (%)	n^o Tasks	n^o Task Eaters
micro 22 (raiz)	50,997	4	2
micro 23	49,002	1	2
total	100	5	4
tempo	10m39.324s		
taxa de erro(roubo)	0		

Tabela 5.2: Execução em 2 máquinas sem usar *Time Goal*

Máquina	Contribuição (%)	n^o Tasks	n^o Task Eaters
micro 13 (raiz)	25,585	20	3
micro 21	25,448	18	2
micro 23	23,911	2	2
micro 16	25,053	46	2
total	100	86	9
tempo	5m13.762s		
taxa de erro(roubo)	4,03		

Tabela 5.3: Execução em 4 máquinas usando *Time Goal*

A distribuição do trabalho foi bastante homogênea, a contribuição de cada máquina no total do trabalho foi semelhante usando ou não a técnica de *time goal*. Os resultados mostrados nas figuras 5.7, 5.8 e 5.9 refletem essa afirmação, que era esperada. Não houve também diferença significativa na comparação entre a contribuição individual usando ou não *time goal* (figuras 5.10, 5.11 e 5.12).

Máquina	Contribuição (%)	n° Tasks	n° Task Eaters
micro 13 (raiz)	25,502	10	3
micro 21	25,483	10	3
micro 23	23,716	5	2
micro 16	25,298	10	2
total	100	35	10
tempo	5m11.433s		
taxa de erro(roubo)	8,75		

Tabela 5.4: Execução em 4 máquinas sem usar *Time Goal*

Máquina	Contribuição (%)	n° Tasks	n° Task Eaters
micro 13 (raiz)	12,345	9	2
micro 14	12,969	36	3
micro 15	12,548	21	3
micro 19	12,732	56	2
micro 22	12,754	36	2
micro 23	12,689	15	3
micro 21	11,309	46	2
micro 16	12,65	29	2
total	100	248	19
tempo	2m41.776s		
taxa de erro(roubo)	8,96		

Tabela 5.5: Execução em 8 máquinas usando *Time Goal*

Máquina	Contribuição (%)	n° Tasks	n° Task Eaters
micro 13 (raiz)	11,855	7	3
micro 14	12,868	25	3
micro 15	12,574	16	3
micro 19	12,799	10	4
micro 22	12,814	10	3
micro 23	12,572	40	3
micro 21	11,904	10	3
micro 16	12,61	41	3
total	100	159	25
tempo	2m40.377s		
taxa de erro(roubo)	5,22		

Tabela 5.6: Execução em 8 máquinas sem usar *Time Goal*

n° máquinas	tempo	<i>SpeedUp</i>
1 (seqüencial)	19min14s	—
1 (XgridApp)	20m39.823s	—
2	10m44s	1,79
4	5m13s	3,68
8	2m41s	7,16

Tabela 5.7: SpeedUp usando *TimeGoal*

n° máquinas	tempo	<i>SpeedUp</i>
1 (seqüencial)	19min14s	—
1 (XgridApp)	20m39.823s	—
2	10m39s	1,81
4	5m11s	3,71
8	2m40s	7,21

Tabela 5.8: SpeedUp sem usar *TimeGoal*

Se por um lado, não houve diferença na comparação entre a contribuição individual usando ou não *time goal*, não pode-se afirmar o mesmo sobre o número de *tasks* computadas. Como se pode constatar nas figuras 5.13, 5.14 e 5.15 o número de tarefas computadas em cada máquina, assim como o seu total, foi muito maior nas execuções que utilizaram *time goal*. Nessas execuções, na primeira vez que um nodo roubava, ele recebia uma *task* com apenas uma porção de trabalho, almejando-se obter uma estimativa de seu desempenho para um futuro roubo. Isso certamente elevou o número de *tasks* criadas. Além disso, o cálculo do *time goal* é muito sensível ao tempo ideal definido pelo programador. É possível que este tempo tenha sido mal calculado e tenha, por conseguinte, aumentado o número de *tasks* ao longo da execução. Por outro lado, as tabelas revelam que nos testes sem *time goal* o número de *Task Eaters* foi maior, o que significa que ocorreram maior número de espera por resultados dos filhos, indicando um dimensionamento não adequado do tamanho do trabalho.

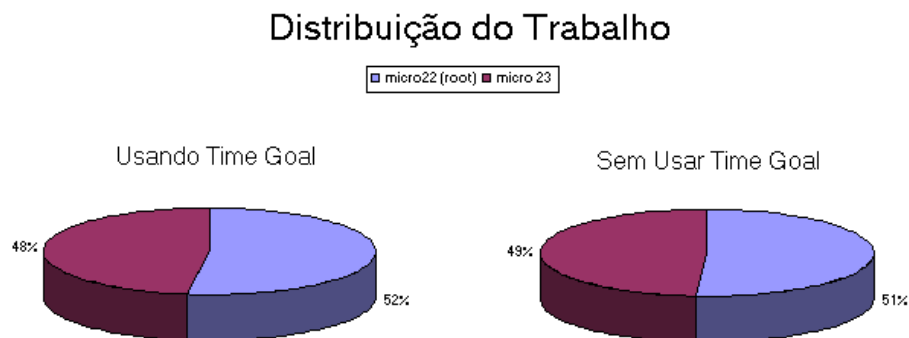


Figura 5.7: Contribuição de cada nodo no cálculo de todo o trabalho, numa execução com 2 máquinas

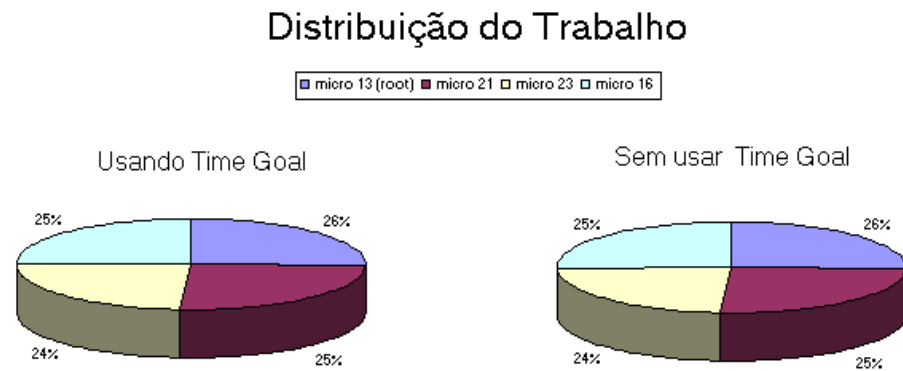


Figura 5.8: Contribuição de cada nodo no cálculo de todo o trabalho, numa execução com 4 máquinas

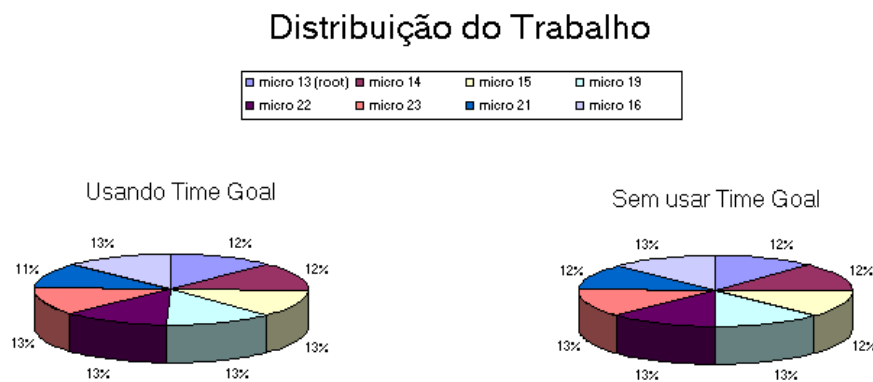


Figura 5.9: Contribuição de cada nodo no cálculo de todo o trabalho, numa execução com 8 máquinas

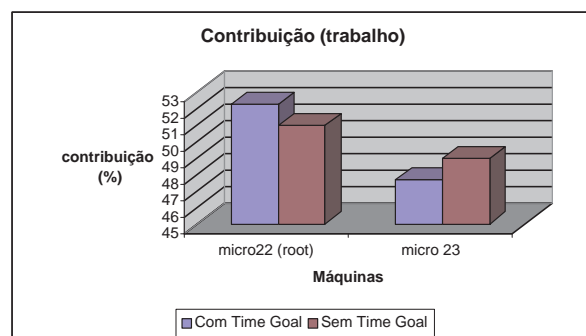


Figura 5.10: Diferença entre a contribuição de cada nodo usando ou não *Time Goal*, numa execução com 2 máquinas

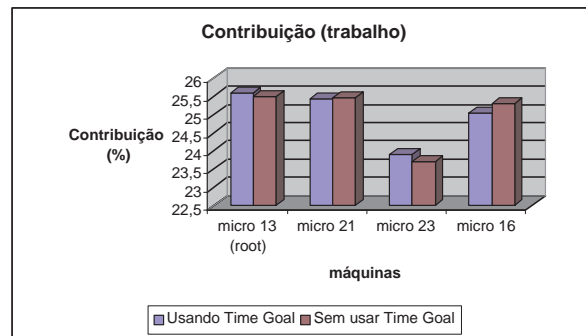


Figura 5.11: Diferença entre a contribuição de cada nodo usando ou não *Time Goal*, numa execução com 4 máquinas

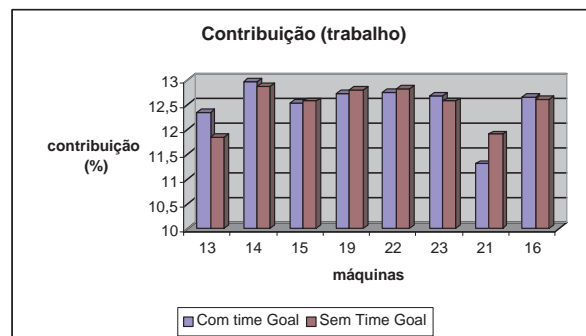


Figura 5.12: Diferença entre a contribuição de cada nodo usando ou não *Time Goal*, numa execução com 8 máquinas

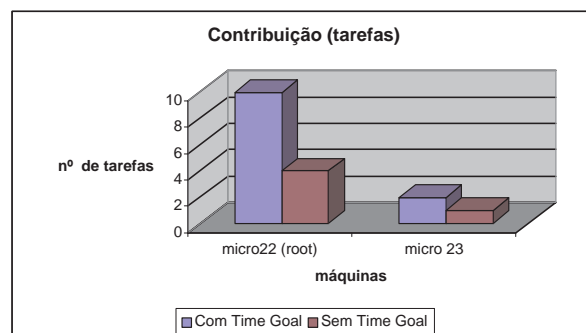


Figura 5.13: Número de tarefas computadas usando ou não *Time Goal*, numa execução com 2 máquinas

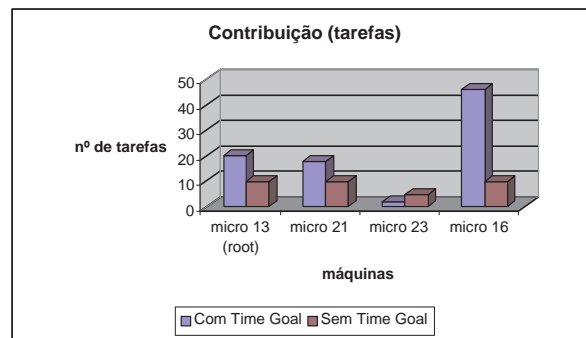


Figura 5.14: Número de tarefas computadas usando ou não *Time Goal*, numa execução com 4 máquinas

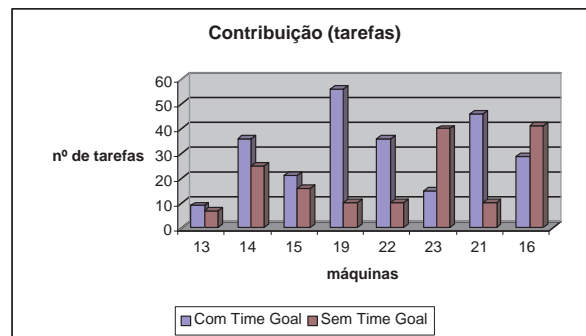


Figura 5.15: Número de tarefas computadas usando ou não *Time Goal*, numa execução com 8 máquinas

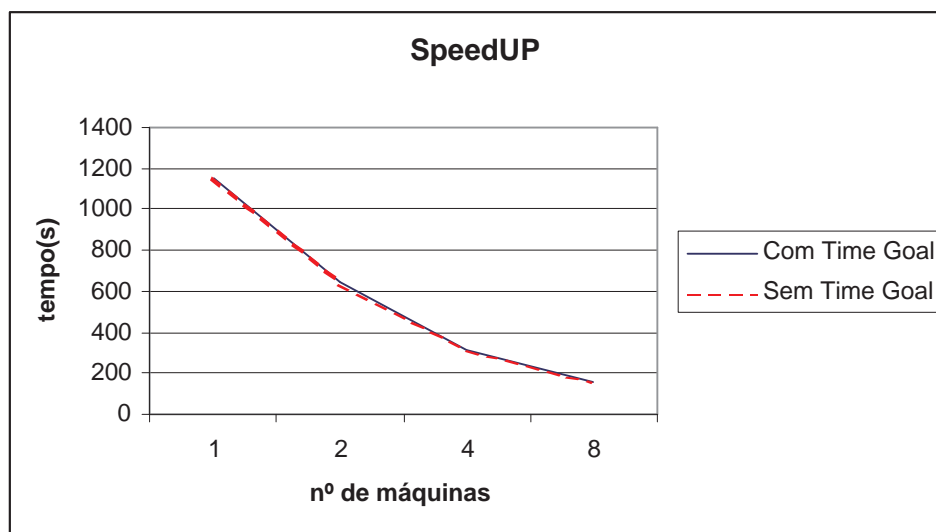


Figura 5.16: *SpeedUp* das execuções em máquinas homogêneas

Observando-se o gráfico da figura 5.16 verifica-se que o *speedup* alcançado foi satisfatoriamente elevado, ficando muito próximo dos valores teóricos. Além disso, os tempos de execução do algoritmo sequencial e do XgridApp em um único nodo foram significativamente próximos, revelando que o custo de gerenciamento do protótipo não tem impacto negativo sobre o desempenho final da aplicação.

6 *Considerações finais*

Grids possibilitam que o termo colaboração seja amplamente inserido no mundo da ciência da computação. A possibilidade de criar organizações virtuais reunindo recursos espalhados pelo globo, utilizar o processamento ocioso dos mesmos e a alta capacidade de processamento disponível são alguns dos aspectos que impulsionaram essa nova área do conhecimento.

Diversos trabalhos a respeito de *grids* já foram realizados em várias partes do mundo. Destes o de maior impacto certamente é Globus. Globus Toolkit objetiva servir de base para a criação de *grids* através de seus serviços, que contemplam vários aspectos como alocação de recursos, segurança, comunicação e transferência de dados. Várias infraestruturas *grid* como Condor e MyGrid empregam algum dos serviços Globus em sua arquitetura.

Numerosos são os desafios e problemas que devem ser resolvidos na concepção de uma infraestrutura *grid* completa. O acesso e autenticação a um ambiente com diferentes domínios administrativos não é uma tarefa trivial. Deve-se garantir a segurança dos recursos contra intrusos e usuários mal intencionados, sem contudo, criar empecilhos a sua utilização. Com um grande nível de heterogeneidade, o balanceamento de carga torna-se vital para atingir um bom aproveitamento dos recursos. A probabilidade de falha, seja na comunicação, distribuição ou execução de aplicações, é bastante alta. Mecanismos tolerantes à falha devem ser providos pela infraestrutura. Uma nova arquitetura necessita também de novas abstrações de programação que facilitem o desenvolvimento de aplicações. Os acordos entre os participantes do *grid* devem ser realizados de forma sofisticada, sem deixar que a burocracia dificulte o compartilhamento.

Dentre os inúmeros desafios que surgem com essa nova tecnologia, este trabalho buscou propor novas abstrações para programação e distribuição de carga eficiente. Foram propostas técnicas de escalonamento baseadas em políticas de *work-stealing* que mostraram-se adequadas ao enfoque de uso de recursos ociosos. Na política de *work-stealing*, quando um nodo fica ocioso ele seleciona um nodo vítima e rouba-lhe tarefas de sua lista local. As tarefas que ficam na lista do escalonador são na verdade tarefas preguiçosas, i.e, não possuem fluxo de execução e não contém trabalho, evitando o desperdício de processamento e espaço de memória.

Em um roubo tipicamente será encaminhado ao ladrão uma tarefa contendo metade do trabalho que resta ser feito pela vítima. O que se imagina é que o ladrão possua a mesma capacidade de processamento que a vítima, recebendo metade do trabalho ambos terminarão ao mesmo tempo. Entretanto, num ambiente heterogêneo, as capacidades podem ser distintas. Propõe-se então a técnica de *time goal*. O programador especifica um tempo ideal ao processamento de uma porção do trabalho. Caso o nodo ladrão, na

execução de uma tarefa anterior, tenha computado uma porção em maior tempo, receberá uma tarefa com poucas porções de trabalho. Já no caso em que tenha computado em menor tempo, receberá tarefas com uma quantidade maior de porções.

No curso deste trabalho foi desenvolvido um protótipo (XgridApp) almejando verificar as potencialidades do modelo em execuções reais. A linguagem de programação empregada no desenvolvimento do protótipo foi Java. O modelo de objetos distribuídos e *threads* nativas à linguagem facilitaram a construção do protótipo. Pretende-se futuramente inserir XgridApp no *middleware* ISAM. A plataforma ISAM passará a realizar a alocação e localização de recursos no *grid*. Em contrapartida, o *middleware* contará com novas abstrações de programação e nova política de escalonamento, além das já existentes.

Como estudo de caso empregou-se o algoritmo de análise probabilística de estruturas de concreto armado, que é trivialmente paralelizável e possui grande utilidade para a sociedade, uma vez que através deste é possível verificar a confiabilidade das estruturas de concreto que nos cercam. Obtiveram-se bons resultados no tocante a *speedup* e utilização de recursos. Os experimentos, contudo, não podem ser considerados conclusivos pois não foram realizados sobre um ambiente altamente heterogêneo e disperso como é um *grid*. Todavia, expressam um bom indicativo das características do protótipo.

6.1 Trabalhos Futuros

Várias são as possíveis contribuições que XgridApp pode receber no futuro. Destaca-se a necessidade de criar mecanismos de detecção e superação de falhas que sejam transparentes ao usuário. A técnica proposta na seção 4.1.1, exige que o programador defina um tempo para cada tarefa realizar sua computação. Cabe também ao programador gerenciar as tarefas, reenviando o trabalho de uma tarefa que excedeu o tempo para outra, em um próximo roubo. Em uma nova implementação, a gerencia pode ser feita pela segunda camada (Camada de *runtime*), cabendo ao programador somente definir o tempo limite.

Acesso e autenticação é outro aspecto crucial para tornar XgridApp aplicável em organizações reais. Restrições de acesso a recursos, limitando seu uso somente ao pessoal autorizado, definindo horários e porcentagens da capacidade que pode ser alocada ao *grid* são imprescindíveis para que organizações queiram compartilhar seus recursos. Pode-se estudar uma integração com o serviço Globus de acesso e autenticação, usando-o como base para mecanismos sofisticados que permitam restrições elaboradas sobre cada recurso.

Poder-se-ia implementar em XgridApp outras políticas de escolha da vítima, empregando heurísticas e monitoramento dos nodos, como também um gerenciamento distribuído dos nodos tal como exemplificado na seção 4.1.3.

Amplios testes devem ser realizados em ambientes altamente heterogêneos e geograficamente distribuídos, usando computadores de várias arquiteturas interligados pela internet. Com os resultados obtidos podem ser detectados pontos fracos que servirão como guia, criando novos focos de pesquisa.

Referências

- [ATH 04] ATHAPASCAN. Athapascan project homepage. **Disponível em** <http://www-id.imag.fr/Logiciels/ath1>, Acessado em maio de 2004.
- [BER 2002] BERSTIS, V. **Redbooks - fundamentals of grid computing**. [S.l.]: IBM, 2002.
- [BLU 94] BLUMOFÉ, D. P. R. Scheduling large-scale parallel computations on networks of workstations. **Third International Symposium on High-Performance Distributed Computing**, 1994.
- [CAV 2004] CAVALHEIRO, G. Princípios da programação concorrente. **Escola Regional de Alto Desempenho (ERAD)**, 2004.
- [CER 04] CERN. Cern project homepage. **Disponível em** <http://public.web.cern.ch/Public/Welcome.html>, Acessado em dezembro de 2004.
- [CIL 04] CILK. Cilk project homepage. **Disponível em** <http://supertech.lcs.mit.edu/cilk>, Acessado em maio de 2004.
- [CIR 2003] CIRNE, W. Grids computacionais: arquiteturas, tecnologias e aplicações. **Escola Regional de Alto Desempenho (ERAD)**, 2003.
- [CIR 04] CIRNE, W. **Mygrid user manual - release 2.0**. [S.l.: s.n.], Acessado em junho de 2004. Disponível em <http://www.mygrid.com.br>.
- [EXE 04] EXEHDA. Projeto exehda. **Disponível em** <http://www.inf.ufrgs.br/~exehda>, Acessado em dezembro de 2004.
- [GAU 99] GAUTIER, F. G. C. R. D. Scheduling parallel programs on non-uniform memory architectures. **Workshop on Parallel Computing for Irregular Applications**, 1999.
- [IBM 04] IBM. The ibm and grid computing homepage. **Disponível em** <http://www.ibm-grid.com>, Acessado em maio de 2004.
- [ISA 04] ISAM. Projeto isam. **Disponível em** <http://www.inf.ufrgs.br/~isam>, Acessado em dezembro de 2004.
- [KES 96] KESSELMAN, I. F. Globus: a metacomputing infrastructure toolkit. **The International Journal of Supercomputer Applications and High Performance Computing**, 1996.
- [LEI 94] LEISERSON, R. B. Scheduling multithreaded computations by work stealing. **35th Annual Symposium on Foundations of Computer Science**, 1994.

- [LIV 2003] LIVNY, D. T. T. Condor and the grid. **Grid Computing : Making the Global Infrastructure a Reality**, 2003.
- [LIV 2002] LIVNY, T. T. W. M. Condor - a distributed job scheduler. **The MIT Press**, v.Beowulf Cluster Computing with Linux, 2002.
- [NAV 2004] NAVAUX, C. D. R. Fundamentos de processamento de alto desempenho. **Escola Regional de Alto Desempenho (ERAD)**, 2004.
- [REA 2002] REAL, R. **Unicluster**: uma proposta de internet computing plataforma para processamento de alto desempenho.
- [REV 2004] REVIRE, J. R. M. D. T. G. K. R. Algorithmes parallèles à grain adaptatif et applications. **Technique el Science Informatiques**, 2004.
- [ROC 98] ROCH, G. C. G. Athapascan-1: parallel programming with asynchronous tasks. **Yale Multithreaded Programming Workshop**, 1998.
- [SAU 2003] SAUVÉ, W. C. B. Grid computing for bag of tasks applications. **Third IFIP Conference on E-Commerce, E-Business and E-Goverment**, 2003.
- [SES 04] SESHADRI, G. **Fundamentals of rmi - ebook**. [S.l.]: jGuru Group, Acessado em agosto de 2004. Livro Eletrônico Disponível em <http://www.jguru.com>.
- [SET 04] SETI. Seti@home homepage. **Disponível em** <http://setiathome.ssl.berkeley.edu/>, Acessado em dezembro de 2004.
- [SIK 2003] SIKORA, Z. **Java - guia prático para programadores**. [S.l.]: Editora Campus, 2003.
- [SIL 2003a] SILVA CARISSINI, S. S. T. S. de Oliveira;Alexandre da. **Sistemas operacionais e programação concorrente**. [S.l.]: Editora Luzzatto, 2003.
- [SIL 2003b] SILVA, L. C. D. **Primitivas para suporte à distribuição de objetos direcionadas à pervasive computing**.
- [SIL 2003] SILVA;IARA AUGUSTIN;CLÁUDIO GEYER, G. F. R. Y. da. Profiles adaptativos para balanceamento de carga no isam. **Workshop em Sistemas Computacionais de Alto Desempenho**, 2003.
- [STA 2003] STALLINGS, W. **Arquitetura e organização de computadores**. [S.l.]: Prentice Hall, 2003.
- [TAN 2003] TANNENBAUM, A. **Sistemas operacionais modernos**. [S.l.]: Editora Makron Books, 2003.
- [TER 04] TERAGRID. Teragrid project. **Disponível em** <http://www.teragrid.org>, Acessado em dezembro de 2004.
- [TUE 2002] TUECKE, I. F. K. M. N. Grid services for distributed system integration. **IEEE - Computer**, 2002.

- [TUE 04] TUECKE, I. F. K. M. N. **The physiology of the grid - an open grid services architecture for distributed systems integration**. Disponível em <http://www.globus.org/research/papers/ogsa.pdf>.
- [TUE 2001] TUECKE, I. F. K. The anatomy of the grid. **Intl J. Supercomputer Applications**, 2001.
- [YAM 2004] YAMIN, A. C. **Arquitetura para um ambiente de execução direcionado às aplicações móveis conscientes contexto em um ambiente de pervasive computing**. 2004. Tese (Doutorado em Ciência da Computação) — Universidade Federal do Rio Grande do Sul.