SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito: 18/06/2001

Assinatura:

Interface Gráfica para um Ambiente de Escalonamento Flexível e Dinâmico

Arion de Campos Junior

Orientadora: *Profa. Dra. Regina Helena Carlucci Santana*

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências de Computação e Matemática Computacional.

USP – São Carlos
Junho/2001

A Comissão Julgadora:

Profa. Dra. Regina Helena Carlucci Santana

Regina H C Santana

Prof. Dr. Odemir Martinez Bruno

A handwritten signature in black ink, appearing to be 'Odemir', written over a horizontal line.

Prof. Dr. Paulo Sérgio Lopes de Souza

A handwritten signature in black ink, appearing to be 'Paulo Sérgio', written over a horizontal line.

DEDICATÓRIA

*Aos meus pais, Lúcia e Arion,
pela dedicação, apoio e por me
confortar em todos os momentos de minha vida*

AGRADECIMENTOS

À prof^a Dr^a Regina Helena C. Santana, pela paciência, acompanhamento e orientação sempre precisa que dedicou a mim e a este trabalho.

Ao prof. Dr. Marcos José Santana, pela colaboração e sugestões neste trabalho.

À minha família e aos meus pais, em especial, que sempre priorizaram a educação de seus filhos.

Aos professores que possibilitaram o afastamento de minhas atividades junto ao Departamento de Informática da UEPG, em especial, os professores Leila, Victor e Furquim e às pessoas que, de certo modo, me substituíram nos afazeres de minhas atividades junto à UEPG.

Às pessoas que me deram apoio e se mobilizaram para que minha liberação fosse efetivada, em especial ao Magnífico Reitor Roberto Merhy, ao Deputado Plauto e seu assessor Larocca.

À Ana Paula, pelo carinho e compreensão e pela colaboração na escrita do *abstract*.

Aos professores Eduardo Marques e Gracinha pela colaboração neste trabalho.

Ao Márcio, pela amizade, pelas críticas (nem sempre bem aceitas, porém úteis) e principalmente, pelo companherismo.

Ao Paulo Sérgio, que sem sua paciência e inestimável ajuda despendida, o desenvolvimento do trabalho não seria possível.

Ao pessoal do ICMC, em especial, Beth, Cabral, Jacques, Laura, Paulinho, Marília, Sônia e funcionários da biblioteca cujo auxílio foi essencial para o desenvolvimento do trabalho.

Aos companheiros e amigos de mestrado: Adenilso, Álvaro, Aletéia, Ana Elisa, Andrezza, Célia, Daniel, Douglas, Eliane, Fernanda, Fernando, Hermes, João Carlos, Jorge, Kalinka, Kleber, Luciano, Luis Fernando, Mara, Mário, Renata, Renato, Reginaldo, Roberta, Rodrigo, Rudinei, Sarita, Simone, Sérgio Viccari (desculpe-me pelo braço), Thiago, Tatiana, Vanderlei e Vera. Em especial, gostaria de agradecer ao Carlos Renato, Omar, Ricardo e Tomás pela amizade sincera e contribuição para o desenvolvimento deste trabalho.

À UEPG, pelo auxílio financeiro.

E sobretudo, a Deus, pela proteção e por me acompanhar em todos os instantes.

SUMÁRIO

1 - INTRODUÇÃO	1
2 - SISTEMAS PARALELOS E DISTRIBUÍDOS.....	5
2.1 – INTRODUÇÃO	5
2.2 – MOTIVAÇÕES E DIFICULDADES INERENTES AO PARALELISMO	8
2.3 – CONCEITOS BÁSICOS RELACIONADOS À COMPUTAÇÃO PARALELA.....	10
2.3.1 – UNIDADES E MEDIDAS PARA MÁQUINAS PARALELAS	10
2.3.2 – TIPOS DE PARALELISMO	11
2.3.2.1 – Considerando o objeto a ser paralelizado	11
2.3.2.2 – Considerando os eventos concorrentes	12
2.3.3 – GRANULOSIDADE.....	13
2.3.4 – <i>SPEEDUP</i> E EFICIÊNCIA	14
2.4 – CLASSIFICAÇÃO DAS ARQUITETURAS.....	15
2.4.1- CLASSIFICAÇÃO DE ARQUITETURAS SEGUNDO MICHAEL J. FLYNN.....	16
2.4.2 – CLASSIFICAÇÃO DE ARQUITETURAS SEGUNDO RALPH DUNCAN	19
2.5 – SISTEMAS DISTRIBUÍDOS	23
2.6 – FERRAMENTAS PARA PROGRAMAÇÃO PARALELA	27
2.7 – PROGRAMAÇÃO PARALELA EM AMBIENTES DE PASSAGEM DE MENSAGENS	31
2.7.1 – MPI.....	32
2.7.2 – PVM	33
2.8 – ESCALONAMENTO DE PROCESSOS	34
2.9 – COMENTÁRIOS FINAIS	39
3 - UM AMBIENTE PARA ESCALONAMENTO DE PROCESSOS	41
3.1 - INTRODUÇÃO	41
3.2 – OBJETIVOS DO AMIGO	43
3.3 – A ESTRUTURA DO AMIGO.....	45
3.3.1 – A CAMADA SUPERIOR DO AMIGO	47
3.3.1.1 – Classes de <i>Software</i>	47
3.3.1.2 – Plataformas Computacionais.....	49
3.3.1.3 – Política de Escalonamento	51
3.3.1.4 – Monitoramento	54
3.3.1.5 - Simulação	55
3.3.2 – A CAMADA INFERIOR DO AMIGO	56
3.4 – COMENTÁRIOS FINAIS	60

4 - INTERFACES GRÁFICAS COM O USUÁRIO	61
4.1 - INTRODUÇÃO	61
4.2 - GUI.....	63
4.3 - O PROJETO DE UMA GUI.....	65
4.3.1 - TCL/TK.....	69
4.3.2 - GTK+.....	70
4.3.3 - QT.....	72
4.3.4 - LESSTIF.....	74
4.4 - COMENTÁRIOS FINAIS	75
5 - FUNCIONALIDADE DA INTERFACE GRÁFICA DO AMIGO	77
5.1 - INTRODUÇÃO	77
5.2 - FUNCIONAMENTO DA PARTE INTERNA DA INTERFACE GRÁFICA	80
5.3 - A FERRAMENTA ADOTADA.....	83
5.4 - COMPONENTES QUE EXECUTAM AS FUNCIONALIDADES.....	84
5.4.1 - <i>CAR.C</i>	85
5.4.2 - <i>CLA.C</i>	86
5.4.3 - <i>HAR.C</i>	87
5.4.4 - <i>BEN.C</i>	88
5.4.5 - <i>MET.C</i>	88
5.4.6 - <i>POL.C</i>	89
5.4.7 - <i>POLCFG.C</i>	91
5.5 - COMENTÁRIOS FINAIS	93
6 - MÓDULO VISUAL DA INTERFACE GRÁFICA DO AMIGO	94
6.1 - INTRODUÇÃO	94
6.2 - A FERRAMENTA ADOTADA.....	95
6.3 - COMPONENTES DO MÓDULO VISUAL	96
6.4 - MENU PRINCIPAL.....	98
6.4.1 - CLASSES DE SOFTWARE.....	99
6.4.2 - PLATAFORMAS COMPUTACIONAIS	103
6.4.3 - POLÍTICAS DE ESCALONAMENTO.....	107
6.4.4 - MONITORAMENTO	115
6.4.5 - SIMULAÇÃO	118
6.5 - COMENTÁRIOS FINAIS	119
7 - CONCLUSÕES E TRABALHOS FUTUROS	120
7.1 - CONSIDERAÇÕES FINAIS E CONCLUSÕES	120
7.2 - CONTRIBUIÇÕES DO TRABALHO	122
7.3 - DIFICULDADES ENCONTRADAS.....	122
7.4 - PROPOSTAS PARA TRABALHOS FUTUROS	123
8 - REFERÊNCIAS BIBLIOGRÁFICAS	125

ÍNDICE DAS FIGURAS

FIGURA 2.1: PARALELISMO DE DADOS	11
FIGURA 2.2: PARALELISMO FUNCIONAL	12
FIGURA 2.3: PARALELISMO DE OBJETOS	12
FIGURA 2.4: AS CURVAS IDEAL E REAL DO <i>SPEEDUP</i> EM UM SISTEMA PARALELO	15
FIGURA 2.5: COMPUTADOR SISD	17
FIGURA 2.6: COMPUTADOR SIMD	18
FIGURA 2.7: COMPUTADOR MISD	18
FIGURA 2.8: COMPUTADOR MIMD.....	19
FIGURA 2.9: CLASSIFICAÇÃO DE DUNDAN DAS ARQUITETURAS PARALELAS	20
FIGURA 2.10: ARQUITETURAS MIMD. A) MEMÓRIA DISTRIBUÍDA; B) MEMÓRIA COMPARTILHADA	22
FIGURA 2.11: ESQUEMA DE UM SISTEMA DISTRIBUÍDO.....	24
FIGURA 2.12: UM SISTEMA DISTRIBUÍDO SEM BALANCEAMENTO DE CARGA (SHIVARATRI ET AL., 1992)	35
FIGURA 2.13: TAXONOMIA DE ESCALONAMENTO DE PROCESSOS (CASAVANT & KUHL, 1988)	37
FIGURA 3.1: A ESTRUTURA INTERNA DO AMIGO (SOUZA, 2000).....	45
FIGURA 3.2: ESTRUTURA DO ARQUIVO DE CONFIGURAÇÃO POLICY.CFG (SOUZA, 2000)	49
FIGURA 3.3: A ESCOLHA DAS POLÍTICAS DA CAMADA SUPERIOR DO AMIGO (SOUZA, 2000)	54
FIGURA 3.4: MÓDULOS DA CAMADA INFERIOR (SOUZA, 2000)	57
FIGURA 3.5: RELACIONAMENTO ENTRE OS MÓDULOS DO AMIGO (SOUZA, 2000)	59
FIGURA 4.1: PROFICIÊNCIA DO USUÁRIO	63
FIGURA 4.2: ATIVIDADES TÍPICAS NO PROCESSO DE DESENVOLVIMENTO DE UMA INTERFACE COM O USUÁRIO.....	67
FIGURA 4.3: A API DO QT.....	73
FIGURA 5.1 – MÓDULOS DA INTERFACE GRÁFICA	78
FIGURA 5.2 – PROCESSO DE FUNCIONAMENTO DA INTERFACE GRÁFICA.....	79
FIGURA 5.3 – EXEMPLO	81
FIGURA 5.4 – CONSOLE DA APLICAÇÃO COM O RETORNO DA FUNCIONALIDADE.....	82
FIGURA 5.5 – ARQUIVO POLICY.CFG (SOUZA, 2000).....	91
FIGURA 6.1 – DIRETÓRIO PADRÃO	97
FIGURA 6.2 – MENU PRINCIPAL	98
FIGURA 6.3 – CADASTRO DE CLASSES	101

FIGURA 6.4 – FLUXO DE INFORMAÇÕES DO CADASTRO DE CLASSES DE SOFTWARE.....	101
FIGURA 6.5 – CADASTRO DE APLICAÇÕES	102
FIGURA 6.6 – FLUXO DE INFORMAÇÕES DO CADASTRO APLICAÇÕES	103
FIGURA 6.7 – CADASTRO DE <i>HARDWARE</i>	104
FIGURA 6.8 – CADASTRO DE <i>BENCHMARKS</i>	106
FIGURA 6.9 – EXECUTA <i>BENCHMARKS</i>	107
FIGURA 6.10 – FRONT END/ BACK END DO ITEM ‘EXECUTA <i>BENCHMARK</i> ’	107
FIGURA 6.11 – CADASTRO DE POLÍTICAS DE ESCALONAMENTO.....	108
FIGURA 6.12 – RELACIONA CLASSES E POLÍTICAS	109
FIGURA 6.13 – FUNCIONAMENTO DA OPÇÃO ‘RELACIONA CLASSES E POLÍTICAS’	110
FIGURA 6.14 – RELACIONAMENTO EFETUADO	110
FIGURA 6.15 – CONFIGURA USO DE POLÍTICAS	112
FIGURA 6.16 – SELECIONA POLÍTICA RESERVA.....	113
FIGURA 6.17 – SELECIONA OUTRAS POLÍTICAS	114
FIGURA 6.18 – RELAÇÃO <i>FRONT END/BACK END</i> DA OPÇÃO ‘SELECIONA POLÍTICAS’	115
FIGURA 6.19 – DEFINIÇÃO DAS POLÍTICAS	115
FIGURA 6.20 – CADASTRO DE MÉTRICAS	117
FIGURA 6.21 – FUNCIONAMENTO DO CADASTRO DE MÉTRICAS	117
FIGURA 6.22 – INÍCIO/ TÉRMINO DO MONITORAMENTO	118

RESUMO

Encontra-se em desenvolvimento no Laboratório de Sistemas Distribuídos e Programação Concorrente do ICMC-USP um ambiente para escalonamento de processos denominado AMIGO (*DynAMical FlexIble SchedulinG EnvirOnment*). Esse ambiente é organizado em duas camadas principais. Uma camada inferior possui os módulos responsáveis pelo escalonamento e pelo relacionamento entre as aplicações dos usuários, as políticas de escalonamento disponíveis e o ambiente de passagem de mensagens. Uma camada superior composta por uma interface gráfica, na qual o usuário tem acesso a várias opções para configurar o ambiente e para configurar o modo que as políticas de escalonamento serão usadas.

Este trabalho tem como objetivo o desenvolvimento da interface gráfica da camada superior do AMIGO, que permite cadastrar classes de software, o hardware utilizado, as políticas de escalonamento disponíveis e as métricas a serem utilizadas para o monitoramento das políticas de escalonamento consideradas pelo AMIGO. A interface gráfica desenvolvida neste trabalho foi projetada considerando dois módulos: *Front End* (desenvolvido em TCL/TK, é responsável por captar os dados definidos pelo usuário e ativar a execução da função solicitada) e *Back End* (desenvolvido em C e responsável por executar as funcionalidades das tarefas definidas pelo usuário).

ABSTRACT

It is being developed in the Laboratory of Distributed Systems and Concurrency Programming of ICMC/USP an environment for process scheduling entitled AMIGO (*DynAMical FlexIble SchedulinG EnvirOnment*). This environment is disposed in two main layers: an inferior layer, composed by modules responsible for scheduling the policies available; and a superior layer composed by a graphical interface, in which the user can access the several options to configure the environment and to configure the way that scheduling policies are going to be used.

This work proposes the development of the graphical interface of the AMIGO superior layer, that allows the registration of software classes, the hardware utilized, the scheduling policies available and the metrics used to monitor the scheduling policies used in AMIGO. The graphical interface considered in this work is projected considering two modules: *Front End* (developed in TCL/TK, it is responsible for capturing the data defined by the user and for activating the execution of the function required) and the *Back End* (developed in C and responsible for executing the features of the tasks defined by the user).

capítulo 1

Introdução

Quando a era dos computadores teve início, em meados de 1945, as máquinas existentes eram muito caras e grandes. Essa situação apresentou pouca alteração até a década de 80, quando dois avanços tecnológicos começaram a reverter essa conjuntura. O primeiro avanço foi o desenvolvimento de microprocessadores com poder de processamento cada vez maior, a um custo relativo cada vez menor. O segundo avanço tecnológico foi a popularização das redes de interconexão, que permitiram a conexão de centenas de máquinas/processadores.

Através dessas tecnologias, hoje em dia é mais fácil construir máquinas paralelas e sistemas computacionais compostos por um grande número de computadores ligados através de redes de alta velocidade. Tais sistemas são denominados sistemas distribuídos, em contraste com os sistemas centralizados, compostos por um único processador, memória, periférico, e terminais (Tanenbaum, 1992).

Os sistemas distribuídos se tornam interessantes para o desenvolvimento de programas paralelos, se comparados aos sistemas verdadeiramente paralelos, quando a relação custo/desempenho é considerada. No contexto deste trabalho, sistemas verdadeiramente paralelos são aquelas máquinas que possuem vários processadores que se comunicam através de uma rede de interconexão e que são utilizadas especificamente para o desenvolvimento de programas paralelos. O fato de essas máquinas serem de uso específico é o que as tornam tão caras e conseqüentemente limitam sua utilização e disseminação.

Considerando que máquinas paralelas e sistemas distribuídos podem ser utilizados para propósitos semelhantes (como o processamento paralelo, por exemplo),

problemas semelhantes também podem ser assinalados. Tanto sistemas distribuídos quanto sistemas verdadeiramente paralelos necessitam de técnicas para balanceamento de carga e em ambos ocorre a degradação de desempenho pelo congestionamento da comunicação (Zaluska, 1991).

É altamente indesejável que em um sistema paralelo, existam alguns processadores com cargas de processamento muito elevadas e outros em estado quase que ocioso. Para conseguir uma distribuição uniforme da carga de trabalho entre os elementos de processamento, e desta forma minimizar o tempo de execução e maximizar a utilização dos recursos do sistema, técnicas de escalonamento devem ser utilizadas (Shivaratri et al., 1992).

Segundo (Rewini et al., 1995), o escalonamento consiste em atribuir processos a processadores e determinar em que ordem estes processos serão executados. Ele é de vital importância para sistemas paralelos e distribuídos, podendo ser considerado um dos problemas mais desafiantes nesta área. Muita pesquisa tem sido feita na área de escalonamento, entretanto, na prática, essas pesquisas não são utilizadas adequadamente. Outro problema relacionado com o escalonamento da carga de trabalho, é que a escolha da política de escalonamento não é trivial, pois os algoritmos propostos são pouco flexíveis e deve-se também considerar a heterogeneidade das plataformas (Souza, 2000). Para amenizar esses problemas foi proposta uma ferramenta para gerenciamento de processos, denominada AMIGO (*dynAMical flexIble schedulinG EnvirOnment*), que está sendo desenvolvida no Grupo de Sistemas Distribuídos e Programação Concorrente do ICMC/USP.

O AMIGO tem por objetivo tornar fácil o uso das políticas de escalonamento e dar “transparência” aos detalhes envolvidos. Não é necessário que o usuário tenha conhecimento profundo das políticas de escalonamento ou precise se preocupar em como aplicá-las. Desta forma, o AMIGO reduz a distância existente entre a teoria sobre balanceamento de carga (largamente pesquisada) e a utilização desta teoria em sistemas reais.

Para alcançar o objetivo definido para o AMIGO é essencial que se tenha um ambiente flexível, dinâmico e de fácil utilização. A flexibilidade é obtida através de um projeto de *software* com objetivos e interfaces bem definidos. O dinamismo é obtido pela possibilidade de alterar políticas de escalonamento e outras características durante sua execução.

A facilidade de utilização requer uma interface amigável com o usuário. O objetivo desta interface é permitir que o usuário informe as características de sua plataforma, de sua aplicação, do tipo de balanceamento pretendido e etc. em um ambiente gráfico sem, no entanto, perder a flexibilidade do ambiente. O projeto e desenvolvimento dessa interface são os objetivos do trabalho de mestrado apresentado nesta dissertação.

Essa interface gráfica permite o cadastro de classes de *software*, o *hardware* utilizado, as políticas de escalonamento disponíveis e as métricas a serem usadas para o monitoramento das políticas de escalonamento usadas no ambiente AMIGO. A partir dessa interface, é possível também definir qual política de escalonamento deve ser usada, e quando e para quais aplicações, determinada política deve ser considerada.

Almejando propiciar modularidade e flexibilidade para a Camada Superior, optou-se por desenvolver os elementos visuais da Interface Gráfica do AMIGO, ou seja, o módulo visual, à parte da funcionalidade oferecida pela mesma. Essa estratégia é explanada nos capítulos 5 e 6 desta dissertação.

Para um melhor entendimento, nesta dissertação, os assuntos propostos são divididos nos seguintes capítulos:

- **capítulo 1:** esta introdução;
- **capítulo 2:** trata os conceitos mais importantes da computação paralela e dos sistemas distribuídos. Também nesse capítulo são apresentados alguns ambientes de passagem de mensagens e a atividade de escalonamento de processos é discutida;
- **capítulo 3:** descreve os principais detalhes e características do AMIGO: uma ferramenta responsável pela gerência do escalonamento de processos;
- **capítulo 4:** apresenta conceitos sobre interfaces com o usuário. Esse capítulo apresenta também características e conceitos relevantes para o desenvolvimento de tais interfaces;
- **capítulo 5:** neste capítulo são enfocados os aspectos relacionados à funcionalidade da Camada Superior;

- **capítulo 6:** descreve o módulo visual da interface gráfica da camada superior do AMIGO;
- **capítulo 7:** apresenta as conclusões e contribuições do trabalho desenvolvido, além de sugestões para trabalhos futuros.

capítulo 2

Sistemas Paralelos e Distribuídos

Neste capítulo serão tratados os Sistemas Paralelos e Distribuídos, bem como a programação concorrente nesses dois ambientes.

O estado atual da computação paralela é discutido em conjunto com as motivações e problemas relacionados aos ambientes paralelos. Também serão tratados aspectos relacionados aos ambientes de passagem de mensagens. Finalizando o capítulo, serão apresentadas algumas noções de escalonamento de processos, uma técnica importante tanto aos sistemas distribuídos quanto sistemas paralelos.

2.1 - INTRODUÇÃO

A história da computação paralela é tão antiga quanto a própria história da computação. Entre os vários fatos que comprovam essa afirmação, os seguintes podem ser citados:

- Charles Babage, que construiu o primeiro computador mecânico por volta de 1850 possuía idéias sobre aritmética paralela e em seus trabalhos literários considerava algum tipo de processamento paralelo;
- por volta de 1920, Lewis F. Richardson propôs que a previsão de tempo poderia ser feita em paralelo por várias pessoas com calculadoras mecânicas sob o comando de um líder (Kuck, 1996);
- ainda em 1920, um pesquisador do MIT (Massachusetts Institute of Technology) chamado Vannevar Bush apresentou um computador analógico capaz de resolver equações diferenciais em paralelo (Navaux, 1989);

- entre 1944 e 1947, G. R. Stibitz e S.B. Willians da Bell Telephone construíram uma máquina paralela com dois processadores chamada MODEL V (Amorim et al., 1988);
- nos anos 50, John V. Newmann faz uma analogia sobre o cérebro humano e os computadores e em seus artigos sugere modelos de computação paralela (Kuck, 1996; Navaux, 1989).

Entre 1960 e 1970 foram desenvolvidos vários projetos de sistemas paralelos. Entretanto, é considerado como marco inicial no surgimento de máquinas para processamento paralelo o desenvolvimento do projeto Illiac IV (Navaux, 1989; Amorin et al., 1988). O Illiac IV foi desenvolvido em 1965 na Universidade de Illinois sob a direção de Dan Slotnick e era formado por 64 processadores de alta velocidade.

Existem várias definições para o termo “*processamento paralelo*”. Duas dessas definições são largamente difundidas. Kay Hwang (Hwang, 1993), formalmente define processamento paralelo como:

“Processamento paralelo é uma forma eficiente do processamento da informação com ênfase na exploração de eventos concorrentes no processo computacional” .

A outra definição, elaborada por George Almasi e Allan Gottlieb (Almasi & Gottlieb, 1994) diz que processamento paralelo é ***“uma coleção de elementos de processamento que podem se comunicar¹ e cooperar para resolver grandes problemas rapidamente”***.

O principal fator que estimulou o aparecimento da computação paralela foi a busca por melhor desempenho (Amorin et al., 1988; Navaux, 1988; Almasi & Gottlieb, 1994). Em certas situações, quando demanda por melhor performance pode rapidamente tornar o equipamento insuficiente para uma tarefa, um único processador pode não ser o mais adequado.

¹ Esses processadores se comunicam entre si para que haja coordenação (sincronização) na execução das diversas tarefas em paralelo.

Computadores cada vez mais rápidos têm surgido desde os anos 40, entretanto, *hardware* cada vez mais rápidos não surgirão para sempre (pelo menos não no ritmo experimentado em décadas passadas) e há sinais que em alguma época, alternativas devam ser encontradas. Isto significa que a arquitetura do computador não é simplesmente um assunto potencialmente útil e interessante, mas o futuro da computação de alto desempenho parece depender especialmente disso (Kuck, 1996).

É nesse contexto que os Sistemas Distribuídos tornam-se atraente. Apesar da motivação principal para o desenvolvimento dos sistemas distribuídos tenha sido o compartilhamento de recursos, rapidamente os projetistas perceberam seu potencial para a computação distribuída.

A computação distribuída é um processo em que um conjunto de computadores conectados por uma rede é usado coletivamente para resolver um grande problema. Cada vez mais organizações possuem redes locais de alta velocidade conectando suas estações de trabalho. Consequentemente, esses recursos computacionais combinados podem exceder a potência de um único computador de alto-desempenho (Geist et al., 1994; Cenapdne, 2000).

O fator mais importante nos sistemas distribuídos é o custo. Ninguém implementaria uma computação distribuída em um sistema distribuído se existissem alternativas mais viáveis (Zaluska, 1991). Grandes processadores maciçamente paralelos (*MPP-Massively Parallel Processors*) custam muito, na ordem de milhões de dólares. Em contrapartida, rodar programas em um conjunto de equipamentos numa rede já existente é muito mais barato (Geist et al., 1994). É claro que essa solução não poderá resolver todos os problemas que uma máquina MPP resolve, entretanto eles serão capazes de resolver problemas maiores e muito mais rapidamente do que se rodassem em apenas uma máquina.

Outro fator que desempenhou importante papel para a proliferação da computação concorrente em sistemas distribuídos foi a eliminação da limitação de memória. Sistemas distribuídos podem obter alta escalabilidade mais facilmente que certas arquiteturas, como por exemplo, as SIMD com memória compartilhada.

Vale ressaltar também que problemas comuns são encontrados tanto em sistemas distribuídos quanto computação concorrente, assim como **balanceamento de carga** e degradação de desempenho pelo congestionamento da comunicação (Zaluska, 1991).

Geralmente tratado como sinônimo de escalonamento de processos, balanceamento de carga é um dos objetivos a ser alcançado pelo escalonamento. O desempenho de um sistema é influenciado pelas suas características de *hardware* e *software*, bem como a carga que ele tem que processar (Calzarossa & Serazzi, 1993).

O escalonamento de processos visa realocar a carga entre os nós do sistema, para atingir um equilíbrio no sistema. Como em um sistema distribuído os elementos de processamento estão unidos, torna-se um objetivo economicamente interessante obter alto desempenho no sistema como um todo (Shivaratri et al., 1992). Não é interessante ter um sistema composto de diversos elementos de processamento, com algumas das máquinas com carga de processamento muito alta e outras em estado quase que ocioso.

2.2 – MOTIVAÇÕES E DIFICULDADES INERENTES AO PARALELISMO

Os sistemas paralelos são utilizados nas mais diversas áreas, principalmente nas que precisam prover serviços de alto desempenho e disponibilidade. Grandes corporações precisam de grande poder computacional de baixo custo, para renderizar gráficos em altas velocidades, prever o clima, fazer simulações, solucionar questões matemáticas complexas, etc. Sistemas paralelos também fornecem alta disponibilidade para empresas que trabalham com serviços de missão crítica que precisam garantir que equipamentos hospitalares ou sistemas públicos de emergência, por exemplo, estejam sempre acessíveis e funcionando.

Dentre as motivações que estimularam o surgimento da computação paralela, destacam-se (Amorin et al., 1988; Skillicorn & Talia, 1998):

- aumento de desempenho. Um dos motivos mais importantes, onde diversos agentes estão operando como uma força-tarefa na execução do programa do usuário;

- o mundo real é inerentemente paralelo, assim é razoável expressar a computação de modo paralelo;
- existem restrições no desempenho da computação seqüencial impostos por limites físicos fundamentais, como a velocidade da luz;
- melhor relação custo/benefício dos sistemas paralelos. A queda no preço dos processadores juntamente com a popularização das redes de computadores possibilita a construção de supercomputadores a um custo mais acessível;
- sistemas paralelos possuem a característica de modularidade. A arquitetura pode ser expandida incrementalmente através da adição ou atualização de módulos;
- tolerância à falhas. Um componente defeituoso de um sistema multiprocessado pode ser substituído enquanto o sistema continua em operação, apesar do desempenho degradado.

Além de vantagens, o uso de arquiteturas paralelas possui também algumas desvantagens. Alguns dos problemas e dificuldades de usar o paralelismo que explica por que suas vantagens não têm (ainda) conduzido a um uso difundido são descritos a seguir (Amorin et al., 1988; Santana et al., 1997; Skillicorn & Talia, 1998):

- a característica não local dos programas paralelos requer uma interligação através de estruturas de comunicação. E o custo de comunicação depende de como a tecnologia é usada e como está implementada;
- a transferência de um programa de um sistema paralelo para outro pode requerer uma grande quantidade de trabalho ou até mesmo a reconstrução do *software*;
- em sistemas paralelos a programação é mais difícil;
- preocupação de se balancear a carga para melhorar a distribuição dos processos entre os vários processadores;
- maior complexidade gerada pela necessidade de comunicação e sincronismo entre os processos;
- maior dificuldade de depuração, pois não existe uma forma eficiente de acompanhar passo-a-passo a alteração das variáveis durante a execução do processamento das diversas tarefas paralelas;

- existem poucas ferramentas e compiladores prontos para paralelização automática que resolvam definitivamente o problema do paralelismo.

2.3 – CONCEITOS BÁSICOS RELACIONADOS À COMPUTAÇÃO PARALELA

Para melhor entendimento dos sistemas paralelos, torna-se necessário a definição de certos conceitos. Os seguintes conceitos são tratados nessa seção: unidades e medidas de desempenho para máquinas paralelas, tipos de paralelismo, granulosidade, linguagens e compiladores paralelos, *speedup* e eficiência.

2.3.1 – Unidades e Medidas para Máquinas Paralelas

As **unidades** de medidas utilizadas em máquinas sequenciais podem ser também utilizadas para medir a velocidade das máquinas paralelas, como por exemplo o MFLOPS (Navaux, 1989). Semelhante ao MIPS (milhões de instruções por segundo), o MFLOPS mede o número de milhões de operações em ponto flutuante em uma máquina. Não existe uma regra definindo quantas instruções perfazem uma operação em ponto flutuante. Dependendo da máquina, uma instrução em ponto flutuante geralmente pode ser executada por duas a cinco instruções (em algumas máquinas, esse número chega a 10). Na área de Inteligência Artificial, uma medida muito comum é a LIPS, que mede o número de inferências lógicas por segundo.

As **medidas** de desempenho podem ser feitas através de *benchmarks* (Navaux, 1989). Dentre os *benchmarks* mais populares pode-se citar o LINPACK do Laboratório de Argonne, o SSC dos Laboratórios Sandia e *benchmarks* mais gerais tais como o *Whetstone* e o *Dhrystone* que são baseados numa coletânea estatística de linguagens. Esses *benchmarks*, por serem gerais não expressam corretamente o poder dos computadores paralelos. Para contornar esse problema, foram desenvolvidos alguns *benchmarks* específicos para computadores paralelos, tais como (Carmona, 1999): **Genesis** (para avaliar programas científicos em sistemas com memória distribuída); **NAS Parallel Benchmarks** (para avaliar sistemas paralelos de alto desempenho que não sejam vetoriais); **Parkbench** (consiste de quatro pacotes distintos, onde todos eles requerem que o PVM ou o MPI estejam adequadamente instalados e configurados).

É difícil projetar bons *benchmarks* sem a devida consideração das várias características da máquina em questão, incluindo tamanho, arquitetura, sistemas operacionais e compiladores. Com estas dependências nos *benchmarks*, observando-se a performance do mesmo *benchmark* em diferentes sistemas, pode-se ter seu valor questionado (Dongarra & Gentzsch, 1993).

2.3.2 – Tipos de Paralelismo

Existem várias maneiras de classificar o paralelismo². Dentre essas várias maneiras, duas se destacam. Uma considera o objeto a ser paralelizado (Blech, 1994; Cenapdne, 2000) e a outra trata dos eventos concorrentes em relação ao número de elementos de processamento, os chamados paralelismo lógico e paralelismo físico (Quinn, 1994).

2.3.2.1 – Considerando o objeto a ser paralelizado

- a) **Paralelismo de Dados:** geralmente aplicado em programas que resolvem sistemas de equações e problemas de elementos finitos. Nesse caso, a mesma instrução é executada, em paralelo, em diferentes instâncias de dados, como esquematizado na figura 2.1.

Figura 2.1: paralelismo de dados

- b) **Paralelismo Funcional:** aplicado em programas dinâmicos e modulares onde cada tarefa será um programa diferente. No paralelismo funcional, diferentes procedimentos ou operações, cada uma com seus próprios dados, podem ser executados em paralelo. O processador executa instruções diferentes que podem ou não operar sobre o mesmo conjunto de dados.

² Embora paralelismo e concorrência sejam tratados muitas vezes como sinônimos, nesse trabalho, paralelismo - ao contrário de concorrência - necessariamente envolverá mais de um elemento de processamento executando tarefas concorrentemente.

Figura 2.2: paralelismo funcional

- c) **Paralelismo de Objetos:** esse modelo se baseia no conceito de objetos distribuídos em uma rede e é o mais recente. Esses objetos são acessados por métodos (funções) em diferentes processadores para uma determinada finalidade.

Figura 2.3: paralelismo de objetos

2.3.2.2 – Considerando os eventos concorrentes

- a) **Paralelismo Lógico:** vários processos são executados alternadamente em um único elemento de processamento, de modo que apenas um está ativo em cada instante (Souza, 1996; Santana et al., 1997).

- b) **Paralelismo Físico:** aqui existem vários elementos de processamento e vários processos são executados ao mesmo tempo³.

Podem existir três tipos de paralelismo físico (Navaux, 1989; Hwang, 1993).

b1) Paralelismo temporal ou de operações sobrepostas (*pipeline*): determinada tarefa é subdividida em uma seqüência de subtarefas, cada uma é executada por um estágio de *hardware* específico que trabalha concorrentemente com os outros estágios do pipeline.

b2) Paralelismo combinado: ocorre quando vários estágios *pipeline* são executados em paralelo (Souza, 1996).

b3) Paralelismo espacial: este tipo de paralelismo pode ser *síncrono* (encontrado, por exemplo na arquitetura matricial) ou *assíncrono* (encontrado na arquitetura multiprocessador). No síncrono, existem várias unidades de processamento supervisionadas por uma unidade de controle, de onde emanam as instruções (Navaux, 1989). No assíncrono, as tarefas são executadas concorrentemente sem nenhum sincronismo. As tarefas executadas pelos processadores não serão as mesmas, nem o momento de iniciar este processamento.

2.3.3 – Granulosidade

Granulosidade, tamanho granular ou nível de paralelismo é uma medida da quantidade de computação envolvida submetida aos processadores (Almasi & Gottlieb, 1994).

De um modo simplificado, a granulosidade pode ser dividida em **fin**a (paralelismo em nível de instruções, com grande quantidade de processadores pequenos e simples); **méd**ia (paralelismo em nível de subrotinas e procedimentos); e **gross**a (paralelismo em nível de programas, com poucos processadores grandes e complexos). O processamento paralelo explora um destes níveis ou uma combinação destes, dependendo da máquina.

³ Caso o número de elementos de processamento seja menor que o número de processos, pode ocorrer ao mesmo tempo o paralelismo lógico e o paralelismo físico, chamado de **paralelismo misto**.

2.3.4 – *Speedup* e Eficiência

Como visto anteriormente, medidas como MIPS, MFLPOS e LIPS podem levar a conclusões errôneas sobre o desempenho de sistemas paralelos se questões como tamanho, arquitetura, sistemas operacionais e compiladores desses sistemas não forem levados em consideração. Por conta disso, torna-se necessária a definição de uma medida de desempenho que meça o ganho de desempenho através da variação do número de processadores.

Sob essas circunstâncias, duas medidas são amplamente aceitas: o *speedup* e a *eficiência*. Segundo Michael Jay Quinn, “Duas importantes medidas da qualidade dos algoritmos paralelos implementados em multiprocessadores e multicomputadores são o *speedup* e a *eficiência*” (Quinn, 1994). Apesar de existirem várias definições na literatura, *speedup* pode ser definido como: a “relação de ganho de desempenho entre um sistema com vários processadores sobre um sistema com um processador” (Kuck, 1996; Almasi & Gottlieb, 1994) ou então, “a relação entre o tempo necessário para o algoritmo seqüencial mais eficiente executar uma computação e o tempo necessário para executar essa computação em uma máquina incorporando paralelismo” (Quinn, 1994). Essa “relação” é dada pela equação:

$$Sp = \frac{T(1)}{T(n)}$$

Onde *Sp* significa o *speedup*; *T*(1) é o tempo de execução em um sistema com um único processador e *T*(*n*) o tempo de execução em um sistema com *n* processadores.

No caso ideal, *SP* deveria ser igual ao número de processadores (ver figura 2.4), porém na prática isso não acontece devido, principalmente, aos seguintes fatores (Kuck, 1996; Almasi & Gottlieb, 1994):

- custo de comunicação entre os processadores;
- Lei de Amdahl: nem todo o código pode ser paralelizado (trecho rigorosamente seqüencial);
- uso de granulosidade inadequada à arquitetura.

Quando o *speedup* excede o número de processadores, diz-se que o *speedup* foi superlinear. Essa é uma situação muito difícil de ocorrer, devido aos fatores já mencionados.

A **eficiência** de um sistema paralelo pode ser definida como uma indicação do grau atual de desempenho alcançado através do *speedup* (Hwang, 1993). Isto é, a relação entre o *speedup* e o seu número de processadores (Almasi & Gottlieb, 1994). A equação de eficiência é a seguinte:

$$EF = \frac{SP}{P}$$

Onde EF é a eficiência, SP é o *speedup* e P é o número de processadores utilizados.

Visto que $1 \leq SP \leq n$, tem-se $(1/n) \leq EF \leq 1$. No caso ideal, ou seja nos casos em que $Sp=P$, tem-se 100 por cento de eficiência, EF teria valor igual a 1.

Figura 2.4: as curvas ideal e real do *speedup* em um sistema paralelo

2.4 - CLASSIFICAÇÃO DAS ARQUITETURAS

Na literatura pode-se encontrar várias maneiras de se classificar as arquiteturas de computadores. A mais amplamente difundida é a de Flynn, que é estudada na seção 2.4.1. Apesar de ser aceita, essa classificação é bastante vaga e imprecisa para classificar as variedades de multiprocessadores e processadores vetoriais existentes (Quinn, 1994; Amorin et al., 1988; Hwang & Briggs, 1984).

Tentando sanar estas falhas, surgiram várias classificações que visaram complementar a classificação de Flynn no que diz respeito às arquiteturas paralelas.

Uma delas, proposta por Händler em 1977 fazia uso de uma notação para diferenciar paralelismo de *pipelining*, nos três níveis de um sistema de computação: unidade de controle, unidade aritmética e nível de controle (Quinn, 1987). Por exemplo, o CRAY-1, que possui uma unidade de controle, 12 unidades funcionais *pipelined* e cada uma possui entre um e 14 estágios e até oito unidades funcionais podem ser encadeadas, poderia ser descrito pelo triplo: $T(\text{Cray-1}) = \{1, 12 \times 8, 64 \times (1 \sim 14)\}$ (Quinn, 1987; Amorin et al., 1988).

Outra, chamada **classificação estrutural** proposta por R. W. Hockney, dividia a classe MIMD em três níveis básicos: sistemas chaveados ou de rede; memória compartilhada ou distribuída; e tipo de rede de chaveamento.

Existem várias outras classificações, como a **funcional**, que dividia as arquiteturas em processamento numérico e processamento simbólico (ex.: máquinas ProLog); a de Gajski que elaborou uma taxonomia segundo as necessidades dos usuários (Amorin, 1988), a classificação dos computadores quanto a relação custo/desempenho (Navaux, 1989), etc.

Uma classificação que engloba a maioria das arquiteturas disponíveis atualmente é a de Duncan, que é apresentada na seção 2.4.2.

2.4.1- Classificação de Arquiteturas segundo Michael J. FLYNN

A taxonomia proposta por Flynn em 1966 (Flynn, 1972), revisada em 1972 (Flynn, 1972) e atualizada em 1996 (Flynn & Kevin, 1996) é um dos métodos mais largamente difundidos para classificar as arquiteturas de computador (Amorin et al., 1988; Almasi & Gottlieb, 1994). Esta classificação parte do conceito de que o processo computacional é a execução de um conjunto de instruções sobre um conjunto de dados.

É uma classificação bastante simples que divide as arquiteturas em quatro categorias conforme a multiplicidade dos fluxos de dados e de instruções. A palavra fluxo é empregada para descrever uma sequência de instruções, ou dados executados num

processador. Portanto, um fluxo de instruções é uma sequência de instruções executadas por uma máquina, enquanto que um fluxo de dados é uma sequência de dados, de entrada e de resultados parciais ou intermediários, utilizados pelo fluxo de instruções.

As quatro categorias⁴ de Flynn são:

a) Single-Instruction stream, Single Data stream (SISD)

Fluxo de Instruções e Dados Únicos:

- *corresponde aos computadores sequenciais convencionais, nos quais só existe uma única unidade de controle que decodifica sequencialmente instruções que operam sobre um conjunto de dados (figura 2.5). Segue o modelo de Von Neumann (Entrada → Processamento → Saída). É a classe que apresenta menor grau de concorrência entre todos os modelos.*

Figura 2.5: Computador SISD

b) Single-Instruction stream, Multiple Data stream (SIMD)

Fluxo de Instruções Único com Vários Fluxos de Dados:

- *corresponde aos processadores matriciais (arrays) paralelos e associativos também chamados de arranjos paralelos e associativos. São vários elementos de processamento supervisionados por uma mesma unidade de controle (figura 2.6). O exemplo clássico de computador paralelo SIMD é o ILLIAC-IV, com cada elemento de processamento recebendo a mesma instrução para ser executada em seus próprios itens de dados.*

⁴ Para as figuras correspondentes à classificação de Flynn, considerar a seguinte legenda: UC: Unidade Controle; EP: Elemento Processador; MEM: Memória; SI: Sequência de Instruções; SD: Sequência de Dados.

Figura 2.6: Computador SIMD

c) Multiple-Instruction stream, Single Data stream (MISD)

Múltiplos Fluxos de Instruções para um Único Fluxo de Dados:

- *corresponde a uma máquina que possui vários elementos de processamento recebendo instruções distintas das unidades de controle, porém processando o mesmo fluxo de dados (figura 2.7). Segundo Flynn, as antigas máquinas com placas de ligação (plugboard) foram formas decadentes de arquiteturas MISD, onde um fluxo de dados (SD) passava pelo passo i de um programa para o passo $i+1$ de outro (MI). Para alguns autores, os processadores vetoriais pipeline (Hayes, 1988; Hwang, 1993; Santana et al., 1997; Cenapdne, 2000) e processadores matriciais sistólicos (Quinn, 1994) são exemplos dessa arquitetura. Para outros, na prática nenhuma máquina se enquadra nessa arquitetura (Amorin et al., 1988; Navaux, 1989).*

Figura 2.7: Computador MISD

d) Multiple-Instruction stream, Multiple Data stream (MIMD)

Múltiplos Fluxos de Instruções e Múltiplos Fluxos de Dados:

- Nessa categoria se enquadram a maioria dos multiprocessadores. Cada processador executa suas instruções independente dos outros processadores (figura 2.8). Os sistemas distribuídos se enquadram nessa categoria.

Figura 2.8: Computador MIMD

2.4.2 – Classificação de Arquiteturas segundo Ralph DUNCAN

Em sua taxonomia, Duncan (Duncan, 1990) satisfaz os seguintes conjuntos de proposições, visando criar uma classificação precisa e ampla para as arquiteturas paralelas:

- excluir arquiteturas que incorporam somente mecanismos de paralelismo de baixo nível;
- englobar a classificação de Flynn, baseada nos fluxos de instruções e dados;
- incluir processadores vetoriais e outras arquiteturas que merecem ser incluídas como arquiteturas paralelas, mas que são difíceis de se acomodar ao esquema de Flynn.

Através da figura 2.9, pode-se observar esquematicamente a classificação proposta por Duncan. No decorrer dessa seção, será tratado cada item dessa taxonomia.

Figura 2.9: Classificação de Dundan das Arquiteturas Paralelas

a) Arquiteturas Paralelas Síncronas:

As Arquiteturas Paralelas Síncronas coordenam as operações concorrentes através de relógios globais, unidades de controle central ou controladores de unidades vetoriais. Enquadram-se nessa classificação os processadores vetoriais, as máquinas SIMD e as arquiteturas sistólicas.

a1) processadores vetoriais: são caracterizados pelas múltiplas unidades funcionais *pipeline*, que implementam operações lógicas (“*booleanas*”) e aritméticas escalares e vetoriais que podem operar concorrentemente. Nessas arquiteturas, o vetor de elementos “flui” através das unidades *pipeline* de saída para as unidades *pipeline* de entrada (processo conhecido como “encadeamento”).

a2) SIMD: arquiteturas SIMD empregam uma unidade de controle central, vários processadores e uma rede de interconexão para a comunicação entre os processadores e entre o processador e a memória. A unidade de controle difunde uma única instrução para todos os processadores e cada um executa as instruções em seus dados locais. Essa categoria pode ainda ser dividida em **processadores matriciais** e **processadores com memória associativa**:

a3) arquiteturas sistólicas: máquinas cujo objetivo é o equilíbrio entre computações intensivas e a largura de banda de entrada/saída disponível. Arquiteturas sistólicas são multiprocessadores *pipeline* nos quais os dados fluem de maneira ritmada da memória através de uma rede de processadores. Um relógio global sincroniza este fluxo de dados *pipeline*. Durante cada intervalo de tempo, esses processadores executam uma curta e invariante seqüência de instruções.

b) Arquiteturas Paralelas Assíncronas:

Nesse tipo de arquitetura, os processadores são independentes entre si e não há um elemento de *hardware* responsável pelo controle centralizado das atividades. As arquiteturas assíncronas são compostas basicamente de máquinas MIMD que, por sua vez, podem ser convencionais ou não (Duncan, 1990).

b1) Arquiteturas MIMD convencionais: arquiteturas MIMD empregam múltiplos processadores que podem executar diferentes fluxos de instruções, usando dados locais. Assim, computadores MIMD suportam soluções em processadores que operam de modo autônomo. Embora os processos executados em arquiteturas MIMD sejam sincronizados por passagem de mensagens através de uma rede de interconexão ou pelo acesso de dados em unidades de memória compartilhada, arquiteturas MIMD são computadores assíncronos, caracterizados pelo controle de *hardware* descentralizado.

Quanto à localidade da memória, as arquiteturas MIMD podem ser organizadas em memória distribuída e memória centralizada:

- memória distribuída: arquiteturas de memória distribuída consistem de vários nós de processamento⁵ conectados por uma rede de interconexão, como esquematizado na figura 2.10a. Os dados são compartilhados por troca de mensagens através da rede de interconexão.
- memória compartilhada: nessas arquiteturas, a coordenação entre os processadores é realizada através de uma memória global e compartilhada que pode ser acessada pelos processadores (esquema ilustrado na figura 2.10b). Computadores com memória compartilhada não têm os mesmos problemas encontrados nas arquiteturas que usam a troca de mensagem, como latência, por exemplo. Entretanto, outros problemas, tais como sincronização de acesso aos dados e coerência de cache devem ser resolvidos.

Figura 2.10: Arquiteturas MIMD. a) memória distribuída; b) memória compartilhada

b2) Arquiteturas MIMD não convencionais: são arquiteturas também conhecidas como **Paradigmas Arquiteturais Baseados em MIMD**, que possuem os princípios das arquiteturas MIMD de operação assíncrona e manipulação concorrente dos múltiplos fluxos de instruções e dados. Entretanto essas arquiteturas são baseadas em princípios de organização distintos. As seguintes arquiteturas fazem parte dessa classe:

- Arquiteturas MIMD/SIMD: são arquiteturas híbridas, onde porções de uma arquitetura MIMD podem ser controladas de modo SIMD

⁵ Consistindo de um processador autônomo com sua própria memória local.

- Arquiteturas de Fluxo de Dados (Dataflow): a característica fundamental das arquiteturas *Fluxo de Dados* é um paradigma de execução no qual as instruções são disponibilizadas para a execução tão logo todos seus operandos tornem-se disponíveis.
- Arquiteturas de Redução: arquiteturas de redução ou dirigidas à demanda implementam um paradigma de execução no qual uma instrução é habilitada para execução quando seus resultados são requeridos como operandos para a execução de outra instrução.
- Arquiteturas Matriciais de Frente de Onda: esse tipo de arquitetura combina o *pipeline* de dados da arquitetura sistólica com o paradigma de execução assíncrono da arquitetura *Dataflow*.

2.5 – SISTEMAS DISTRIBUÍDOS

Na literatura podem ser encontradas várias definições para “Sistemas Distribuídos”. Coulouris o define como “*uma coleção de computadores autônomos unidos por uma rede, com software projetado para produzir um recurso computacional integrado*”(Coulouris et al., 1994). Para Tanenbaum, “*um sistema distribuído é aquele que roda em um conjunto de máquinas sem memória compartilhada, máquinas estas que mesmo assim aparecem como um único computador para seus usuários*” (Tanenbaum, 1992). A figura 2.11 ilustra esquematicamente um Sistema Distribuído.

Independente da definição, os seguintes avanços tecnológicos tiveram grande influência no desenvolvimento dos Sistemas Distribuídos (Tanenbaum, 1992; Mullender, 1993; Coulouris, 1994):

- criação das primeiras estações de trabalho mono usuário;
- o desenvolvimento de microprocessadores com poder de processamento cada vez maior;
- o desenvolvimento da *Ethernet* que possibilitou a criação das redes locais de alta velocidade (LAN - *Local Area Network*);
- criação de servidores de impressão e de arquivos;

- desenvolvimento de vários sistemas distribuídos experimentais, como o *Cambridge Distributed Computing System (CDCS)* em 1979, por exemplo.

Figura 2.11: Esquema de um Sistema Distribuído

A literatura apresenta várias características para os Sistemas Distribuídos. Entretanto, algumas se destacam e são amplamente aceitas (Tanenbaum, 1992; Mullender, 1993; Coulouris, 1994):

- **múltiplos computadores (nós):** os sistemas distribuídos possuem múltiplos computadores independentes (e não apenas múltiplos processadores em um mesmo computador que é domínio da computação paralela). Estes nós podem variar desde computadores pessoais até estações de trabalho de alto desempenho.
- **compartilhamento de recursos:** a razão mais comum para se conectar um conjunto de computadores para operar como um sistema distribuído é permitir que eles compartilhem recursos físicos e computacionais. Várias

são as aplicações para os recursos compartilhados, como acesso a sistemas de arquivos contendo base de dados, programas, documentação, etc. Dispositivos de *hardware* como impressoras, discos de alta capacidade e outros periféricos são compartilhados para reduzir custos.

- **abertura:** é a característica que determina se um sistema pode ser estendido de várias maneiras. Isso provê ao sistema a habilidade para evoluir e se adaptar à realidade em que reside. Um sistema pode ser aberto ou fechado no que concerne ao:
 - a) *software*: adição de serviços de gerenciamento de recursos, protocolos de comunicação, etc;
 - b) *hardware*: adição de periféricos, memórias, dispositivos de comunicação, etc.
- **escalabilidade:** um sistema distribuído pode variar desde duas estações de trabalho e um servidor de arquivos a várias centenas de máquinas em uma LAN com vários servidores de arquivo, impressão e de propósito específico. Frequentemente essas LANs são interconectadas unindo milhares de computadores em um sistema distribuído.
- **controle descentralizado:** não há um único computador necessariamente responsável pela configuração, administração ou controle de políticas para o sistema.
- **tolerância à falhas:** a tolerância à falhas em sistemas distribuídos é obtido de duas maneiras:
 - a) redundância de *hardware*: através do uso de componentes redundantes;
 - b) recuperação de *software*: projeto de programas capazes de se recuperar de falhas.
- **troca de mensagens:** os *software* nos diferentes computadores se comunicam via passagem de mensagens estruturadas, construídas sobre algum protocolo de rede (como o TCP/IP, por exemplo) rodando em

qualquer tecnologia de comunicação (como ethernet ou modem). Essas mensagens podem ser comandos, pedidos de serviços, notificação de eventos, conteúdos de arquivos e até mesmo programas inteiros.

- **transparência:** considerada uma das características mais importantes dos sistemas distribuídos. Todo o conjunto de máquinas é visto por seus usuários como se fosse simplesmente um único sistema de tempo compartilhado. Segundo Coulouris (Coulouris et al., 1994), podem existir vários níveis de transparência, tais como transparência de acesso, transparência de localização, transparência de concorrência, transparência de replicação, etc.

Os sistemas distribuídos apresentam várias vantagens em relação aos sistemas centralizados. A economia propiciada por estes sistemas é um ponto fundamental. Com a constante evolução dos microprocessadores, estes oferecem uma relação custo/benefício melhor do que os *mainframes*, pois os microprocessadores estão cada vez mais velozes e acessíveis a uma gama maior de usuários.

Outras vantagens dos sistemas distribuídos podem ser citadas (Tanenbaum, 1992):

- **velocidade:** sistemas distribuídos podem ter poder de processamento equivalente e muitas vezes superior ao *mainframe*;
- **confiabilidade:** mesmo que uma máquina saia do ar, o sistema se mantém operante;
- **escalabilidade:** o poder computacional pode crescer gradativamente através da inclusão de novos nós ou atualizações de componentes específicos;
- **compartilhamento:** vários usuários podem acessar os mesmos periféricos e dados concomitantemente;
- **flexibilidade:** a carga de trabalho pode ser dividida entre os elementos de processamento;

Apesar das vantagens mencionadas acima, é prudente antes de decidir pela implementação de um sistema distribuído, considerar alguns aspectos:

- **software:** deve-se verificar quais sistemas operacionais, linguagens de programação e aplicações estão disponíveis para necessidades específicas;
- **segurança:** apesar das vantagens do compartilhamento de recursos, isso possibilita que os dados secretos tornem-se acessíveis mais facilmente. Os servidores devem ser capazes de verificar a identidade do cliente;
- **rede de interconexão:** deve existir uma preocupação em manter a rede não saturada. Se isso ocorrer, o desempenho degradará e tornar-se-á necessário a substituição da rede ou até mesmo a adição de uma rede complementar.

No que concerne ao desenvolvimento de programas concorrentes, existem várias técnicas e métodos para esse propósito que são tratados na seção 2.6.

2.6 – FERRAMENTAS PARA PROGRAMAÇÃO PARALELA

Atualmente, podem ser encontradas várias maneiras de se desenvolver um programa concorrente, cuja função é explorar os recursos paralelos disponíveis em um dado sistema. Um programa concorrente especifica dois ou mais programas seqüenciais⁶ que podem ser executados concorrentemente como processos paralelos.

Existem pelo menos três maneiras de se desenvolver um algoritmo paralelo para resolver um problema (Quinn, 1994):

- *através da exploração de qualquer paralelismo inerente que exista em um algoritmo seqüencial:* embora em alguns casos não seja tão óbvio tornar paralelo um algoritmo seqüencial, esta solução pode ser viável, pois para a maioria dos problemas já existe uma solução seqüencial;
- *através da criação de um novo algoritmo paralelo:* em algumas situações é melhor desenvolver o algoritmo paralelo a partir do zero;
- *pela adoção de qualquer algoritmo paralelo existente que resolva um problema similar:* o estudo de algoritmos paralelos pode resultar no desenvolvimento de técnicas que podem ser aplicadas para resolver uma grande variedade de problemas.

O termo programação concorrente é utilizado para designar os casos em que várias computações iniciaram e ainda não terminaram. Um sistema que tenha apenas um processador pode se beneficiar da programação concorrente através da **multiprogramação**; que disponibiliza o processador durante um intervalo de tempo para cada processo em execução. Sistemas que possuem mais de um processador podem executar programas paralelamente, isto é, mais de um processo pode ser executado ao mesmo tempo. Nesses ambientes, um algoritmo paralelo pode ser construído através do **multiprocessamento**: se dois processadores compartilham uma memória comum; ou através do **processamento distribuído**: se os processadores são interligados através de uma rede de comunicação. Um algoritmo paralelo pode também ser desenvolvido de forma híbrida. Por exemplo: os processadores de um sistema distribuído freqüentemente são multiprogramados.

O desenvolvimento de programas paralelos implica basicamente em decompor o algoritmo e/ou dados; distribuir as tarefas e os dados entre os vários processadores que trabalham simultaneamente; e coordenar os processos em execução e a comunicação entre os processadores (Cenapdne, 2000). A implementação de um programa paralelo pode seguir os seguintes enfoques:

- *paralelização automática*: um compilador de paralelização transforma um programa serial em um programa paralelo. O programador não se preocupa em modificar o programa ou aprender uma nova linguagem. A paralelização automática exige o mínimo do programador, entretanto o aumento de desempenho pode ser modesto.
- *extensões paralelas para linguagens imperativas*: apesar de exigir mais trabalho do programador que a paralelização automática, este método elimina a necessidade de se aprender completamente uma nova linguagem ou de modificar completamente o programa original. O programador deve apenas aprender uma extensão de uma linguagem já familiar. Podem ser citados como exemplos: o *Express*, *PVM* e *Linda*. Este método é o mais difundido para desenvolver algoritmos paralelos, entretanto o *speedup*

⁶ Um programa seqüencial determina uma execução seqüencial de uma lista de declarações.

obtido pode ser insignificante caso o algoritmo original seja estritamente seqüencial.

- *linguagens não imperativas*: nesse modelo, o programador deve aprender uma nova linguagem desvinculando-se do modelo computacional de Von Neumann e criar o programa nessa nova linguagem. Apesar de possuir muito estudo, essa área ainda não é difundida na prática. A linguagem *Sisal* é um exemplo de linguagem não imperativa.

Os processos executados em paralelo devem se comunicar e sincronizar (Quinn, 1994; Almasi & Gottlieb, 1994), permitindo que um processo influencie na execução de outro. A comunicação entre processos depende da arquitetura de memória utilizada no ambiente.

Em um sistema com memória compartilhada, a comunicação interprocessos é baseada no uso de variáveis compartilhadas (variáveis que podem ser referenciadas por mais de um processador). Em sistemas com memória distribuída, a comunicação se dá através da troca de mensagens. Nessas arquiteturas, os processadores operam independentemente e cada um possui sua própria memória. Podem ser observadas algumas vantagens nessas arquiteturas, como a não existência de um limite para o número de processadores e acesso à memória local sem interferência. Entretanto, algumas desvantagens também se destacam, como sobrecarga nos meios de comunicação e sobrecarga para o programador, que deve tratar do envio e recebimento das mensagens.

A sincronização tem por objetivo assegurar que dois ou mais processos estejam coordenados em tempo ou ação.

A troca de mensagem representa um ponto de sincronização entre os dois processos. Os seguintes métodos são utilizados para obter a sincronização em ambientes de passagem de mensagem (Tanenbaum, 1992; Hwang, 1993; Almasi & Gottlieb, 1994; Santana et al., 1997):

- ***RPC (Remote Procedure Call)***: a chamada de procedimento remoto integra o mecanismo de solicitação e resposta dos clientes aos servidores com a tradicional programação procedimental, permitindo que clientes se

comuniquem com os servidores através de chamadas a procedimentos semelhante ao convencional uso das chamadas a procedimentos das linguagens de alto nível;

- *rendezvous*: nessa estratégia, se o processo transmissor for executado antes do receptor, aquele é bloqueado até que este esteja disponível para receber a mensagem. O mesmo ocorre quando o processo receptor é executado antes do processo transmissor;
- *ponto a ponto*: a comunicação ponto-a-ponto é caracterizada por quatro aspectos básicos, que são:
 - *bloqueio ou não de uma rotina* → rotinas não bloqueantes possibilitam sobrepor a transmissão (ou a recepção) de uma mensagem, conseguindo maior desempenho em determinadas aplicações e evitando deadlocks⁷. Por outro lado, rotinas bloqueantes retornam apenas após o seu término, possibilitando maior segurança;
 - *modo de comunicação* → define diferentes semânticas para uma operação de comunicação ponto-a-ponto. Exemplos de semânticas de comunicação são: síncrona, onde o receptor confirma o recebimento da mensagem, e bufferizada, onde a comunicação é implementada através de buffers;
 - *persistência de requisições* → é comum que uma rotina de comunicação se repita várias vezes em um mesmo programa paralelo, de maneira que todos os seus parâmetros mantenham-se os mesmos, variando apenas o conteúdo da mensagem. Uma requisição persistente de comunicação tem o objetivo de evitar a sobrecarga inicial e constante das consecutivas chamadas a uma mesma rotina, permitindo que os procedimentos iniciais relacionados a uma operação de comunicação sejam executados apenas quando a requisição persistente é criada (Snir et al., 1996).
 - *sentido duplo de comunicação* → é relativamente comum em algoritmos paralelos a ocorrência de um padrão de comunicação onde dois ou mais processos trocam mensagens através da execução de

⁷ *Deadlocks* ocorrem quando os processos estão em um estado de espera por eventos que jamais ocorrerão.

operações de transmissão e recepção de mensagens em rotinas consecutivas (comunicação pingpong).

2.7 – PROGRAMAÇÃO PARALELA EM AMBIENTES DE PASSAGEM DE MENSAGENS

Em ambientes de passagem de mensagens, a comunicação se dá através do envio e recebimento de mensagens através da rede, seguindo regras do protocolo de comunicação entre vários processadores que possuem memória própria. Juntas, as primitivas *send/receive* executam ações de passagem de mensagens entre um par de processos (Coulouris et al., 1994). Nesses ambientes, o programador é responsável pela sincronização dos processos.

A programação em ambientes de passagem de mensagens se dá através do uso de bibliotecas de comunicações que são extensões das linguagens convencionais como o C ou Fortran. Inicialmente essas bibliotecas foram desenvolvidas para uso com processamento maciçamente paralelo (MPP) como uma tentativa de se padronizar um ambiente para possibilitar a portabilidade entre os sistemas de vários fabricantes. A idéia era definir um conjunto de funções independente da máquina que está sendo utilizada e implementá-las em várias plataformas de *hardware* (Geist et al., 1994; Santana et al., 1997).

Ao contrário do MPP, os sistemas distribuídos fornecem um ambiente heterogêneo onde podem variar a arquitetura; o formato de dados; a potência computacional; a carga de trabalho em cada máquina e a carga de trabalho nos meios de comunicação. Apesar das dificuldades causadas pela heterogeneidade dos sistemas distribuídos, a computação distribuída pode oferecer várias vantagens (Geist et al., 1994):

- pelo uso de *hardware* já existente, o custo dessa computação pode ser baixo;
- o desempenho pode ser melhorado pela atribuição de determinada computação a uma arquitetura específica;
- através da exploração da diversidade do ambiente, pode-se tirar proveito de bases de dados diferentes, por exemplo;
- a máquina virtual é escalável e pode crescer conforme novas tecnologias surjam;

- ➡ fornece uma computação segura, por conta da estabilidade das estações de trabalho;
- ➡ facilita o trabalho em grupo;
- ➡ um ambiente tolerante à falhas pode ser criado.

Os fatores mencionados contribuem para diminuir o tempo de desenvolvimento e depuração de programas paralelos em ambientes distribuídos.

Existem vários grupos de pesquisa desenvolvendo bibliotecas para ajudar programadores a usar a computação distribuída. Embora existam vários outros sistemas com capacidade similar, as mais conhecidas são: P4, Express, as implementações da especificação MPI, e PVM (Hwang, 1993; Almasi & Gottlieb, 1994; Geist et al., 1994; Santana et al., 1997).

Dentre os ambientes de passagem de mensagens, os mais utilizadas são MPI e PVM e serão detalhados nas próximas seções.

2.7.1 – MPI

O MPI (Message Passing Interface) é um padrão para *software* para programação paralela em multicomputadores, ou seja, em ambientes com memória distribuída. Trata-se de uma biblioteca de rotinas de comunicação e outros utilitários (MPI, 1995).

Uma questão que deve ser considerada é que o MPI não é uma implementação e sim uma especificação padrão. Uma especificação descreve o que deve ser feito, mas não como fazê-lo. O órgão responsável pela definição da sintaxe e da semântica do padrão MPI é o **MPI Forum**. Os principais objetivos desse comitê são a padronização para a maioria dos fabricantes de *hardware* e plataformas portáteis.

O MPI não oferece suporte para tolerância a falhas e assume que o meio de comunicação é confiável. Um conjunto de rotinas responsáveis pela comunicação ponto-a-ponto entre os pares de processos forma o núcleo do MPI. São implementadas rotinas bloqueantes e não bloqueantes para enviar e receber mensagens (Santana et al., 1997).

O Forum MPI definiu as seguintes características do MPI:

- facilidade: definir uma interface semelhante às já existentes PVM, Express, P4, etc. e acrescentar algumas extensões que permitam maior flexibilidade;
- eficiência: projetado para executar com eficiência em máquinas diferentes;
- portabilidade: compatível para sistemas de memória distribuída;
- transparência: permite a portabilidade para sistemas e implementações heterogêneas, sem mudanças significativas;
- segurança: fornece uma interface de comunicação confiável. O usuário não precisa se preocupar com falhas de comunicação;
- escalabilidade: vários níveis de escalabilidade. Por exemplo, uma aplicação pode criar subgrupos de processos que permitam operações de comunicação coletiva para melhorar o alcance dos processos.

2.7.2 – PVM

O PVM (Parallel Virtual Machine) é uma consequência dos avanços nas pesquisas em computação em redes heterogêneas. O PVM é um conjunto de ferramentas e bibliotecas de *software* integrados que emulam um ambiente de computação de propósito geral, flexível e concorrente em computadores interconectados de arquiteturas variadas (Geist et al., 1994).

O objetivo principal do sistema PVM é permitir que uma coleção de computadores com arquiteturas diferentes possam trabalhar em cooperação formando uma única máquina paralela virtual para que possa ser usada para computação concorrente e paralela. O PVM é baseado nos seguintes princípios:

- conjunto de computadores configurado pelo usuário: as tarefas computacionais das aplicações executam em um conjunto de máquinas que são selecionadas pelo usuário;
- acesso transparente ao hardware: um programa pode explorar os recursos de uma máquina específica ou de um conjunto delas;
- computação baseada em processos: a unidade de paralelismo do PVM é uma tarefa;
- modelo de passagem de mensagens explícito: coleções de tarefas computacionais cooperam pelo explícito envio e recebimento de mensagens

de uma para outra. O tamanho da mensagem é limitado pela quantidade de memória disponível;

- heterogeneidade: o sistema PVM suporta heterogeneidade em termos de máquinas, redes e aplicações. O PVM permite que mensagens contenham diferentes representações de dados;
- suporte a multiprocessador: o PVM usa os recursos de passagem de mensagens nativos dos multiprocessadores, pois os fabricantes freqüentemente fornecem sua própria versão de PVM otimizada para seus sistemas, que podem ainda comunicar com a versão pública do PVM.

O modelo de comunicação do PVM provê *send* bloqueante assíncrono, *receive* bloqueante assíncrono e *receive* não bloqueante. E em sua versão 3, o PVM foi projetado para resistir à maioria das falhas envolvendo *hosts* e redes (Santana et al., 1997).

2.8 – ESCALONAMENTO DE PROCESSOS

A atividade de escalonamento consiste em selecionar uma tarefa e o local onde esta tarefa será executada. Essa atividade é comum tanto aos sistemas distribuídos quanto aos sistemas com multiprocessadores (Zaluska, 1991).

Apesar dos termos “*escalonamento de processos*” e “*balanceamento de carga*” freqüentemente serem empregados como sinônimos, o escalonamento de processos deve ser visto como uma atividade que tem como objetivo o balanceamento da carga entre os vários elementos de processamento (Wang & Morris, 1985; Casavant & Kuhl, 1988; Shivaratri, 1992; Rewini et al., 1995).

O balanceamento de carga tenta distribuir uniformemente os recursos compartilhados, evitando a situação em que um elemento de processamento está sobrecarregado enquanto que outro está com a carga de processamento leve. Uma transferência de tarefa pode ser preemptiva ou não-preemptiva. O primeiro caso envolve a transferência de uma tarefa parcialmente executada. Esta operação é geralmente cara, pois envolve a coleta de informações sobre o estado da tarefa, sendo que essas podem ser grandes e/ou complexas. O segundo caso envolve somente tarefas que não tenham sido iniciadas. Em

ambos os casos, informações sobre o ambiente no qual a tarefa irá executar devem ser transferidas para o nó remoto. A figura 2.12 ilustra um sistema distribuído sem balanceamento de carga (Shivaratri et al., 1992).

Figura 2.12: um sistema distribuído sem balanceamento de carga (Shivaratri et al., 1992)

É desejável que uma estratégia de escalonamento de processos possua as seguintes propriedades (Wang & Morris, 1985):

- desempenho ideal do sistema como um todo: a capacidade total de processamento deve ser maximizada;
- equidade de serviço: o desempenho deve ser uniforme, independente da origem das tarefas;
- tolerância a falhas: a robustez do desempenho deve ser mantida na presença de falhas parciais do sistema.

O balanceamento de carga pode ser dividido em fisicamente distribuído e fisicamente não distribuído. No *fisicamente distribuído*, cada elemento de processamento possui autonomia para decidir a respeito do escalonamento, ou seja, o processo de decisão é distribuído entre os elementos de processamento do sistema. O processo de escalonamento pode ser feito de forma cooperativa (considerando o estado dos outros processadores que fazem parte do sistema) ou não cooperativa (não considerando o estado dos outros processadores). Já no escalonamento *fisicamente não distribuído*, existe um único processador responsável pelo balanceamento em todo o sistema. Essa categoria engloba três grupos; **a) abordagem determinística**: mecanismo mais simples, onde o algoritmo envia o i -ésimo processo para o processador $i \bmod N$. Esse algoritmo funciona bem em sistemas com poucos processadores e processos independentes um dos outros; **b) abordagem aleatória**: semelhante

ao anterior, entretanto, quando um processo é gerado, é verificado o estado de carga desse processador. Caso sua carga seja baixa, o processo é executado, caso contrário o processo é distribuído aleatoriamente a outro processador; c) *distribuição mestre-escravo (farmer)*: existe um processo mestre que distribui um conjunto de dados de um problema para um conjunto de processos escravos que residem em outros processadores. A distribuição dos processos é feita durante a compilação e os dados são distribuídos durante a execução.

Embora não haja um consenso na determinação dos componentes de um escalonador de processos, os seguintes são bastante difundidos na literatura (Shivaratri et al., 1992):

1. *política de transferência*: determina se um nó está apto a participar do processo como emissor ou como receptor, conforme sua carga;
2. *política de seleção*: escolhe a tarefa a ser transferida (geralmente a iniciada mais recentemente);
3. *política de localização*: responsável por encontrar um parceiro de transferência (emissor ou receptor) adequado para um nó, assim que a política tenha decidido que este nó é um emissor ou um receptor;
4. *política de informação*: decide quando as informações sobre os estados de outros nós no sistema devem ser coletadas, de onde serão coletadas, e quais informações serão coletadas. Existem três tipos de políticas de informação:
 - *política orientada à demanda*: um nó coleta o estado dos outros nós somente quando ele se torna um emissor ou receptor;
 - *política periódica*: as informações são coletadas de tempos em tempos;
 - *política orientada à mudança de estado*: as informações dos nós são divulgadas conforme mude o grau de seu estado.

No que concerne à classificação e taxonomia dos escalonadores de processos, também não existe uma regra ou consenso geral. Os algoritmos de escalonamento podem ser classificados de várias maneiras. Uma classificação bem simples, proposta por Shivaratri (Shivaratri et al., 1992) classifica os algoritmos de balanceamento de carga em três tipos: algoritmos iniciados pelo transmissor, algoritmos iniciados pelo receptor e algoritmos iniciados simetricamente. A saber:

a) *algoritmos iniciados pelo transmissor*: a decisão de balanceamento de carga é iniciada por um elemento de processamento que está carregado que tenta enviar tarefas para os elementos de processamento que estejam ociosos;

b) *algoritmos iniciados pelo receptor*: a atividade de balanceamento de carga é iniciada pelo elemento de processamento que está ocioso, que tenta receber tarefas de elementos de processamento mais sobrecarregados;

c) *algoritmos iniciados simetricamente*: nesses algoritmos, tanto o emissor quanto o receptor dão início às atividades de distribuição de carga. Este tipo de algoritmo apresenta vantagens sobre os anteriores. Se a carga do sistema está baixa, os elementos transmissores podem ter mais sucesso em encontrar elementos de processamento com pouca carga de trabalho; quando a carga é alta, os receptores são mais indicados para encontrar elementos de processamento sobrecarregados.

Uma outra taxonomia de escalonamento de processos mais abrangente e completa foi proposta por Thomas Casavant e Jon Kuhl (Casavant & Kuhl, 1988) e está ilustrada na figura 2.13 e será detalhada a seguir.

Figura 2.13: Taxonomia de escalonamento de processos (Casavant & Kuhl, 1988)

- **Local ou global:** escalonamento local está relacionado com a distribuição dos processos em um único processador através da alocação de fatias de tempo para o processador executar o processo. Já o escalonamento global consiste em escolher qual processador executará um processo e o sistema operacional trata do escalonamento local desse processo no processador escolhido.
- **estático:** nesse nível são tomadas as decisões sobre escalonamento do tempo disponível para cada processo no processador. No caso de escalonamento estático, as informações relativas sobre o total de processos, todas suas subtarefas envolvidas são assumidas que estão disponíveis até que os módulos de objeto de programa sejam unidos em módulos de carga, o que determina uma tarefa estática para um processador;
- **dinâmico:** no escalonamento dinâmico é feita uma suposição mais realista das necessidades de recursos de um processo, mesmo que seja desconhecido o que os processos deverão executar, nenhuma decisão é tomada até que o processo inicie.

*Niranjan Shivaratri (Shivaratri et al., 1992) propõe ainda que os algoritmos de distribuição de carga podem ser caracterizados como **adaptativos**: os algoritmos adaptativos compõem uma classe especial dos algoritmos dinâmicos. Esse tipo de algoritmo adapta sua atividade dinamicamente, mudando seus parâmetros ou mesmo sua política. É interessante utilizar esses algoritmos em sistemas que estejam uniformemente sobrecarregados, pois a transferência de carga entre os nós não traria benefício algum. Nessa situação, o algoritmo adaptativo suspenderia sua atividade por algum tempo.*

- **ótimo:** nesse caso todas as informações relativas ao estado do sistema assim como os recursos necessários por um processo são conhecidos. Uma tarefa ótima é aquela onde são minimizados o tempo de conclusão de processo, maximizando a utilização dos recursos do sistema, ou maximizando o processamento do sistema;
- **sub-ótimo:** quando a minimização do tempo de conclusão e a maximização dos recursos não são alcançados satisfatoriamente.
- **sub-ótimo aproximado:** nesse caso, uma solução *boa* é considerada satisfatória;

- **sub-ótimo heurístico:** procura através de uma métrica evolutiva a melhor solução para atingir a otimização dos recursos disponíveis para execução dos processos.
- **global estático ótimo e global estático subótimo aproximado:** existem quatro categorias básicas de algoritmos de alocação de tarefas que podem ser usados para se conseguir a atribuição dos processos aos processadores:
 1. enumerativo;
 2. teoria dos grafos;
 3. programação matemática;
 4. teoria das filas .
- **fisicamente distribuído:** as tarefas são distribuídas fisicamente entre os processadores;
- **fisicamente não distribuído:** este nível define se a alocação da tarefa deverá residir fisicamente em um único processador.
- **fisicamente distribuído cooperativo:** os componentes distribuídos cooperam entre si;
- **fisicamente distribuído não cooperativo:** os processadores tomam decisões independentemente das ações dos outros processadores.

2.9 – COMENTÁRIOS FINAIS

Tendo como maior motivação a busca por melhor desempenho, a computação paralela é uma forma eficiente de processamento da informação com ênfase na exploração de eventos concorrentes no processo computacional. A maior complexidade dos sistemas paralelos em relação aos sistemas seqüenciais torna mais evidente a importância de estudar o desempenho desses sistemas.

Além de suas características de *software* e *hardware*, a carga de trabalho que um sistema tem que processar também influencia seu desempenho. Para evitar a situação em que alguns elementos de processamento estão sobrecarregados enquanto que outros estão ociosos, técnicas de escalonamento de processos são empregadas.

O balanceamento de carga tenta distribuir uniformemente os recursos compartilhados, evitando a situação em que um elemento de processamento está sobrecarregado enquanto que outro está com a carga de processamento leve. Entretanto, no processo de escalonamento da carga de trabalho, a escolha da política de escalonamento não é simples. Para tornar mais flexível o gerenciamento dessas tarefas é que se propõe o AMIGO (*dynAMical flexIble schedulinG enviroNment*), que é uma ferramenta de *software* aberta, responsável pela gerência do escalonamento de processos. Tal ferramenta é estudada no capítulo seguinte.

capítulo 3

AMIGO

Um Ambiente para Escalonamento de Processos

Como visto no capítulo anterior, o escalonamento de processos é uma atividade de grande importância para os sistemas computacionais, independente de sua arquitetura (MIMD, SIMD, etc.).

Este capítulo trata dos componentes, características e principais detalhes de um ambiente de escalonamento denominado AMIGO, especificado em um projeto de doutorado do Grupo de Sistemas Distribuídos e Programação Concorrente do Instituto de Ciências Matemáticas e de Computação - ICMC/USP (Souza et al., 1998; Araújo et al., 1999a; Araújo et al., 1999b; Araújo, 1999c; Souza et al., 1999a; Souza et al., 1999b; Souza et al., 1999c; Souza et al., 1999c; Souza et al., 1999d; Souza et al., 1999e; Souza, 2000). Todas as figuras desse capítulo são creditadas a (Souza, 2000).

3.1 - INTRODUÇÃO

Uma quantidade considerável de trabalhos vem sendo realizada na área de escalonamento de processos, entretanto, a aplicação dessas pesquisas não vem sendo realizada com a mesma intensidade. Isso ocorre devido a três motivos principais (Feitelson & Rudolph, 1996):

- a) falta de uma nomenclatura coesa;
- b) dificuldade de se relacionar trabalhos diferentes;
- c) falta de uma maior flexibilidade dos objetivos inseridos nos algoritmos.

Por conta das dificuldades expostas, a escolha da política de escalonamento não é trivial. Além disso, a escolha correta da política de escalonamento tem forte influência

no desempenho final alcançado pelo sistema e é vital para se usar os recursos disponíveis com eficácia⁸.

Surge então a questão: qual é a melhor política de escalonamento? A resposta para esta pergunta não é única, uma vez que não existe uma única política de escalonamento que forneça picos de otimização para todos os casos (Souza et al., 1999a). O escalonamento de processos deve tratar diversos fatores que influenciam o desempenho, tais como (Souza et al., 1999b):

- *plataforma utilizada*: se a computação está sendo executada em uma arquitetura paralela ou em um sistema distribuído;
- *hardware disponível*: os recursos de computação e comunicação utilizados;
- *ambientes multiusuários*: caso vários usuários compartilhem os recursos computacionais;
- *diferentes classes de aplicação*: se a aplicação é em lote, interativa ou em tempo real;
- *carga de trabalho*: ambientes com índices alto e baixo de carga se comportam de maneiras diferentes.

Uma ferramenta flexível e dinâmica para suportar atividades de escalonamento pelo agrupamento das melhores políticas de escalonamento em um modo fácil, coerente e organizado é proposto pelo AMIGO (dynAMical flexIble schedulinG envirOnment). O AMIGO permite reduzir a distância entre a teoria e a prática, e também permite a obtenção de um pico de desempenho devido à correta utilização da melhor política de escalonamento.

A flexibilidade do AMIGO é mantida devido à capacidade de se incluir e escolher os vários fatores que têm influência no processo de escalonamento, incluindo aqui as políticas mais ajustáveis para as necessidades dos usuários. Essa ferramenta também é dinâmica, pois possui a capacidade de alterar completamente a política de escalonamento adotada em tempo real.

⁸ No contexto desse trabalho, eficácia está relacionada com as necessidades atuais dos usuários e não pode ser quantificada uniforme e globalmente para todas as situações.

Atualmente, vários projetos de mestrado e doutorado estão em andamento e tem por objetivo contribuir para a implementação do ambiente de escalonamento AMIGO. O projeto de mestrado, que é melhor discutido no capítulo 5, trata do desenvolvimento de uma interface para o AMIGO.

3.2 – OBJETIVOS DO AMIGO

Como visto na seção introdutória, o AMIGO é um ambiente de escalonamento flexível e dinâmico.

A flexibilidade é obtida através de um projeto de *software* aberto, isto é, com objetivos bem especificados e interfaces bem definidas, permitindo futuras inserções e atualizações. Além disso, seu código fonte é aberto e está disponível para quem se interessar. Já a característica dinâmica⁹ do AMIGO é devido à sua capacidade de mudar a política de escalonamento utilizada durante a execução da aplicação.

No início de seu projeto, o AMIGO foi proposto para trabalhar em ambientes de computação paralela distribuída rodando em plataformas heterogêneas (uma rede de estações de trabalho baseada em UNIX), gerenciado por um ambiente de passagem de mensagens como PVM ou MPI. Entretanto, a adoção do AMIGO não é restrita a esse tipo de ambiente e pode ser facilmente adotado em um ambiente distribuído executando aplicações sequenciais. Além disso, devido à sua flexibilidade, o AMIGO pode ser empregado em máquinas paralelas reais.

A escolha por ambientes paralelos distribuídos se justifica por sua heterogeneidade, visto que a tarefa de escalonamento tem maior influência nesse tipo de ambiente. Essa afirmação é constatada pela observância das características tanto dos ambientes verdadeiramente paralelos (que normalmente são formados por recursos homogêneos tanto em sua arquitetura como configuração); quanto dos sistemas distribuídos (que devem lidar com a heterogeneidade em termos de configuração, por exemplo, a quantidade de memória e velocidade dos processadores, quanto em termos de arquitetura, por

exemplo, se os processadores são CISC ou RISC). A heterogeneidade pode se dar ainda em vários outros níveis, tais como os *software* básicos utilizados, a topologia de rede adotada, a tecnologia de comunicação empregada, entre outros.

Outro objetivo do AMIGO é tornar fácil o uso das políticas de escalonamento e dar “transparência” aos detalhes envolvidos. Ou seja, o usuário não precisa ter conhecimento profundo das políticas de escalonamento ou se preocupar em como aplicá-las. Entretanto, para tornar o AMIGO específico para um ambiente local particular, é necessário que a configuração seja feita por uma pessoa com mais conhecimento na área (por exemplo, um gerente de recursos); e que essa pessoa tenha certos direitos de acesso às atividades do sistema, caso esse privilégio de acesso seja necessário.

Vale ressaltar que nada impede que um usuário comum configure o ambiente. E essa estratégia possui vantagens e desvantagens. A vantagem é que o usuário pode otimizar a configuração do ambiente segundo as suas necessidades reais, por conta da facilidade em determinar a política e o mecanismo mais apropriados para a sua aplicação. Como desvantagem, é fácil perceber que esse usuário pode ter uma visão egoísta do ambiente utilizado e talvez prejudicar o funcionamento do ambiente como um todo.

Levando em consideração que a configuração seja feita por uma pessoa especializada, como o gerente de recursos da plataforma, por exemplo, novamente vantagens e desvantagens podem ser assinaladas. Como vantagens, pode-se observar: a) que o escalonamento será feito através de uma visão mais democrática dos recursos disponíveis (pois o gerente de recursos almeja a satisfação coletiva e não a satisfação de apenas um usuário em específico); e b) maior transparência obtida, pois os usuários finais, que seriam responsáveis apenas por suas aplicações, estariam utilizando o escalonamento sem se preocupar como isso seria feito, pois essa atividade seria gerenciada pelo AMIGO. Em contrapartida, pode-se assinalar como desvantagem o fato de que algumas atividades podem ser dificultadas, como por exemplo, a avaliação do desempenho de alguma aplicação que necessite controle total da plataforma utilizada.

⁹ Diferentemente de sua utilização nos “*algoritmos de escalonamento dinâmicos*”, o termo *dinâmico* no AMIGO referencia sua capacidade de alterar dinamicamente as políticas que estão sendo usadas no escalonamento durante a execução da aplicação do usuário.

3.3 – A ESTRUTURA DO AMIGO

O AMIGO é estruturado em duas camadas principais, conforme ilustrado na figura 3.1, denominadas de camada superior e camada inferior. A camada superior é composta por uma interface gráfica amigável na qual o usuário tem acesso a muitas opções para configurar o ambiente e ajustar a maneira que as políticas de escalonamento serão usadas. A camada inferior possui os módulos responsáveis pelo escalonamento e também a interface entre as aplicações do usuário e as políticas de escalonamento disponíveis e eventualmente ativas.

Figura 3.1: A estrutura interna do AMIGO (Souza, 2000)

Procurar alcançar uma maior modularidade e facilitar a portabilidade para outras plataformas são os principais argumentos para justificar a divisão em camadas. Em relação à dependência das camadas no que diz respeito à plataforma utilizada, a inferior é mais dependente do que a superior. Isso se deve ao fato de que a camada inferior utiliza as primitivas necessárias para a comunicação, obtenção da carga computacional, execução de novos processos, entre outras características da plataforma que está sendo executada. Em um nível bem menor, a dependência da camada superior se limita a alguns fatores inerentes à

linguagem de programação e ao sistema operacional. Por meio de pequenas adaptações ela pode ser executada facilmente em outras plataformas que não a usada inicialmente.

O acesso à camada superior, independente dos privilégios que o usuário possua¹⁰ (seja ele gerente de recursos ou usuário comum), se dá através da interface gráfica pertencente a essa camada. Essa interface possibilita ao usuário o uso, escolha e definição de cinco itens¹¹:

1 - Classes de *software*:

- cadastro de Classes de *Software*;
- cadastro de Aplicações.

2 - Plataformas:

- cadastro de *hardware*;
- detecção automática do *hardware* disponível;
- *benchmark's* para avaliar os processadores (cadastro e execução).

3 - Políticas de escalonamento:

- cadastro de políticas;
- relacionamento classes & políticas;
- configura uso de políticas;
- escolha da política:
 - padrão;
 - assistido;
 - livre.

4 - Monitoramento:

- cadastro de métricas;
- controle do monitoramento (início/ativação e término/suspensão).

5 – Simulação:

- aplicações;
- plataformas;
- políticas;
- simula.

Na seção 3.3.1, maiores detalhes serão fornecidos sobre a camada superior.

¹⁰ É interessante, porém não obrigatório que o usuário tenha algum conhecimento para que possa configurar o ambiente mais adequado para a sua realidade.

¹¹ O usuário não fará o escalonamento propriamente dito, pois tal tarefa é função e exclusiva da camada inferior.

Quanto à camada inferior, o acesso pelo usuário se dá através do ambiente de passagem de mensagens utilizado. O usuário executa da maneira corriqueira o ambiente de passagem de mensagens e este, por sua vez, inicia a execução do AMIGO, através de um *daemon* denominado AMIGOD. A seção 3.3.2 que trata da camada inferior do AMIGO explica melhor o componente AMIGOD. Vale ressaltar que o funcionamento da camada inferior independe do funcionamento da camada superior.

Depois que o ambiente de passagem de mensagens esteja devidamente configurado e executando, o acesso do usuário ao AMIGO se dá através das aplicações paralelas.

3.3.1 – A Camada Superior do AMIGO

Como visto na seção anterior, a camada superior do AMIGO é formada basicamente por uma interface gráfica que permite ao usuário o gerenciamento da flexibilidade inserida no AMIGO. Esse gerenciamento é realizado em vários itens (visualizados na figura 3.1) que são tratados nas subseções a seguir.

3.3.1.1 – Classes de *Software*

A inclusão das classes de *software* foi projetada para conectar as políticas de escalonamento às classes de aplicação, porque não existe uma política de escalonamento perfeita para todos os tipos de aplicações. Como é difícil detalhar todas as aplicações existentes, a conexão das políticas não é feita diretamente para alguma aplicação em específico. A inclusão de classes de *software* vem de encontro à questão do custo para a obtenção da informação sobre a aplicação a ser escalonada. Nem sempre é possível (viável) que informações detalhadas sobre a aplicação sejam obtidas antes ou durante o escalonamento. Dessa maneira, as classes de *software* amenizam esse custo e viabilizam a obtenção das informações.

As classes não se preocupam com uma aplicação em especial, e sim com um grupo afim de aplicações. Além disso, as classes de *software* permitem a inclusão de informações extras sobre o comportamento futuro de uma aplicação e permite também a

classificação dos diversos tipos de aplicações. Caso uma aplicação não cadastrada venha a ser escalonada, a mesma irá utilizar o escalonamento padrão determinado para o momento da execução.

Os tipos de aplicação podem variar, conforme o enfoque adotado. As aplicações IO-bound e CPU-bound representam aplicações que envolvem mais comunicação ou mais processamento, respectivamente. Dependendo do grau de interação com o usuário durante a execução da aplicação, essa pode ser interativa ou de lote. Entre outros tipos de aplicações.

Apesar de não ser obrigatória, a inclusão de aplicações diretamente vinculadas às classes de *software* privilegia o escalonamento feito com maiores informações. Entretanto, a pouca ou nenhuma informação não impossibilitará o escalonamento das aplicações.

O gerenciamento das informações sobre as classes de *software* e suas aplicações é feito pelas opções **cadastro de classes** e **cadastro de aplicações**. O cadastro de classes é composto pelas seguintes informações e é armazenado no arquivo '*classoft.dad*': **a) código da classe**: formado por uma numeração seqüencial e crescente de classes cadastradas (de 001 até 999) e que será usado posteriormente para o relacionamento com políticas e aplicações; **b) nome da classe**: formado por uma cadeia de caracteres que representa o nome destinado à classe de *software* cadastrada; e **c) características**¹²: formadas por uma estrutura com conteúdo variável que armazena as características mais importantes sobre a classe (por exemplo, interatividade com o usuário, prazos determinados para o término, comunicação e computação, entre outras).

O cadastro das aplicações é feito pela opção **cadastro de aplicações** e são armazenadas no item [*applications*] do arquivo '*policy.cfg*' (figura 3.2). As seguintes informações devem ser fornecidas para incluir a aplicação a ser escalonada: **a) nome da aplicação**; **b) os seus arquivos executáveis**; e **c) código da classe de software que a aplicação pertence**.

¹² É através das informações armazenadas em *características* que a relação entre classes de software e políticas poderá ser feita automaticamente.

A consistência das informações nos dois arquivos '*classsoft.dad*' e '*policy.cfg*' é mantida toda vez que uma classe de *software* é excluída ou alterada.

Figura 3.2: Estrutura do arquivo de configuração policy.cfg (Souza, 2000)

3.3.1.2 – Plataformas Computacionais

O objetivo de se manter um registro com as informações de *hardware* (no caso, as plataformas computacionais) é o de permitir a avaliação individual da capacidade de cada elemento de processamento, considerando dessa forma a heterogeneidade encontrada nas plataformas.

O item **plataformas** permite a inclusão de fatores, como por exemplo arquitetura e conexão dos processadores para que os mesmos sejam considerados durante a escolha das políticas de escalonamento ativas. A opção **Cadastro de Hardware** possibilita o gerenciamento (acesso e inclusão) das informações sobre a plataforma armazenada no arquivo '*hardware.dad*'. As seguintes informações são armazenadas nesse arquivo: a) código identificador para a máquina (como o endereço IP), b) nome da máquina; c) arquitetura; d) sistema operacional e) número de processadores; f) quantia de memória principal disponível;

g) existência ou não de acesso a disco e h) características especiais (que tem a finalidade de informar outras características sobre o processador).

A inclusão dos processadores através da opção **Cadastro de Hardware** é feita pelo usuário de forma manual ou automática (através da opção **detecta hardware**). Se manual, o usuário informa quais elementos de processamento devem ser considerados, atualizando o registro do *hardware* disponível. Já na detecção automática, o AMIGO oferece uma lista de elementos disponíveis ao usuário, atualizando o registro de *hardware*.

O AMIGO permitirá também a simulação do *hardware*, tornando possível uma análise prévia do comportamento das políticas de escalonamento. Após detectar o *hardware*, são utilizadas as primitivas do sistema operacional como o *rsh* (remote shell) para executar alguns processos nos processadores encontrados para obter as demais informações necessárias ao cadastro da plataforma computacional, as quais não puderam ser obtidas diretamente durante a detecção, em função de limitações e/ou dificuldades no sistema operacional. Após coletarem as informações desejadas, esses processos são finalizados.

O AMIGO faz uso também de *benchmark's* que avaliam as diferenças de desempenho dos elementos de processamento disponíveis no registro de *hardware*. Dependendo da informação desejada, esses *benchmark's* podem apresentar grandes variações, pois o item avaliado pode ser o poder de processamento, a capacidade dos meios de comunicação ou qualquer outro item mensurável. Como visto na seção 2.3.1 do capítulo dois, existem vários *benchmark's* disponíveis para diferentes plataformas, como o NAS Parallel Benchmarks, Parkbench e o Genesis.

Diferentes *benchmark's* podem ser incluídos e executados no AMIGO através da opção **cadastro de benchmarks** do item **Plataformas**. Os resultados obtidos dessas ferramentas são armazenados em um arquivo de configuração denominado *bench.dad*. Esse arquivo está disponível para as políticas de escalonamento, que o utilizam para obter informações sobre a heterogeneidade do ambiente, de modo fácil e direto (por meio da opção **executa benchmarks** no item **plataformas**). A falta desse arquivo de configuração não implica na não execução da aplicação paralela. Sua ausência apenas faz a política de escalonamento ativar pressupor que todos os elementos de processamento possuem igual poder de processamento.

Para desenvolver um *benchmark* para o AMIGO é necessário apenas observar as características da plataforma utilizada (arquitetura, topologia, entre outros) e estabelecer como valor de saída (retorno) o tempo obtido durante a execução. A relação de melhor/pior será feita do menor para o maior tempo respectivamente.

3.3.1.3 – Política de Escalonamento

Quanto às políticas de escalonamento, no AMIGO elas são definidas por meio de um registro das políticas disponíveis. O item **Políticas de escalonamento** merece destaque em relação aos demais devido à importância das seguintes atividades associadas à esse item, tais como:

- viabilizar a inclusão de diferentes políticas de escalonamento e dessa forma torná-las disponíveis;
- estabelecer o relacionamento automático destas com os demais itens (como classes de *software*, *hardware* e outros);
- selecionar (ativar) as políticas que devem realmente ser utilizadas pela camada inferior.

O acesso ao cadastro de políticas de escalonamento é feito pela opção **Cadastro de Políticas** e seus respectivos dados são incluídos no arquivo '*policies.dad*'. Esse arquivo armazena as seguintes informações: **a) código**; **b) nome da política**; **c) nome do arquivo executável** (com o caminho completo) e **d) as principais características da política**. Embora a inclusão das características não seja obrigatória, a mesma é de vital importância para que a política possa ser relacionada automaticamente com as diferentes classes de *software*. É também através das *características* das políticas, que serão determinadas quais delas devem permanecer ativas, em função dos fatores que estão influenciando a plataforma no momento.

O item **relaciona classes & políticas** permite a associação das classes de *software* com as políticas de escalonamento. Para cada classe de *software* cadastrada é possível relacionar uma única política de escalonamento, mas uma mesma política pode estar vinculada a diferentes classes de *software*. Para fazer o relacionamento, a interface exibe as classes de *software* previamente cadastradas e permite que cada uma seja vinculada a uma

política apropriada. Após feito o relacionamento, o mesmo é armazenado em *[policies]* no arquivo '*policy.cfg*' (figura 3.2). Armazenando o relacionamento de classes com políticas dessa maneira, facilita a recuperação das informações pela camada inferior, diminuindo assim a sobrecarga da mesma.

O uso das políticas de escalonamento é configurado pela opção **configura uso de políticas** e essa configuração é armazenada no item *[config]* do arquivo '*policy.cfg*' (figura 3.2). Após configurar o uso do escalonamento, o usuário pode determinar se é possível: **a)** a existência de mais de uma política ativa ao mesmo tempo (*more_policies*); **b)** ativar políticas sob demanda (*by_demand*); **c)** mudar as políticas ativas automaticamente em tempo de execução (*change policy*); **d)** utilizar *cache* no AMIGOD para o índice de carga obtido em cada política ativa (*load_index_cache*); **e)** a troca de mensagens diretamente entre as políticas, sem passar pelo AMIGOD (*direct_route*); **f)** o uso de um arquivo *profile* (chamado de '*amigo.pfl*'), o qual é responsável por armazenar informações sobre o escalonamento feito na plataforma em um período preestabelecido.

Através da opção **seleciona políticas**, o usuário pode determinar quais políticas de escalonamento devem estar ativas no momento. É permitido que várias políticas de escalonamento estejam ativas em um mesmo instante, dividindo-as em três grupos:

- política reserva (*reserve policy*): representa a política utilizada em último caso, apenas se todas as outras políticas ativas não puderam ser executadas. A política reserva atua quando problemas impedirem a execução normal das outras políticas ativas. Um dos principais objetivos da política reserva é viabilizar o conceito de tolerância a falhas, de maneira transparente ao usuário. Mesmo que não haja o escalonamento mais apropriado para o momento, os processos terão outra oportunidade de serem escalonados sob o AMIGO;
- política independente (*independent policy*): quando não for possível utilizar uma das políticas específicas, o ambiente indica a política independente para escalonar os diferentes processos, independente da classe de *software* que esses processos se enquadrem;
- políticas específicas (*specific policies*): são as primeiras políticas de escalonamento a serem procuradas no escalonamento. Elas só poderão ser utilizadas se: **a)** a opção *more_policies* habilitar a execução de mais de uma

política de escalonamento; **b)** a aplicação estiver vinculada a uma política de escalonamento ativa, através da sua classe de *software*; e **c)** a opção *by_demand* permitir a execução de políticas que não estão ativas, quando essas foram solicitadas por uma aplicação em particular.

Seguindo o princípio do AMIGO, as políticas devem ser específicas e pequenas (e não grandes e genéricas), isto é, não deve existir uma única política para todos os tipos de aplicação (por exemplo, IO-Bound, CPU-Bound, etc.). Essa característica permite a obtenção do melhor desempenho para uma dada plataforma. A escolha da política de escalonamento é feita pela opção **seleciona políticas** através de um dos seguintes modos (figura 3.3):

- **padrão**: indicado para usuários com pouco ou nenhum conhecimento sobre escalonamento de processos. Essa opção fornece uma política de escalonamento baseada na heurística adotada pelo AMIGO, sem a interferência do usuário. A heurística aplicada aqui considera as classes de *software*, a eficiência da comunicação, a distância entre os elementos de processamento, a carga de trabalho, tempo para iniciar a execução e questões administrativas;
- **assistida**: permite a interferência do usuário na seleção da política de escalonamento feita pelo AMIGO. A heurística seguida nessa opção é a mesma utilizada na *opção padrão*, mas o usuário pode dar maior prioridade para algumas características, mudando a escolha da política;
- **livre**: indicado para o usuário que tenha total domínio da teoria do escalonamento de processos, essa opção permite a indicação de qual política deve ser usada, quando e para quais aplicações ela pode ser adotada. A responsabilidade nesse caso está totalmente nas mãos do usuário.

Para que a heurística seja empregada, são utilizadas as informações vindas do arquivo *profile* (*'amigo.pfl'*), se este estiver disponível, além das características inseridas nos diversos cadastros. Com o *'amigo.pfl'* e as características cadastradas, há dados locais e específicos suficientes para decidir quais políticas ativar.

Figura 3.3: A escolha das políticas da Camada Superior do AMIGO (Souza, 2000)

3.3.1.4 – Monitoramento

Mesmo não fazendo parte do escalonamento em si, a atividade de monitoramento é necessária para verificar como se comporta uma plataforma computacional (principalmente a multiusuária). O monitoramento permite que erros sejam corrigidos e que seja verificado se os objetivos estão sendo alcançados com sucesso. De acordo com a heurística adotada pelo AMIGO, o monitoramento é uma ferramenta que avalia o desempenho da plataforma de computação, indicando um possível problema e também qual política deve ser aplicada.

Através das informações obtidas pelo monitoramento é possível afirmar se as políticas de escalonamento ativas devem ou não ser substituídas. O monitoramento pode ser

feito de diversas formas, como por exemplo: uso de um *hardware* específico, alterações feitas no sistema operacional e o monitoramento feito por *software* fora do sistema operacional.

Para realizar o monitoramento, há a necessidade de se utilizar métricas que quantifiquem o quanto os objetivos foram realmente atingidos. Por esse motivo, a escolha da métrica de monitoramento deve estar relacionada com os objetivos propostos.

No AMIGO, o monitoramento está disponível através da inclusão de métricas pela opção **cadastro de métricas** do item **monitoramento** que define qual índice de desempenho será adotado em cada caso. As informações sobre as métricas são armazenadas no arquivo '*metrics.dad*' e são compostas por: **a) código;** **b) nome;** **c) nome do arquivo executável** (com o caminho completo); **d) nome do arquivo** que irá armazenar os valores obtidos (com o caminho completo) e **e) características específicas** da métrica. As métricas estão implementadas em arquivos executáveis, os quais são chamados pela rotina de monitoramento para coletar as informações desejadas. Uma vez coletada a informação, a mesma é tratada pela rotina de monitoramento, para ser visualizada de maneira gráfica e/ou textual pelo usuário e em seguida armazenada nos arquivos que guardam os resultados obtidos (especificados no cadastro de métricas).

Por meio da opção **início/término** no item **monitoramento**, o usuário inicia e controla o monitoramento. Além disso, nessa opção o usuário pode:

- escolher a métrica a ser utilizada;
- determinar o horário para funcionamento;
- estabelecer os intervalos para a coleta das informações;
- visualizar o desempenho, expresso gráfica e/ou textualmente.

3.3.1.5 - Simulação

O item **simulação** foi incluído com o intuito de se verificar o comportamento das políticas de escalonamento sem a necessidade de executá-las realmente.

O uso da simulação no escalonamento de processos é interessante porque permite, entre outras coisas:

- uma avaliação preliminar de uma política antes da sua implementação, diminuindo assim os custos dessa implementação;
- avaliar o comportamento de uma política já implementada ou não, sob outros fatores como: outras plataformas, outras classes de *software* ou variações na carga computacional;
- analisar previamente as alterações na implementação de políticas já existentes (com o intuito de melhoria de desempenho);
- analisar o comportamento do AMIGO com o uso simultâneo de várias políticas de escalonamento.

Em função de sua abrangência e de constituir uma área de pesquisa à parte, o item simulação não será detalhado nesse trabalho. Cabe ressaltar apenas que a simulação faz parte do projeto AMIGO e que a mesma será especificada e implementada em trabalhos futuros.

3.3.2 – A Camada Inferior do AMIGO

Enquanto a camada superior do AMIGO está envolvida com toda a configuração do ambiente no processo de escalonamento, através de sua interface gráfica, a camada inferior oferece a flexibilidade e o escalonamento dinâmico propostos. Vale ressaltar que as duas camadas trabalham independentemente uma da outra, ou seja, uma delas pode operar mesmo que a outra não esteja ativa. Para que a camada inferior funcione, esta precisa somente dos arquivos de configuração produzidos pela camada superior. A ausência desses arquivos implica na adoção de valores *default* pela camada inferior, sem penalizar os usuários.

A camada inferior oferece suporte para a flexibilidade e o escalonamento dinâmico propostos por meio de três módulos: o ambiente de passagem de mensagens, o *daemon* AMIGO (AMIGOD) e as políticas de escalonamento (conforme ilustrado na figura 3.4). O ambiente de passagem de mensagens é um cliente do AMIGO, pois esses ambientes (computadores paralelos distribuídos) solicitam ao AMIGO os elementos de processamento necessários para suas aplicações paralelas. Algumas poucas mudanças são necessárias aos ambientes de passagem de mensagens para permitir a interação com o AMIGO. Considerando, por exemplo o MPI e o PVM, as seguintes alterações são necessárias:

- conforme novos elementos de processamento são iniciados ou parados, AMIGOD's devem ser incluídos ou removidos, respectivamente;
- toda a vez que a aplicação paralela solicita recursos computacionais, novas tarefas devem ser atribuídas aos elementos de processamento;
- na preempção de processos, quando esses iniciam e ainda não estão concluídos.

Figura 3.4: Módulos da Camada Inferior (Souza, 2000)

Se para os ambientes de passagem de mensagens algumas poucas mudanças são necessárias para a interação com o AMIGO, para a aplicação do usuário nenhuma mudança no código fonte é requerida. Todos os programas que estejam funcionando corretamente no PVM, MPI ou qualquer outro ambiente de passagem de mensagens funcionará no AMIGO. Essa característica mantém a transparência para o usuário. E o caráter dinâmico é obtido através da implementação das políticas de escalonamento como processos independentes. Assim, as políticas podem ser substituídas sem parar a aplicação ou mesmo informar sobre a mudança.

Responsável por interligar o ambiente de passagem de mensagens às políticas de escalonamento ativas, o AMIGOD é o componente central da camada inferior do AMIGO. Esse componente é um processo servidor que atua como um roteador e controlador das mensagens entre as políticas de escalonamento e o ambiente de passagem de mensagens.

O AMIGOD é formado basicamente por alguns poucos mecanismos de comunicação entre processos (*IPC*). Por conta disso, o AMIGOD é um componente pequeno

e “enxuto”, característica necessária, devido ao fato dele ser distribuído entre as máquinas envolvidas na atividade de escalonamento.

Uma característica muito importante do AMIGOD é que ele é totalmente distribuído entre os elementos de processamento participantes do escalonamento. Não existe uma hierarquia (como mestre/escravo, por exemplo) entre os vários AMIGOD's. O fato dos AMIGOD's ser distribuído se justifica, por uma maior possibilidade de tolerância a falhas e por evitar “gargalos” no sistema, compensando os aumentos de complexidade do projeto e a sobrecarga na comunicação.

As políticas de escalonamento são implementadas independentemente no AMIGOD e podem ter qualquer característica e objetivos. Isto ocorre porque eles são dinâmicos (isto é, eles atuam em tempo de execução da aplicação do usuário) e seguem o protocolo de comunicação especificado para as trocas de mensagens entre os vários módulos do AMIGO. As características mais importantes das políticas implementadas para o AMIGO são:

- independência nos ambientes de passagem de mensagens, uma vez que as políticas tem acesso somente ao AMIGOD;
- código independente, uma vez que o objetivo é ter políticas concisas e eficientes.

Embora o módulo AMIGOD seja completamente distribuído, as políticas de escalonamento não são necessariamente distribuídas. Elas podem ser centralizadas, e isso novamente exemplifica a alta flexibilidade do AMIGO. Atualmente, várias políticas podem ser implementadas, embora o AMIGO possa lidar somente com as dinâmicas.

O mecanismo IPC na camada inferior do AMIGO é feito por meio de *sockets* usando o protocolo TPC/IP. Três tipos de conexões são consideradas para a comunicação na camada inferior:

- AMIGOD↔POLÍTICA: as mensagens trocadas entre o AMIGOD e a Política são identificadas através de macros cujo prefixo é AP_;
- AMIGOD↔AMBIENTE: através do prefixo AA_, as mensagens trocadas entre o AMIGOD e o Ambiente são identificadas;

- AMIGOD $\leftrightarrow$ AMIGOD: as mensagens entre o AMIGOD e a Política comunicam-se através de mensagens rotuladas por macros iniciadas por AM_;

A figura 3.5 mostra o esquema de comunicação entre os módulos do AMIGO, onde pode-se observar que a comunicação entre os módulos pertencentes ao mesmo elemento de processamento são feitos através de *sockets* TCP e a comunicação feita entre elementos de processamento distintos é feita através de *sockets* UDP.

A escolha pelo protocolo TCP para os módulos em um mesmo elemento de processamento deve-se, principalmente, a necessidade de alta confiabilidade e a reduzida quantidade de módulos. Em contrapartida, para a comunicação entre os AMIGOD's, optou-se por *sockets* UDP. Essa escolha se justifica por dois motivos principais:

1. possibilitar a escalabilidade: pois em uma máquina com n *hosts*, cada AMIGOD deveria ter $n-1$ conexões, onde cada transmissão aberta com TCP consome um descritor de arquivos e alguns sistemas operacionais limitam o número de arquivos abertos;
2. diminuir a sobrecarga: TCP gasta tempo extra para fazer as conexões necessárias.

Figura 3.5: Relacionamento entre os módulos do AMIGO (Souza, 2000)

Cabe ressaltar que a camada inferior é independente do funcionamento da camada superior. Para uma funcionar a outra não precisa, necessariamente, estar funcionando. O que une as duas camadas são os arquivos de configuração gerados pela camada superior e utilizados na camada inferior.

3.4 – COMENTÁRIOS FINAIS

Apesar de existirem bons projetos na área de escalonamento de processos, poucos são utilizados na prática. O AMIGO tem como objetivo diminuir a distância entre a teoria e a prática.

Para tal finalidade, o AMIGO foi projetado para ser uma ferramenta flexível e dinâmica para suportar atividades de escalonamento pelo agrupamento das melhores políticas de escalonamento em um modo fácil, coerente e organizado.

A flexibilidade do AMIGO é mantida através de sua capacidade de se incluir e escolher os vários fatores que tem influência no processo de escalonamento. Tais fatores podem ser alterados em tempo de execução, o que caracteriza seu aspecto dinâmico. Essas características, por si só não garantem que o AMIGO não seja apenas mais um bom projeto na área de escalonamento de processos entre tantos outros sub-utilizados. Entretanto, o AMIGO possui um diferencial que justifica sua adoção pelo usuário final: os detalhes de implementação e as decisões tomadas podem ser totalmente transparentes para ele. Através de uma interface gráfica, o responsável pela plataforma computacional pode tomar todas as decisões necessárias para o escalonamento da aplicação paralela.

Devido à sua importância no projeto do AMIGO, o próximo capítulo terá como alvo de estudo as interfaces gráficas.

capítulo 4

Interfaces Gráficas com o Usuário

Como visto no capítulo anterior, o ambiente AMIGO é dividido em duas camadas, uma denominada superior e outra inferior.

O principal componente da camada superior é a interface gráfica, que é responsável pela comunicação entre o usuário e o ambiente de escalonamento. O objetivo deste projeto de mestrado é o desenvolvimento dessa interface gráfica. Desta forma, torna-se importante o estudo deste tópico.

4.1 - INTRODUÇÃO

As interfaces com o usuário são os elementos dos sistemas computacionais que permitem que uma pessoa acesse os recursos oferecidos pelo computador (Thimbleby, 1990). Ela é um conjunto de protocolos e técnicas para a troca de informações entre uma aplicação computadorizada e o usuário humano. A interface com o usuário é responsável pelos comandos solicitados por ele e por mostrar os resultados da aplicação de um modo compreensível (Larson, 1992). Sem esse mecanismo, o computador torna-se inútil. Além do mais, uma interface mal projetada frequentemente leva o usuário a cometer erros desnecessários.

Uma interface gráfica típica é um conjunto de regras que formalmente especifica a apresentação visual dos dados e funções, a interação com o usuário, a organização lógica e o comportamento da informação (Marcus, 1992). Ela pode fornecer um ambiente amigável, menos intimidador e menos confuso. Elas são distinguidas pela aparência de suas janelas e pelo modo como as ações do operador e as opções de entrada são manuseadas.

As opções de entrada para os programas de computador podem ser projetadas como um conjunto de ícones, os quais são símbolos gráficos que se parecem com a opção de processamento que eles representam, como arquivos ou lixeira, por exemplo. As vantagens desses sistemas é que um ícone pode ocupar menos espaço que seu correspondente textual e que o comando pode ser mais rapidamente compreendido e memorizado, se bem projetado (Marcus, 1992; Peddie, 1992).

Quando as interface gráficas com o usuário (GUI¹³) começaram a se popularizar, por volta de 1992, estudos mostraram que os usuários nesses ambientes trabalham mais rapidamente, com mais exatidão e com menores níveis de fadiga e frustração que usuários em um ambiente baseado em caracteres. A produtividade teve um aumento de 35 por cento, e a exatidão dos trabalhos concluídos teve um aumento na ordem de 74 por cento em alguns ambientes (Peddie, 1992). Alguns autores referenciam os sistemas de menu baseados em caracteres ou as linhas de comando como **interface comum com o usuário** (CUI – *Common User Interface*).

Através da figura 4.1 é possível observar por meio das curvas de aprendizado que os usuários conseguem uma maior proficiência mais rapidamente através de uma GUI. Isso se deve principalmente à padronização do modo de operação das GUI. Estudos indicam que usuários que utilizam aplicações baseadas em caracteres levam mais tempo para aprender a usar o aplicativo. Conseqüentemente, usuários de GUI podem aprender a utilizar mais aplicações que usuários de CUI, em um mesmo período de tempo (Peddie, 1992).

¹³ O termo **interface gráfica com o usuário** é referenciado como GUI que é a sigla em inglês para *Graphical User Interface*.

Figura 4.1: Proficiência do usuário

4.2 - GUI

Sistemas com interfaces gráficas com o usuário não é uma idéia nova. Em seus artigos escritos em 1945, Vannervar Bush já propunha sistemas gráficos. Nas décadas de 50 a 70, cientistas do MIT (Massachusetts Institute of Technology) e do PARC (Xerox's Palo Alto Research Centre) investigavam a interatividade das interfaces com o usuário. Suas pesquisas mostraram que as pessoas poderiam aprender a usar aplicações com GUI mais rapidamente do que aprendendo comandos de linha (Peddie, 1992). Em 1983 a maioria dos grandes fabricantes de *workstations* tinha seus sistemas de janelas proprietários. Porém, foi em 1984, quando a Apple lançou o Macintosh que os sistemas baseados em janelas tornaram-se realmente populares.

Também em 1984, a partir de um projeto do MIT chamado Athena, surgiu o sistema X Window (Xwindows ou simplesmente X), que tinha por objetivo o uso de estações gráficas no auxílio de estudantes. A idéia desse sistema era o de possibilitar que os usuários executassem aplicações como processadores de textos ou planilhas eletrônicas localmente em

suas máquinas enquanto que as bibliotecas gráficas e os documentos eram requisitados de fontes remotas. O sistema X Window não é específico a um fabricante e apesar de não ser uma GUI ele atua como uma base, nas quais as GUIs são construídas (assim como o OSF/Motif e o DECwindows).

No ano seguinte, percebendo a potencialidade comercial dos ambientes gráficos, a Microsoft cria o MS-Windows que foi uma alternativa orientada a janelas ao MS-DOS para computadores compatíveis com o IBM-PC. O Windows sofreu a primeira atualização significativa em 1995. Em 1998 as mudanças da então nova versão foram superficiais. Recentemente, foi lançado o Windows 2000, que é a nova versão da linha de sistemas operacionais corporativos da Microsoft, a evolução do Windows NT. O Windows 2000 Professional é a evolução do Windows NT Workstation 5.0 e o Windows 2000 Server é a do Windows NT Server 5.0. Para pequenos escritórios e para o usuário final, está prevista uma atualização do Windows 98 para o segundo semestre deste ano, denominado Windows Me.

Além dos sistemas gráficos mencionados, existem vários outros, como o OS2, NextStep, Open Look GUI, Commodore AMIGA, etc.

Os estudos das interfaces gráficas concluíram que o usuário poderia obter mais do computador em menos tempo, porque ele gastaria seu tempo fazendo o trabalho e não formalizando suas necessidades ao computador.

Independente do sistema gráfico¹⁴, uma interface gráfica com o usuário deve considerar os seguintes componentes chaves:

- uma visão que capture a essência do sistema;
- organização apropriada dos dados, funções, ferramentas, etc.;
- esquema de navegação eficiente;
- qualidade da aparência (tamanho, forma, cor, localização) de cada elemento visual na tela;
- ajuda on-line.

¹⁴ apesar das várias interfaces gráficas não serem exatamente iguais, elas possuem os mesmos simbolismos básicos.

Considerando os vários sistemas gráficos disponíveis atualmente, uma característica comum a todos é a facilidade de uso. Facilidade de uso inclui quão fácil os usuários são capazes de aprender a utilizar a aplicação, com que facilidade os usuários poderão empregar suas aplicações e a frequência de erros na interação com a aplicação. A facilidade de uso é um dos maiores fatores que influenciam no modo como o usuário percebe e trabalha uma aplicação; e seu sucesso é determinado pela facilidade com que a aplicação pode ser aprendida e usada. Por mais poderosa que seja uma aplicação, isso não é garantia de que ela será um sucesso se sua interface com o usuário for pobre.

Embora sejam encontradas várias vantagens nas interfaces gráficas com o usuário, como facilidade de uso, menor tempo de aprendizagem, funcionalidade, produtividade, etc., algumas dificuldades e precauções devem ser consideradas na adoção de uma GUI. Entre elas (Peddie, 1992):

- **custo:** a utilização de interfaces gráficas demanda um maior uso de instruções de processador. Esse argumento tem perdido força devido ao contínuo incremento da velocidade do *clock* dos processadores;
- **treinamento:** o uso das GUIs não é óbvio e intuitivo para todas as pessoas. Sempre será necessário o treinamento, pelo menos das funções mais específicas.
- **stress:** um estudo psicológico em vários usuários de computador constatou um aumento de *stress* em alguns deles. Como as interfaces gráficas permitem uma maior produtividade por parte do usuário, ele tem mais tempo livre disponível para realizar mais e mais trabalho.

4.3 – O PROJETO DE UMA GUI

Um projeto pode ser considerado tanto como um **produto** quanto um **processo** (Wood, 1998). O *produto* é um artefato projetado para um propósito específico, dado um conjunto de componentes, recursos e obrigações com o qual o projetista deve trabalhar. O *processo* consiste de técnicas e procedimentos para a construção de um desejado produto.

Projetar uma interface gráfica não é trivial. Embora pareça óbvio, no desenvolvimento de uma interface com o usuário, o projetista deve ter em mente que ela deve

ser **útil**, no sentido de utilizável. Para ser útil, uma interface deve fornecer acesso às funções e características de uma aplicação, permitindo a interação entre ela e o usuário de modo intuitivo e natural.

A interação entre homem-computador deve ser baseada em como as pessoas interagem com as coisas e com outras pessoas. Esse modelo é considerado “natural” e a interação pode ser modelada como um tipo de diálogo, uma sequência de ações e respostas. Em um diálogo homem-computador o computador apresenta ao usuário as possibilidades, o usuário faz um pedido e o computador responde com resultados ou confirmações. Assim como no diálogo humano, o diálogo homem-computador deve ser no nível correto: nem muito alto para não se tornar difícil de compreender, nem muito baixo a ponto de ser limitado. O desafio do projetista é selecionar o nível correto do diálogo, relacionado com as tarefas, objetos e ações da aplicação.

Existem vários fatores que influenciam a boa aceitação ou não de uma interface pelo usuário. O projetista deve considerar os seguintes aspectos que podem ser encontrados no desenvolvimento de uma GUI (Larson, 1992):

- **grau de aprendizado do usuário:** o desenvolvedor da interface deve conhecer o grau de familiaridade do usuário com a aplicação e dessa forma projetar uma interface que seja apropriada às necessidades do usuário;
- **conveniência de uso:** as interfaces não devem ser convenientes do ponto de vista do projetista e sim do usuário. Geralmente, os programadores gastam anos aprendendo como escrever programas, se esquecendo de como os usuários utilizam o computador para concluir seu trabalho;
- **avaliação da interface:** nem sempre o desenvolvedor se preocupa em realizar bons testes com sua interface. Uma GUI deve ser prototipada e cuidadosamente testada e avaliada várias vezes antes de ser entregue ao usuário final;
- **interface personalizada:** a maioria das aplicações são utilizadas por uma variedade de usuários com diferentes hábitos, preferências, estilos de trabalhar, etc. Uma interface que possa ser customizada tende a tornar o trabalho dos usuários mais produtivo;

- ➔ **consistência entre as interfaces:** o usuário precisa de uma interface uniforme e consistente para todas as aplicações que ele usa. Comandos similares devem possuir funções similares entre as diversas aplicações.

As atividades típicas no desenvolvimento de um projeto de interface gráfica com o usuário são ilustrados na figura 4.2 (Wood, 1998).

Figura 4.2: Atividades típicas no processo de desenvolvimento de uma interface com o usuário

Atualmente existe uma variedade muito grande de ferramentas disponíveis que podem ser usadas para construir interfaces gráficas. Essas ferramentas incluem processadores de comandos, sistemas de menus, sistemas de formulários, sistemas de entrada de dados, construtores de telas, etc. Essas ferramentas também ajudam o projetista de interface a especificar a aparência, checar a consistência e corretude do projeto e simular partes do projeto.

As ferramentas para projetar interfaces com o usuário ajudam a resolver os três maiores problemas relacionados à esta área nos sistemas aplicativos (Larson, 1992; Peddie, 1992):

- ➔ **a interface com o usuário pode ser facilmente construída e modificada:** através do uso de linguagens de alto nível, o tempo para implementar uma interface é menor, pois o projetista não precisa escrever e depurar linhas de código em baixo nível. Frequentemente o código gerado pela ferramenta de auxílio é mais confiável que o código criado por implementadores humanos. Além disso, algumas vezes a mesma interface de usuário pode

ser usada por mais de uma aplicação no sistema, economizando tempo dos projetistas;

- **diferentes interfaces com o usuário podem ser construídas para classes de usuários diferentes:** como visto, os usuários possuem estilos de trabalho, hábitos e preferências próprias. Tentar forçar o usuário a mudar suas características resultam em frustração e queda de produtividade do mesmo. O projeto de uma interface pode ser customizado para as diversas classes de usuários. Por exemplo, os usuários podem ser classificados em novatos e experientes dependendo de sua familiaridade com a computação: usuários novatos preferem menus e formulários em suas interfaces, enquanto que usuários experientes preferem usar sua aplicação através de teclas de atalho.
- **interfaces podem ser padronizadas para cada classe de usuário:** uma das maiores frustrações para o usuário é ter que aprender a usar diferentes interfaces para aplicações distintas ou então aprender comandos similares que podem fazer coisas diferentes. Interfaces com **aparência e impressão** (*look and feel*) similares são necessárias para cada classe de usuário. A *aparência* da interface se refere à maneira como o computador mostra as informações ao usuário; informações similares devem ser apresentadas usando símbolos e formatos similares e devem ser posicionados em localização familiar. A *impressão* da interface se refere à maneira como o usuário entra com suas informações e pedidos.

Além do mais, a ferramenta de programação para a interface gráfica deve possuir características como: recursos necessários ao desenvolvimento do projeto, facilidade de utilização/manutenção, ter bom desempenho, ser de domínio público, entre outras.

Algumas ferramentas satisfazem esses requisitos e quatro delas são alvo de estudo de caso. As ferramentas selecionadas foram: TCL/TK (Scriptics, 2001), o GTK+ (GTK, 2001), o QT (Troll-Tech, 2001), e o LessTif (Lesstif, 2001); por satisfarem às necessidades pertinentes ao desenvolvimento de interfaces gráficas. Tais ferramentas serão discutidas nas seções subseqüentes.

4.3.1 – TCL/TK

O TCL (*Tool Command Language*) é uma linguagem para escrever scripts, similar a outras linguagens de comando utilizadas em "shells" do UNIX. Como qualquer shell, o TCL permite a execução de outros programas e fornece variados recursos de programação, como variáveis, procedimentos e controle de fluxo. Possibilita assim a construção de scripts complexos que podem ser agregados a programas existentes para melhor atender às necessidades do programador.

O TCL é um interpretador usado para traduzir (em tempo de execução) um programa em linguagem de alto nível de um usuário para código de máquina. Ele consiste de um pacote de bibliotecas embutido em ferramentas (assim como editores, depuradores, etc.) como um interpretador de comandos básicos.

Resumidamente, o TCL provê: **a)** um analisador gramatical para uma linguagem de comandos textual; **b)** uma coleção de comandos utilitários embutidos; e **c)** uma interface C que serve para aumentar os comandos embutidos com comandos específicos à uma ferramenta. O TCL é particularmente interessante quando integrado com uma biblioteca de *widget*¹⁵ de um sistema de janelas: os recursos de programação em uma *widget* são aumentados pela inclusão de mecanismos para variáveis, procedimentos, expressões, etc. Isso permite que os usuários programem tanto a aparência quanto a ação dos *widgets* e oferecem um mecanismo de comunicação simples e poderoso entre programas que se interagem.

O Tk é uma ferramenta (*ToolKit*) que define comandos em TCL que permite a criação e manipulação de interfaces. TK é uma ferramenta para programação baseada em janelas. Ela foi projetada para o sistema de janelas X, mas já existem versões (a mais recente é a 8.33) para outros sistemas como o Windows NT e Windows 95/98/ME. Ele fornece um conjunto de comandos de TCL que criam e manipulam *widgets*, permitindo a criação de interfaces gráficas com o usuário.

Assim como outras ferramentas, o TK serve a dois propósitos gerais:

¹⁵ Um widget é um elemento de uma interface gráfica que tem aparência e comportamento particulares. Tipos de widgets são botões, barras de rolagem, menus e janelas de texto.

- **estrutura:** ele fornece uma estrutura que permite que aplicações sejam construídas a partir de vários pequenos elementos de interface (os *widgets*). Essa estrutura torna possível também o desenvolvimento de *widgets* independentes, que podem ser usados em muitas situações diferentes sem redesenhá-los.
- **conveniência:** o Tk fornece uma variedade de soluções pré-construídas para a maioria das necessidades comuns às aplicações dos sistemas de janelas. Muitas aplicações podem ser criadas pela construção de um ou dois novos tipos de *widgets* e combinando-os com os *widgets* existentes do Tk.

Tk e Tcl também simplificam os ambientes de janelas tornando uma única linguagem de comando em tempo de execução disponível em qualquer lugar. Há menos necessidade dos desenvolvedores de aplicações inventar linguagens de propósito específicas ou protocolos para manipular situações particulares: a aplicação pode usar meramente o Tcl. Ao invés de aprender diferentes comandos de linguagens para cada aplicação, um usuário precisa aprender apenas o Tcl. O usuário estará então capaz de programar qualquer aplicação baseada em Tk simplesmente aprendendo as primitivas da aplicação.

A popularidade do Tcl e Tk deve-se à sua capacidade e flexibilidade para se construir programas. Isso em razão de seu poder de programação de se construir ferramentas que podem ser intercambiadas. O uso do Tcl com o Tk significa que uma única linguagem de programação está disponível em tempo de execução para controlar todos os aspectos de interação de uma aplicação, desde sua **aparência e impressão** (*look and feel*) até suas funções.

4.3.2 – GTK+

O GTK+ (GIMP ToolKit) é uma biblioteca para criar interfaces gráficas com o usuário (GUI). Essa ferramenta é chamada de GIMP toolkit porque ela foi originariamente escrita para o desenvolvimento do Programa de Manipulação de Imagens GNU (*GNU Image Manipulation Program – GIMP*). Entretanto, atualmente o GTK+ é usado em uma grande variedade de projetos de *software*, incluindo o projeto GNOME (*GNU Network Object Model Environment*). O GTK+ é construído em cima do GDK (*GIMP Drawing Kit*) o qual é

basicamente um invólucro das funções de baixo nível para acessar as funções de manipulação de janelas (a Xlib no caso do sistema X Windows).

Essencialmente, o GTK+ é uma API (Application Programmers Interface) orientada a objetos. Embora escrita completamente em C, ela é implementada usando a idéia de classes e funções recursivas (ponteiros para funções). Isso quer dizer que todos os *widgets* são contidos em objetos (as variáveis da linguagem C) e são acessíveis através de seus métodos (as funções em C).

A popularidade e difusão do GTK+ se devem principalmente ao fato dele ser uma ferramenta GUI com código aberto e multi-plataforma. Isso permite que desenvolvedores criem aplicações gráficas com toda a liberdade possível. A aplicação criada com o GTK+ pode ser aberta, livre ou até mesmo comercial e proprietária sem a preocupação de se pagar por licenças ou direitos autorais (*royalties*).

O GTK+ pode ser definido em três partes distintas:

- **GLIB**: uma biblioteca de funções de utilidade geral, que pode ser usada independentemente do GTK+. Inclui funcionalidades adicionais como um alocador de memória com suporte estendido à depuração, funções para uso de listas ligadas, funções para aumentar a portabilidade do programa, entre outras. As substituições nas funções são feitas para aumentar a portabilidade do GTK+, porque algumas funções não estão disponíveis ou não são padrões para todos os UNIX. Algumas alterações visam também a melhoria de certos comandos, assim como o `g_malloc` que acrescenta os utilitários para depuração;
- **GDK**: um conjunto de funções gráficas, úteis para que o programador não precise lidar diretamente com a Xlib (biblioteca gráfica do Xfree86¹⁶);
- **GTK**: contém os *widgets* que podem ser utilizados pelo programador, além de toda estrutura para a criação de novos *widgets*.

¹⁶ Versão X distribuída gratuitamente para os sistemas UNIX baseados no processador 80386, como o Linux, por exemplo.

Apesar do GTK+ ser bastante difundido com a linguagem de programação C, existem ligações GTK+ para outras linguagens incluindo C++, Guile, Perl, Python, TOM, Ada95, Objective C, Free Pascal e Eiffel.

Atualmente, o GTK+ se encontra na versão 1.2 (estável) e tem como principais melhorias, as seguintes:

- suporte a threads, incluindo threads padrão POSIX e Solaris;
- novo suporte *DND-drag and drop* (arrastar e soltar) tanto para o protocolo MOTIF quanto XDND através de uma interface simples e unificada;
- internacionalização de diálogos padrão;
- maior suporte a *widgets* para texto;
- melhoria da funcionalidade de muitos *widgets* da versão 1.0 do GTK+.

4.3.3 – QT

O Qt (acrônimo para “cute” - atraente) é um produto da empresa norueguesa Troll Tech. Ele foi lançado em 1995 e é particularmente usado em larga escala no Linux, onde forma a base do popular ambiente gráfico KDE.

Qt é uma estrutura de aplicação em C++ para múltiplas plataformas. É implementado como uma biblioteca de classes e fornece uma rica API para desenvolvedores de aplicações. O Qt fornece uma grande variedade de funcionalidades geralmente úteis, mas o foco principal está no GUI.

Muitas ferramentas de GUI operam com dois tipos diferentes de elementos de GUI: os controles são os elementos básicos como botões e barras de rolagem, enquanto que os recipientes são os elementos que contêm os controles, como caixas de diálogo e janelas da aplicação. No Qt não há nenhuma diferença fundamental entre os controles e os recipientes: qualquer *widget* pode funcionar como um ou outro. Isso é expresso através de uma relação pai-filho: um *widget* que contém outro *widget* é chamado de pai do *widget* contido. Uma classe de *widget* provê uma API para ter acesso aos conteúdos do *widget*. Por exemplo, a classe de *widget* QPopupMenu provê um método `insertItem()` que acrescenta um novo item ao menu.

Um dos principais aspectos do Qt é o fato dele ser multi-plataforma, no sentido de que suas classes de bibliotecas são implementadas para vários sistemas operacionais e de janelas distintos. A API é idêntica para todas as plataformas. Isto significa que uma aplicação escrita com Qt em uma plataforma pode ser executada em outra simplesmente recompilando-a na nova plataforma, e unindo a aplicação com a biblioteca de Qt para a nova plataforma. Assim, com o Qt, produtores de *software* podem desenvolver e manter uma aplicação para múltiplas plataformas através do desenvolvimento e manutenção de um único código fonte base. A figura 4.3 ilustra a estrutura do Qt.

Atualmente, o Qt é implementado para dois grupos principais de sistemas operacionais¹⁷:

- **UNIX**: incluindo Linux, HP-UX, Sun Solaris, Digital UNIX, SGI Irix, IBM AIX, SCO UNIX, e outras variantes de BSD. A biblioteca do Qt é implementada usando as bibliotecas X11 do sistema de Janelas X;
- **Windows**: incluindo Windows 95, 98, e NT. A biblioteca do Qt é implementada usando o Windows GDI API, e usa o sistema de janelas Microsoft Windows.

Figura 4.3: A API do Qt

As principais características do Qt são:

- uso efetivo dos princípios da programação orientada a objetos. Por exemplo, todas as classes de *widget* (tanto padrão quanto personalizada) herdam da mesma classe *QWidget* básica. Assim, todas as *widgets* têm um grande conjunto de funções comuns e imediatamente utilizáveis;
- fornecer características não encontradas em todos os sistemas de janelas suportados. Por exemplo, o Qt deixa as aplicações rotacionar e transformar textos: das plataformas atualmente suportadas pelo Qt, apenas o Windows

NT provê esta funcionalidade nativamente, porém, o Qt implementa isso internamente para Windows 95/98 e X11;

- ➔ flexibilidade em tempo de execução. As aplicações baseadas em Qt não dependem de qualquer arquivo externo, de recurso externo ou similar. Todos os aspectos da GUI podem ser alterados ou adicionados em tempo de execução.

Atualmente, o Qt está na versão 2.0 e existe uma versão comercial do mesmo. A diferença entre a versão comercial e a livre, é que aquela possui suporte técnico, garantias pelo fabricante e em algumas circunstâncias, é necessário o pagamento de *royalties*.

4.3.4 – LessTif

O LessTif é uma implementação livre compatível com o Motif (principalmente com a versão 1.2). Assim como o Motif, o LessTif consiste de um conjunto de *widgets*, um gerenciador de janelas e uma implementação UIL.

A implementação UIL (User Interface Language) consiste de um compilador que traduz uma descrição textual de uma aplicação Motif em uma forma legível para o computador. Atualmente esse trabalho está em progresso embora não seja uma área de desenvolvimento muito ativa.

O gerenciador de janelas é responsável pela manipulação de janelas na tela e executa as seguintes operações: alocação, movimento, redimensionamento e controle de entrada de janelas.

’ Apesar de esforços estarem sendo despendidos para o desenvolvimento do gerenciador de janelas (denominado mwm), é o conjunto de *widgets* o objetivo primário e a área de principal desenvolvimento pela equipe responsável pelo LessTif.

Esse conjunto de *widgets*, denominado LessTif Toolkit, possibilita o desenvolvimento de interfaces gráficas com o usuário através da integração de suas bibliotecas com uma linguagem de programação usual, comumente a linguagem C. A

¹⁷ São planejadas implementações para outros sistemas operacionais gráficos.

primeira versão do LessTif (denominada 0.75) foi disponibilizada em dezembro de 1996 e a versão estável mais recente é a 0.92.26, que foi lançada em março de 2001. LessTif provê um *toolkit* de alto nível, e possui muitas características pertinentes ao desenvolvimento de GUI. Por exemplo, “cortar e colar”, navegação em arquivos e mecanismos de “arrastar e soltar”.

O LessTif fornece *widgets* para a maioria dos componentes de interfaces gráficas, incluindo botões, menus e barras de rolagens. LessTif disponibiliza também *widgets* que tem por função o controle de outros *widgets*, possibilitando assim o desenvolvimento de interfaces mais requintadas.

Atualmente, o LessTif está disponível para as seguintes plataformas:

- Linux;
- FreeBSD;
- Sun;
- OS/2 (requer XFree86);
- AIX;
- HP/UX
- Win NT
- BeOS.

4.4 - COMENTÁRIOS FINAIS

Interfaces com o usuário são importantes porque os usuários finais precisam interagir com as aplicações computacionais. Existem vários problemas no projeto de uma interface gráfica, como por exemplo, uma interface pobremente projetada, falta de uniformidade das interfaces entre as aplicações, falta de adaptação da interface às necessidades das várias classes de usuários, etc.

Para amenizar esses problemas, existem várias ferramentas que auxiliam o projetista de uma interface gráfica, entre elas, quatro foram apresentadas no decorrer desse capítulo.

No capítulo vindouro é apresentado o funcionamento da parte interna da interface gráfica do AMIGO.

capítulo 5

Funcionalidade da Interface Gráfica do AMIGO

Esse capítulo enfoca o funcionamento da interface gráfica e os componentes que possibilitam que as funções da referida interface sejam realizadas.

Como explanado no decorrer dessa dissertação, o objetivo deste projeto de mestrado é uma contribuição a um trabalho de doutorado desenvolvido no Grupo de Sistemas Distribuídos e Programação Concorrente do ICMC-USP. Por conta disso, os objetivos enfocados nesse capítulo seguem as especificações formuladas no referido trabalho (Souza, 2000).

5.1 – INTRODUÇÃO

Com o intuito de se obter maior flexibilidade, o desenvolvimento da Interface Gráfica do AMIGO foi dividido em duas partes: uma parte responsável pelos elementos visuais da camada superior e outra pelas funções da interface gráfica.

A figura 5.1 ilustra os dois módulos que compõem a Interface Gráfica da Camada Superior do AMIGO. Por meio dos elementos visuais da Camada Superior, o usuário interage com a Interface Gráfica fornecendo dados e informações. Tais informações são tratadas por funcionalidades específicas (programas executáveis independentes) que efetuam a gerência da Camada Superior.

Figura 5.1 – Módulos da Interface Gráfica

A primeira parte da interface, formada pelos elementos visuais, possibilita a interação do usuário com a Camada Superior e é denominada de **Front End**. A segunda parte é responsável pela execução das funções da interface e é denominada de **Back End**.

As necessidades para a implementação de cada uma destas partes são bastante diferentes. Desta forma, optou-se por adotar ferramentas diferentes em cada um dos casos: *Tool Command Language/ToolKit* (TCL/TK) para o desenvolvimento dos elementos visuais e a linguagem C, padrão ANSI, para o desenvolvimento da funcionalidade da Interface Gráfica.

Outra particularidade que motivou essa escolha é o fato da Interface Gráfica e seus elementos visuais (projetados em TCL/TK) serem desenvolvidos à parte da funcionalidade propriamente dita (esta projetada em ANSI C), criando-se desse modo, uma

certa modularidade e flexibilidade entre os elementos visuais da camada superior e a funcionalidade oferecida pela mesma.

Por intermédio da figura 5.2, pode-se visualizar o comportamento do *Front End* e do *Back End* da Interface Gráfica. Os elementos visuais da Interface Gráfica (*Front End*) solicitam serviços e fornecem subsídios (dados) para que as funcionalidades sejam executadas.

Cabe à funcionalidade da interface (*Back End*) a execução de serviços pertinentes à gerência da Camada Superior do AMIGO, tais como o cadastro e alteração dos arquivos de dados e a manipulação dos arquivos de configuração. Eventos são disparados através de comandos TCL/TK a partir do *Front End*. Tais comandos são tratados em forma de argumentos (argv, argc da linguagem C) pelo *Back End* da interface, que por sua vez, executa a funcionalidade solicitada. Após sua execução, o *Back End* devolve ao *Front End* informações sobre a atividade realizada (sucesso da operação, dados atualizados, etc.).

Figura 5.2 – Processo de Funcionamento da Interface Gráfica

Neste capítulo tem-se a descrição da funcionalidade da interface gráfica (*back end*) e a justificativa do uso da linguagem C para o desenvolvimento dessa funcionalidade. O próximo capítulo descreve os componentes visuais que compõem o módulo visual da Interface Gráfica (*front end*).

5.2 – FUNCIONAMENTO DA PARTE INTERNA DA INTERFACE GRÁFICA

A Interface Gráfica do AMIGO possui um módulo visual que contém os elementos gráficos da Camada Superior. Essa camada gráfica, ou seja, o *Front End* da aplicação é responsável principalmente pela interação do usuário com a Camada Superior do AMIGO.

Uma vez que o usuário interage com a interface gráfica, cabe a ela a evocação dos programas que realmente executarão as funcionalidades da Camada Superior do AMIGO. Tais funcionalidades são executadas por programas desenvolvidos em linguagem de programação C. O conjunto dessas funcionalidades corresponde ao *Back End* da Camada Superior.

Sempre que algum evento que esteja associado à configuração da Camada superior é disparado, uma aplicação é executada e os arquivos de configuração são acessados pela respectiva aplicação. É responsabilidade da aplicação a atualização do respectivo arquivo de configuração e o retorno de valores ao *Front End* da Camada Superior.

Para ilustrar essa lógica de funcionamento, considere a inclusão de uma política de escalonamento fictícia que possui os seguintes dados: **código:** *1*; **nome:** *Uma Nova Política*; **executável:** */politica*; **características especiais:** *comunicacao*. Os dados são inseridos através de entradas na Interface Gráfica, conforme ilustrado na figura 5.3.

Figura 5.3 – exemplo

Quando o usuário dispara o evento para incluir o registro (no caso, um clique do *mouse* no botão **Inclui**), um comando em TCL/TK associado ao evento é disparado pelo *front end* (ver seta 1 na figura 5.2) com a seguinte sintaxe: `set retorno [exec ./pol inclui $caminho $cp_cod $cp_nome $cp_exe $cp_ce]`. Onde,

retorno: variável que receberá o resultado da operação. Em caso de sucesso, a *string* “INCLUÍDO” é associada à variável *retorno*;

`exec ./pol`: responsável pela execução da aplicação. Nesse exemplo, cabe ao arquivo executável *pol*, o cadastro e manutenção das políticas de escalonamento no respectivo arquivo de configuração;

inclui, *\$caminho*, *\$cp_cod*, *\$cp_nome*, *\$cp_exe*, *\$cp_ce*: são parâmetros passados à aplicação. Nesse caso, *inclui* é a funcionalidade a ser efetuada, *\$caminho* é uma variável que representa o diretório padrão e os quatro últimos argumentos representam os dados inseridos pelo usuário: código, nome, executável e características especiais da política a ser incluída.

Cabe ao programa desenvolvido em C e que faz parte do *back end* (nesse exemplo, o executável *pol*) o tratamento dos argumentos, a execução da funcionalidade e o retorno do que foi efetuado. No cadastro de uma nova política (caso considerado neste exemplo) o arquivo *policies.dad* será alterado pelo *back end* (seta 4 da figura 5.2) e o retorno indicando se a operação teve sucesso é enviado para o *front end* (seta 2 da figura 5.2).

Outros eventos são disparados com sintaxe muito semelhante à essa, variando apenas alguns detalhes. Se o intuito for a exclusão de um registro, o processo seria o seguinte: `set retorno [exec ./pol exclui $caminho $codigo]`. Onde: *retorno* poderia receber a informação

de que o registro foi excluído ou que o arquivo está vazio; *pol* é o programa executável responsável pelas alterações das políticas de escalonamento; *exclui* é a funcionalidade solicitada; *\$caminho* é o diretório padrão; e, *\$codigo* é código do registro a ser excluído.

Finalizando os exemplos, de modo a explicar como seria um caso em que a Interface Gráfica é atualizada pela funcionalidade da mesma, considere o seguinte: o arquivo que armazena as políticas de escalonamento possui vários registros incluídos. Em um instante qualquer o usuário está no segundo registro (que tem, por exemplo, o código número 2) e clica com o *mouse* no botão **Anterior**. Seguindo a sintaxe já explicada, tal evento dispararia o seguinte comando: *set retorno [exec ./pol btn_ant \$caminho \$codigo]*. *Btn_ant* representa a funcionalidade solicitada ao executável *pol*; e *\$codigo* armazena o número 2. A aplicação deve tratar os argumentos e retornar os dados do registro anterior, no caso o registro 1 (pode-se observar como seria esse comportamento em um console de terminal através da figura 5.4):

```
1
Uma Nova Política
/politica
comunicação
```

```
[arion@lasdpc04 ~/amigo]$ ./pol btn_ant /tmp/amigo/ 2
1
Uma Nova Política
/politica
comunicação
[arion@lasdpc04 ~/amigo]$
```

Figura 5.4 – Console da aplicação com o retorno da funcionalidade

Tais dados são simples cadeias de caracteres (*strings*) que são informadas pelo programa executável (no caso da linguagem C, um mero *printf*). Essas *strings* são armazenadas em forma de lista na variável *retorno*. De posse desses valores, a Interface Gráfica atualiza seus componentes, mostrando o conteúdo do registro anterior ao usuário, e dessa forma finalizando o processo.

Por intermédio dos exemplos citados, é possível perceber uma vantagem da modularidade da Interface Gráfica mencionada no capítulo anterior: as funcionalidades podem ser desenvolvidas em qualquer outra linguagem de programação, desde que desempenhem tais funções do modo descrito acima.

5.3 – A FERRAMENTA ADOTADA

Para o desenvolvimento dos componentes que executam as tarefas de gerenciamento da Camada Superior, optou-se pela linguagem de programação C, mais especificamente o padrão ANSI.

Dennis Ritchie desenvolveu a versão original da linguagem C no começo dos anos setenta. No final dessa década e início dos anos 80 haviam sido desenvolvidas centenas de implementações C para os mais variados tipos de arquiteturas, desde microcomputadores até mainframes. Por conta disso tornou-se necessária a padronização dessa linguagem (Plauser, 1994).

O *American National Standards Institute*, ANSI, é o órgão responsável pela padronização de linguagens de programação nos Estados Unidos e autorizou um comitê a desenvolver o padrão para o C. O *International Standards Institute*, ISO, possui a mesma responsabilidade na área internacional. O ISO formou um comitê técnico para proceder à revisão e ampliação do trabalho realizado pelo ANSI. Hoje, o ISO adota um padrão para o C que é essencialmente o mesmo do ANSI.

O propósito do padrão ANSI é promover portabilidade, confiança, manutenção e eficiência na execução de programas em linguagem C em uma variedade de máquinas (Kelley & Pohl, 1995).

Vários fatores motivaram a escolha dessa linguagem, entre elas, algumas merecem destaque:

- confere os recursos necessários ao desenvolvimento do projeto;
- bom desempenho;
- o fato de existirem versões livres do compilador tanto para MS-Windows (o *Bloodshed Dev-C++* é um exemplo) quanto Linux (o *GNU CC – gcc* – por exemplo);
- *C é uma linguagem de programação de finalidade geral que permite a economia de expressão, fluxos de controle e estruturas de dados e um rico conjunto de operadores* (Ritchie & Vernighan, 1987; Plauger, 1995; Kelley & Pohl, 1995).

Visto que os programas foram desenvolvidas seguindo o padrão ANSI - e sem chamadas de sistema específicas - o mesmo código-fonte poderá ser compilado tanto na plataforma MS-Windows quanto em Linux (as duas plataformas utilizadas no desenvolvimento desse projeto). Não há necessidade de alteração do código para o funcionamento das aplicações.

5.4 – COMPONENTES QUE EXECUTAM AS FUNCIONALIDADES

A funcionalidade da Camada Superior foi dividida - para depuração mais eficiente dos fontes, que ficaram, em média, com pouco mais de 300 linhas cada - em sete programas (arquivos executáveis), todos seguindo o processo de funcionamento mencionado na seção 6.2.

Esses programas são compilados a partir dos seguintes fontes em linguagem de programação C: *car.c*, *cla.c*, *har.c*, *ben.c*, *pol.c*, *met.c* e *polcfg.c*. Cada um destes programas é responsável pela execução de uma ou mais funcionalidades da Camada Superior do AMIGO.

A partir de comandos recebidos do *front end*, estes programas irão manipular os arquivos de configuração e de dados, que fazem a integração entre as camadas inferior e superior do AMIGO. Esses arquivos são: *classoft.dad*, *caract.dad*, *hardware.dad*, *bench.dad*, *policies.dad*, *policy.cfg*, *metrics.dad*, que foram definidos na seção 3.3.1 do capítulo 3.

Nas seções subseqüentes são apresentadas as funções que são tratadas em cada um desses programas. As funções são ajustadas às necessidades da Interface Gráfica.

5.4.1 – CAR.C

Esse programa é responsável pela definição das descrições das características que são necessárias para o cadastro das classes de *software*. O arquivo *caract.dad* é o repositório dessas informações. Esse arquivo possui dois campos (*código* e *nome*) e armazena descrições como *CPU-Bound*, *IO-Bound*, *Aplicação em Lote*, etc.

Esse programa obedece a seguinte sintaxe: *car* <função> <caminho> <argumento(s)>. Onde *car* é o executável dessa aplicação; *caminho* é o diretório onde está armazenado o arquivo de configuração, *argumento(s)* são dados adicionais enviados pelo *front end*. Tais argumentos podem variar de um simples código (utilizado para excluir um registro, por exemplo) até uma lista completa de informações (utilizada, por exemplo, para incluir um registro). Para algumas funções, não há necessidade de se passar argumentos (quando a função *init* é solicitada, por exemplo). Todos os programas obedecem essa filosofia. Finalmente, *função* é um parâmetro que é tratado pela funcionalidade e pode ser um dos procedimentos a seguir:

init: se o arquivo de dados não estiver vazio, retorna o primeiro registro. Caso contrário associa a *string* “VAZIO” ao retorno;

obtem_cod: gera um código automático e retorna o próximo código disponível para ser associado a um registro quando da sua inclusão;

inclui: adiciona um registro ao arquivo *caract.dad* baseado nas informações contidas em *argumento(s)* (descrito na sintaxe acima);

exclui: exclui um registro do arquivo *caract.dad* (o código do registro é especificado em *argumento(s)*). Se o arquivo estiver vazio, a *string* “VAZIO” é associada ao retorno;

todos: lista todos os registros;

btn_ant: retorna o registro anterior ao do código solicitado em *argumento(s)*.

Caso contrário associa a *string* “PRIMEIRO” ao retorno;

btn_pro: retorna o registro seguinte ao do código solicitado em *argumento(s)*.

Caso contrário associa a *string* “ULTIMO” ao retorno.

5.4.2 – CLA.C

Compilado a partir do fonte *cla.c*, o cadastro das classes de *software* (armazenado no arquivo *classoft.dad*) é composto pelas seguintes informações: código da classe, nome da classe, características da classe (uma ou um conjunto de características).

Esse programa obedece a seguinte sintaxe: *cla* <função> <caminho> <argumento(s)>. Onde *cla* é o executável dessa aplicação; *caminho* é o diretório onde está armazenado o arquivo de configuração, *argumento(s)* são dados adicionais enviados pelo *front end*; e *função* é um parâmetro que é tratado pela funcionalidade e pode ser um dos procedimentos a seguir:

init: se o arquivo de dados não estiver vazio, retorna o primeiro registro. Caso contrário associa a *string* “VAZIO” ao retorno;

obtem_cod: gera um código automático e retorna o próximo código disponível para ser associado a um registro quando da sua inclusão;

inclui: adiciona um registro baseado nas informações contidas no parâmetro *argumento(s)* ao arquivo *classoft.dad*;

exclui: exclui um registro do arquivo *classoft.dad* (o código do registro a ser excluído está contido em *argumento(s)*). Se o arquivo estiver vazio, a *string* “VAZIO” é associada ao retorno;

todos: lista todos os registros;

btn_ant: retorna o registro anterior do código enviado como parâmetro. Caso contrário associa a *string* “PRIMEIRO” ao retorno;

btn_pro: retorna o registro seguinte do código enviado como parâmetro. Caso contrário associa a *string* “ULTIMO” ao retorno;

nomes: esse procedimento lista os nomes das classes de software cadastradas. Esses nomes são necessários no cadastro de aplicações, onde é preciso associar uma classe de *software* à aplicação.

5.4.3 – HAR.C

Permite a inclusão e o acesso das informações sobre a plataforma computacional no arquivo *hardware.dad*. As informações armazenadas nesse arquivo são: código da máquina, nome da máquina, arquitetura, sistema operacional, número de processadores, quantia de memória principal disponível, existência ou não de acesso a disco e características especiais.

Esse programa segue a seguinte sintaxe: *har* <função> <caminho> <argumento(s)>. Onde *har* é o executável dessa aplicação; *caminho* é o diretório onde está armazenado o arquivo de configuração, *argumento(s)* são dados adicionais enviados pelo *front end*; e *função* é um parâmetro que é tratado pela funcionalidade e pode ser um dos procedimentos a seguir:

init: se o arquivo de dados não estiver vazio, retorna o primeiro registro. Caso contrário associa a *string* “VAZIO” ao retorno;

obtem_cod: gera um código automático e retorna o próximo código disponível para ser associado a um registro quando da sua inclusão;

inclui: adiciona um registro ao arquivo *hardware.dad* (as informações do registro estão contidas em *argumento(s)*);

exclui: exclui um registro do arquivo *hardware.dad* (*argumento(s)* contém o código do registro a ser excluído). Se o arquivo estiver vazio, a *string* “VAZIO” é associada ao retorno;

todos: lista todos os registros;

btn_ant: retorna o registro anterior do código contido em *argumento(s)*. Caso contrário associa a *string* “PRIMEIRO” ao retorno;

btn_pro: retorna o registro seguinte do código contido em *argumento(s)*. Caso contrário associa a *string* “ULTIMO” ao retorno.

5.4.4 – BEN.C

Esse programa é gerado a partir do arquivo *ben.c* e é responsável pelo cadastro de *benchmarks*. Essa aplicação armazena no arquivo *bench.dad* as seguintes informações sobre os *benchmarks*: código, nome, nome do *benchmark* executável (com o caminho completo) e principais características.

O referido programa obedece a seguinte sintaxe: **ben** <função> <caminho> <argumento(s)>. Onde **ben** é o executável dessa aplicação; **caminho** é o diretório onde está armazenado o arquivo de configuração, **argumento(s)** são dados adicionais enviados pelo *front end*; e **função** é um parâmetro que é tratado pela funcionalidade e pode ser um dos procedimentos a seguir:

init: se o arquivo de dados não estiver vazio, retorna o primeiro registro. Caso contrário associa a *string* “VAZIO” ao retorno;

obtem_cod: gera um código automático e retorna o próximo código disponível para ser associado a um registro quando da sua inclusão;

inclui: adiciona um registro baseado nas informações enviadas em **argumento(s)** ao arquivo *bench.dad*;

exclui: exclui o registro (cujo código está contido em **argumento(s)**) do arquivo *bench.dad*. Se o arquivo estiver vazio, a *string* “VAZIO” é associada ao retorno;

todos: lista todos os registros;

btn_ant: retorna o registro anterior do código solicitado (contido em **argumento(s)**). Caso contrário associa a *string* “PRIMEIRO” ao retorno;

btn_pro: retorna o registro seguinte do código solicitado (contido em **argumento(s)**). Caso contrário associa a *string* “ULTIMO” ao retorno;

exec_b: esse procedimento é evocado quando da execução dos *benchmarks*. O retorno é uma lista dos executáveis dos *benchmarks* cadastrados.

5.4.5 – MET.C

Para efetuar o monitoramento e dessa forma avaliar se os objetivos do escalonamento estão sendo atingidos, é necessário o uso de métricas que quantifiquem tais objetivos. O cadastro de métricas armazena no arquivo *metrics.dad* as seguintes informações:

código, nome, nome do arquivo executável (com o caminho completo), nome do arquivo que irá armazenar os valores obtidos (com o caminho completo) e características específicas da métrica.

Esse programa obedece a sintaxe a seguir: **met** <função> <caminho> <argumento(s)>. Onde **met** é o executável dessa aplicação; **caminho** é o diretório onde está armazenado o arquivo de configuração, **argumento(s)** são dados adicionais enviados pelo *front end*; e **função** é um parâmetro que é tratado pela funcionalidade e pode ser um dos procedimentos a seguir:

init: se o arquivo de dados não estiver vazio, retorna o primeiro registro. Caso contrário associa a *string* “VAZIO” ao retorno;

obtem_cod: gera um código automático e retorna o próximo código disponível para ser associado a um registro quando da sua inclusão;

inclui: adiciona um registro baseado nas informações enviadas em **argumento(s)** ao arquivo *metrics.dad*;

exclui: exclui um registro do arquivo *metrics.dad*. Se o arquivo estiver vazio, a *string* “VAZIO” é associada ao retorno. O registro a ser excluído é baseado no código enviado em **argumento(s)**;

todos: lista todos os registros;

btn_ant: retorna o registro anterior baseado em um código enviado em **argumento(s)**. Caso contrário associa a *string* “PRIMEIRO” ao retorno;

btn_pro: retorna o registro posterior baseado em um código enviado em **argumento(s)**. Caso contrário associa a *string* “ULTIMO” ao retorno;

exec_b: esse procedimento é evocado quando da execução do início/término do monitoramento. O retorno é uma lista dos executáveis das métricas cadastradas;

nomes: esse procedimento lista as descrições dos nomes das métricas cadastradas. Esses nomes são disponibilizados para o usuário selecionar a métrica adotada quando o monitoramento for efetuado.

5.4.6 – POL.C

Funcionalidade responsável pelo cadastro de políticas de escalonamento (armazenados no arquivo *policies.dad*). Esse arquivo armazena as seguintes informações:

código, nome da política, nome do arquivo executável (com o caminho completo) e as principais características da política.

Esse programa segue a seguinte sintaxe: *pol* <função> <caminho> <argumento(s)>. Onde *pol* é o executável dessa aplicação; *caminho* é o diretório onde está armazenado o arquivo de configuração, *argumento(s)* são dados adicionais enviados pelo *front end*; e *função* é um parâmetro que é tratado pela funcionalidade e pode ser um dos procedimentos a seguir:

init: se o arquivo de dados não estiver vazio, retorna o primeiro registro. Caso contrário associa a *string* “VAZIO” ao retorno;

obtem_cod: gera um código automático e retorna o próximo código disponível para ser associado a um registro quando da sua inclusão;

inclui: adiciona um registro com as informações contidas em *argumento(s)* ao arquivo *policies.dad*;

exclui: exclui um registro (baseado no código enviado em *argumento(s)*) do arquivo *policies.dad*. Se o arquivo estiver vazio, a *string* “VAZIO” é associada ao retorno;

todos: lista todos os registros;

btn_ant: retorna os dados do registro anterior (baseado em um código enviado em *argumento(s)*). Caso contrário associa a *string* “PRIMEIRO” ao retorno;

btn_pro: retorna os dados registro seguinte (baseado em um código enviado em *argumento(s)*). Caso contrário associa a *string* “ULTIMO” ao retorno;

exec_b: esse procedimento providencia uma lista com os executáveis das políticas de escalonamento cadastradas. Essa lista é utilizada, por exemplo, quando uma política reserva é definida. Esse item (política reserva) armazena o caminho de uma política de escalonamento que deve ser executada quando não existam políticas independente ou específicas ativas;

nomes: esse procedimento lista as descrições dos nomes das políticas de escalonamento cadastradas. Essa lista é necessária quando o usuário associa uma classe de software a uma política, por exemplo;

cod_nome: retorna o nome da política de escalonamento de um dado código (o referido código está contido em *argumento(s)*);

cod_cam: retorna o caminho do executável da política de escalonamento de um dado código (esse código está contido em *argumento(s)*). Em algumas circunstâncias, quando

o usuário seleciona um nome de política de escalonamento, é o caminho do executável dessa política que realmente interessa à funcionalidade.

5.4.7 – POLCFG.C

Essa aplicação é a principal responsável pela configuração da Camada Superior do AMIGO. Ela tem por objetivo a manutenção do arquivo *policy.cfg* (a figura 5.5 ilustra a estrutura do arquivo *policy.cfg*).

Figura 5.5 – arquivo *policy.cfg* (Souza, 2000)

A referida aplicação segue a seguinte sintaxe: **polcfg** <item> <caminho completo> <acao> <argumentos>. Onde **polcfg** é o executável da aplicação; **item** é o item a ser configurado/gerenciado; **caminho** é a localização do arquivo de configuração, **ação** se resume a duas possibilidades (atualizar o item ou visualizar sua configuração); e **argumentos** são os dados a serem atualizados no item quando a ação for uma atualização.

Como dito no parágrafo acima, a ação pode ser um argumento com duas finalidades:

m: para mostrar/visualizar os dados de algum item do arquivo *policy.cfg*; e

a: para atualizar/gerenciar o conteúdo de algum item com os novos argumentos passados pela própria sintaxe.

Os itens que podem ser atualizados e visualizados do arquivo *policy.cfg* são quatro:

r - sigla para o item *[reserve]*. Nesse item é armazenado o caminho completo da política de escalonamento que será definida como reserva. O AMIGO procura por essa política quando não existem política independente e políticas específicas definidas. Essa política reserva tem por objetivo possibilitar a atividade de escalonamento de processos mesmo que não existam políticas de escalonamento ativas;

c - sigla para o item *[config]*. Esse item armazena o número 0 e 1 para, respectivamente, definir como inativa e ativa as seguintes opções do item *[config]*: mais de uma política ativa ao mesmo tempo, políticas sob demanda, mudar políticas de escalonamento em tempo de execução, utilizar cachê no AMIGOD, troca de mensagens diretamente entre as políticas e o uso de um arquivo de *profile* (nessa opção, 0 define pelo não uso de um arquivo de *profile* e qualquer outro número representa o período de tempo que informações coletadas sobre o escalonamento são armazenadas no arquivo de *profile*); Esse item é atualizado através da opção **Configura Uso de Políticas** presente no **menu Pol. de Escalonamento**. Cada uma dessas opções é explicada na seção 6.4.3 (item “*Configura Uso de Políticas*”) do capítulo 6.

p - sigla para o item *[policies]*. Esse item define o relacionamento entre as classes de *software* e políticas de escalonamento além de especificar se tal relacionamento está ativo e de que maneira ele está ativado. Cada relacionamento é formado por quatro parâmetros: o código da classe de *software*, o código da política de escalonamento, o caminho completo da política de escalonamento; e mais um quarto parâmetro que pode receber o valor **independent** para especificar que essa é a política de escalonamento *independente*, **active** para especificar que essa política é uma política específica ou um espaço em branco para especificar que existe uma relação entre classe de *software* e política de escalonamento, porém não ativa.

a – sigla para o item *[applications]*. Esse item armazena as informações sobre a aplicação a ser escalonada: o nome da aplicação, o nome do executável (ou executáveis) da aplicação e o código da classe de *software* que a aplicação pertence.

5.5 - COMENTÁRIOS FINAIS

Como visto no decorrer do capítulo, a adoção da linguagem C padrão ANSI fornece uma certa portabilidade à aplicação, inerente à própria padronização da referida linguagem.

Entretanto, como a Camada Superior teve os seus componentes visuais desenvolvidos distintamente da funcionalidade propriamente dita, nada impede que a funcionalidade atualmente desenvolvida em C ou novas funções sejam feitas em outra linguagem de programação. Basta apenas que tal funcionalidade retorne valores em forma de cadeia de caracteres que serão capturados pelos módulos visuais.

Do mesmo modo, a Interface Gráfica pode ser desenvolvida utilizando outras ferramentas. Nesse caso, a Interface Gráfica terá que tratar da interação do usuário e considerar os retornos das aplicações responsáveis pela funcionalidade da Camada Superior.

O próximo capítulo trata dos componentes que formam o módulo visual da Interface Gráfica do AMIGO.

capítulo 6

Módulo Visual da Interface Gráfica do AMIGO

Neste capítulo são apresentados o projeto e a implementação da Interface Gráfica no que concerne aos elementos visuais da Camada Superior do AMIGO. Serão abordados, ainda, aspectos relacionados ao desenvolvimento de tal interface, tais como a ferramenta adotada e a disposição dos componentes visuais que formam a interface gráfica.

6.1 – INTRODUÇÃO

Como descrito no capítulo anterior, a funcionalidade da Interface Gráfica foi projetada em linguagem de programação C e optou-se pela ferramenta TCL/TK para o desenvolvimento dos elementos que compõem o módulo visual da interface gráfica.

Os principais fatores que motivaram a adoção de ferramentas distintas para o desenvolvimento dos elementos visuais e da funcionalidade da Interface Gráfica foram os seguintes:

- a) modularização entre a parte que provê a interação com o usuário e a parte que efetivamente executa o gerenciamento da Camada Superior. Novos componentes podem ser desenvolvidos por qualquer outra ferramenta que possua a capacidade necessária para tal;

- b) através do TCL/TK é possível criar *widgets* com maior eficiência que o C. De um modo simples o TCL/TK permite a especificação e disposição de vários componentes que formam uma interface gráfica;
- c) a linguagem de programação C fornece mecanismos mais eficientes que linguagens interpretadas para a manipulação e acesso de arquivos.
- d) a adoção do padrão ANSI permite maior portabilidade entre plataformas computacionais distintas, visto que chamadas específicas de sistema não são utilizadas.

A próxima seção descreve a ferramenta TCL/TK e apresenta a justificativa de sua escolha.

6.2 – A FERRAMENTA ADOTADA

A adoção da ferramenta TCL/TK (*Tool Command Language/ToolKit*) para o desenvolvimento do módulo visual da Camada Superior do AMIGO baseou-se nas características apresentadas na pesquisa bibliográfica descritas no capítulo anterior.

As vantagens e desvantagens da utilização do TCL/TK e a razão de sua escolha são descritos a seguir.

Apesar da ferramenta escolhida apresentar algumas desvantagens (que são comuns às linguagens interpretadas), tais como a geração de aplicações com maior tempo de execução (quando comparadas às linguagens compiladas) e a maior dificuldade de depurar aplicações interpretadas (particularmente no desenvolvimento deste projeto, que em TCL/TK possui cerca de 3.000 linhas), aspectos positivos motivaram sua escolha. Dentre esses aspectos, alguns merecem ser citados:

- *o TCL/TK é livre*: é possível obter os códigos fonte de várias versões do TCL/TK por diversos métodos (*ftp*, *http*, distribuições em CD-ROM, etc.), assim como as versões binárias para as plataformas Windows e Macintosh (Scriptics, 2001);

- **linguagem de scripts de alto nível:** através do TCL/TK é possível desenvolver aplicações elaboradas com menos código, quando comparada às linguagens compiladas;
- **multiplataformas:** aplicações TCL/TK são, em sua grande maioria, independentes de plataforma. Existem versões para UNIX (Linux, inclusive), Windows (NT/95/98/ME) e Macintosh.
- **TCL/TK é interpretada:** o código pode ser executado diretamente sem a necessidade de compilação e *link-edição* (embora compiladores Tcl estejam disponíveis);
- **TCL/TK pode ser estendido:** o programador pode incluir seus próprios comandos (por exemplo, em C ou até mesmo em Tcl).

As características do TCL/TK justificam sua escolha, uma vez que é desejável que a ferramenta para o desenvolvimento da Interface Gráfica forneça os recursos necessários ao desenvolvimento do projeto, como facilidade de utilização, bom desempenho e o fato de ser de domínio público.

A seção subsequente descreve os componentes que formam os elementos visuais da Interface Gráfica, desenvolvidos em TCL/TK.

6.3 – COMPONENTES DO MÓDULO VISUAL

Todos os componentes visuais da Camada Superior do AMIGO foram projetados de modo a prover a interação com o usuário. Essa interação se dá através de entradas que o usuário fornece ao módulo visual. Por outro lado, o módulo visual, através de suas janelas e elementos visuais evoca as aplicações que fazem o gerenciamento e atualização do ambiente AMIGO, propriamente dito.

Em algumas circunstâncias, a atualização dos elementos visuais se dá através das funcionalidades da interface e não pelo usuário. Nesses casos, os retornos das funcionalidades servem como fontes de dados que são recebidos pelos componentes visuais.

É interessante comentar que visando explorar a característica do TCL/TK ser multiplataforma, o projeto desenvolvido neste trabalho de mestrado foi produzido para funcionar tanto em ambiente MS-Windows quanto ambiente UNIX. No momento, no caso do ambiente Windows, não é possível utilizar o trabalho efetivamente, pois não existe implementação da Camada Inferior do AMIGO para esse ambiente de janelas.

Quando o ambiente gráfico é iniciado, este, através de funções TCL verifica qual é o tipo de sistema de arquivos que está sendo utilizado. Quando da definição do caminho, compete ao usuário a responsabilidade de obedecer à estrutura de arquivos do respectivo Sistema Operacional: para o Linux, ponto de montagem e / (barra normal) e para Windows, unidade de disco e \ (barra invertida).

O diretório padrão, ilustrado na figura 6.1, é o primeiro item da Camada Superior a ser configurado. Ele é configurado antes mesmo de se ter acesso às opções de gerenciamento da interface gráfica do AMIGO.

Figura 6.1 – Diretório Padrão

O diretório padrão serve como local de armazenamento dos arquivos de configuração e de dados gerados pela Camada Superior. Tais arquivos são acessados e utilizados pela Camada Inferior que executa o escalonamento baseado nas informações contidas em tais arquivos.

Nas seções subsequentes deste capítulo serão tratadas as partes gráficas que compõem o módulo visual acompanhadas de suas respectivas imagens capturadas no instante

de sua execução. Esse capítulo não tem o intuito de detalhar cada opção, pois mais detalhes sobre os itens da Camada Superior foram descritos no capítulo 3.

6.4 – MENU PRINCIPAL

A partir da tela principal, apresentada na figura 6.2, o usuário tem acesso a todas as opções de gerenciamento da Camada Superior do AMIGO. As opções presentes nessa tela encontram-se detalhadas na seção 3.3.1, subseções 3.3.1.1 até 3.3.1.5, do capítulo 3. A partir dessa tela o usuário poderá definir políticas de escalonamento, cadastrar plataformas e classes de software, determinar ações, dentre outras operações.

Figura 6.2 – Menu Principal

As opções disponíveis seguem a estrutura a seguir:

CLASSES DE SOFTWARE:

- Cadastro de Classes;
- Cadastro de Aplicações.

PLATAFORMAS COMPUTACIONAIS:

- Cadastro de Hardware;
- Detecta Hardware;
- Cadastro de *Benchmarks*;
- Executa *Benchmarks*.

POLÍTICAS DE ESCALONAMENTO:

- Cadastro de Políticas;
- Relaciona Classes & Políticas;

- Configura Uso de Políticas;
- Seleciona Políticas
 - reserva;
 - independente;
 - específicas.

MONITORAMENTO:

- Cadastro de Métricas;
- Início/Término.

SIMULAÇÃO:

- Aplicações;
- Plataformas;
- Políticas;
- Simula.

AMIGO:

- Responsável por algumas preferências do ambiente, entre elas o diretório padrão citado anteriormente.

SAI:

- Finaliza o ambiente de gerenciamento.

As próximas seções descrevem os detalhes de cada uma dessas opções. Também são apresentadas as telas dessas opções e algumas figuras que explicam as relações entre *front end* e *back end* (por apresentarem comportamentos semelhantes, nem todos os relacionamentos são ilustrados).

6.4.1 – Classes de Software

Através desse item é possível a inclusão de classes de software (tipos de aplicações afins, tais como CPU-Bound, I/O-Bound, interativa, batch, entre outras) e das próprias aplicações a serem escalonadas, sendo que depois, ambas serão relacionadas às políticas de escalonamento (Souza, 2000). Através desta opção tem-se acesso ao Cadastro de Classes e ao Cadastro de Aplicações.

• Cadastro de Classes

O cadastro de classes de software tem por objetivo flexibilizar a caracterização dos tipos de aplicação (isto é, não há classes de software fixas) que serão escalonados e viabiliza de maneira prática e objetiva a inclusão de qualquer classe de *software* que venha a

ser necessária. Esta opção é responsável pela manutenção do arquivo *classoft.dad* que mantém informações sobre as classes de software e é descrito na seção 3.3.1.1 do capítulo 3.

Como ilustrado na figura 6.3, uma classe de *software* possui um código, um nome e as características mais importantes da classe. As características de cada classe de *software* podem ser adicionadas ou removidas a partir de uma lista de características previamente cadastradas e das características da classe em questão, respectivamente. Tal lista pode ser alterada a partir da opção Descrição de Características contida no item Classes de *Software*.

A navegação entre as classes de software cadastradas é possível através dos botões *Anterior* e *Próximo*, que atualizam os campos, respectivamente, com o registro imediatamente anterior e posterior da classe visualizada. Estes botões evocam os procedimentos *btn_ant* e *btn_pro* do programa *cla.c* (seção 5.4.2 do capítulo 5).

A exclusão de registros cadastrados e a inclusão de novos registros são realizadas através dos botões *Exclui* e *Inclui*, respectivamente. Os referidos botões chamam os procedimentos *exclui* e *inclui* do programa *cla.c*. O botão *Atualiza* é responsável por atualizar eventuais alterações que o usuário venha a fazer em registros já cadastrados. Para que a atualização seja efetuada, os procedimentos *exclui* e *inclui* do programa *cla.c* são executados (o registro a ser atualizado é apagado para depois ser incluído com os novos dados). Finalmente, o botão *Sair* fecha a janela, retornando ao Menu Principal.

Figura 6.3 – Cadastro de Classes

A figura 6.4 ilustra como estão relacionados o *front end* e o *back end* do cadastro de classes. O fluxo A da ilustração corresponde às solicitações ou atualizações do *front end* para o *back end*. O fluxo B da ilustração são os retornos do *back end* ao *front end*. Esses retornos fornecem subsídios para que o *front end* seja atualizado. O fluxo C da ilustração corresponde à efetivação das atualizações ou consultas que o *back end* realiza do arquivo *classoft.dad*.

Figura 6.4 – Fluxo de informações do cadastro de Classes de software

• Cadastro de Aplicações

Esta opção relaciona as classes de software disponíveis às aplicações escalonadas. Este vínculo não é obrigatório, mas quando feito, possibilita um escalonamento com mais informações.

Através da figura 6.5, pode-se verificar que a aplicação a ser escalonada possui um nome e é possível adicionar e remover (através dos botões *Adicionar/Remover*, respectivamente) os arquivos executáveis (com o caminho completo) que compõem tal aplicação. Após definir o nome e os executáveis da aplicação, deve-se associá-la a uma classe de *software* previamente cadastrada. Na tela apresentada na figura 6.4 tem-se a possibilidade de escolha entre seis classes de *software* (a lista com as classes disponíveis é obtida pelo procedimento *nomes* do programa *cla.c*). No caso, foi selecionada a opção “Classe_Muita_Comunicação”.

A inclusão desta aplicação no cadastro de aplicações se dá através do botão *Atualizar* que irá modificar o arquivo *policy.cfg* (descrito na seção 3.3.1.1 do capítulo 3). O processo de atualização do arquivo *policy.cfg* é descrito na seção 5.4.7 do capítulo 5. É possível retornar ao Menu principal através do botão *Sair*.

Figura 6.5 – Cadastro de Aplicações

A figura 6.6 ilustra como estão relacionados o *front end* e o *back end* do cadastro de aplicações. Inicialmente, o *front end* recebe a lista classes de *software* disponíveis.

A seta C da ilustração representa essa lista obtida do arquivo *classoft.dad* (seta E). O fluxo A da ilustração corresponde às solicitações ou atualizações do *front end* para o *back end*. O fluxo B da ilustração são os retornos do *back end* ao *front end*. Esses retornos fornecem subsídios para que o *front end* seja atualizado. O fluxo C da ilustração corresponde à efetivação das atualizações que o *back end* realiza do arquivo *policy.cfg*.

Figura 6.6 – Fluxo de informações do cadastro aplicações

6.4.2 – Plataformas Computacionais

Considerando que a plataforma computacional utilizada no escalonamento é um fator decisivo na escolha da política de escalonamento, torna-se necessário o cadastro de fatores que permitam que a heterogeneidade da plataforma seja considerada.

Além da plataforma computacional, é possível cadastrar *benchmarks* que poderão ser utilizados para medir a potência computacional do *hardware* utilizado no escalonamento. Segundo especificado em (Souza, 2000): “A responsabilidade da camada superior em relação aos *benchmarks* é apenas torna-los disponíveis eficientemente para a camada inferior, permitindo que os resultados dos mesmos possam ser recuperados rapidamente e com pouca sobrecarga para as políticas”.

Através desta opção tem-se acesso ao Cadastro de Hardware, ao Detecta Hardware, ao Cadastro de Benchmarks e Executa Benchmarks.

• Cadastro de Hardware

Permite a inclusão e o acesso das informações sobre a plataforma computacional no arquivo *hardware.dad* (descrito na seção 3.3.1.2 do capítulo 3). Como

ilustrado na figura 6.7, pode-se observar os vários campos que compõem cada registro da plataforma. Eventualmente, o *hardware* cadastrado pode possuir particularidades que poderão ser incluídas em *características especiais*.

A navegação entre os registros cadastrados é possível através dos botões *Anterior* e *Próximo*, que atualizam os campos, respectivamente, com o registro imediatamente anterior e posterior da plataforma visualizada. Estes botões evocam os procedimentos *btn_ant* e *btn_pro* do programa *har.c*.

A exclusão de registros cadastrados e a inclusão de novos registros são realizadas através dos botões *Exclui* e *Inclui*, respectivamente (tais procedimentos se encontram no arquivo *har.c* com os nomes *exclui* e *inclui*). O botão *Atualiza* é responsável por atualizar eventuais alterações que o usuário venha a fazer em registros já cadastrados (inicialmente o registro é apagado para posterior inclusão com os novos valores). Finalmente, o botão *Sair* fecha a janela, retornando ao Menu Principal.

Figura 6.7 – Cadastro de Hardware

• Detecta Hardware

A função de detectar o *hardware* em que o AMIGO está executando é responsabilidade da Camada Superior do AMIGO. Essa função ainda não foi implementada e devido à sua complexidade, por si só gera uma linha de pesquisa. Cabe apenas registrar que uma vez detectado o *hardware* (independente do método adotado), a inclusão dos valores

obtidos (e eventual atualização do arquivo *hardware.dad*) poderá ser feito pela própria funcionalidade da Camada Superior.

Em relação à detecção de *hardware*, essa funcionalidade se dá através de programas específicos. Tais programas podem utilizar vários protocolos para detecção de *hardware*, como os protocolos de gerenciamento de redes (snmp, por exemplo). Independente do aplicativo de detecção utilizado, este deverá fornecer as seguintes informações sobre o nó detectado: **a)** código identificador para a máquina; **b)** nome da máquina; **c)** arquitetura; **d)** Sistema Operacional; **e)** quantidade de processadores; **f)** quantidade de memória principal disponível; **g)** acesso (1) ou não acesso (0) a disco; e **h)** características especiais.

Para cadastrar cada nó detectado no arquivo *hardware.dad* e, dessa forma, tornar tais máquinas disponíveis ao AMIGO é necessário executar a funcionalidade seguindo a seguinte sintaxe: **har inclui** <CAMINHO> <informações detectadas>. Onde, **har** é o executável responsável pela funcionalidade em questão, **inclui** é a ação a ser realizada, <CAMINHO> é o diretório padrão que armazena os arquivos de configuração e <informações detectadas> são os dados de cada nó detectado. Observar que o <CAMINHO> deve seguir a estrutura de arquivos do sistema operacional utilizado. Para Linux, ponto de montagem e / (barra normal) e para MS-Windows, unidade de disco e \ (barra invertida).

• Cadastro de Benchmarks

Este item tem por objetivo flexibilizar os *benchmarks* utilizados para medir a potência computacional da plataforma. Compete a esse item a atualização do arquivo *bench.dad* que armazena as informações sobre os *benchmarks* cadastrados. A figura 6.8 ilustra os campos que formam esse cadastro: o código, o nome e o arquivo executável (com o caminho completo) de cada *benchmark* cadastrado. Eventualmente, os *benchmarks* podem possuir particularidades que poderão ser incluídas em *características especiais*.

A navegação entre os registros cadastrados é possível através dos botões *Anterior* e *Próximo*, que atualizam os campos, respectivamente, com o registro imediatamente anterior e posterior do *benchmark* visualizado. O acionamento desses botões ativarão, respectivamente, os procedimentos *btn_ant* e *btn_pro* contidos no programa *ben.c*.

A exclusão de registros cadastrados e a inclusão de novos registros são realizadas através dos botões *Exclui* e *Inclui*, respectivamente. Tais ações evocam os procedimentos *exclui* e *inclui* do programa *ben.c*. O botão *Atualiza* é responsável por atualizar eventuais alterações que o usuário venha a fazer em registros já cadastrados (o sistema exclui o registro antigo e inclui um novo com os dados atualizados). Finalmente, o botão *Sair* fecha a janela, retornando ao Menu Principal.

Figura 6.8 – Cadastro de *Benchmarks*

- Executa *Benchmarks*

Uma vez cadastrados e disponibilizados, os *benchmarks* podem ser executados para gerar informações sobre o desempenho da plataforma, possibilitando desse modo a diferenciação da mesma em função de seu desempenho.

A lista dos *benchmarks* previamente cadastrados é ilustrada na figura 6.9. Esta lista é obtida pelo procedimento *exec_b* contido no programa *ben.c*. Uma vez selecionado o *benchmark* é possível executá-lo através do botão *Executar*. O botão *Cancelar* fecha a janela, retornando ao Menu Principal.

Figura 6.9 – Executa Benchmarks

A figura 6.10 ilustra como são disponibilizados os *benchmarks* cadastrados. Toda vez que o usuário deseja executar algum *benchmark*, o *front end* solicita a lista com os *benchmarks* cadastrados (seta A). Cabe ao *back end* obter essa lista (seta C) do arquivo *bench.dad* e envia-la ao *front end* (seta B). De posse da lista de *benchmarks* cadastrados, o usuário pode executar qualquer um que esteja disponível.

Figura 6.10 – Front end/back end do item 'executa benchmark'

6.4.3 – Políticas de Escalonamento

Considerado o item mais importante da camada superior do AMIGO devido à importância de algumas de suas atividades. A partir desse item é possível viabilizar o cadastro e tornar disponível várias políticas de escalonamento; é possível estabelecer a relação entre as políticas de escalonamento e os demais itens que influenciam a atividade de escalonamento; e selecionar as políticas que estarão ativas para a camada inferior.

Através da opção *Políticas de Escalonamento* tem-se acesso às seguintes opções: Cadastro de Políticas, Relaciona Classes e Políticas, Configura Uso de Políticas e Seleciona Políticas.

• Cadastro de Políticas

Este é um dos cadastros mais importantes para o escalonamento de processos. Esse item possui quatro campos - conforme pode-se observar na figura 6.11 - e é responsável pela atualização do arquivo *policies.dad*. Tal arquivo é descrito na seção 3.3.1.3 do capítulo 3 e mantém informações sobre as políticas de escalonamento cadastradas.

A navegação entre os registros cadastrados é possível através dos botões *Anterior* e *Próximo*, que atualizam os campos com o registro imediatamente anterior e posterior da política de escalonamento visualizada. Estes procedimentos disparam os procedimentos *btn_ant* e *btn_pro* do programa *pol.c*.

A exclusão de registros cadastrados e a inclusão de novos registros são realizadas através dos botões *Exclui* e *Inclui*, respectivamente (procedimentos *exclui* e *inclui* do programa *pol.c*). O botão *Atualiza* é responsável por atualizar eventuais alterações que o usuário venha a fazer em registros já cadastrados. A atualização ocorre pela exclusão do registro a ser atualizado e posterior inclusão com os novos valores. Finalmente, o botão *Sair* fecha a janela, retornando ao Menu Principal.

Figura 6.11 – Cadastro de Políticas de Escalonamento

• Relaciona Classes e Políticas

Permite o vínculo das classes de *software* com as políticas de escalonamento. Para cada classe de *software* cadastrada é possível relacionar uma única política, mas uma mesma política de escalonamento pode estar vinculada a diferentes classes de *software*.

A figura 6.12 ilustra no lado esquerdo as classes de *software* previamente cadastradas e no lado direito as políticas de escalonamento cadastradas anteriormente. A lista de classes de *software* é obtida pelo procedimento *nomes* do programa *cla.c*. E a lista de políticas de escalonamento é obtida pelo procedimento *nomes* do programa *pol.c*.

O botão *Relacionar* associa uma classe de *software* a uma política de escalonamento. Assim, o item *[policies]* do arquivo *policy.cfg* é atualizado com o código da classe de *software*, a política de escalonamento à essa classe associada e o caminho completo dessa política de escalonamento.

Figura 6.12 – Relaciona Classes e Políticas

A figura 6.13 ilustra como estão associados o *front end* e o *back end* do item “relaciona classes de *software* e políticas de escalonamento”. Inicialmente, o *front end* recebe a lista das classes de *software* cadastradas e das políticas de escalonamento disponíveis. A seta C da ilustração representa essa lista obtida dos arquivos *classoft.dad* e *policies.dad* (através dos programas *cla.c* e *pol.c*, representados nas setas E). O fluxo A da ilustração

corresponde às solicitações ou atualizações do *front end* para o *back end*. O fluxo B da ilustração são os retornos do *back end* ao *front end*. Esses retornos fornecem subsídios para que o *front end* seja atualizado. O fluxo C da ilustração corresponde à efetivação dos relacionamentos entre classes de *software* e políticas de escalonamento que o *back end* realiza do arquivo *policy.cfg* (através do programa *polcfg.c*).

Figura 6.13 – Funcionamento da opção 'Relaciona Classes e Políticas'

A opção *Ver Relações* presente nesse item permite a verificação de como o relacionamento está atualmente definido. Os dados ilustrados na figura 6.14 são obtidos a partir do arquivo *policy.cfg*. O item *[policies]* do arquivo *policy.cfg* contém os códigos da classe de software e da política de escalonamento associada à essa classe. De posse do código da classe, o procedimento *nomes* do programa *clã.c* é executado e obtém-se a descrição da referida classe de software. Para obter a descrição da política de escalonamento, executa-se o procedimento *cod_nome* do programa *pol.c*.

O botão *Sair* fecha a janela, retornando ao Menu Principal.

Figura 6.14 – Relacionamento Efetuado

• Configura Uso de Políticas

O uso das políticas de escalonamento é configurado por essa opção que atualiza o arquivo *policy.cfg* (item *[config]*), que é acessado pela Camada Inferior do AMIGO. A atualização do arquivo *policy.cfg* é descrita na seção 5.4.7 do capítulo 5. Ao configurar o uso do escalonamento, como ilustrado pela figura 6.15, o usuário pode determinar os seguintes itens (Souza, 2000):

- ***mais de uma política ativa ao mesmo tempo***: essa opção autoriza que políticas vinculadas às classes de software sejam executadas de maneira concorrente à política determinada pelo AMIGO. Impedir o uso simultâneo de várias políticas de escalonamento pode ser útil quando se deseja fixar a maneira como os escalonamentos serão feitos sobre a plataforma.
- ***ativar políticas sob demanda***: essa opção permite que uma política (que não está ativa) seja executada quando uma aplicação solicita o escalonamento de processos. A ativação sob demanda só é possível se for permitida a existência de mais de uma política ativa ao mesmo tempo.
- ***mudar as políticas de escalonamento em tempo de execução***: essa opção permite ou não que as políticas que estão em atividade possam ser substituídas pelo AMIGO, toda vez que este determinar que o escalonamento não está sendo vantajoso (através do monitoramento), ou então, toda vez que as políticas forem alteradas pela opção <Seleciona Políticas> do item <Políticas de Escalonamento>.
- ***utilizar cache no AMIGOD para o índice de carga obtido em cada política ativa***: essa opção tem por objetivo otimizar a troca de informações entre as políticas de escalonamento em cada processador.
- ***a troca de mensagens diretamente entre as políticas***: essa opção possibilita o aumento de desempenho na comunicação da camada inferior do AMIGO, por reduzir os saltos necessários para que cada mensagem seja enviada entre os módulos da camada inferior.
- ***o uso de um arquivo de “profile”***: O uso de um arquivo *profile* tem por objetivo armazenar informações sobre o escalonamento feito na plataforma

em um período preestabelecido. Se a opção estiver habilitada, o usuário fornece o período máximo para coleta das informações (em minutos). Se o período for de 120 minutos, por exemplo, o AMIGO irá armazenar as 2 últimas horas de execução, onde as informações mais antigas serão as primeiras a serem descartadas após as 2 horas de coleta.

Quando a interface gráfica está sendo executada é possível obter uma descrição mais detalhada sobre os itens a serem configurados, bastando clicar o botão direito do *mouse* no item desejado.

O botão *Atualizar* efetiva eventuais alterações no arquivo *policy.cfg* e o botão *Sair* fecha a janela, retornando ao Menu Principal.

Figura 6.15 – Configura Uso de Políticas

• Seleciona Políticas

Várias políticas de escalonamento podem estar ativas em um mesmo instante: política reserva, política independente e políticas específicas, que podem ser acessadas pela opção *Seleciona Políticas*.

- Política Reserva

A política reserva (disponível em “Definir política reserva”) representa a política que será utilizada caso todas as outras políticas não possam ser executadas.

A figura 6.16 ilustra uma lista com as políticas de escalonamento previamente cadastradas. O botão *Defenir* associa como reserva a política selecionada. O item *[reserve]* do arquivo *policy.cfg* é atualizado com o caminho completo da política escolhida como reserva (obtém-se o caminho da política com o procedimento *cod_cam* do programa *pol.c*). O processo de atualização do item *[reserve]* do arquivo *policy.cfg* é descrito na seção 5.4.7 do capítulo 5.

O botão *Sem Pol. Reserva* especifica que não existe política reserva (o item *[reserve]* do arquivo *policy.cfg* fica vazio). O botão *Sair* fecha a janela, retornando ao Menu Principal.

Figura 6.16 – Selecciona Política Reserva

- Políticas Independente e Específicas

As políticas independente e específicas podem ser manualmente configuradas através da opção “Livre” disponível em “Definir Outras Políticas” em *Selecciona Políticas*. A figura 6.17 ilustra uma lista de como as políticas de escalonamento e as classes de *software* estão relacionadas (lado esquerdo). Essa lista disponibiliza as políticas que poderão ser definidas como políticas independente e específicas.

Na parte superior do lado direito da figura em questão é definida a política independente (através do botão *Definir* respectivo). O botão *Remover* define que não haverá

política independente. No item *[policies]* do arquivo *policy.cfg* é adicionada a string *independent* à classe especificada como independente.

Na parte inferior do lado direito da mesma figura é possível definir a(s) política(s) específica(s). É permitida a definição de uma ou mais política específica. No item *[policies]* do arquivo *policy.cfg* é adicionada a string *active* à(s) política(s) de escalonamento especificada(s) como específica(s). O botão *Remover* define que não haverá política específica.

O botão *Atualizar* efetiva as alterações no arquivo *policy.cfg* (O processo de atualização do arquivo *policy.cfg* é descrito na seção 5.4.7 do capítulo 5) e o botão *Sair* fecha a janela, retornando ao Menu Principal.

Figura 6.17 – Seleciona Outras Políticas

A figura 6.18 ilustra como estão relacionados o *front end* e o *back end* da opção 'Seleciona Políticas Independente e Específicas'. Inicialmente, o *front end* recebe as informações de como estão associadas as classes de *software* com as políticas de escalonamento (seta B). A seta C da ilustração representa essas informações, que são obtidas do arquivo *policy.cfg*, através do programa *polecfg.c*. O fluxo A da ilustração corresponde às atualizações vindas do *front end* (definições da política independente e das políticas

específicas) para o *back end*, que também são efetivadas pelo programa *policy.cfg* (seta C, que efetiva as opções selecionadas pelo usuário).

Figura 6.18 – Relação front end/back end da opção 'Seleciona Políticas'

No momento em que as políticas são atualizadas (arquivo *policy.cfg*, item *[policies]*), é possível verificar como as políticas foram definidas, como ilustrado na figura 6.19. O procedimento *cod_nome* do programa *pol.c* converte em nomes das políticas os códigos das políticas definidas como independente e específicas.

Figura 6.19 – Definição das Políticas

A especificação do AMIGO prevê que esses relacionamentos poderão ser feitos automaticamente com mais ou menos influência do usuário, dependendo da estratégia adotada (assistida e padrão, respectivamente).

Um trabalho de doutorado que está sendo desenvolvido no LaSDPC propõe um estudo de um sistema inteligente de auxílio na escolha de política de escalonamento baseado no conceito de agentes inteligentes. O referido trabalho poderá ser empregado no AMIGO.

6.4.4 – Monitoramento

Através desta opção, o usuário poderá verificar, em tempo de execução, se as políticas de escalonamento estão atingindo seus objetivos. Com informações fornecidas pelo monitoramento é possível avaliar se as políticas de escalonamento atualmente ativas devem ser substituídas. Através desta opção tem-se acesso ao Cadastro de Métricas e ao Início e Término do monitoramento.

• Cadastro de Métricas

Esta opção está disponível no item Monitoramento e tem por objetivo o cadastro de métricas que quantifiquem o quanto dos objetivos foram realmente atingidos. Devido à variedade de métricas, o usuário poderá criar seu próprio cadastro de métricas (arquivo *metrics.dad* - conforme definido na seção 3.3.1.4 do capítulo 3, este arquivo contém o cadastro das métricas disponíveis).

Por intermédio da figura 6.20 pode-se observar os campos que compõem esse cadastro: o código, o executável com caminho completo, o arquivo responsável por armazenar os dados do monitoramento (com o caminho completo) e as eventuais características especiais da métrica.

A navegação entre os registros cadastrados é possível através dos botões *Anterior* e *Próximo*, que atualizam os campos, respectivamente, com o registro imediatamente anterior e posterior da métrica visualizada. Os procedimentos responsáveis por essas funções são *btn_ant* e *btn_pro* contidos no programa *met.c*.

A exclusão de registros cadastrados e a inclusão de novos registros são realizadas através dos botões *Exclui* e *Inclui*, respectivamente. Tais funcionalidades são obtidas pelos procedimentos *exclui* e *inclui* do programa *met.c*. O botão *Atualiza* é responsável por atualizar eventuais alterações que o usuário venha a fazer em registros já cadastrados (o registro com os novos valores é incluído após o mesmo ter sido apagado). Finalmente, o botão *Sair* fecha a janela, retornando ao Menu Principal.

Figura 6.20 – Cadastro de Métricas

A figura 6.21 ilustra como estão relacionados o *front end* e o *back end* do cadastro de métricas. O fluxo A da ilustração corresponde às solicitações ou atualizações do *front end* para o *back end*. O fluxo B da ilustração são os retornos do *back end* ao *front end*. Esses retornos fornecem subsídios para que o *front end* seja atualizado. O fluxo C da ilustração corresponde à efetivação das atualizações ou consultas que o *back end* realiza do arquivo *metrics.dad*.

Figura 6.21 – Funcionamento do Cadastro de Métricas

• Início/Término

Essa opção permite que a atividade de monitorar seja ativada e paralisada, além de definir vários outros itens. Como visualizado na figura 6.22, é possível observar que existe uma lista de métricas cadastradas e a partir dessa lista, seleciona-se a métrica desejada para a atividade de monitoramento. Esta lista é obtida pelo procedimento *nomes* presente no programa *met.c*.

Pode-se também, definir um horário de funcionamento para a atividade de monitoramento e especificar o intervalo de tempo que as informações serão coletadas. Um campo informa textualmente a atividade desempenhada pela monitoração.

O botão *Iniciar* dá início à atividade de monitoramento e o botão *Terminar* paralisa a atividade de monitoramento. O botão *Cancelar* cancela a atividade de monitoramento e fecha a janela, retornando o sistema ao Menu Principal.

Figura 6.22 – Início/Término do Monitoramento

A atividade de monitorar uma plataforma computacional não é trivial, e por si só, constitui uma linha de pesquisa, a qual está sendo explorada em um trabalho de doutorado no Grupo de Sistemas Distribuídos e Programação Concorrente do ICMC/USP.

6.4.5 – Simulação

O objetivo de incluir um item chamado “Simulação” no AMIGO é permitir que o comportamento das políticas de escalonamento possa ser analisado, sem a necessidade de executá-las realmente.

Em função da abrangência da simulação, constituindo uma área de pesquisa à parte, alguns trabalhos correlatos foram e estão sendo desenvolvidos no LaSDPC.

6.5 – COMENTÁRIOS FINAIS

Considerando o que foi referenciado no Capítulo 4, que tratava as interfaces gráficas, o desenvolvimento da Interface Gráfica do AMIGO procurou obedecer às especificações do referido capítulo. Através da Interface Gráfica espera-se que os usuários tenham um ganho de produtividade e vejam com maior clareza as características dos resultados obtidos pela mesma.

Por meio dos componentes visuais é possível a interação do usuário com o ambiente AMIGO, no que se refere ao gerenciamento do ambiente em sua Camada Superior.

É importante ressaltar que algumas funcionalidades da Camada Superior não estão atualmente desenvolvidas. Esse aspecto pode ser considerado positivo, pois significa que vários outros trabalhos (e linhas de pesquisa) poderão ser desenvolvidos de modo a disponibilizar a funcionalidade da Camada Superior em sua plenitude.

capítulo 7

Conclusões e Trabalhos Futuros

Este trabalho considera o projeto e implementação da interface gráfica da camada superior do AMIGO - um ambiente flexível e dinâmico de escalonamento de processos. O projeto foi desenvolvido seguindo as orientações encontradas na literatura especializada para o desenvolvimento de interfaces gráficas e seguindo também as especificações definidas na tese de doutorado que propôs o AMIGO (Souza, 2000).

Neste capítulo estão descritas as contribuições do trabalho, as dificuldades encontradas e as propostas para trabalhos futuros concluindo, dessa forma, esta dissertação.

7.1 – CONSIDERAÇÕES FINAIS E CONCLUSÕES

O objetivo principal desta dissertação de mestrado é o desenvolvimento da interface gráfica de um ambiente de escalonamento flexível e dinâmico (AMIGO). Para atingir esse objetivo, diversas etapas foram consideradas:

1 – Projeto da interface gráfica

Nesta etapa foram definidos os módulos necessários para a implementação da interface e quais tarefas cada um deles deveria efetuar. A divisão adotada (*front end/back end*)

é tradicional e permitiu modularizar o projeto e a implementação da interface em questão e obter maior flexibilidade.

O projeto da interface gráfica foi discutido na seção 5.1 e 5.2 do capítulo 5.

2 – Definição das ferramentas a serem utilizadas

A divisão da interface gráfica em dois módulos permite que cada um dos módulos utilize as ferramentas mais apropriadas para cada caso:

- a linguagem de programação *C* fornece mecanismos mais eficientes que linguagens interpretadas para a manipulação e acesso de arquivos;
- de um modo mais simples e prático que o *C*, o TCL/TK permite a especificação e disposição dos vários componentes que formam uma interface gráfica.

Uma discussão mais completa sobre a definição das ferramentas encontra-se nas seções 5.3 do capítulo 5 e 6.1 e 6.2 do capítulo 6.

3 – Projeto dos módulos da Interface Gráfica

Cada um dos módulos da interface gráfica foi definido ressaltando-se as funções e a estrutura de cada um deles. O *front end* foi projetado de modo a permitir uma interação amigável entre o usuário e o ambiente de escalonamento. O projeto do *back end* visou explorar a eficiência da linguagem *C* para manipular os arquivos necessários para configurar a camada superior do AMIGO.

4 – Implementação dos Módulos

Nesta última etapa foram implementados e testados os módulos da interface gráfica.

A divisão do desenvolvimento da interface gráfica em módulos (*front end/back end*) apresenta vantagens como a modularidade e flexibilidade entre os componentes visuais e as funcionalidades da interface gráfica. Entretanto, isso não significa, necessariamente, que essa abordagem facilitou o desenvolvimento do projeto. Principalmente, quando é considerado que a depuração e teste de quase 3.000 linhas de código (que foram necessárias para o desenvolvimento do *front end* do projeto) em linguagem interpretada não é trivial.

7.2 - CONTRIBUIÇÕES DO TRABALHO

Como contribuição principal deste trabalho, destaca-se o projeto e implementação da interface gráfica que permite uma interação mais amigável entre o usuário e o ambiente de escalonamento de processos no que concerne ao acesso das funções e à gerência de características que influenciam a atividade de escalonamento.

Embora este trabalho não seja a primeira contribuição para o projeto do AMIGO, pode-se considerar que esta é a primeira vez que um projeto aborda a Camada Superior do AMIGO - outros trabalhos já concluídos enfocam aspectos da Camada Inferior. Desse modo, este trabalho é uma contribuição quando utilizado como base para outras pesquisas relacionadas à Camada Superior do AMIGO.

Seguindo a filosofia do AMIGO no que diz respeito à flexibilidade, este trabalho permite que novas funções sejam adicionadas e alteradas às já implementadas (inclusive pelo uso de ferramentas distintas das aqui utilizadas). Isso devido ao desenvolvimento separado dos controles visuais e da funcionalidade propriamente dita da Interface Gráfica.

Os aspectos considerados a respeito da computação paralela, escalonamento de processos e interfaces com o usuário também contribuem para os trabalhos futuros da área.

7.3 - DIFICULDADES ENCONTRADAS

A implementação de projetos se valendo de ferramentas de domínio público é por si só complexa. Não que não haja ferramentas de qualidade, muito pelo contrário. A dificuldade é que tais ferramentas não têm como filosofia o desenvolvimento de aplicações rápidas (ou seja, um ambiente RAD - *Rapid Application Development* - como o Kylix e Delphi da Borland ou Visual Basic da Microsoft, por exemplo).

Visto que o AMIGO foi primeiramente projetado para trabalhar em computação de sistema distribuído rodando em plataformas heterogêneas (uma rede de computadores baseada em Linux), optou-se pelo uso de uma linguagem de programação compatível com Linux, no caso o C. Sob essas circunstâncias, o projeto das funcionalidades não foi trivial, pois embora seja eficiente, esse ambiente não é amigável.

Uma das maiores dificuldades encontradas no desenvolvimento do trabalho foi a depuração do módulo desenvolvido em TCL/TK (cerca de 3.000 linhas de código). Dificuldade essa inerente às linguagens interpretadas, pois não existe um modo eficaz de interromper a execução e analisar o estado das variáveis.

7.4 - PROPOSTAS PARA TRABALHOS FUTUROS

O trabalho desenvolvido nesta dissertação de mestrado está inserido em um projeto maior do Grupo de Sistemas Distribuídos e Programação Concorrente do ICMC/USP. Diversos outros trabalhos já contribuíram para o projeto e implementação dos diferentes módulos do AMIGO. Outros trabalhos estão em desenvolvimento e certamente, novos projetos serão ainda definidos. Desta forma, embora as propostas aqui apresentadas relacionam-se com o desenvolvimento da interface gráfica, diversas outras poderiam ser citadas em relação ao projeto e implementação do AMIGO.

Assim, como continuidade deste trabalho, é possível traçar algumas sugestões para trabalhos futuros que completam ou aprimoram a interface gráfica:

- ♦ Verificar mecanismos que possibilitem a detecção do hardware que compõe o ambiente paralelo para que as informações do mesmo sejam automaticamente cadastradas junto à base de dados que fazem parte da Camada Superior do AMIGO.

- ♦ Estudo e integração com a Camada Superior do AMIGO de um sistema inteligente para o auxílio na escolha de políticas de escalonamento (independente e específicas) baseado no conceito de agentes inteligentes.
- ♦ Desenvolvimento de técnicas de monitoramento que poderão ser integradas à Interface Gráfica.
- ♦ Projeto de ambientes de simulação ou integração com ambientes de simulação já existentes com a Camada Superior do AMIGO.

capítulo 8

Referências Bibliográficas

- (Almasi & Gottlieb, 1994) ALMASI, G.; GOTTLIEB, A. **Highly Parallel Computing**, 2ª edição, The Benjamin/Cummings Publishing Company, Redwood City, Califórnia, 1994.
- (Amorim, 1988) AMORIM, C. L. **Uma Introdução à Computação Paralela e Distribuída**, VI Escola de Computação, Campinas, 1988.
- (Araújo et al., 1999a) ARAÚJO, A.P.F.; SANTANA, M.J.; SANTANA, R.H.C.; SOUZA, P.S.L. **A New Dynamical Scheduling Algorithm**, International Conference on Parallel and Distributed Processing Techniques and Applications - PDPTA'99, Las Vegas, Nevada, U.S.A., 28 de junho a 1 de julho de 1999.
- (Araújo et al., 1999b) ARAÚJO, A.P.F.; SANTANA, M.J.; SANTANA, R.H.C.; SOUZA, P.S.L. **DPWP - A New Load Balancing Algorithm**, 5th International Conference on Information Systems Analysis and Synthesis - ISAS'99, Orlando, U.S.A., 31 de julho a 4 de agosto de 1999.
- (Araújo, 1999c) ARAÚJO, A. P. F. **DPWP – Uma nova abordagem para o Escalonamento Dinâmico em Computação Paralela Virtual**. Dissertação de mestrado, ICMC/USP, São Carlos, 1999.
- (Blech, 1994) BLECH, R. A. **An Overview of Parallel Processing Parallel Computing with PVM Workshop**, Ohio Aerospace Institute, NASA Lewis Research Center, 7 a 8 de fevereiro de 1994.
- (Calzarossa & Serazzi, 1993) CALZAROSSA, M.; SERAZZI, G. **Workload Characterization: A Survey**. **IEEE Computer Performance Evaluation**, v. 81, n. 8, pp.1136-1150, agosto de 1993.
- (Carmona, 1999) CARMONA, O. A. **Desenvolvimento e Avaliação de Algoritmos Numéricos Paralelos**. Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação – USP, São Carlos, São Paulo, 1999.
- (Casavant & Kuhl, 1988) CASAVANT, T.; KUHL, J. **A Taxonomy of Scheduling in general-Purpose Distributed Computing Systems**. **IEEE Transactions on software Engineering**, v. 14, n. 2, pp.141-154, fevereiro de 1988.
- (Cenapdne, 2000) Centro Nacional de Processamento de Alto Desempenho do Nordeste. [Disponível em: <http://cenapdne.br>. Acessado em 18 de maio de 2000]

- (Coulouris et al., 1994) COULOURIS, G.; DOLLIMORE, J; KINDBERG, T. **Distributed Systems Concepts and Desing**, 2ª edição, Addison-Wesley, 1994.
- (Dongarra & Gentzsch, 1993) DONGARRA, J. J.; GENTZSCH, W. **Computer Benchmarks - Advances in Parallel Computing**. Ed. North Holland. Amsterdam, Netherlands, 1993.
- (Duncan, 1990) DUNCAN, R. **A Survey of Parallel Computer Architectures**, IEEE Computer, pp. 5-16, fevereiro de 1990.
- (Feitelson & Rudolph, 1996) FEITELSON, D. G.; RUDOLPH, L. **Toward Convergence in Job Schedulers for Parallel Supercomputers**, IPPS' 96 Workshop on Job Scheduling Strategies for Parallel Processing, Honolulu, Hawaii, Lecture Notes in Computer Sciences 1162, abril de 1996.
- (Flynn, 1966) FLYNN, M.J. **Very High Speed Computing Systems**. Proceedings of the IEEE 54, pp. 1901-1909, dezembro de 1966.
- (Flynn, 1972) FLYNN, M. J. **Some Computer Organizations and Their Effectiveness**, IEEE Transactions on Computers, v. c-21, n. 9, pp. 948-960, setembro de 1972.
- (Flynn & Kevin, 1996) FLYNN, M. J.; KEVIN, R. W. **Parallel Architectures**, ACM Computing Surveys, v. 28, , n. 1, pp. 67-70, março de 1996.
- (Geist et al., 1994) GEIST, A.; BEGUELIN, A.; DONGARRA, J.; JIANG, W.; MANCHEK, R.; SUNDERAM, V. **PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing**, The MIT Press, Londres, 1994.
- (GTK, 2001) The GIMP ToolKit. [Disponível em: <http://www.gtk.org>. Acessado em 18 de junho de 2001]
- (Hayes, 1988) HAYES, J. P. **Computer Architecture and Organization**, 2ª edição, Nova Iorque, 1988.
- (Hwang & Briggs, 1984) HWANG, K.; BRIGGS, F.A. **Computing Architecture and Parallel Processing**, Mcgraw-Hill, 1984.
- (Hwang, 1993) HWANG, K. **Advanced Computer Architecture: Parallelism, Scalability, Programmability**, Mcgraw-Hill, 1993.
- (Kelley & Pohl, 1995) KELLEY, A.; POHL, I. **A book on C: Programming in C**. Addison-Wesley Publishing Company, 3ª Ed., 1995.
- (Kuck, 1996) KUCK, D. J. **High Performance Computing – Challenges for Future Systems**. Oxford University Press, Nova Iorque, 1996.
- (Larson, 1992) LARSON, J. A. **Interactive Software: tools for building interactive user interfaces**. Prentice Hall – Yourdon Press Computing Series, Nova Jersey, 1992.
- (Lesstif, 2001) LessTif. [Disponível em: <http://www.lesstif.org>.. Acessado em 18 de junho de 2001]
- (Marcus, 1992) MARCUS, A. **Graphic Design for Electronic Documents**. ACM Press Books, Nova Iorque, 1992.
- (MPI, 1995) Message Passing Interface Forum. **MPI: A Messagee-Passing Interface Standard**. [Obtido em junho de 1995 a partir de <http://www.unix.mcs.anl.gov/mp/MPI-1.1>]

- (Mullender, 1993) MULLENDER, S. **Distributed Systems, Scheduling**: Hermann Kopetz, pp. 491-509, 2ª edição, Nova Iorque, 1993.
- (Navaux, 1989) NAVAUX, P. O. A. **Introdução ao Processamento Paralelo**. *Revista Brasileira de Computação*, v. 5, n. 2, pp. 31-43, outubro/dezembro de 1989.
- (Peddie, 1992) PEDDIE, J. **Graphical User Interfaces and Graphics Standards**. McGraw-Hill, Nova Iorque, 1992.
- (Plauger, 1995) PLAUGER, P. J. **A Biblioteca Standard C**. Rio de Janeiro, Editora Campus, 1995.
- (Quinn, 1987) QUINN, M. J. **Designing Efficient Algorithms for Parallel Computers**, McGraw-Hill Book Company, Nova Iorque, 1987.
- (Quinn, 1994) QUINN, M. J. **Parallel Computing – Theory and Practice**, 2a. edição, McGraw-Hill, Nova Iorque, 1994.
- (Rewini et al., 1995) REWINI, H. E.; ALI, H. H.; LEWIS, T. **Task Scheduling in Multiprocessing Systems**. *IEEE Computer*, pp. 27-37, dezembro de 1995.
- (Ritchie & Vernighan, 1987) RITCHIE, D.; VERNIGHAN, B. C – **A Linguagem de Programação**. Editora Campus, 3ª. Edição, 1987.
- (Santana et al., 1997) SANTANA, R. H. C.; SANTANA, M. J.; SOUZA, M. A.; SOUZA, P. S. L.; PIEKARSKI, A. E. T. **Computação Paralela. Material Didático do Grupo de Sistemas Distribuídos e Programação Concorrente**. ICMC/USP, setembro de 1997.
- (Scriptics, 2001) Scriptics Corporation. [Disponível em: <http://www.scriptics.com>. Acessado em 18 de junho de 2001]
- (Shivaratri et al., 1992) SHIVARATRI, N. G.; KRUEGER, P.; SINGHAL, M. **Load Distribution for Locally Distributed Systems**. *IEEE Computer*, pp. 33-44, dezembro de 1992.
- (Skillicorn & Talia, 1998) SKILLICORN, D. B.; TALIA, D. **Models and Languages for Parallel Computation**, pp. 123-169, *ACM Computing Survey*, Vol. 30, N.º 2, Junho de 1998.
- (Snir et al., 1996) SNIR, M.; OTTO, S; HUSS-LEDERMAN, S; WALKER, D; DONGARRA, J. **MPI: The Complete Reference**. The MIT Press, London, England. 1996.
- (Souza, 1996) SOUZA, M. A. **Avaliação das Rotinas de Comunicação Ponto-a-Ponto do MPI**, Dissertação de Mestrado, Instituto de Ciências Matemáticas e de Computação (ICMC-USP), dezembro, 1996.
- (Souza et al., 1998) SOUZA, P. S. L.; ARAÚJO, A. P. F.; SANTANA, M. J.; SANTANA, R. H. C. **Escalação de Processos: Características, Divergências e Soluções**. Semana Paraense de Informática – SEPAI'98, novembro de 1998.
- (Souza et al., 1999a) SOUZA, P.S.L.; SANTANA, M.J.; SANTANA, R.H.C. **A New Scheduling Environment for Near-Optimal Performance**, International Conference on Parallel and Distributed Processing Techniques and Applications - PDPTA'99, Las Vegas, Nevada, U.S.A., 28 de junho a 1 de julho de 1999.
- (Souza et al., 1999b) SOUZA, P.S.L.; SANTANA, M.J.; SANTANA, R.H.C. **AMIGO – A Dynamical Flexible Scheduling Environment**. A ser publicado, Proceedings of the 5th

International Conference on Information Systems, Analysis and Synthesis: ISAS'99, Orlando, Florida, junho de 1999.

- (Souza et al., 1999c) SOUZA, P.S.L.; SANTANA, M.J.; SANTANA, R.H.C. **PVM and Viable and Flexible Scheduling**, artigo submetido para a EuroPVM/MPI'99, Espanha, 1999.
- (Souza et al., 1999d) SOUZA, P. S. L.; ARAÚJO, A. P. F.; SANTANA, M. J.; SANTANA, R. H. C. **Migração de Processos como Técnica de Distribuição de Cargas para Sistemas Localmente Distribuídos**. SEMINF'99 - Semana de Informática da UFPa, abril de 1999.
- (Souza et al., 1999e) SOUZA, P.S.L.; SANTANA, M.J.; SANTANA R.H.C.; ARAÚJO, A.P.F. **PVM and a Viable and Flexible Scheduling**. A ser publicado, Proceedings of Eleventh IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'99) Cambridge, MA, USA, 3-6 de novembro, 1999.
- (Souza, 2000) SOUZA, P.S.L. **AMIGO: Uma Convergência para a Área de Escalonamento de Processos**. Tese de Doutorado defendida em junho de 2000 no Instituto de Física de São Carlos (IFSC), Universidade de São Paulo, 2000.
- (Tanenbaum, 1992) TANENBAUM, A. S. **Sistemas Operacionais Modernos**. LTC-Livros Técnicos e Científicos Editora S.A., Rio de Janeiro, 1992.
- (Thimbleby, 1990) THIMBLEBY, H. **User Interface Design**. ACM Press Frontier Series, Maryland, 1990.
- (Troll-Tech, 2001) TROLL-TECH. [Disponível em: <http://www.troll-tech.com>. Acessado em 18 de junho de 2001]
- (Wang & Morris, 1985) WANG, Y-T; MORRIS, R. J. T. **Load Sharing in Distributed Systems**. IEEE Transactions on Computers, v. c-34, n. 3, pp. 204-217, março de 1995.
- (Wood, 1998) WOOD, L. E. **User Interface Desing: bridging the gap from user requirements to design**. CRC Press, Boca Raton, 1998.
- (Zaluska, 1991) ZALUSKA, E. J. **Research Lines in Distributed Computing Systems and Concurrent Computation**, Workshop em Programação Concorrente, Sistemas Distribuídos e Engenharia de Software, pp. 132-142, ICMC/USP, São Carlos/SP, dezembro de 1991.