Projeto de Uma Ferramenta de Auxílio na Depuração de Programas Paralelos

Edmilson Marmo Moreira

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 – Área: Ciências de Computação e Matemática Computacional.

USP – São Carlos
Março de 2000

Dedico esta Dissertação à minha esposa
Karina da Costa Cinquetti Moreira.

A *DEUS,*

Inteligência Suprema causa primária de todas as coisas

e a Jesus,

Mestre dos Mestres, que passou entre os homens

sem nada cobrar por Seus Divinos Ensinamentos.

Agradecimentos

A minha orientadora *Profa. Dra. Regina Helena Carlucci Santana,* pela cuidadosa orientação e pela confiança que sempre depositou em mim...

Ao *Prof. Dr. Marcos José Santana*, pelos conselhos e sugestões...

Aos meus Pais *Antonio e Terezinha,* meus primeiros mestres... Eu amo vocês...

Ao meu filho *Lucas*, minha maior conquista...

Aos meus irmãos *Leonardo e Eduardo,* acima de tudo amigos...

Ao meu irmão *Mauro*, amigo de tantos caminhos e tantas jornadas...

Aos Professores *Alexandre e Marly,* pelo incentivo durante todos esses 11 anos que nos conhecemos...

Ao amigo *Tomás Dias Santana*, companheiro de testes e depuração de programas paralelos...

Aos meus *alunos da Unifenas,* razão por eu estar aqui...

Aos *funcionários do ICMC,* pelo carinho e respeito...

Aos *colegas do LASDPC*, a família que encontrei em São Carlos...

Aos meus amigos *Clóvis, Jonatas, Rony, Giovana, Toninho e Alexandre,* companheiros do meu ideal maior...

A *Universidade de São Paulo,* pela oportunidade...

A *Universidade de Alfenas,* meu segundo lar...

Resumo

O uso da programação paralela tem crescido muito nos últimos anos. Isso se deve, entre outros fatores, ao aumento da utilização dos sistemas distribuídos. Entretanto, esse tipo de programação apresenta maior complexidade em relação à programação seqüencial, o que dificulta a sua popularização. Um problema encontrado na programação paralela é o não determinismo global, que torna a depuração desses programas uma tarefa difícil. Além disso, a aprendizagem dos conceitos que envolvem a programação paralela por usuários sem muita experiência não é uma tarefa trivial.

Dentro desse contexto, este trabalho apresenta o projeto de uma ferramenta para depuração de programas paralelos. Essa ferramenta, além de permitir a depuração de um programa, auxilia os usuários sem muita experiência a analisar o código de seus programas, conduzindo-os a uma reformulação de suas técnicas de programação. Esse procedimento permite a aquisição de novos conhecimentos sobre a prática da programação paralela ou ainda consolidar conceitos anteriormente adquiridos.

Um protótipo da ferramenta proposta foi desenvolvido com o objetivo de avaliar a *interface* e a facilidade com que os usuários interagem com o ambiente, verificando assim o potencial da ferramenta.

Abstract

The use of parallel programming had increased over the last years. This is mainly due, to the increase in the use of distributed systems. However, the parallel programming presents larger complexity in relation to the sequential programming, making its popularization difficulty. One problem found in the development of parallel software is the non-determinism that turns the debugging of those programs a difficult task. Furthermore, the learning of the concepts involved with parallel programming for inexperienced users is not a trivial task.

Thus, this MSc. dissertation presents the project of a tool aiming at debugging parallel programs. This tool, besides allowing the debugging of a parallel program, helps the inexperienced users in the analyzes of their program code, driving them towards a reformulation of their programming techniques. This procedure allows either the acquisition of new knowledge on the practice of the parallel programming or it can consolidate previous knowledge.

A prototype was also developed to evaluate both the interface built and the easiness in the users environment interaction, verifying the potential of the tool.

Sumário

Capítulo 3

Capítulo 4

Capítulo 5

Capítulo 6

Capítulo 7

Lista de Figuras

Relação de Listagens

Lista de Tabelas

Capítulo 1
Introdução

Esta é uma época incrível. Pode-se notar, através da história, que a evolução humana caminha em progressão geométrica. E isto ocorre porque há cada descoberta do homem, cada conhecimento novo cria condições para novos saltos, conquistas ainda maiores, e estas novas conquistas não se realizariam sem as bases anteriores.

Dando luz a essas considerações, o *Homo Sapiens* surgiu na Terra por volta de 35000 anos atrás. Nesta época, o homem já possuía alguns rudimentos de aquisição intelectual, haja visto que nas paredes das cavernas desse tempo, foram encontrados os primeiros testemunhos pictóricos expressando sentimentos fortemente enraizados [Seleções, 1975]. Dividindo-se esses 35000 anos de história por 60, que representa a expectativa média de vida, encontra-se um valor aproximado de 590 gerações. Sessenta anos é um valor aleatório, já que a expectativa de vida mudou muito durante esses anos, visto que os primeiros homens viviam pouco, um em cada dez sobrevivia aos 40 anos, raramente se encontraria alguém com 50 anos. Destas quase 600 gerações, o homem viveu a maior parte delas dentro de cavernas, em uma vida nômade, sobrevivendo quase que exclusivamente da caça e da pesca. O domínio do conhecimento da agricultura só apareceu por volta de 10000 anos a.C. há quase 200 gerações.

Cerca de 6000 a.C. no Irão e na Turquia, os habitantes de Çatal Hüyük, no sul da Turquia, já fundiam cobre e chumbo [Seleções, 1975]. Procedimento que viria a ter conseqüências enormes em toda a Humanidade: a libertação do homem de sua longa idade da pedra (há 140 gerações).

Na sua maioria, as grandes realizações da humanidade, que perduraram até hoje, ocorreram entre as grandes civilizações, que existiram nos últimos 5000 anos (aproximadamente 85 gerações). A civilização iniciou-se entre os povos que viviam no Médio

Oriente, em solos férteis particularmente propícios à agricultura que, em conseqüência, exigia uma vida mais sedentária, já que as sementeiras necessitavam de tempo para se desenvolverem. Os agricultores construíram as primeiras vilas, que viraram cidades; a palavra "civilização" significou, primitivamente, "viver em cidades".

Sem dúvida que a grande descoberta humana desta época foi a escrita. O homem, por viver em sociedade, sempre buscou formas de melhorar a sua comunicação. Com o surgimento da escrita, as ciências como a matemática e medicina, tiveram um progresso significativo. A grande dificuldade de entender o comportamento do homem primitivo, é justamente, não existir relatos que possam ser interpretados além das pinturas das cavernas.

A história da humanidade é uma saga de descobrimentos, lutas, conquistas, passando pelas civilizações antigas, como as cidades gregas e o glorioso império romano. Mas foi no *Renascimento,* com a invenção da imprensa por Hans Guttenberg (1398-1468), publicando a primeira obra: a Bíblia latina (1456), que a humanidade realmente disparou na curva do progresso. Desde esse momento, a informação foi se tornando cada vez mais presente na vida de cada homem. E isto aconteceu a apenas 24 gerações.

Quanto crescimento realizado nas últimas 20 gerações. As observações de Galileu Galilei (1564-1642). A genialidade de Newton (1642-1727). A seleção natural de Charles Darwin (1809-1882). A descoberta da natureza do átomo por John Dalton (1766-1844). As invenções de Thomas Alva Edison (1847-1931), patenteadas foram 1093. Quantos homens, quantas vitórias.

Ao olhar ao seu redor, o homem pode perceber que a maioria das coisas que são utilizadas foram criadas na última geração (últimos sessenta anos). O homem do século passado nascia e morria sobre o lombo de um cavalo. E olhando para um futuro próximo, daqui a dois ou três anos, é impossível prever como estará a humanidade.

A ciência da computação, como se apresenta hoje, é uma área moderna. Não participou de todas estas conquistas da história, mas é impossível imaginar um mundo a partir de hoje sem a sua presença.

O conhecimento não surge do nada. É preciso criar condições, propor hipóteses, testar, negar. O mundo que se observa nos dias atuais, é fruto de uma grande caminhada que começou antes mesmo do surgimento do *Homo Sapiens.* A informática não surgiu das lutas desse século, fez parte das técnicas que os pastores primitivos usavam para contar suas ovelhas. Está ligada nos fundamentos que promoveu a criação do ábaco. Acelerou a sua evolução com Blaise Pascal (1623-1662) e sua máquina de calcular em 1642, passando pelo projeto de Charles Babbage (1792-1871) de uma máquina para realizar cálculos poderosos.

Os teclados atuais, com tantas funções e as impressoras poderosas, fazem recordar a primeira máquina de escrever *Remington,* concebida por C. Sholes (1816-1867) com um teclado simples que só permitia escrever maiúsculas. As discussões sobre a velocidade de comunicação que as redes utilizam reportam aos estudos de investigação dos fenômenos electromagnéticos, realizados na Europa e na América no início do século passado. Esses estudos resultaram nas descobertas que viriam a dar origem ao telégrafo, ao telefone, ao rádio e a televisão.

Qualquer descoberta científica não é um fato isolado. É resultado de anos de experiência. Para se realizar um trabalho de pesquisa é preciso contextualizá-lo. Acima de tudo, é preciso entender que todo conhecimento novo necessita de um conhecimento anterior que o sustente. Por isso o homem levou tanto tempo para sair das cavernas, mas nos dias de hoje, surgem descobertas novas a cada momento.

Contribuir de alguma forma com a ciência é deixar de ser mero espectador para atuar no mundo.

1.1 – Motivação

O homem caminha do egocentrismo para o altruísmo. Necessita viver em sociedade e enfrenta os desafios que a cooperação impõe. Em várias ocasiões, não consegue trabalhar em equipe, e encontra dificuldades para trocar informações.

Na computação isto também acontece. Muito mais fácil criar um programa seqüencial, que realiza uma série de tarefas para se atingir um fim comum, do que vários programas que possam cooperar entre si, diminuindo a carga individual e aumentando a produtividade e o desempenho. Isto porque é difícil pensar em termos de conjunto, de compartilhar tarefas, de sincronizar ações, de dividir o prêmio pelo sucesso.

A programação paralela apresenta inúmeras vantagens em relação à programação seqüencial, entretanto, os motivos que ainda restringe o seu uso são relativamente fortes.

Um motivo que já está saindo de moda é a necessidade de *hardware* específico, ou seja, de máquina paralela para executar um programa paralelo. Esta situação existia até bem pouco tempo, porém com o advento dos sistemas distribuídos que estão ao alcance de todos, a programação paralela passou a ser, não só uma realidade, mas principalmente, uma necessidade para que esses sistemas possam ser usados com maior eficiência.

Outro motivo, é que programar paralelamente exige conhecimentos específicos, que muitos programadores não possuem. Isto restringe o número de profissionais que produzem esse tipo de *software*. Além disso, os conceitos que envolvem a programação paralela ainda se

encontram circunscritos nas Universidades e Centros de Pesquisa, dificultando a popularização desses conhecimentos.

Em adição, as ferramentas de desenvolvimento de programas paralelos que existem não são tão fáceis de utilizar e nem apresentam tantos recursos quanto às ferramentas de desenvolvimento seqüenciais, que já estão fortalecidas no mercado.

Desta forma, despender esforços para popularizar a programação paralela, através do desenvolvimento de ferramentas de auxílio aos usuários e ampliar o estudo das técnicas envolvidas no desenvolvimento de um software paralelo, objetivando encontrar soluções mais simples e práticas, proporciona um ambiente propício e a motivação necessária para o desenvolvimento de um trabalho como esse.

1.2 – Objetivos

Diante das dificuldades que os estudantes de programação concorrente encontram na depuração de seus códigos, é proposto este trabalho que envolve duas áreas de atuação diferentes: a depuração de programas paralelos e o ensino da programação paralela.

O objetivo é criar uma ferramenta que auxilie um usuário de programação concorrente, sem muita experiência, a localizar erros em programas paralelos. Mas a ferramenta deve ainda, possibilitar o aumento do conhecimento do usuário durante a sua utilização. Ou seja, a ferramenta a ser desenvolvida não é um mero depurador de programas paralelos, mas uma ferramenta que induza o usuário a descobrir seus erros. Desta forma, acredita-se que além de localizar erros em seus programas, o programador estará aprendendo como fazê-lo.

Criar ferramentas que auxiliem o usuário resolver os problemas que surgem durante o desenvolvimento de um *software* paralelo é importante, mas é fundamental capacitá-lo a desenvolver *software* mais confiáveis, através do treinamento que pode ser alcançado com a própria análise de seus erros com sugestões e comentários da forma ideal de se realizar aquele trabalho.

Muitas atividades têm sido desenvolvidas recentemente sobre computação paralela, porém os rumos de tal tecnologia de programação ainda estão se definindo, por falta de uma metodologia efetiva de programação. Desenvolver programas eficientes para computadores paralelos é difícil devido à complexidade e variedade das arquiteturas paralelas.

É preciso um esforço conjugado na busca de soluções simples e práticas para auxiliar na construção de *software* paralelos. Estas ferramentas precisam ser divulgadas para que as idéias não fiquem apenas na Academia, mas possam ser efetivamente utilizadas por aqueles que necessitam de apoio na confecção de seus sistemas.

A depuração de programas paralelos não é uma tarefa simples. E sua difusão está diretamente relacionada com a popularização dos *software* paralelos e o surgimento de ferramentas mais amigáveis para o seu desenvolvimento.

A ferramenta é composta de módulos funcionais que resolvem tipos diferentes de problemas encontrados nos programas paralelos como *deadlock* e sincronismo entre as tarefas.

Inicialmente é definido um modelo de alto nível com as relações dos diversos módulos. As especificações desses módulos permitem que eles possam ser implementados nas diversas linguagens e bibliotecas existentes para a criação de *software* paralelo, particularmente em sistemas distribuídos ou ambientes com mecanismos de troca de mensagem.

1.3 – Organização do Trabalho

O trabalho está organizado da seguinte forma:

O próximo capítulo faz uma apresentação geral dos conceitos que envolvem a programação paralela. O capítulo 3 apresenta um estudo sobre depuração de programas paralelos, mostrando as técnicas que existem nos diferentes tipos de ferramentas para depuração. No capítulo 4 é discutido o problema denominado *deadlock*, enfocando sua ocorrência nos programas paralelos. O capítulo 5 faz uma introdução dos assuntos *software* tutor e *interface* de *software*. O capítulo 6 descreve o projeto da ferramenta de auxílio na depuração de programas paralelos. No capítulo 7 é apresentado o desenvolvimento de um protótipo. E o último capítulo apresenta o comentário final deste trabalho, mostrando as suas contribuições e propostas para trabalhos futuros.

Alguns termos foram mantidos conforme o original em inglês, na tentativa de não prejudicar a legibilidade do texto.

Capítulo 2
Programação Paralela

Até bem pouco tempo, não se ouvia falar em programação paralela com muita freqüência. A maioria dos desenvolvedores não tinha acesso a máquinas paralelas e as técnicas de programação que eles utilizavam se baseavam nos conceitos da programação seqüencial.

Nos dias atuais, a programação paralela está cada vez mais correlacionada aos sistemas distribuídos e o aumento na sua utilização está diretamente relacionada com a crescente popularização desse tipo de sistema. Ela está deixando os ambientes "elitizados" onde se encontrava até bem pouco tempo, como as universidades, centros de pesquisa e grandes empresas que tinham condições de adquirir supercomputadores que a maioria das pessoas sequer conheciam, para uma maior atuação nos diversos campos de aplicação da ciência em geral.

É fácil entender o grande interesse pela programação paralela, afinal ela está associada com o alto desempenho que todos os usuários buscam. Durante anos, o desempenho foi sinônimo de aumento no poder de processamento dos computadores. Atualmente, o desempenho pode ser conseguido através da otimização dos programas, além da paralelização dos códigos, na maioria seqüenciais, para o uso em sistemas distribuídos, que podem simular máquinas paralelas.

Muitas empresas possuem a estrutura para implantar um sistema distribuído, mas não fazem isso porque seus *software* não exploram o paralelismo, sendo códigos estritamente seqüenciais e seus programadores não sabem trabalhar com programas paralelos. Aliás, esta é a grande dificuldade para a popularização da computação paralela e de certa forma, também dos sistemas distribuídos, ou seja, a falta de conhecimento dos desenvolvedores, com relação

a estas técnicas.

Implementar um algoritmo sobre um computador paralelo, ou mesmo sobre um sistema distribuído, necessita de recursos de programação paralela para expressar o paralelismo entre as tarefas e incluir mecanismos para sincronização e comunicação destas tarefas.

2.1 – Tipos de Programação

Qual é a dificuldade encontrada pelos programadores para mudar o paradigma da programação seqüencial para a programação paralela? Antes de responder, é preciso diferenciar esses dois tipos de programação. Um Programa seqüencial é uma lista de comandos que são executados, instrução após instrução, para se atingir um objetivo proposto. Um processo é geralmente definido como um programa seqüencial que está executando. Um programa paralelo é um conjunto de programas seqüenciais que trabalham juntos e que possuem mecanismos que permitem a esses programas trocarem informações e se sincronizarem, para atingir um objetivo comum. Em um sistema com multiprogramação podem existir muitos processos executando em um determinado momento, mas não formam necessariamente um programa paralelo, pois cada um está buscando atingir um objetivo que lhe é próprio. Neste trabalho, os processos que compõem um programa paralelo serão chamados de tarefas, para diferenciar dos processos seqüenciais.

É comum ainda, a utilização do termo programação concorrente. Quando um programa paralelo executa em uma máquina com apenas um processador, o sistema operacional necessita escalonar as tarefas do programa paralelo, criando um pseudoparalelismo, permitindo que o desenvolvimento das tarefas possa se dar de forma concorrente. Daí o termo programação concorrente. Quando uma máquina paralela possui um número de processadores em quantidade menor que o número de tarefas que compõe um programa paralelo, ocorrem as duas situações, ou seja, existem várias tarefas concorrendo ao uso dos processadores em um ambiente paralelo.

Em termos de programação não existem diferenças entre programação concorrente e programação paralela, sendo os dois termos utilizados indistintamente nesta monografia. É evidente que as técnicas de programação recebem influência da arquitetura utilizada, provocando algumas diferenças na codificação. Por exemplo, em um programa paralelo, sendo executado em uma máquina com apenas um processador, portanto concorrente, existe apenas um endereçamento de memória, que é compartilhado por todas as tarefas da aplicação. Nesse caso, as políticas de sincronização e comunicação que deverão ser utilizadas serão

aquelas próprias para os sistemas fortemente acoplados, como semáforos e monitores, que serão discutidos no item 2.4. Se o mesmo programa executar em um sistema distribuído, a comunicação só poderá ser feita por mecanismos de troca de mensagens.

Respondendo a pergunta efetuada no primeiro parágrafo desta seção (2.1). Existem várias razões para explicar a dificuldade que os programadores encontram para migrar da programação seqüencial para a programação paralela. Uma delas é a própria característica das linguagens seqüenciais que evoluíram muito, proporcionando um ambiente de desenvolvimento amigável, que facilita a criação dos programas e a sua depuração caso seja constatado algum problema. Muitas linguagens paralelas, ainda são escritas em editores com poucos recursos (se comparadas as linguagens seqüenciais), separados do compilador que, geralmente, é acionado por linhas de comandos complexas.

É preciso considerar ainda, que as linguagens paralelas não possuem a mesma quantidade bibliográfica se comparadas às linguagens puramente seqüenciais. Esta situação desencoraja o programador ainda inexperiente e que sente dificuldade em entender os conceitos que envolvem a programação paralela. Além disso, um programa que executa como se espera em uma arquitetura paralela, se for compilado para executar em uma outra, pode não alcançar o mesmo desempenho, ou seja, o desenvolvedor, além de saber programar, deve entender que não se faz programação paralela separada da arquitetura paralela.

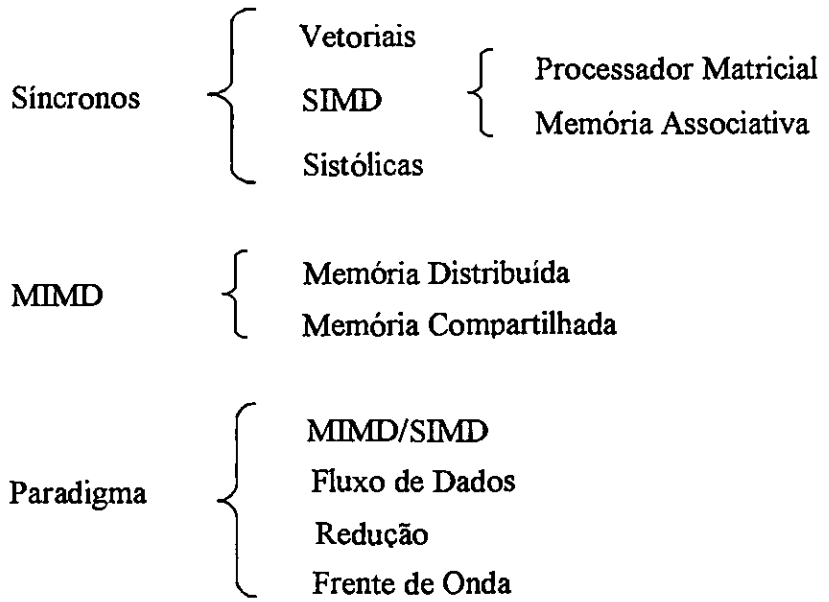
Já que a arquitetura em que um programa paralelo está executando interfere diretamente nas técnicas de programação, será feita uma breve introdução sobre o tema.

2.2 – As Arquiteturas Paralelas

As arquiteturas paralelas têm se apresentado como o futuro para computação de alto desempenho nas mais diversas aplicações. A barreira para sua difusão, como já foi colocado, é que escrever programas paralelos que sejam eficientes e portáveis é muito difícil. A programação paralela apresenta uma complexidade muito maior que a programação seqüencial porque programas paralelos não somente expressam a computação seqüencial, mas também as interações entre as tarefas que definem o paralelismo [Browne et al., 1995].

Diversas taxonomias para arquiteturas paralelas foram propostas. As duas mais utilizadas são a de Flynn [Flynn, 1972] e a de Duncan [Duncan, 1990]. Com o objetivo de abranger os computadores vetoriais e as arquiteturas híbridas, que não foram detalhados na classificação de Flynn, Duncan apresentou uma taxonomia mais abrangente, mostrada na figura 2.1.

Figura 2.1 – Taxonomia de Duncan.



A classificação de Flynn enxergava o processo computacional como uma relação entre os fluxos de instrução e os fluxos de dados. Por fluxo de instrução, deve-se entender a seqüência de instruções executadas sobre um fluxo de dados relacionados. Flynn considerou a unicidade e multiplicidade dos fluxos de dados e instruções. Através das combinações possíveis Flynn chegou a quatro classes de arquiteturas:

(1) **SISD** (*Single Instruction Stream / Single Data Stream*) que representa o modelo tradicional von Neumann. Nesta classificação um processador executa, de maneira seqüencial, um conjunto de instruções sobre um conjunto de dados;

(2) **SIMD** (*Single Instruction Stream / Single Data Stream*) que possui vários processadores controlados por uma única unidade de controle. Executam simultaneamente a mesma instrução em diversos conjuntos de dados;

(3) **MISD** (*Multiple Instruction Stream / Single Data Stream*) que envolve vários processadores executando diferentes instruções em um único conjunto de dados; e

(4) **MIMD** (*Multiple Instruction Stream / Multiple Data Stream*) que apresenta vários processadores executando diferentes instruções em diferentes conjuntos de dados, de maneira independente. Esta classe engloba a maioria dos computadores paralelos.

Em relação à classificação de Flynn, Duncan ainda excluiu as arquiteturas que somente apresentassem fluxo único de instrução e fluxo único de dados (SISD – *Single Instruction Single Data Stream*), por se tratar de uma abordagem seqüencial e as arquiteturas MISD (*Multiple Instruction Stream / Single Data Stream*), pois não existe nenhuma

arquitetura com esta abordagem. Além disso, Duncan separou as arquiteturas em Síncronas e Assíncronas. As primeiras dizem respeito às arquiteturas que coordenam as operações paralelas sincronamente em todos os processadores, utilizando o mesmo relógio global e unidades de controle únicas. Já as arquiteturas assíncronas possuem o controle descentralizado do *hardware*, proporcionando uma maior independência entre os processadores.

Entre os tipos de arquiteturas Síncronas, além das arquiteturas de classificação SIMD (*Single Instruction Multiple Data Stream*) proposta por Flynn, estão os Processadores Vetoriais que possuem um *hardware* específico para a manipulação de vetores e as Arquiteturas Sistólicas que possuem vários processadores dispostos em uma seqüência (*pipeline*), realizando o acesso a memória através dos processadores localizados nas extremidades desta cadeia.

Duncan ainda subdividiu a arquitetura SIMD em Processadores Matriciais, criados para computação sobre matrizes de processadores, e Memória Associativa, onde o acesso à memória é feito através do seu conteúdo e não através de endereçamento que é o mais comum.

Em sua taxonomia, Duncan dividiu as arquiteturas Assíncronas em MIMD e Paradigma MIMD. A arquitetura MIMD (*Multiple Instruction, Multiple Data Stream*), também já havia sido considerada por Flynn, mas Duncan ainda subdividiu esse tipo de arquitetura em Memória Distribuída e Memória Compartilhada. Esta divisão está relacionada à existência de uma memória única, onde todos os processadores podem ler e escrever em todas as posições, no caso de memória compartilhada, ou a existência de várias memórias, uma para cada processador, e que somente pode ser endereçada pelo processador correspondente, para as arquiteturas com memória distribuída.

As arquiteturas Paradigma MIMD englobam todas as arquiteturas que, apesar de possuírem características MIMD, possuem detalhes muito específicos em seus projetos. Esse tipo de arquitetura foi subdividido em MIMD/SIMD ou híbridas, Fluxo de Dados, Redução e Frente de Onda.

2.3 – Granulosidade

Granulosidade, ou nível de paralelismo, é o termo usado para especificar o tamanho das tarefas que compõe um programa paralelo. A granulosidade está relacionada diretamente com a arquitetura utilizada e é dividida em grossa, média e fina. A granulosidade grossa relaciona o paralelismo ao nível de programas. Em sistemas distribuídos, por exemplo, é a

granulosidade ideal, devido à necessidade de haver pouca comunicação entre as tarefas que estão executando em máquinas distintas na rede. A granulosidade fina relaciona o paralelismo ao nível de instruções, aplicando-se a arquiteturas fortemente acopladas através de memória centralizada e que possuem um grande número de processadores simples. A granulosidade média situa-se entre as duas anteriores.

A arquitetura e a granulosidade das tarefas repercutem diretamente nas técnicas a serem adotadas para programar paralelamente, devido aos mecanismos para comunicação que devem ser utilizados pelas tarefas que compõem a aplicação. A solução adotada, sob o ponto de vista de *software*, para uma arquitetura e sua granulosidade, pode ser completamente incompatível em uma arquitetura paralela com características diferentes. Por exemplo, uma transformação elementar que se deseja realizar em uma matriz de grandes dimensões. Se o programa estiver executando em uma arquitetura SIMD com hardware específico para a manipulação de matrizes e um grande número de processadores simples, será interessante dividir a grande matriz em muitas matrizes pequenas, distribuindo-as para os diversos processadores. Se a mesma solução fosse adotada em uma arquitetura MIMD com memória distribuída, possuindo poucos processadores e se comunicando através de uma rede com pequena largura de banda, seria necessário muita comunicação entre os processadores, aumentando o gargalo da rede e produzindo um efeito muito pior. Uma solução seria a divisão do programa em um número de matrizes menores, na quantidade de processadores, mas ainda grandes o suficiente para compensar a transferência desses valores entre eles.

A arquitetura adotada influencia diretamente nas técnicas de comunicação e sincronismo que devem ser utilizadas em um programa paralelo. A seguir serão discutidos alguns dos mecanismos mais usados e em que situação eles se aplicam.

2.4 – Mecanismos de Sincronização e Comunicação

Como as tarefas que compõem um programa paralelo necessitam trabalhar em conjunto, buscando atingir um objetivo comum, fica evidente a necessidade destas tarefas trocarem informações. Também pode ser observado que em um trabalho coletivo, principalmente em um ambiente heterogêneo, as tarefas não executam com a mesma velocidade. Isto pode ser causado por vários fatores, como a carga do processador ou seu poder de processamento. Esta diferença na velocidade, em muitas situações, provoca a necessidade de sincronizar as tarefas através de bloqueios.

As técnicas que garantem a comunicação entre as tarefas (ou processos) e o acesso de recursos compartilhados são chamados de *mecanismos de sincronização*. A maioria dos

problemas que ocorrem nos programas paralelos está vinculada a uma má utilização desses mecanismos durante a codificação.

Outro conceito importante que surge quando se estuda o problema da sincronização e comunicação entre os processos é a chamada *Região Crítica*. A região crítica é uma área do código de um processo que faz referência a dados compartilhados. Muitos problemas podem surgir quando vários processos realizam o acesso a dados ou recursos compartilhados de forma paralela ou concorrente. Alguns desses problemas serão vistos no capítulo 3, sobre a depuração de programas paralelos. Os mecanismos de sincronização devem evitar a concorrência nas regiões críticas, permitindo que somente um processo (ou tarefa) esteja executando sua região crítica de cada vez. Essa idéia de exclusividade de acesso é denominada *Exclusão Mútua* [Axford, 1989] [Ben-Ari, 1990] [Shay, 1996] [Tanenbaum, 1992].

Diversos mecanismos de sincronização são discutidos na literatura. Nas próximas seções serão apresentados os mecanismos por troca de mensagens (utilizados em sistemas com memória distribuída), as soluções de *hardware*, semáforos e monitores (utilizados em sistemas com memória compartilhada).

2.4.1 – Soluções de *Hardware*

Uma solução de *hardware* bem simples para implementar o problema da exclusão mútua é fazer com que o processo que necessita entrar em sua região crítica desabilite todas as interrupções externas e as reabilite quando deixar a região crítica. Como a mudança de estados dos processos só pode ser realizada através de interrupções, após um processo desabilitar as interrupções não será possível a troca de contexto até que elas sejam novamente habilitadas.

Essa solução traz uma série de inconvenientes, sendo o maior deles o comprometimento do sistema operacional se um processo desabilitar as interrupções e não reabilitá-las.

Alguns processadores possuem instruções atômicas que lêem uma variável e armazenam o seu conteúdo em uma outra área de memória atribuindo um novo valor a essa variável. Esse tipo de instrução também permite a criação de mecanismos de controle para o acesso da região crítica de um processo.

2.4.2 – Semáforos

As primeiras soluções por *software* apresentadas tinham a deficiência da *espera ocupada*. Na espera ocupada, quando um processo deseja entrar em sua região crítica ele fica

perguntando para o sistema se o recurso já está liberado. Essa solução é inadequada, por que o processo não é bloqueado caso o recurso não esteja disponível. Ele volta para a fila do processador aguardando por uma nova chance de executar podendo retornar ao processador antes que o recurso esteja disponível. Ou seja, essa solução acaba provocando um alto custo no sistema como um todo.

O conceito de semáforo foi introduzido por Dijkstra citado em [Axford, 1989] [Ben-Ari, 1990] [Tanenbaum, 1992], sendo uma solução mais simples.

Um semáforo é uma variável inteira, não negativa, que só pode ser manipulada pelas primitivas atômicas DOWN e UP (listagem 2.1). Essas primitivas funcionam como protocolos de entrada e saída no acesso à região crítica.

Listagem 2.1 – As primitivas DOWN e UP.

```
Procedimento DOWN (S : Semáforo)
inicio
      se S = 0 então Bloqueia_Processo
      senão S ← S - 1
fim

Procedimento UP (S : Semáforo)
inicio
      se Tem_Processo_Bloqueado
            então Libera_Um_Processo
      senão S ← S + 1
fim
```

O funcionamento é bastante simples: se um processo deseja entrar em sua região crítica, ele realiza a operação DOWN em um semáforo criado para controlar o acesso àquele código. Se o semáforo estiver valendo 1, ele será decrementado e o processo continua sua execução, ou seja, entra na região crítica. Se, ao contrário, o semáforo estiver valendo 0, significa que já existe outro processo na região crítica e o processo que executou a operação DOWN é bloqueado. Quando um processo termina a execução da sua região crítica, a primitiva UP é executada, liberando a região crítica para outro processo. A primitiva UP verifica inicialmente se existe algum processo bloqueado. Se isto ocorre ela libera um deles, caso contrário incrementa o semáforo, sinalizando a disponibilidade da região crítica.

Semáforos que se aplicam ao problema da exclusão mútua são também chamados de *mutex* ou semáforos binários, pois assumem apenas os valores 0 e 1. Porém, os semáforos podem ser utilizados para resolver uma série de outros problemas de sincronização entre processos, podendo assumir qualquer valor inteiro positivo.

2.4.2 – Monitores

O uso de semáforos exige dos programadores bastante atenção, pois os problemas que podem resultar do uso incorreto dos semáforos podem ser difíceis de reproduzir devido ao não determinismo das tarefas. Dessa forma, foi proposto por Hoare, citado em [Ben-Ari, 1990], o mecanismo denominado monitor.

Um monitor é um conjunto de procedimentos, variáveis e estruturas de dados definido dentro de um módulo (semelhante a uma classe, da programação orientada ao objeto). A característica mais importante do monitor é a implementação automática da exclusão mútua, pois somente um processo pode estar executando os procedimentos do monitor em um determinado instante. Se um processo requisitar um procedimento do monitor, será verificado se já existe outro processo executando algum procedimento do monitor. Se isto ocorre o processo fica aguardando até o monitor estar disponível novamente.

A principal diferença entre monitores e semáforos é que a exclusão mútua, no caso do monitor, é implementada automaticamente pelo compilador e não pelo programador, como é o caso dos semáforos.

2.4.3 – Troca de Mensagens

Em sistemas com memória distribuída, se torna inviável o uso de semáforos ou monitores, pois os processos que executam em processadores diferentes não possuem acesso ao mesmo endereçamento de dados. Nesse caso a única maneira de haver comunicação é através da troca de mensagens que é realizada pelas primitivas SEND e RECEIVE. Estas primitivas permitem realizar a sincronização e a comunicação entre os processos.

Existem alguns problemas que surgem em ambientes com troca de mensagens. O principal deles é a perda de uma mensagem. Uma maneira de se garantir que as mensagens cheguem no destino, é forçando o processo receptor a enviar uma mensagem de confirmação para o emissor, que aguarda o recebimento dessa mensagem por um determinado tempo. Se isto não ocorrer, ele envia novamente a mensagem.

Existem, basicamente, duas formas de comunicação entre processos pela troca de mensagens: a comunicação síncrona e a comunicação assíncrona.

A comunicação síncrona acontece quando um processo que envia uma mensagem fica esperando até que o processo receptor leia a mensagem ou quando o processo receptor aguarda pelo envio de uma mensagem por algum processo. A comunicação assíncrona ocorre quando nenhum processo fica bloqueado, nem o processo emissor e nem o receptor. Nesse caso é necessário a utilização de *buffers* para o armazenamento temporário das mensagens.

A vantagem da comunicação assíncrona é o maior paralelismo entre as tarefas. Já a comunicação síncrona permite a realização da sincronização entre os processos.

2.5 – A Elaboração de Um Programa Paralelo

Normalmente, existem vários caminhos para se resolver um problema através da programação seqüencial. O mesmo ocorre com a programação paralela. Infelizmente, nem sempre a lógica que foi expressa na resolução seqüencial de um problema funciona para a solução paralela. Às vezes, o resultado pode ser bem pior, não alcançando a eficiência almejada. Assim, muitas vezes é necessária uma abordagem do problema totalmente nova.

Para se resolver um problema utilizando programação paralela, pode-se utilizar as seguintes abordagens:

(1) **Estudo das partes**: essa abordagem parte do programa seqüencial e procura localizar as partes que podem ser realizadas em paralelo. A granulosidade depende da arquitetura e também da linguagem em que está sendo desenvolvido o programa. Mas pode ser explorado desde uma granulosidade fina até uma granulosidade média, de acordo com a característica do programa.

(2) **Começar do zero**: criar um novo algoritmo paralelo permite alcançar um melhor desempenho, pois é possível desenvolver uma solução que explore com muito mais eficiência o paralelismo que a arquitetura a ser utilizada proporciona.

(3) **Utilizar outro algoritmo paralelo**: com essa abordagem, procura-se melhorar o desempenho do algoritmo paralelo existente. Essa solução exige um esforço menor do programador em identificar as partes paralelas e pode alcançar um bom desempenho.

Quando se projeta um algoritmo paralelo deve-se preocupar também com a quantidade de comunicação a ser utilizada pelo programa, pois isto aumenta o custo. É importante lembrar que dependendo da escolha adotada na concepção do algoritmo, pode-se desenvolver um programa muito pior em eficiência que a versão seqüencial ou outra versão paralela já existente.

2.5.1 – O Ciclo de Vida de Um Programa Paralelo

Um programa paralelo é, geralmente, bem mais complexo que um programa seqüencial. Portanto, na concepção de um programa paralelo é importante ter em mente os aspectos particulares da programação paralela, com vistas a não se criar um programa inadequado.

As etapas na criação de um programa paralelo podem ser resumidas em:

(1) **Criação do algoritmo paralelo**: os aspectos que envolvem a criação de um algoritmo paralelo, dizem respeito a adaptação do problema a ser resolvido com a arquitetura que será usada para se implementar o algoritmo. Como já foi visto, as técnicas de comunicação e sincronismo estão diretamente relacionadas com a arquitetura e a granulosidade utilizadas. A abordagem também é importante, pois muitos algoritmos que necessitam de uma versão paralela já possuem uma versão seqüencial, o que pode ser um bom começo.

(2) **Desenvolvimento do Programa**: este é o momento de se escolher a linguagem a ser utilizada, como serão ativados os processos paralelos, e como serão realizadas às comunicações e os sincronismos entre as tarefas. Muitas linguagens já possuem paralelização automática, o que facilita a implementação. Essa etapa é o momento do usuário se familiarizar com a ferramenta que será adotada ou escolher aquela em que ele já possui experiência, se for possível utilizá-la.

(3) **Mapeamento**: o mapeamento é uma exclusividade dos programas paralelos. Após o desenvolvimento do programa é preciso definir em quais processadores as tarefas que compõem o programa serão executadas. Essa etapa é fundamental para se conseguir alcançar eficiência através de um bom balanceamento de carga.

(4) **Teste e depuração**: o ideal é que um programa seja testado com vários conjuntos de valores de entrada possíveis. Em grandes programas é difícil cobrir todas as possibilidades devido ao grande número de combinações que os valores de entrada podem assumir. Em programas paralelos, isto é ainda pior devido ao não determinismo das aplicações, como será visto no próximo capítulo. Existem vários tipos de testes [Pressman, 1992]:

- Teste de caixa preta: faz uma análise exclusiva dos resultados obtidos durante o teste.
- Teste de caixa branca: baseia-se em um exame detalhado dos procedimentos.

Para se testar programas paralelos é possível usar uma das quatro estratégias abaixo:

- Teste de tarefas: os testes de caixa preta e caixa branca são projetados e executados para cada tarefa, ou seja, preocupa-se com a parte seqüencial do código. Esse tipo de teste pode revelar erros de lógica e erros funcionais, mas não revela erros de comunicação e sincronismo.
- Teste comportamental: através do uso de modelos do sistema, é possível simular o comportamento do programa paralelo. Esse modelo pode servir

como a base para a definição dos casos de testes que serão realizados.

- Teste do relacionamento entre as tarefas: o objetivo desse tipo de teste é, após o isolamento dos testes anteriores, verificar a possível ocorrência de erros de comunicação e sincronismo.
- Teste do sistema: esse tipo de teste tenta verificar o conjunto *hardware/software*, pois estão fortemente casados na programação paralela.

A depuração é uma conseqüência de um teste bem sucedido, ou seja, quando um erro foi detectado. A depuração de programas paralelos será discutida em detalhes no próximo capítulo.

(5) **Avaliação de desempenho**: o objetivo da programação paralela é o aumento da velocidade do processamento. Pouco adianta um programa paralelo que executa sem erros, mas não consegue maior eficiência que o seu correspondente seqüencial. Para medir o ganho computacional de um programa paralelo em relação a um programa seqüencial são usados dois tipos de medidas: *speedup* e eficiência [Almasi e Gottlieb, 1994]. *Speedup* é o aumento da velocidade quando se executa um determinado programa em p processadores em relação à execução desse mesmo processo em um processador. Eficiência relaciona o *speedup* com o número de processadores.

2.6 – Plataformas de Portabilidade

Existem várias ferramentas para o desenvolvimento de programas paralelos. O usuário, ao escolher a ferramenta, deve se pautar no tipo de aplicação e na arquitetura. Segundo Almasi [Almasi e Gottlieb, 1994], há três tipos de ferramentas: ambientes de paralelização automática, extensões paralelas para linguagens seriais e as linguagens concorrentes. Os compiladores capazes de gerar um código paralelo proporcionam uma codificação mais simples por parte do programador, mas nem sempre é possível alcançar um bom desempenho. As extensões das linguagens paralelas são bibliotecas contendo várias instruções para se realizar as operações paralelas, complementando as linguagens seriais já existentes. As linguagens especialmente criadas para a programação concorrente proporcionam um alto desempenho e a criação de códigos bem estruturados.

As plataformas de portabilidade ou bibliotecas para programação distribuída, que se enquadram nas extensões das linguagens paralelas, foram criadas inicialmente para facilitar o desenvolvimento de aplicações paralelas em máquinas com processamento maciçamente paralelo, mas passaram a dar suporte a programação paralela em sistemas distribuídos,

expandindo a utilização dos sistemas distribuídos, que inicialmente objetivavam apenas o compartilhamento de recursos.

Essas bibliotecas fornecem uma extensão para algumas linguagens seqüenciais mais comuns (como C e Fortran), permitindo a elaboração de aplicações paralelas. Outra característica interessante, é que executando em um sistema distribuído, é possível utilizar computadores heterogêneos o que traz vantagens e desvantagens, conforme discussão a seguir. Essa característica diferencia bastante um sistema distribuído de uma arquitetura maciçamente paralela, que possui todos os processadores homogêneos.

As maiores dificuldades encontradas em sistemas heterogêneos são:

- A diferença de potência computacional. Isto cria obstáculos a um balanceamento de carga eficiente, devido a dificuldade de se estabelecer comparações nos diversos tipos de processadores;
- Os formatos dos dados podem ser diferentes. Dados diferentes enviados de um computador para outro devem ser traduzidos para permitir a comunicação;
- Um sistema distribuído pode ser composto por máquinas de arquiteturas diferentes e muitas vezes é difícil prever todas as características das máquinas em que um programa executará, dificultando o alcance de um programa que explora todo o potencial das máquinas envolvidas no sistema.

Apesar disso, a computação paralela sobre sistemas distribuídos oferece grandes vantagens como:

- Menor custo, se comparado às arquiteturas paralelas;
- Escalabilidade;
- Possibilidade de se alcançar um alto desempenho, atribuindo para cada tipo de arquitetura a tarefa mais apropriada.

Um fator importante, particularmente para o enfoque deste trabalho, é que a depuração paralela em um ambiente de troca de mensagens tende a ser mais fácil que em uma linguagem explicitamente paralela.

Exemplos dessas bibliotecas são: P4, Parmacs, Express e principalmente PVM (*Parallel Virtual Machine*) e MPI (*Message Passing Interface*).

Capítulo 3
Depuração de Programas Paralelos

Construir um *software* que alcance um alto grau de confiabilidade é freqüentemente muito difícil, e para alcançar esse objetivo, o custo envolvido pode tornar-se muito alto. Vários sistemas existentes não são confiáveis, mesmo assim, são muito utilizados [Axford, 1986]. Nota-se que entre as diversas razões que contribuem para essa situação, uma decisiva é que a alta confiabilidade não é o objeto de interesse principal, sendo colocada em segundo plano quando se defronta o custo adicional que as fases de testes e depuração acrescentam. É curioso notar, entretanto, que sistemas não confiáveis podem multiplicar o custo gasto para o seu desenvolvimento de forma inaceitável, tornando os usuários menos tolerantes a baixa confiabilidade com o passar do tempo.

Os primeiros computadores foram usados em pesquisas científicas e trabalhos específicos, sendo comum o desenvolvimento de *software* de certa forma especulativa e com baixa confiabilidade. Não existiam técnicas claramente confiáveis e o desenvolvimento de um *software* era uma tarefa artesanal. As máquinas eram lentas e caras e, normalmente, o desempenho era o foco principal.

Nos dias atuais, a situação tem mudado radicalmente. O *hardware* apresenta um custo mais acessível e um desempenho superior e o *software* adquiriu uma função crucial em várias aplicações com muita demanda de tempo. Essa atenção que vem sendo dada ao *software* é essencial e aumenta a preocupação que se deve ter na criação de sistemas que não sejam apenas eficientes, mas que também apresentem um alto grau de confiabilidade.

Essa confiabilidade só poderá ser atingida se forem observados dois objetivos básicos:

(1) Melhorar o nível dos procedimentos envolvidos nas etapas de desenvolvimento do *software*;

(2) Aperfeiçoar as técnicas para se testar e corrigir programas, fortalecendo o refinamento de possíveis falhas.

Em princípio, esses objetivos têm sido alcançados quando se trata de programas seqüenciais. Entretanto, um aumento considerável na aplicação dos programas paralelos, devido ao crescimento dos ambientes paralelos e principalmente na popularização do uso das redes de computadores e dos sistemas distribuídos, vem sendo observado. Considerando ainda a grande complexidade desses programas, surge a necessidade de se criar programas para a depuração de códigos paralelos mais eficientes que os programas responsáveis na depuração dos códigos seqüenciais [McDowell e Helmbold, 1989]. Têm surgido várias técnicas para se alcançar o primeiro objetivo (desenvolvimento de *software*), mas as pesquisas sobre as técnicas de depuração ainda são bastante teóricas. Devido ao fato de existirem vários ambientes e linguagens para o desenvolvimento de sistemas paralelos, algumas das poucas ferramentas de depuração que existem acabam caindo no desuso, pois algumas linguagens, ou mesmo plataformas, com o passar do tempo se tornam pouco utilizadas.

É curioso notar que a depuração tem sido considerada mais uma arte do que uma ciência [Cheung et al., 1990]. Muitas vezes a depuração é um processo intuitivo. O problema é que nem sempre é possível garantir que a intuição está correta, assim, muitos desenvolvedores encontram grande dificuldade para localizar erros em seus sistemas. Isto é provocado por várias situações diferentes [Pressman, 1992]. Em muitas ocasiões os sintomas surgem em um determinado ponto da execução do programa, entretanto a causa pode se localizar em outra parte, o que dificulta a sua correção. Outra situação bastante encontrada é que alguns erros são causados por problemas de *hardware*, podendo os sintomas desaparecer.

Quando um sistema é extremamente grande, reproduzir as entradas que provocaram a execução errada é uma tarefa quase impossível. Essa situação chama atenção, pois, sistemas paralelos tendem a ser enormes, e se já é difícil reproduzir a execução em um ambiente seqüencial, em um ambiente paralelo a situação torna-se ainda mais complexa, devido ao não determinismo das tarefas paralelas.

Em adição, é interessante comentar sobre os aspectos psicológicos relacionados com a busca de erros nos sistemas. Muitos programadores encontram dificuldade em admitir que seus programas possam estar errados e que necessitam de depuração. Quanto a isso Shneiderman, citado em [Pressman, 1992] afirma:

> *"A depuração é uma das partes mais frustrantes da programação. Ela tem elementos de solução de problemas ou de complicações, combinados com o aborrecido reconhecimento de que se cometeu um erro. A elevada ansiedade e a pouca disposição*

para aceitar a possibilidade de erros aumenta a dificuldade da tarefa. Felizmente, vem um grande suspiro de alívio e uma diminuição da tensão quando o bug é por fim... corrigido".

É importante salientar a necessidade de se evitar os erros durante o desenvolvimento do programa, pois muitos programadores repetem os mesmos erros em sistemas diferentes. Geralmente, isto ocorre devido à falta de uma metodologia adequada durante o desenvolvimento e, também, durante a depuração. Os programadores realizam várias tentativas até conseguir que o programa funcione conforme as especificações, e quando isto ocorre, eles não conseguem perceber o que realmente estava provocando a falha. Por isso, é importante que as ferramentas de depuração não sejam somente "detetives" na busca de "culpados", mas que assumam também a função de "professores", alertando aos usuários quanto aos motivos que geraram os erros. Essa abordagem permite a aquisição de conhecimento pelo desenvolvedor durante a etapa de depuração, já que uma forma de também melhorar o processo de desenvolvimento é aumentar o senso crítico do desenvolvedor, bem como seus conhecimentos na área.

3.1 – Características Especiais na Depuração dos Programas Paralelos

A depuração de programas paralelos, em sua essência, é similar à depuração dos programas seqüenciais, ou seja, é a busca por problemas que provocam um comportamento não esperado no programa. Entretanto, a programação paralela tende a ser muito mais complexa, e muitos usuários, principalmente principiantes, encontram grande dificuldade em se adaptar aos conceitos inerentes a esse tipo de programação.

A adoção de linguagens concorrentes no desenvolvimento de *software* e as dificuldades desse tipo de programação, têm acentuado a necessidade de se criar um conjunto integrado de ferramentas para auxiliar neste desenvolvimento [Baiardi et al., 1986].

Pode-se observar várias diferenças entre programas paralelos e programas seqüenciais. A maior delas é o não determinismo global. Em programas seqüenciais, quando é selecionado um conjunto de valores de entrada, o fluxo de controle dos programas sempre segue por um mesmo caminho, dessa forma, o processamento sempre provoca resultados iguais. Entretanto, quando se trata de programas paralelos, isto nem sempre é verdadeiro, pois a necessidade de sincronização e comunicação das várias tarefas que compõem o programa e o tempo gasto por cada tarefa, pode resultar em comportamentos diferentes, se o programa não for bem codificado.

É possível agrupar em 4 categorias as diferenças que dificultam a depuração em programas paralelos e torna esse tipo de programação uma tarefa trabalhosa:

(1) **Tipo do processo**: as características de cada processo interferem diretamente na velocidade em que ele é processado. Um processo pode necessitar somente da utilização da CPU, como por exemplo, os programas que realizam muitas operações numéricas. Esse tipo de processo é chamado de *CPU-bound* e o seu tempo de execução depende exclusivamente da velocidade do processador e o tempo de espera na fila. Outro tipo de processo é o denominado *I/O-bound*. Essa classe de processos possui a característica de realizar muito acesso aos mecanismos de armazenamento de informação como acontece nos sistemas de gerência de banco de dados. Os processos podem ainda conter ambas as características, o que é muito comum [Tanenbaum, 1992]. Essas diferenças não permitem afirmar com precisão a ordem em que os eventos ocorrerão.

(2) **Vários processadores**: o tempo gasto para a execução de cada tarefa pode resultar em comportamento diferente no sistema como um todo. Isto é, o tempo total de cada tarefa, depende da velocidade do processador, o seu tempo de espera na fila e da quantidade de processadores no ambiente paralelo, pois um maior número de processadores tende a diminuir o tamanho da fila de espera de cada tarefa. Como cada processo que compõe um programa paralelo pode executar em diferentes processadores fisicos, torna-se ainda mais difícil descobrir o comportamento errôneo do programa e a gerência dos processos de depuração.

(3) **Tempo de comunicação**: os processadores nos quais um programa paralelo executa, podem estar geograficamente dispersos, como é o caso dos sistemas distribuídos. Isto pode provocar atrasos na comunicação tornando algumas operações de depuração impraticáveis. Normalmente já existe um retardo entre a ocorrência de um erro e a sua descoberta, mesmo nos programas seqüenciais. O grande problema nos programas paralelos é que um erro pode se propagar bastante, até que seja detectado [Cheung et al., 1990].

(4) **Tamanho do Sistema**: os sistemas paralelos tendem a ser grandes, ou seja, possuírem um número de processos elevado, além de possuírem muito mais recursos envolvidos. Assim, quanto maior o sistema, mais difícil é sua depuração, pois é difícil realizar testes que possam gerar todas os caminhos possíveis em um sistema.

(5) ***Probe Effect***: um outro ponto que deve ser considerado na depuração de

programas paralelos é a interferência do depurador nos resultados do programa. Quando a depuração interfere na execução de um determinado processo, o comportamento de outros processos poderá também ser influenciado. Dessa forma, o programa para depuração deverá oferecer mecanismos de análise sem afetar o comportamento do programa, ou seja, a presença do depurador deverá ser transparente ao funcionamento do programa [Baiardi et al., 1986]. Nos programas paralelos, esse esforço em obter mais informações do comportamento de um programa, em muitas situações acaba provocando uma interferência perniciosa, ou seja, o programa de depuração interfere na reprodução do funcionamento "natural" do programa que está sendo depurado. Essa situação é conhecida como *probe effect* [Damitio e Turner, 1995].

(6) **Baixa Visibilidade**: é um termo usado por muitos autores. Quando o programa é executado sobre um sistema distribuído, a ordem de execução das instruções pertencentes aos diferentes processos do programa não pode ser deduzida. Esse fator é fundamental, afinal o grande crescimento da aplicação de programas paralelos vem, principalmente, do aumento da aplicabilidade dos sistemas distribuídos. A sincronização nos sistemas distribuídos é bastante complexa devido à dificuldade de padronizar o relógio das diversas estações que o compõem.

Quando os processos paralelos estão executando com as mesmas entradas, seus resultados podem ser radicalmente diferentes. Essas diferenças são causadas pelas *condições de corrida*, que ocorrem quando duas atividades estão progredindo em paralelo. Por exemplo, um processo deve escrever em uma posição da memória enquanto o segundo processo deverá ler a mesma célula. O segundo processo poderá ler o valor novo ou o antigo, dependendo da ordem de acesso, que pode ser alterada devido à velocidade de execução de cada processo, que como já foi discutido depende do processador, tamanho da fila de processos, tipo de processos, etc.

É interessante notar que se a probabilidade de um mau funcionamento ocorrer for muito pequena, a situação de erro poderá nunca ser recriada.

Essas diferenças tornam a depuração dos programas paralelos uma tarefa extremamente delicada e uma área de pesquisa fascinante.

3.2 – Abordagens Clássicas na Depuração de *Software*

A abordagem clássica para se depurar um programa seqüencial envolve várias paradas durante a execução – *breakpoints*, com o objetivo de examinar o estado das variáveis ou o

fluxo do programa, muitas vezes, sendo necessária uma nova execução de um determinado trecho. Esse estilo de depuração é chamado de *depuração cíclica*. Em cada parada é possível examinar ou mesmo modificar, de forma interativa, parte dos processos que estão sendo examinados. Infelizmente, programas paralelos nem sempre podem ser reproduzidos corretamente, pois é muito difícil definir paradas em termos da precisão dos estados globais. Essa abordagem pressupõe, ainda, um bom conhecimento de todo o sistema para saber os melhores pontos de parada do programa.

A abordagem da *depuração cíclica* freqüentemente falha para programas paralelos porque essas situações de indeterminação não podem ser previstas quando o programa é novamente executado. Em outras palavras, a recriação das circunstâncias que levaram ao erro, pode nunca ser conseguida. Quando o programa é testado com uma entrada, uma simples execução é insuficiente para determinar a exatidão do programa para essa entrada. Durante a depuração de uma execução errônea desse programa, não há garantia de que esta execução será repetida quando o programa for executado novamente, repetindo-se os dados de entrada. Isto é um problema crítico, porque freqüentemente é necessário repetir uma execução errônea da mesma forma para coletar informações adicionais para depuração. Durante uma execução, é difícil salvar todas as informações de depuração que podem ser necessárias para localizar a causa do erro.

Outra abordagem, também muito comum, é a depuração de saída. É a abordagem mais primitiva e, provavelmente, todo o programador já a utilizou. Essa abordagem consiste na apresentação dos estados das variáveis durante a execução do programa. A depuração de saída requer a alteração do programa, o que não ocorre na abordagem de *breakpoints*, e pode provocar o indesejado *probe effect*. Um exemplo fácil de perceber esse efeito é quando dois processos entram em uma situação de um abraço fatal – *deadlock* –, que será visto no próximo capítulo por se tratar de um problema clássico que deve ser observado por um depurador. Nessa abordagem, durante a busca do erro deve ser acrescentada alguma linha de código em um dos processos, com o objetivo de se obter informações desse processo ou de apresentar o estado de uma variável. Esse simples comando pode ser suficiente para sincronizar os dois processos, ou seja, atrasar o processo modificado, fazendo com que o outro processo saia da sua região crítica sem apresentar o problema, criando um comportamento errôneo com relação a sua codificação.

Muitos depuradores apresentam a técnica de *tracing* que é uma junção das duas abordagens anteriores. O depurador utiliza de algum sinal facilmente fornecido pelo sistema operacional, compilador ou ambiente de programação para apresentar informações

selecionadas e que pode ser habilitado ou desabilitado. Mas essa abordagem, quando trabalhando sobre um sistema distribuído, enfrenta os problemas da dificuldade de se ter um estado global.

3.3 – Classificação dos Depuradores para Programas Paralelos

Alguns pesquisadores distinguem entre monitoração e depuração tradicional [McDowell e Helmbold, 1989]. **Monitoração** é o processo de reunir informações sobre um programa em execução. **Depuração** é o processo de localização, análise, correção e eliminação de falhas, onde a falha é definida como sendo uma condição acidental causadora de um erro em um programa que executou alguma de suas funções [Stewart e Gentleman, 1997]. Apesar dessa diferenciação, um processo monitor é freqüentemente um processo que tem como objetivo localizar procedimentos incorretos. Devido ao auxílio que pode ser conseguido com a utilização de um processo monitor, o monitor é visto como uma ferramenta de depuração.

As técnicas de depuração de programas concorrentes têm sido organizadas em quatro grupos:

(1) Técnica de depuração tradicional, que pode ser aplicada com algum sucesso em programas paralelos.

(2) Técnica de depuração baseada em eventos, que encara a execução de um programa paralelo como uma seqüência (ou várias seqüências paralelas) de eventos.

(3) Técnica da visualização do fluxo de controle e distribuição de dados associados com programas paralelos.

(4) Técnica de análise estática baseada na análise do fluxo de dados dos programas paralelos.

A maioria dos programas de depuração atuais se baseia nas técnicas de depuração tradicional e nas técnicas baseadas em eventos, mas existem programas que possuem ambas as características.

Um programa de depuração de programas paralelos com base nas técnicas tradicionais é geralmente, o mais fácil de ser implementado e, portanto, o que proporciona uma solução mais rápida. Entretanto, a grande desvantagem desse tipo de depurador é que em programas com muitos processos concorrentes se torna muito difícil identificar e compreender o que realmente está acontecendo no nível dos processos. Essa técnica de depuração funciona bem na observação do comportamento do programa em nível de instrução ou em nível de procedimento. Isto ocorre porque, geralmente, um depurador paralelo tradicional é formado

por um conjunto de depuradores seqüenciais, sendo normalmente, um para cada tarefa. Outro limitador para essa técnica é a possibilidade de acontecer o *probe effect*, devido a necessidade de interação do programa de depuração com o sistema examinado.

Os depuradores baseados em eventos proporcionam uma melhor abstração que os depuradores tradicionais. Também podem provocar *probe effect*, pois forçam uma execução determinística dos programas paralelos. Enquanto alguns sistemas exibem apenas os eventos ocorridos, outros registram um histórico de eventos, contendo todos os eventos gerados pelo programa. Esse histórico pode ser examinado pelo usuário após o término do programa. O histórico dos eventos pode também ser utilizado para guiar uma nova execução do programa, permitindo a reprodução da computação errônea. Mas, como sempre, nem tudo são vantagens, se o programa for muito grande pode ser impossível armazenar tanta informação.

Diversos sistemas para depuração permitem que um programa seja executado novamente (*replay*) sob o controle de um histórico de eventos. O *replay* não é uma simulação, mas a execução com as mesmas entradas que foram armazenadas no histórico, com um mecanismo que força a ordem dos eventos na execução original [Cheung et al., 1990]. Se o histórico refletir corretamente a comunicação ocorrida entre os processos durante a execução original, então a nova execução produzirá os mesmos resultados. Outra vantagem é que se o histórico é suficientemente completo, um único processo pode ser depurado isoladamente com o histórico fornecendo a comunicação necessária.

Um ponto importante a ser considerado é que a sincronização envolvida durante a nova execução pode diminuir o desempenho do programa paralelo. Em sistemas de tempo real, essa reexecução proporciona diversos problemas. Além da comunicação entre processos a entrada e saída e as interrupções devem também ser registradas, uma vez que sistemas de tempo real geralmente dependem muito do tempo, é importante simular o tempo durante a nova execução, o que requer mais eventos no histórico. Além disso, um problema que já foi apresentado é a necessidade de tornar o depurador transparente, e nem sempre é fácil fazer com que o processo monitor, responsável pela manutenção do histórico de eventos, não interfira na sincronização entre os processos da tarefa examinada.

Os depuradores baseados na técnica de análise estática não apresentam a etapa de teste, pois não necessitam da execução do programa. Geralmente a análise é feita checando as falhas estruturais ao invés de se verificar as falhas funcionais permitindo a eliminação de erros antes que eles ocorram.

As ferramentas de análise estática evitam o *probe effect*, porém não executam programas. Possuem o potencial para identificar grandes classes de erros difíceis de encontrar

em alguns programas usando as técnicas dinâmicas. Essas ferramentas permitem localizar tipos básicos de erros: aqueles causados por problemas de sincronização e os erros nas operações com dados. Os primeiros se referem, por exemplo, aos problemas de *deadlock* e espera infinita. Já os erros causados nas operações com dados, dizem respeito aos erros usualmente seqüenciais como leitura de variáveis não inicializadas e erros no acesso paralelo dos dados compartilhados.

O problema básico com a maioria dos algoritmos para a análise estática é que geralmente, no pior caso, a complexidade do algoritmo é freqüentemente exponencial.

É possível ainda combinar as técnicas de análise estática com a depuração dinâmica. Uma forma de se combinar estas técnicas é partir da análise estática para projetar os casos de teste para uso em conjunto com um depurador dinâmico. Dessa forma, as partes do programa que pela análise estática se mostrarem corretas, não precisarão passar pela monitoração dinâmica, o que reduz o *overhead* associado com a depuração. O contrário também é possível, ou seja, a informação obtida a partir da monitoração dinâmica poderia ser usada para dirigir uma análise estática parcial, pois a análise estática completa produz muitos estados. Essa união pode apresentar as melhores soluções e diversos pesquisadores têm apresentado técnicas que combinam o uso da análise estática e da análise dinâmica [McDowell e Helmbold, 1989].

3.4 – Apresentação dos Resultados

Nos programas seqüenciais a apresentação dos resultados durante ou após, a depuração é uma tarefa razoavelmente simples. As informações podem ser apresentadas como um único texto seqüencial, já que apresenta a ordem em que foram executadas. Outro detalhe importante é o fato, dos dados estarem localizados em um mesmo local, podendo ser acessados facilmente quando necessário.

Os programas paralelos possuem uma abordagem bastante diferente. Eles possuem diversos processos executando concorrentemente e os dados se encontram logicamente e, as vezes, fisicamente separados.

Um importante objetivo no estudo da depuração de programas paralelos está em se encontrar a forma de apresentar os dados distribuídos sem causar *probe effect*, e de uma forma que possibilite aos usuários a sua compreensão.

Existem algumas técnicas de apresentação das informações resultantes da depuração paralela [McDowell e Helmbold, 1989], entre elas:

(1) Apresentação dos dados de maneira textual, podendo envolver cor com o objetivo

de apresentar o fluxo de controle da informação. É a técnica mais comum e presente nos sistemas atuais. É a forma utilizada pelos depuradores paralelos tradicionais. Em sistemas baseados em eventos, a apresentação seqüencial de um processo específico pode ser útil. Em sistemas baseados em objetos, uma lista seqüencial dos objetos é criada na ordem em que os mesmos são acessados.

(2) Diagramas de Processo x Tempo que apresentam a execução do programa paralelo em saídas bidimensionais. Esses tipos de diagramas são importantes para a visualização das atividades dos sistemas paralelos. Para a maioria dos sistemas, um relógio global é necessário. Sua limitação está no número de processos que podem exceder o tamanho da tela, mas isto pode ser resolvido através da filtragem ou ocultamento de processos.

(3) Uma alternativa para o diagrama anterior é criar uma animação da execução do programa, em que ambas as dimensões são espaciais, mostrando imagens específicas do programa em intervalos de tempo. Essa apresentação consiste em colocar cada processo em um ponto diferente do vídeo, de forma que todo o vídeo apresente o sistema em um único instante de tempo. As alterações de eventos são sucessivamente mostradas no vídeo, gerando uma espécie de animação. A posição dos processos pode ser arbitrária, sob controle do usuário ou em função da estrutura do programa. Alguns sistemas permitem controlar o tempo de apresentação de cada imagem e apresentar o conteúdo de mensagens trocadas entre os processos.

(4) Uma pequena variação seria a utilização de várias janelas, permitindo várias imagens simultâneas do programa sendo depurado. Esse tipo de apresentação envolve a criação de uma janela para cada processo.

Nenhuma dessas técnicas, utilizada de forma isolada, é suficiente para detectar todos os erros que podem surgir em programas paralelos. Por exemplo, a animação permite uma boa visão dos estados do sistema em um determinado instante. Para um único instante de tempo, mais informações podem ser apresentadas simultaneamente. Também permite a observação de padrões de comportamento. Mas, não mostra claramente os padrões de comportamento que ocorrem em função do tempo. Em contrapartida, o diagrama processo X tempo apresenta padrões de comportamento em função do tempo, e torna-se útil para descobrir problemas relacionados a desempenho. A principal desvantagem dessa abordagem é que poucas informações podem ser mostradas para cada processo em um determinado instante de tempo.

3.5 – Considerações Sobre os Sistemas Distribuídos

Até esse momento foi abordado o problema da depuração de programas paralelos considerando sistemas distribuídos e arquiteturas paralelas quase que sem distinção, porém existem algumas considerações que devem ser levadas em conta quando um programa paralelo está executando em um ambiente distribuído.

Nos sistemas distribuídos encontra-se o conceito de estado global. A sincronização global do *clock* é um problema de pesquisa clássica em sistemas distribuídos. Sem uma sincronização global do *clock*, pode ser difícil determinar com precisão a ordem dos eventos que estão ocorrendo em processadores diferentes. Se cada computador executa em um *clock* diferente, como saber exatamente qual evento ocorreu primeiro? Em um sistema centralizado já existe o conceito de consistência interna, que representa a ordem cronológica em que dois eventos ocorrem, mesmo que o *clock* da máquina não represente o tempo real, todos os processos dessa máquina possuem o mesmo *clock* como referência [Tanenbaum, 1992]. Em contrapartida, os processos que executam em sistemas distribuídos possuem características próprias, pois as informações que são relevantes estão espalhadas em diversas máquinas e o processo necessita tomar decisões observando as informações locais. A necessidade de manter essas informações trafegando pela rede de comunicação é a maior causa dos problemas de *deadlock* nos sistemas paralelos que executam sob os sistemas distribuídos. Dois processos não precisam concordar com o tempo atual, mas devem saber exatamente a ordem em que os eventos ocorrem.

Assim, alguns dos problemas que surgem nos programas paralelos estão relacionados com a necessidade de compartilhamento de recursos e informações. Um processo, mesmo que esteja executando em paralelo com outros processos, se não necessitar de recursos comuns ou mesmo de trocar informações, não apresentará os erros que estão sendo abordados nesse trabalho. Os erros que eventualmente ocorram diz respeito, única e exclusivamente, aos problemas de ordem seqüencial, como, por exemplo, a falta de inicialização de variáveis ou falhas no fluxo do processo, situações que são resolvidas com a utilização de depuradores seqüenciais.

Em sistemas com memória centralizada o estudo de possíveis situações de conflito se baseia na análise de técnicas de sincronização como semáforos e monitores. Nos sistemas distribuídos, onde os processos que executam em máquinas diferentes não têm acesso às memórias dos demais processos, as comunicações e os sincronismos são realizados através de primitivas de envio e recebimento de mensagens, isto torna a busca de erros uma tarefa muito mais trabalhosa.

O objetivo de se analisar os problemas que surgem em um sistema distribuído, vem do fato de que o foco principal de um depurador paralelo é a interação entre os processos e não sua lógica interna. Isto porque os problemas internos podem ser removidos através de depuradores seqüenciais [Cheung et al., 1990].

No capítulo sobre *deadlock* serão apresentadas algumas soluções para alguns desses problemas encontrados nos sistemas distribuídos.

Capítulo 4
Deadlock

Quando os sistemas eram não preemptivos um processo começava a sua execução e só era retirado da memória quando terminava ou quando o seu tempo de execução excedia muito a expectativa de término do sistema operacional. Dessa forma todos os recursos disponíveis na arquitetura estavam inteiramente disponíveis para o processo que estava executando. Isto não causava nenhum conflito e o gerenciamento desses recursos era mais simples.

A possibilidade de várias tarefas estarem ativas em um determinado momento só foi possível com o advento dos sistemas de tempo compartilhado, onde foi introduzido o conceito de tempo compartilhado para simular o paralelismo entre as tarefas. Nesse tipo de sistema os processos não executam indefinidamente até terminarem. O sistema implementa a preempção e interrompe um processo que está executando após uma pequena fatia de tempo, salva sua parte volátil e transfere o fluxo para outro processo, após recuperar a parte volátil pertencente ao novo processo. E esse processo, por sua vez, também executa por uma pequena fatia de tempo. Esse rodízio segue indefinidamente, enquanto existe processo executando, criando um paralelismo virtual.

Com o compartilhamento do tempo da CPU e a possibilidade de cada processo mudar de estado a CPU passou a ser melhor utilizada, pois enquanto um processo aguarda o resultado da busca de dados em algum dispositivo de armazenamento, como o disco rígido, o sistema operacional pode escalonar um novo processo.

Como existem vários processos executando existe a possibilidade de vários deles necessitar dos recursos disponíveis no ambiente. E o que pode acontecer quando a quantidade de recursos não é suficiente para todos os processos? Se cada processo só utilizar um recurso de cada vez, basta criar um protocolo de acesso ao recurso, bloqueando o próximo processo

que necessitar do recurso antes desse estar desocupado. Mas essa situação nem sempre é verdadeira, pois muitos processos precisam ter acesso simultâneo e exclusivo a vários recursos. E essa necessidade de utilização de vários recursos pode provocar algumas anomalias, sendo a principal delas o *deadlock*, que é uma situação em que um processo aguarda por um recurso que nunca estará disponível.

Como exemplo, considere a situação em que um processo *P1* solicita e consegue permissão para usar um recurso *R1*, e um processo *P2* solicita, por sua vez o recurso *R2*, também conseguindo permissão para utilizá-lo. Se o processo *P2* requisitar o recurso *R1*, sem liberar o recurso *R2*, e considerando que o processo *P1* ainda não terminou de utilizá-lo, o processo *P2* será bloqueado. Se após esse bloqueio o processo *P1*, também necessitar do recurso *R2*, estará caracterizada a ocorrência de um *deadlock*. Nessa situação é fácil perceber a anomalia, pois estão envolvidos apenas dois processos. Mas o problema do *deadlock* pode envolver vários processos.

Deadlock pode ser formalmente definido como [Tanenbaum, 1992]:

"Um conjunto de processos, esperando por eventos que somente outro processo pertencente ao mesmo conjunto poderá fazer acontecer".

Aqui, recurso é algo que só pode ser usado por um único processo em um determinado instante de tempo. Pode ser um dispositivo de *hardware*, como, por exemplo, a impressora, ou uma informação, como um registro em uma base de dados.

4.1 – *Deadlock* e Depuração de Programas Paralelos

Deadlock é um problema clássico no estudo dos sistemas operacionais, sendo uma situação de difícil correção [Axford, 1986]. A dificuldade em torno dos *deadlocks* é que eles possuem a propriedade de serem estáveis, ou seja, uma vez que ocorram, persistirão até que seja detectado e quebrado. Os processos envolvidos não poderão dar continuidade em sua execução até que algum processo externo quebre o *deadlock*, liberando algum recurso [Craveiro, 1998]. Fica a cargo do projetista do sistema decidir por qual técnica escolher. Geralmente, a escolha recai entre algoritmos de prevenção ou algoritmos de detecção e correção, ou ainda em não fazer absolutamente nada (o famoso algoritmo do Avestruz: enfiar a cabeça na terra e fingir que nada está acontecendo). Essas abordagens apresentam vantagens e desvantagens e são adequadas a sistemas com características específicas.

A atenção que se dá ao problema do *deadlock* nesse trabalho vem do fato de que são considerados programas que possuem vários processos executando paralela ou concorrentemente, disputando por recursos comuns.

A maioria dos artigos e estudos feitos sobre *deadlock* trata do assunto em ambiente de sistema e não analisam a possibilidade de ocorrer *deadlock* entre os processos que compõe um programa paralelo. O *deadlock* que ocorre em nível de sistema será definido como *deadlock* externo. Esse tipo de *deadlock* é difícil de resolver, porque o controle pertence ao sistema operacional, que se não possuir recursos para resolvê-lo, nada poderá ser feito. A aplicação do usuário não pode auxiliar na solução do *deadlock* externo, pois não tem conhecimento de quais processos estão executando e que recursos eles estão utilizando. Essa tarefa é de responsabilidade do sistema operacional.

O *deadlock* que ocorre em nível de programa do usuário, provocado pelos próprios processos que compõe a aplicação será definido como *deadlock* interno. O estudo desse tipo de ocorrência possibilita corrigir erros de sincronização e de requisição de recursos, que podem ser controlados pelo próprio fluxo do programa, não havendo necessidade do envolvimento do sistema operacional.

4.2 – Condições Para a Ocorrência de *Deadlocks*

É claro que a ocorrência de *deadlock* é uma situação indesejada, pois quando o *deadlock* ocorre os processos não terminam sua execução e os recursos do sistema ficam presos, sem serem utilizados.

Como identificar, então, se um programa com diversas tarefas, ou mesmo um conjunto de processos independentes competindo pela utilização dos recursos do sistema, poderão causar *deadlock*? Quatro condições são necessárias para se caracterizar uma situação de *deadlock* [Tanenbaum, 1992], [Shay, 1996] e [Craveiro, 1998]:

(1) **Condição da Exclusão Mútua**: cada recurso só poderá estar sendo utilizado por um processo de cada vez, ou estar disponível.

(2) **Condição de Posse e de Espera**: um processo detendo um recurso aguarda por outro recurso.

(3) **Condição de Não Preempção**: um recurso não pode ser retirado de um processo, sem que esse libere o mesmo voluntariamente.

(4) **Condição de Espera Circular**: deve existir um conjunto de processos bloqueados, onde cada um dos processos pertencentes a esse conjunto, espera por um recurso alocado para um processo do próprio conjunto. Essa situação foi ilustrada na apresentação do capítulo.

Diante dessas situações, a solução encontrada por diversos projetistas é a tentativa de prevenir pelo menos a ocorrência de uma dessas condições, já que para a ocorrência de

deadlock é preciso que todas as quatro condições estejam presentes [Tanenbaum, 1992].

A prevenção dessas condições acarreta um alto custo para o sistema e evitar algumas dessas situações pode acarretar um problema muito maior que o próprio *deadlock*. Por exemplo, a eliminação da exclusão mútua, certamente retira toda a possibilidade de ocorrer *deadlock*, porém, no capítulo anterior foram apresentadas as anomalias que podem ocorrer quando um recurso não preemptivo está sendo utilizado concorrentemente por vários processos [Brawer, 1989].

Outra situação complexa é a tentativa de se evitar a condição de posse e espera. Uma primeira forma de se evitar esta condição é retirar os recursos já alocados a um processo quando este solicitar novos recursos, o que não faz sentido algum, já que um recurso não preemptivo não pode ser acessado concorrentemente. Uma segunda forma de se evitar a posse e espera é fazer com que o processo solicite todos seus recursos de uma vez só, antes de ser executado. Esta opção é totalmente inadequada, pois se um processo utilizar um arquivo em disco durante uma hora e depois imprimi-lo, a impressora será alocada a este processo juntamente com o disco, ficando uma hora sem utilização. Outro problema relacionado a solicitação de todos os recursos de uma só vez é o caso de um processo necessitar de muitos recursos, esses recursos podem não estar disponíveis simultaneamente sendo que o processo ficará esperando por uma ação que nunca acontecerá.

A remoção da não preempção pode ser conseguida testando o estado dos recursos solicitados. Se eles estão livres, são alocados. Se eles estão ocupados, então todos os recursos que o processo já havia alocado são liberados, e retornam para a lista dos recursos disponíveis. O processo voltará a executar quando conseguir alocar seus velhos recursos juntamente com o novo recurso sendo solicitado. É claro que esse algoritmo só pode ser aplicado em recursos cujo estado pode ser facilmente salvo e reposto mais tarde.

A espera circular é a última condição e também não é fácil de ser evitada. Isto será possível se os processos não puderem pedir recursos em tempos diferentes, mas como discutido nos parágrafos anteriores esse método é inviável. Outra forma de evitar é fazer com que cada processo use um recurso de cada vez. Também já foi comentada a impossibilidade dessa abordagem.

Pode-se observar que a prevenção de *deadlock* é difícil, devido aos novos problemas que surgem quando utilizam-se os métodos para amenizar o problema.

4.3 – Detecção de *Deadlocks*

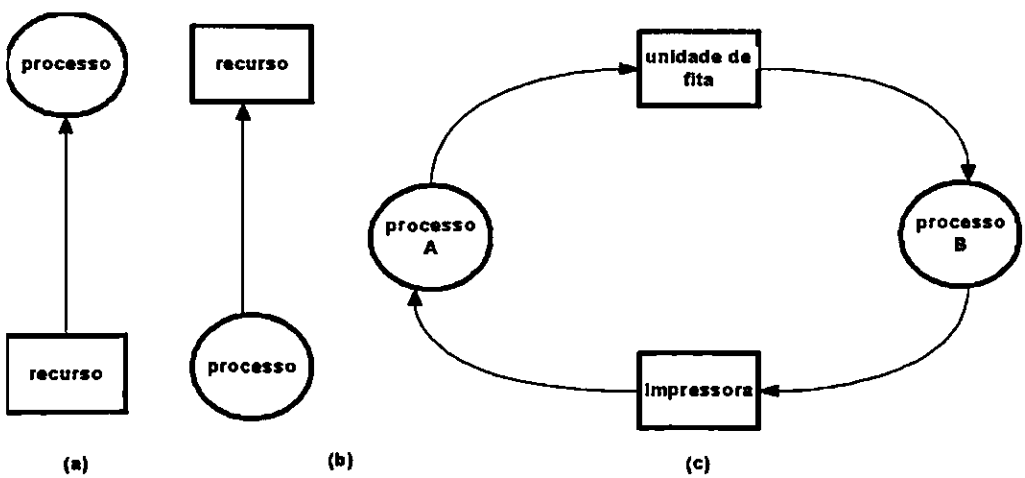
A prevenção de *deadlocks* é uma abordagem utilizada em ambiente de sistema

operacional visando os *deadlocks* externos. As técnicas de detecção se aplicam ao problema do *deadlock* interno. Dessa forma, as técnicas relacionadas a detecção de *deadlock* é de grande importância para este trabalho.

A abordagem de detecção mais simples ocorre quando no sistema só existe um recurso de cada tipo, como uma impressora, uma unidade de fita, uma *plotter*, etc. Para esse tipo de análise a criação de um grafo dirigido (dígrafo) de recursos, como ilustrado na figura 4.2, é a maneira mais fácil de se detectar a ocorrência de *deadlock*.

Em um dígrafo como os das figuras 4.1, os nós representam os processos (círculos) e os recursos (quadrados) envolvidos e os arcos representam a dependência de cada um, como pode ser visto na figura 4.1. Na figura 4.1(a) encontra-se a representação de um arco partindo de um nó quadrado (recurso) em direção a um nó em formato de círculo (processo) representando um processo de posse de um recurso. A figura 4.1(b) apresenta um processo aguardando por um recurso não disponível. Observando a figura 4.1(c) verifica-se a ocorrência de um ciclo no dígrafo, demonstrando a ocorrência de *deadlock*.

Figura 4.1 – Grafos de alocação de recursos [Tanenbaum, 1992].



(a) Um processo de posse de um recurso

(b) Um processo aguardando por um recurso

(c) *Deadlock*

A figura 4.2 apresenta um exemplo [Tanenbaum, 1992] com sete processos identificados de A até G e seis recursos de R a W. Analisando-se o exemplo, observa-se que:

1. O processo A está de posse do recurso R e precisa de S.
2. O processo B não está de posse de nenhum processo, mas precisa de T.
3. O processo C não está de posse de nenhum recurso, mas precisa de S.

4. O processo D está de posse de U, e precisa de S e de T.
5. O processo E está de posse de T, e precisa de V.
6. O processo F está de posse de W, e precisa de S.
7. O processo G está de posse de V, e precisa de U.

Toda vez que em um dígrafo de recursos for encontrado um ciclo, os processos que fizerem parte desse ciclo encontram-se em *deadlock*. Dessa forma, pode-se observar na figura 4.2, que os processos D, E e G estão em *deadlock*.

Existem vários algoritmos para detectar ciclos em grafos dirigidos. O grande problema é que a maioria desses algoritmos possui complexidade exponencial.

Essa técnica só é adequada quando existe apenas um recurso de cada tipo. Quando se trabalha com mais de um recurso de um mesmo tipo é necessária a utilização de estruturas de dados capazes de identificar cada recurso do sistema. Essas estruturas devem conter informações sobre quais recursos estão disponíveis e quais estão alocados, além das informações sobre os recursos a alocar e recursos alocados para cada processo.

Nessa abordagem para detecção de processos em *deadlock*, os processos começam desmarcados. Cada processo será marcado se ele puder terminar a sua execução sem provocar *deadlock*. Quando o programa de análise terminar, os processos que ainda estiverem desmarcados estarão em *deadlock*.

Inicialmente o algoritmo procura por um processo desmarcado *P*, e verifica se a quantidade de recursos que ele necessita é menor do que a quantidade de recursos disponíveis. Isto indica que esse processo terá condições de terminar a sua execução. Assim, o processo é colocado para executar até terminar. Se todos os processos estiverem em condições de executar, nenhum deles entrará em *deadlock*. Se existirem processos que nunca irão possuir condições para entrar em execução, estarão em *deadlock*.

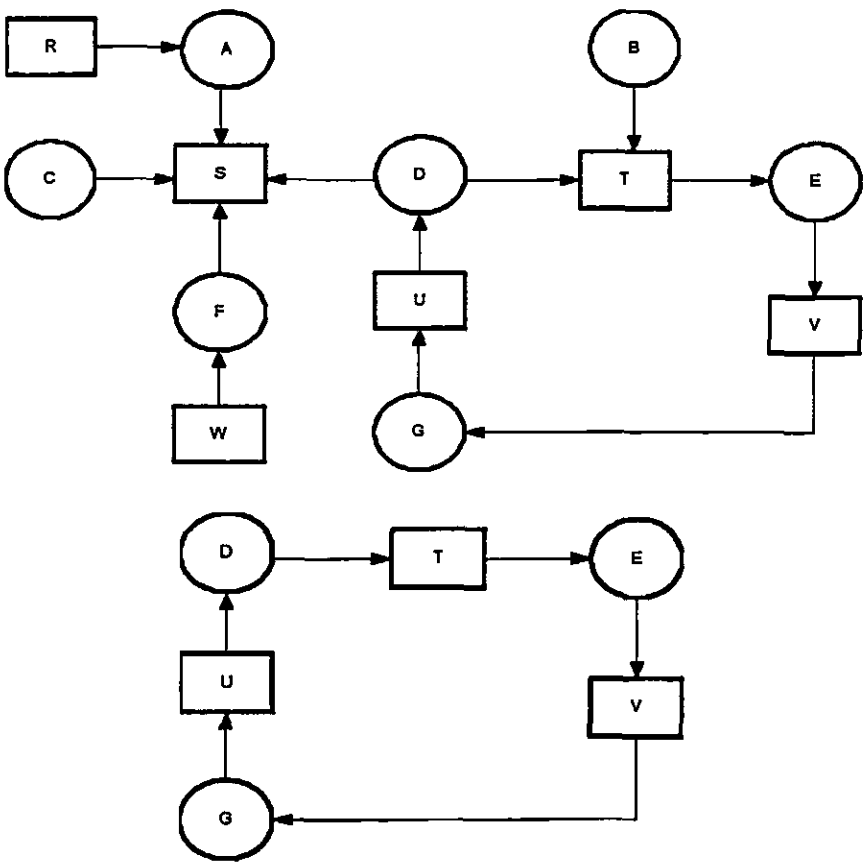
Uma característica interessante dessa técnica é que apesar do algoritmo ser não determinístico, o resultado é sempre o mesmo. O problema é que o sistema deve saber desde a criação do processo, os recursos que ele vai utilizar durante a sua execução. Em termos de depuração, isto não é um problema muito difícil, pois em uma análise estática do código é possível conseguir essas informações.

Os algoritmos que implementam a detecção do *deadlock* verificam a ocorrência da espera circular, percorrendo toda a estrutura, toda vez que um processo precisa de um recurso e esse pode ser imediatamente garantido.

O problema do *deadlock* tende a se tornar cada vez mais freqüente considerando a evolução que os sistemas operacionais tem alcançado no sentido de implementar o

paralelismo de forma intensiva, permitindo uma maior alocação de recursos.

Figura 4.2 – Um dígrafo de recursos com a ocorrência de um ciclo [Tanenbaum, 1992].



Ciclo extraído do dígrafo anterior.

A detecção do *deadlock* se baseia na tentativa de se buscar estados seguros durante a execução de um processo. Existe um estado seguro em relação a um processo, quando esse processo puder executar e não entrar em *deadlock* alocando todos os recursos de que necessita.

Quando um sistema operacional adota a alternativa de detecção são, automaticamente, utilizadas técnicas de recuperação do *deadlock*. Existem várias formas de se realizar essa recuperação. Uma maneira seria através da preempção do recurso, quando isto é possível, transferindo o controle para um outro processo e devolvendo em seguida. Outra forma é retornar para um estado anterior a ocorrência de *deadlock*, pois quando ocorre o *deadlock*, o sistema percebe quais são os recursos que vão causar, ou que causaram a situação. Quando não é possível utilizar nenhuma dessas alternativas, a solução mais simples é "matar" um dos

processos envolvidos liberando os seus recursos. Mesmo assim, não são todos os tipos dê processos que podem ser eliminados sem provocar um problema ainda maior.

Nesse trabalho, não serão abordados em detalhes, as formas de se recuperar os *deadlocks* após a sua detecção, pois o que interessa saber é se um programa paralelo pode entrar em *deadlock* interno, ou seja, causado pela má administração da alocação de recursos entre as suas próprias tarefas.

4.4 – *Deadlocks* em Sistemas Distribuídos

O problema do *deadlock* nos sistemas distribuídos possui a mesma conceituação em relação aos sistemas centralizados. O *deadlock* ocorre principalmente durante a troca de mensagens entre os processos que estão executando nas diversas máquinas. Entretanto, se as mensagens forem tratadas como recursos, tem-se uma situação bastante parecida em relação aos outros recursos do sistema. Em sistemas distribuídos, a prevenção de *deadlock* é ainda mais difícil, devido aos componentes estarem fisicamente separados.

Para se detectar a ocorrência de *deadlock* é necessário eleger um processo coordenador. Isto porque em cada máquina é possível a criação de um grafo de recurso próprio, mas é extremamente complicado manter um grafo que envolva todos os recursos de todas os componentes computacionais. O coordenador manteria um estado global do sistema, mas se torna um ponto único de falhas.

Dentro de um programa paralelo, o controle é mais simples de ser feito, pois a responsabilidade pode ficar a cargo da própria aplicação. Essa é a vantagem de um depurador que consiga perceber a ocorrência de *deadlock* durante a fase de teste.

Capítulo 5
Interface Homem-Máquina e Sistemas Tutores

Quando um projeto computacional adquire funções educacionais, como é o caso desse trabalho, inevitavelmente os temas ***interface*** **homem-máquina** e **sistemas tutores** são abordados. O sucesso de uma ferramenta que pretende ensinar algum conteúdo para um usuário, está diretamente relacionado com a facilidade que esse usuário encontra na interação com a ferramenta. Além da *interface*, a ferramenta precisa ter um sistema tutor que possa conduzir o usuário de uma forma suave, sem provocar saltos na apresentação dos conteúdos. Esses saltos, em várias situações, provocam conclusões erradas por parte dos usuários.

Um dos objetivos desse projeto é permitir que os usuários iniciantes da programação concorrente possam adquirir novos conhecimentos com o exame das possíveis falhas em seus programas paralelos. Na busca desse objetivo, devem ser focalizados os temas: *interface* homem-máquina e sistemas tutores para que o esforço não seja em vão.

5.1 – *Interface* Homem-Máquina

As *interfaces* homem-máquina vêm ganhando destaque nos últimos anos. Os primeiros programas para computador foram desenvolvidos por engenheiros, sendo por eles utilizados. Não existia a preocupação de se criar um ambiente amigável. O essencial era o programa apresentar as respostas esperadas, mesmo que essas respostas só fossem compreensíveis aos programadores e as entradas só pudessem ser realizadas por quem entendia o funcionamento do programa em detalhes, que geralmente eram os próprios programadores.

Mesmo hoje é possível verificar a diferença entre dois programas desenvolvidos pelo mesmo programador. Quando um dos programas está sendo desenvolvido para o uso do

próprio programador, a preocupação com a *interface* fica para segundo plano, pois o programador sabe os passos que precisa efetuar para se realizar alguma tarefa em seu programa e compreende os resultados apresentados. Quando o programa está sendo criado para uma terceira pessoa, a preocupação com a interação do usuário com o programa deve ser maior. O desenvolvedor não deve esquecer que para o usuário o programa que ele utiliza é uma caixa preta.

A *interface* homem-máquina é o comportamento do usuário e do computador que pode ser observado externamente, existindo uma linguagem de entrada, outra de saída e um protocolo de interação [Chi, 1985]. Em outras palavras, a *interface* homem-máquina é a parte de um *software* responsável pela interação entre o usuário e o computador, traduzindo as ações do usuário em ativações das funcionalidades do sistema, e pela apresentação em forma adequada dos resultados.

5.1.1 – Aspectos da Percepção Humana

O ser humano interage com o meio em que vive por meio de um sistema sensorial bem desenvolvido [Pressman, 1992]. Em se tratando de *interface* homem-máquina, os sentidos predominantes são o visual, o tátil e o auditivo. Desses o mais importante é o sentido visual, já que a maioria das informações que são apresentadas aos usuários é feita através de imagens (monitor) ou relatórios.

Uma pessoa interpreta uma informação através de uma imagem, muito mais rapidamente que a mesma informação em um texto. Ainda assim, alguns dos sistemas computacionais interagem com o usuário na forma textual, mesmo aqueles baseados em ambientes gráficos.

Um bom tratamento visual de um *software* com objetivos educacionais permite uma maior assimilação por parte do educando, facilitando o processo ensino-aprendizagem.

Um detalhe importante no estudo das *interfaces*, é que cada ser humano possui personalidade única. Nem tudo que agrada a uma pessoa agrada a outra. Apesar disto existem padrões que devem ser observados, mas a *interface* ideal será aquela que possuir flexibilidade para permitir ao próprio usuário algumas características que adaptem ao seu gosto.

5.1.2 – *Interface User-friendly*

Um dos objetivos do projeto de *interface* homem-máquina é criar uma *interface* que seja amigável. Habermann apresenta uma semântica para definir uma *interface* "amigável" [Habermann, 1991]:

(1) A *interface* deve ser transparente, ou seja, o usuário concentra-se nas tarefas que

pretende realizar;

(2) Deve ser previsível e ao mesmo tempo flexível. Nem sempre o usuário possui condições de recorrer ao manual para entender o que realiza uma opção;

(3) O usuário deve gostar dela. Muitos usuários finais preferem sistemas que são inferiores, em termos de eficiência, mas que são fáceis de usar.

O projeto de *interfaces* é um importante ramo dentro da ciência da computação, pois a interface atua diretamente na produtividade do usuário. A popularização da informática deve-se muito a evolução das técnicas de projeto de *interface*.

Além do preço do *hardware* e o custo de desenvolvimento do *software*, um sistema computacional ainda necessita treinar os usuários. Quanto mais difícil o aprendizado mais caro fica o sistema, além de aumentar no usuário a rejeição ao sistema.

5.1.3 – *Interface* e Aprendizagem

A *interface* alcança grande destaque quando se trata de informática e educação, ou aprendizagem através do computador. Uma *interface* bem projetada pode facilitar o aprendizado do usuário enquanto que uma *interface* em que o usuário sinta dificuldade de interação pode criar situações difíceis de serem resolvidos.

Quando a *interface* possui o objetivo de servir de apoio ao aprendizado, é interessante a análise das características que possam tornar a comunicação a mais simples possível. A *interface* deve fazer uso de recursos multimídia, linguagem natural etc, explorando bastante o sentido auditivo, além do visual que já é bastante utilizado.

5.1.4 – Construção de Uma *Interface*

O desenvolvimento de uma *interface* é um processo iterativo. A cada iteração, as versões do projeto são refinadas e testadas com os usuários que interagem com o desenvolvimento, melhorando a definição da *interface*. A *interface* só deve ser implementada após várias iterações com o usuário. Mesmo assim, poderá ser necessário realizar novas alterações quando a ferramenta já estiver pronta, pois a implementação pode revelar dificuldades que não foram observadas durante o projeto.

O projeto de *interface* define o comportamento e a apresentação de uma *interface* envolve fatores da psicologia humana, tecnologia dos dispositivos de entrada e saída, tipos de diálogos, análise de tarefas, princípios, padrões, regras, protótipos e avaliação de sistemas existentes.

5.1.5 – Modelos de Projeto de *Interface*

Basicamente, existem quatro modelos de projeto de *interfaces* e cada modelo representa a *interface* sob um ponto de vista diferente [Pressman, 1992]. O primeiro é o **modelo de projeto**, que é a especificação criada pelo desenvolvedor contendo a funcionalidade de todo o sistema. O segundo modelo é o **modelo de usuário**, que descreve o perfil dos usuários, como nível de conhecimento, educação, capacidades físicas, idade, sexo, etc. O objetivo desse modelo é focalizar a que usuário a ferramenta se destina. O terceiro modelo é a **percepção do sistema**, que representa a imagem que o usuário tem do sistema. Por fim, o último modelo é a **imagem do sistema**, que é a interface final do sistema. Se a imagem do sistema e a percepção do sistema forem iguais, o usuário terá satisfação ao utilizar o sistema.

5.2 – Sistemas Tutores e Sistemas Tutores Inteligentes

Os sistemas tutores inteligentes (S.T.I.) surgiram da interseção da Ciência da Computação, da Psicologia Cognitiva e da Pedagogia. A Ciência da Computação fornece as técnicas para representar e manipular o conhecimento. A Psicologia Cognitiva apresenta as formas de construção desse conhecimento pelo ser humano e a Pedagogia estuda como melhorar as formas de ensino-aprendizagem [Freire, 1998].

Os sistemas tutores inteligentes são uma evolução natural dos sistemas tutores. Os sistemas tutores se limitam a apresentar tópicos que são selecionados pelos usuários com intuito de obter as informações referentes. O grande problema que surge com esse tipo de sistema é a falta de flexibilidade que é extremamente importante em um processo educacional.

Um sistema inteligente para o ensino deve ser capaz de perceber certas deficiências do usuário, remontando-o a tópicos básicos para maior fixação do conteúdo. O sistema não pode permitir que o aluno interaja com tópicos que ainda não possui condições de assimilar, para que não surjam bloqueios dificultando a aprendizagem. Deve, ainda, proporcionar um diagnóstico detalhado dos erros do usuário, ao invés de mostrar simplesmente como realizar a tarefa.

Além disso, o usuário deve se sentir diante de um "professor" podendo propor questionamentos novos e responder às questões que o sistema formula. Quanto menos o usuário percebe que está diante de uma máquina, maior é o alcance do sistema.

5.2.1 – A Arquitetura de um Sistema Tutor Inteligente

Geralmente, um sistema tutor inteligente é composto dos seguintes módulos [Freire, 1998]:

(1) **Módulo Especialista**: que possui os predicados e as regras de um domínio particular. É a origem do conhecimento que deverá ser absorvida pelo usuário.

(2) **Modelo do Estudante**: armazena o desempenho do estudante, sendo a estrutura básica para as tomadas de decisão do sistema.

(3) **Módulo Tutorial**: regula as interações de aprendizagem do sistema com o estudante. Esse módulo define as atividades pedagógicas que serão propostas ao aluno pelo sistema.

(4) **Módulo de *Interface***: é o módulo da *interface* homem-máquina, ou seja, a *interface* do sistema tutor com o usuário.

Um sistema baseado em conhecimento faz uso intenso de conhecimento especializado. O objetivo desses sistemas é tentar resolver problemas da mesma forma que um especialista humano resolveria.

Um sistema tutor inteligente também é classificado como um sistema baseado em conhecimento, visto que o sistema deve armazenar o conhecimento sobre as características do usuário, bem como o conhecimento especialista sobre determinado domínio. Um sistema baseado em conhecimento constitui-se de:

(1) **Núcleo de Sistema Baseado em Conhecimento**: que realiza a interação com o usuário ou equipamentos externos, produz um conhecimento baseando-se em alguma linha de raciocínio e apresenta as conclusões a partir desse raciocínio;

(2) **Base de Conhecimento**: é onde fica armazenado o conhecimento especialista;

(3) **Base de Dados**: é onde se armazenam as conclusões intermediárias e as provas realizadas durante a execução do programa, permitindo fornecer ao usuário as provas correspondentes às conclusões obtidas.

5.2.2 – Aquisição de Conhecimento

A aquisição de conhecimento é um tópico importante em um sistema inteligente. O conhecimento adquirido agiliza as tomadas de decisões futuras. Em se tratando de *software* tutores, a aquisição de conhecimento facilita as interações entre o usuário e o computador, permitindo um aprendizado mais fácil por parte do usuário.

5.2.3 – O Ensino da Programação de Computadores Através de Um S.T.I.

Do ponto de vista da cognição, a programação de computadores é uma tarefa difícil

para iniciantes. Isto ocorre por duas razões: a falta de conhecimento dos princípios que envolvem a programação e a falta de perícia do aluno [Pimentel e Direne, 1998]. Como qualquer área, as técnicas de programação de computadores também podem ser ensinadas utilizando-se um Sistema Tutor Inteligente. É possível, por exemplo, desenvolver um sistema que auxilie um professor de Linguagens e Técnicas de Programação, através de exercícios que possam ser modificados pelo próprio sistema, aumentando as alternativas do estudante. O sistema auxiliaria o usuário na correção dos erros que ele ainda estiver cometendo. Um sistema desses permite minimizar o tempo de aprendizado possibilitando a utilização de diversas estratégicas pedagógicas.

A aprendizagem da programação paralela tende a se tornar mais suave com o auxílio de um sistema tutor. O objetivo das ferramentas não é eliminar a figura do professor, mas ampliar o potencial pedagógico durante o processo de ensino-aprendizagem. A utilização da informática no processo pedagógico ainda não logrou grande sucesso. Isto se deve ao fato de se usar o computador da mesma forma que se usa um livro ou o quadro negro. É preciso utilizar o computador explorando o seu potencial característico, sem desmerecer os outros mecanismos pedagógicos, mas utilizando a computação com as técnicas que ela possui de especial.

A grande dificuldade existente na criação de uma ferramenta assim é como selecionar os exemplos em que o aluno está apto a resolver, ou seja, como definir o grau de dificuldade de um programa. A engenharia de *software* usa medidas como linhas de código (LOC) ou pontos por função (FP) [Pressman, 1992], mas essas medidas não são precisas porque nem sempre um programa extenso é mais difícil de se desenvolver que um pequeno.

Em adição, é possível unir um sistema tutor inteligente para o aprendizado das técnicas de programação com um sistema de depuração proporcionando um sistema de aprendizado bastante eficaz.

Capítulo 6
O Projeto de Um Ambiente Para Depuração de Programas Paralelos

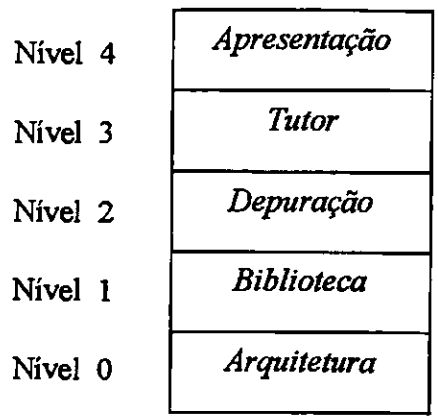
Devido à diversidade de linguagens para a criação de programas paralelos e à existência de várias bibliotecas para a criação de ambientes de portabilidade, são poucas as ferramentas para a depuração de programas paralelos. O foco principal das pesquisas que estão em desenvolvimento nessa área, procura enfatizar os problemas que estão envolvidos com a depuração de programas paralelos, apresentando conceitos que definem as regras para se depurar um programa, mas não envolvem o desenvolvimento de uma ferramenta que integre a parte de teste e depuração do programa em um ambiente de aprendizagem.

As ferramentas de depuração existentes não possuem uma boa *interface* e não têm a preocupação de conduzir o usuário a uma melhor utilização da linguagem. Normalmente, as ferramentas disponíveis são específicas para um sistema e visam apenas detectar erros.

O objetivo deste trabalho é projetar uma ferramenta que auxilie um usuário de programação paralela, sem muita experiência, a localizar erros em seus programas. A ferramenta deverá conter um ambiente tutor, para que durante a localização dos possíveis problemas que estão provocando os erros, conduza o usuário a uma reformulação de suas técnicas de programação. Esse procedimento proporciona a aquisição de novos conhecimentos sobre a prática da programação paralela, ou ainda consolida conceitos anteriormente adquiridos. A utilização de ilustrações que mostrem alternativas para se resolver as situações que surgem quando se trabalha com um programa paralelo são consideradas.

A ferramenta está estruturada através de uma arquitetura em camadas como pode ser visto na figura 6.1. Cada camada se comunica com a seguinte através de protocolos, permitindo uma maior independência na definição e implementação do projeto.

Figura 6.1 – As Camadas que Definem o Projeto.



A camada de Arquitetura, localizada na base da estrutura e por isso no nível zero, define o ambiente em que o programa paralelo irá executar (não necessariamente o ambiente em que a ferramenta está executando). Com essas informações é possível traçar a granulosidade adequada para o programa. Essas informações são utilizadas pelas camadas de depuração e também pela camada tutor, para verificar se as técnicas de comunicação e sincronização usadas no desenvolvimento do programa estão adequadas à arquitetura utilizada.

A segunda camada, Biblioteca, é semelhante à primeira. Define as ferramentas utilizadas no programa que será depurado, isto é, as linguagens e as bibliotecas que foram usadas na elaboração do programa.

O depurador se localiza na terceira camada da arquitetura, ou seja, a camada de Depuração (nível 2). Essa camada se divide em módulos funcionais. Cada módulo tem o objetivo de resolver um tipo de problema específico. Por exemplo, um módulo para resolver problemas de *deadlock* e espera infinita e outro que resolva problemas de sincronização e comunicação. Além disso, deve ter um módulo para a busca de possíveis problemas de ordem seqüencial que podem estar afetando o programa como um todo. Por exemplo, se uma tarefa possui um *loop* codificado de forma incorreta gerando uma repetição "infinita" e logo em seguida a esse *loop*, essa tarefa precisa enviar uma mensagem para todos as outras que compõe a aplicação. A tarefa que está aguardando pela mensagem ficará esperando indefinidamente, devido a um problema na parte seqüencial do código.

O depurador possui características estáticas pela própria necessidade de interação do usuário com o sistema tutor.

Com o resultado que a camada anterior produziu, a camada Tutor (nível 3) proporciona ao usuário um mecanismo de ensino, facilitando o entendimento das falhas que

foram realizadas durante a implementação e criando um ambiente que melhora o aprendizado da programação paralela. Esse ambiente tutor pode, futuramente, se tornar um sistema tutor inteligente.

Finalmente, a camada de Apresentação (nível 4) possui como objetivo criar uma representação gráfica do programa que está sendo analisado, permitindo a apresentação de várias janelas com imagens de cada passo da execução do programa, permitindo montar uma animação do seu funcionamento, pois isto não é possível na camada de depuração visto que o depurador se baseia em técnicas estáticas. Além disso, é possível trabalhar essa camada para criar testes para os programas paralelos.

A seção 6.2 mostra o modelo proposto para a ferramenta e apresenta o relacionamento entre as camadas descritas nessa seção e os processos do modelo.

6.1 – O Modelo da Ferramenta

O projeto de um sistema ou de uma ferramenta, como a proposta nesse trabalho, pode ser visto como a construção de um edifício, que obedece a fases e procedimentos bem definidos. Em um primeiro instante, trabalha-se a criatividade, ou seja, parte-se de uma idéia inicial que é a base de tudo. Nesse caso, o ponto de partida foi a estrutura em camadas apresentada na seção anterior (figura 6.1). Entretanto, para ir além é necessário uma metodologia. Uma *metodologia* pode ser definida como um conjunto de métodos e técnicas com os quais uma tarefa pode ser cumprida, em outras palavras, uma metodologia estabelece um caminho único no desenvolvimento de projetos novos ou na evolução de um projeto já existente [Machado e Abreu, 1995].

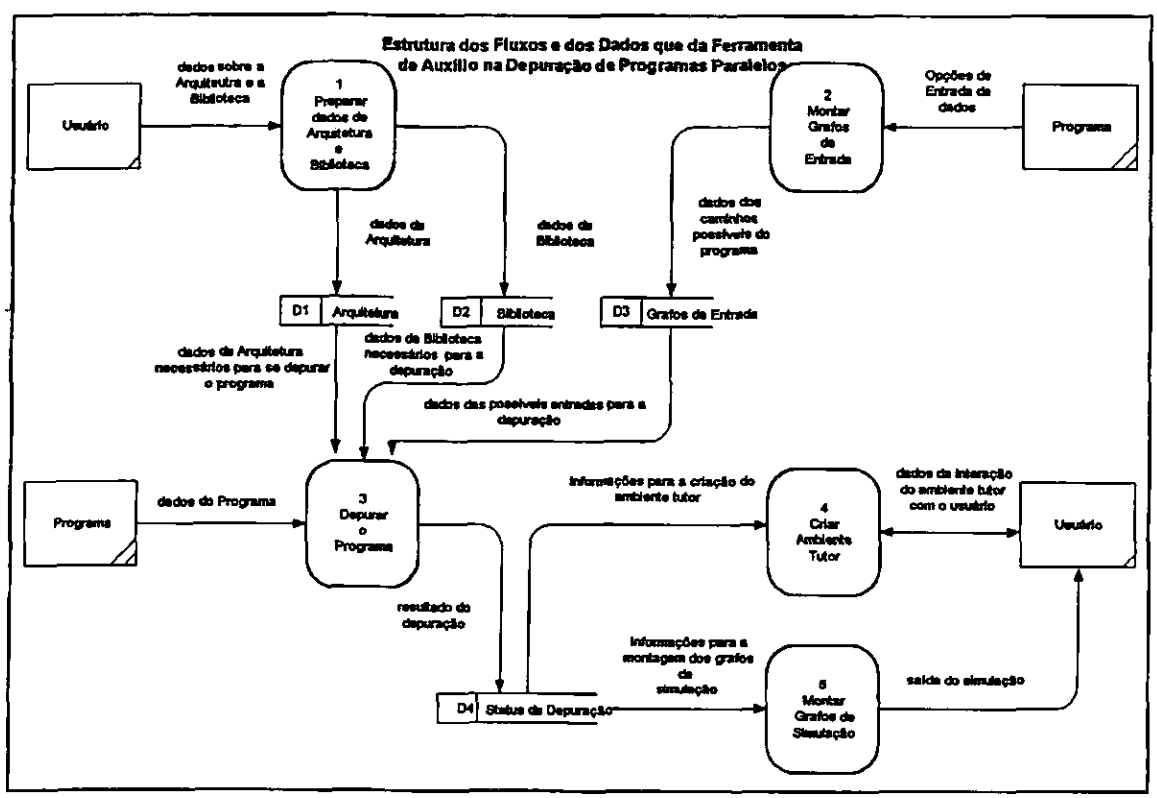
Uma metodologia de projeto consiste na construção de um modelo do domínio de um problema e da adição de detalhes de implementação ao modelo durante o projeto do sistema.

Para a criação de um modelo inicial que possibilite a visualização e definição das tarefas que compõem a ferramenta proposta nesse trabalho, além do inter-relacionamento entre elas, está sendo utilizada a ferramenta de projeto de sistemas denominada *DFD – Diagramas de Fluxos de Dados* [Davis, 1994] [Gane, 1988] [Gane e Sarson, 1988] definida na metodologia da análise estruturada de sistemas.

O diagrama da figura 6.2 representa os processos e as informações que trafegam nas 5 camadas que compõem a ferramenta. O *processo 1 – Preparar Dados de Arquitetura e Biblioteca*, está contido nas camadas 1 e 2 e possui a função de atualizar as informações da arquitetura (*hardware*) onde está executando a aplicação a ser depurada e as informações referentes às linguagens e bibliotecas utilizadas (*software*). *O processo 2 – Montar Grafos de*

Entrada faz parte da terceira camada (camada de depuração). Esse processo é responsável em criar duas classes de grafos: grafos de fluxo de instruções ou grafos de caminho e grafos de sincronização. Os grafos de caminho representam todos os caminhos possíveis de serem executados pelo programa e os grafos de sincronização representam as alternativas dos caminhos existentes nas comunicações entre os processos. A definição desses grafos será vista no item 6.2.2. Esses grafos são armazenados em um arquivo para a posterior utilização pelo *processo 3 – Depurar o Programa*, responsável pela depuração propriamente dita. O processo 3 possui subprocessos, cada um representando uma função, ou seja, resolvendo uma classe de erros em programas paralelos. Ao final de sua ação, esse processo armazena os resultados da depuração para serem utilizados pelos processos 4 e 5. O *processo 4 – Criar Ambiente Tutor* representa a quarta camada da arquitetura (camada Tutor). Com base nos erros encontrados, esse processo cria um ambiente de comunicação com o usuário, promovendo alternativas para solucionar o problema. O *processo 5 – Montar Grafos de Simulação* representa a quinta camada da arquitetura (camada Apresentação). Esse processo permite ao usuário verificar o efeito que os erros encontrados provocam no programa, através da simulação da execução do programa pelos grafos correspondentes. Esse processo permite a elaboração de testes para o programa, que apesar de não ser o foco inicial da ferramenta, é possível de ser realizado através da composição dos dados que passam pelo processo.

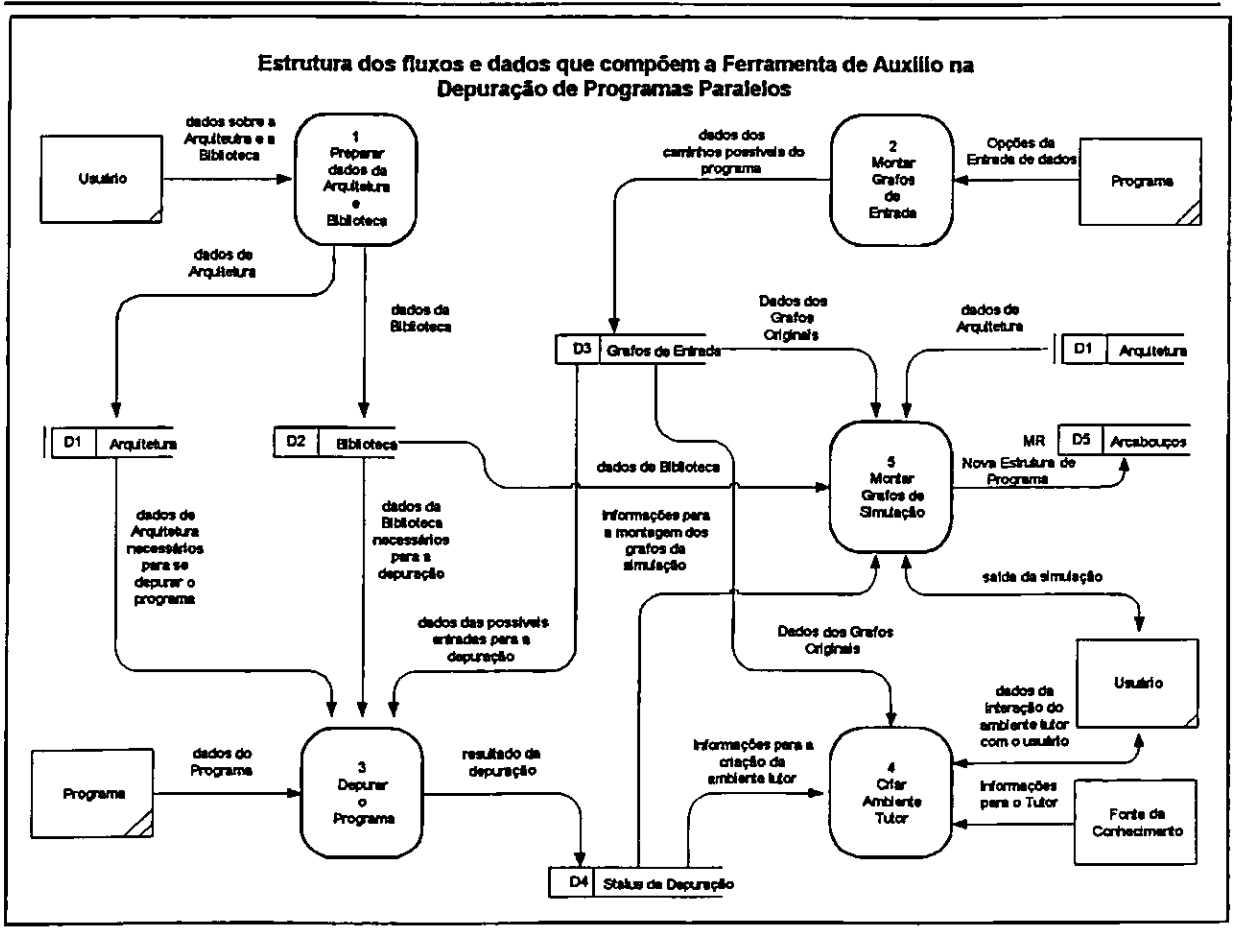
Figura 6.2 – Estrutura dos Fluxos e dos Dados da Ferramenta (1° Modelo)



Através da análise desse modelo é possível verificar a flexibilidade que ele oferece para explorar outras funcionalidades, como por exemplo, a paralelização automática do código que pode ser obtida pelo processo 5 (Montar Grafos de Simulação).

Pode parecer estranho o desejo de paralelizar um programa paralelo. Entretanto, a ferramenta é destinada aos usuários iniciantes. Possivelmente o código que está sendo depurado apresenta trechos seqüenciais que poderiam ser codificados paralelamente e que passou "despercebido" pelo usuário. O processo 5 pode avaliar as precedências nos grafos que representam o fluxo de instruções do programa e sugerir uma melhor divisão das tarefas. Para criar essa funcionalidade é necessário que o processo tenha acesso aos grafos que representam o programa, além das informações contidas nos depósitos: *Biblioteca (D2)* e *Arquitetura (D1)*. Assim, o modelo pode ser melhorado com a adição dos fluxos desses depósitos no processo 5.

Figura 6.3 – Estrutura dos Fluxos e dos Dados da Ferramenta (Modelo Final).



Outra mudança que pode ser feita no modelo original, é o acréscimo de um depósito de dados diretamente relacionado com o processo 5 (Montar Grafos de Simulação). Esse depósito terá a função de armazenar os grafos que serão o resultado da paralelização

automática ou modificações que forem realizadas pelos usuários diretamente nos grafos. Com essa saída, outras ferramentas que trabalham com geração automática de código poderão utilizar esses dados para gerar novas estruturas de programas a partir dos grafos corrigidos [Branco, 1999]. O resultado dessas modificações pode ser visto na figura 6.3.

A seguir será detalhada a estrutura interna de cada processo que compõem o modelo.

6.1.1 - Processo 1 – Preparar Dados de Arquitetura e Biblioteca

A camada de arquitetura tem como objetivo definir o ambiente em que o programa paralelo está implementado. Essas informações dizem respeito às características da arquitetura paralela, por exemplo, se o ambiente é fortemente acoplado ou fracamente acoplado; ou ainda, se representa um sistema distribuído ou uma máquina paralela, além de outras informações relevantes, como a quantidade de elementos de processamento, quantidade de memória, etc.

O desenvolvimento das arquiteturas paralelas visa melhorar o desempenho dos sistemas computacionais. Para alcançar esse objetivo, surgem obstáculos como o alto custo do desenvolvimento do *hardware* e do *software,* além da dificuldade encontrada pelos programadores em adequar o *software* paralelo à arquitetura, maximizando o paralelismo do programa e utilizando uma granulosidade mais adequada à arquitetura.

As arquiteturas paralelas podem ser definidas como "um conjunto de elementos de processamento que podem se comunicar e cooperar para resolver grandes problemas rapidamente" [Almasi e Gottlieb, 1994]. Essa definição cria uma abstração sobre o conceito de arquiteturas paralelas, permitindo que sejam elaboradas questões relativas aos elementos de processamento, aos mecanismos de comunicação, a cooperação entre esses elementos de processamento e sobre o que significa resolver grandes problemas rapidamente [Calônego, 1997]. Analisar todas as arquiteturas paralelas com as possíveis combinações para os dispositivos de entrada e saída, que crescem exponencialmente, não é uma tarefa fácil e nem recomendável. Entretanto, o estudo dessas informações aumenta consideravelmente a aplicabilidade dessa ferramenta, pois possibilita uma análise mais detalhada de um programa paralelo.

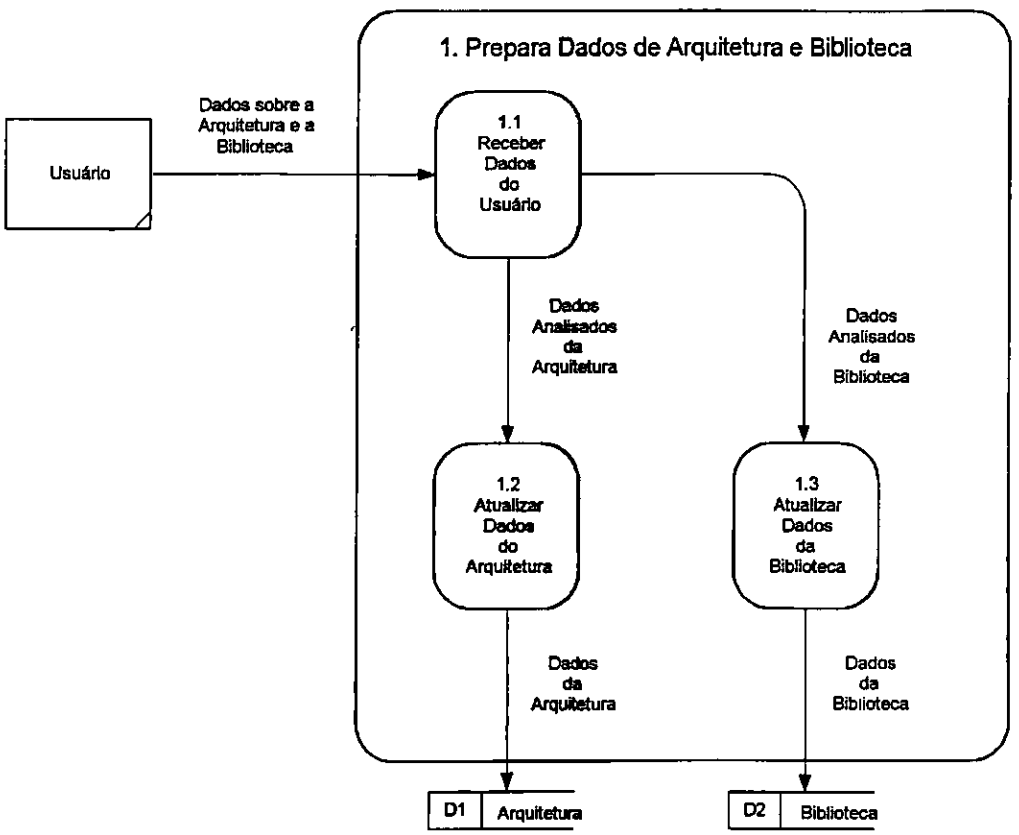
A busca de erros em um programa se restringe à análise semântica e sintática, aos testes de caixa preta e na verificação da adequação dos objetivos alcançados com a análise de requisitos, que é a especificação do que o software deve fazer. Um *software* paralelo possui outros aspectos que devem ser analisados, como a sua eficiência.

Um estudo da eficiência de um *software* paralelo deve se pautar na análise do grau de paralelismo alcançado no programa. Além disso, é fundamental o casamento entre o *software*

e o *hardware*, ou seja, a granulosidade explorada no programa deve estar sincronizada com as características do *hardware*.

O protocolo de saída dessa camada será utilizado pelas camadas de depuração e também pela camada tutor, com o objetivo de verificar se as técnicas de comunicação e sincronização usadas no desenvolvimento do programa estão adequadas. Além disso, com essas informações é possível traçar a granulosidade ideal do programa paralelo. Isto possibilita à ferramenta perceber as falhas relacionadas com o desempenho do programa e sugerir novas estratégias na divisão das tarefas que compõem a aplicação.

Figura 6.4 – Explosão do Processo 1 – Preparar Dados de Arquitetura e Biblioteca.



A figura 6.4 apresenta o detalhamento desse processo. É possível observar que o processo *1.1 – Receber Dados do Usuário* faz parte da *interface* com o usuário, sendo responsável pela verificação da consistência dos dados recebidos. Os processos *1.2 – Atualizar Dados da Arquitetura* e *1.3 – Atualizar Dados da Biblioteca* são responsáveis pelo armazenamento dos dados recebidos. Futuramente, será possível acrescentar um depósito de dados com informações de arquiteturas que foram avaliadas por *software benchmarks*. Essas informações poderão proporcionar um maior realismo durante a simulação do comportamento do programa, através da definição de unidades de tempo proporcionais às diferenças de tempo

dos processadores. Essas diferenças poderão ser percebidas pelos usuários durante a simulação, pois o tempo de simulação de cada tarefa estará associado à unidade de tempo atribuída ao processador na qual ela está alocada.

A seguir serão apresentadas as informações que devem ser tratadas na camada de arquitetura:

(1) **Arquitetura do Ambiente:** Esse item define o tipo de estrutura física do ambiente paralelo, ou seja, a arquitetura onde o programa executará: Sistema Distribuído, SP2, etc.

(2) **Elementos de Processamento:** Fixa o número de processadores que compõem a máquina paralela e os atributos desses processadores como: capacidade de endereçamento, *clock*, fabricante, família a que pertence, dentre outras.

(3) **Memória:** As informações da memória se dividem em dois grupos: memória fortemente acoplada e memória fracamente acoplada. No primeiro grupo os atributos são a velocidade de acesso e a capacidade máxima. O segundo grupo possui os mesmos atributos, mas por se tratar de várias memórias, uma para cada elemento de processamento, há a subdivisão desses atributos em cada módulo.

(4) **Barramento:** Define o tipo de barramento que está sendo usado no momento da análise do desempenho do programa, como por exemplo: ISA, EISA, PCI, etc. O objetivo é definir a topologia de comunicação que está sendo utilizada, visto que a comunicação entre processos influencia diretamente no desempenho do programa. A informação sobre o barramento também incorpora o atributo da *inferface* utilizada para a conexão entre os componentes e seus respectivos meios de transmissão de dados. Em várias situações uma mesma *interface* pode ser utilizada por protocolos de comunicação e meios de transmissão distintos.

(5) **Periféricos:** são os dispositivos de entrada e saída de dados aos quais o processador não tem acesso direto.

(6) **Informações Adicionais:** Quando se tratar de um ambiente heterogêneo (Sistema Distribuído) será necessário informar quais os tipos de computadores que fazem parte do sistema, permitindo uma análise do balanceamento da carga utilizado na definição do programa paralelo.

Um estudo mais detalhado sobre as características das arquiteturas paralelas pode ser encontrado em [Calônego, 1997].

Enquanto a camada de Arquitetura se responsabiliza pelo *hardware*, a camada de Biblioteca define as linguagens e bibliotecas que foram usadas na elaboração do programa e o

número de tarefas que compõem a aplicação paralela.

Esta camada trata das seguintes informações:

(1) **Linguagem Utilizada:** Define a linguagem em que o programa foi codificado para que seja utilizada a biblioteca correta durante o processo de depuração. As linguagens que são apresentadas nessa camada só poderão estar disponíveis se já existir a implementação dos respectivos módulos para depuração.

(2) **Biblioteca:** Esse atributo é utilizado quando se trata de bibliotecas para troca de mensagem como, por exemplo, PVM.

(3) **Sistema Operacional:** Possui informações intrínsecas relativas à adaptação de componentes como *drivers* controladores dos dispositivos de comunicação e outros elementos disponíveis num dado hardware.

(4) **Quantidade de tarefas:** Esse atributo define o número de tarefas projetadas pelo usuário na elaboração do programa. Desse modo, é possível verificar se os comandos para criação de tarefas escravas possuem falhas e, ainda, verificar as relações das tarefas com o número de elementos de processamento que foram definidos na camada de arquitetura. Dependendo do programa que está sendo analisado, não será necessário especificar explicitamente esse atributo, pois a quantidade de tarefas pode ser extraída diretamente do código. Entretanto, a utilização explícita desse item permite verificar se o objetivo do usuário com relação às divisões das tarefas do programa foi atingido.

(5) **Arquitetura do Sistema:** Diz respeito à forma como as tarefas estão divididas: SPMD (*Simple Program Multiple Data*) ou MPMD (*Multiple Program Multiple Data*).

Esse processo é responsável pela consistência das informações que os usuários inserem no sistema com relação à arquitetura e à biblioteca. Durante a implementação das versões finais para as linguagens existentes será necessária a existência de uma gramática livre de contexto ou BNF (Backus-Naur Form), para especificar a sintaxe das linguagens. Além de especificar a sintaxe da linguagem, uma gramática livre de contexto pode ser usada como auxílio para guiar a tradução de programas [Aho et al., 1995]. Dessa forma, um programa paralelo que apresente um alto grau de correctude na análise de seus grafos pode ser traduzido para outras linguagens que possuem abordagem de paralelismo semelhante.

6.1.2 - Processo 2 – Montar Grafos de Entrada

Dado um conjunto de objetos pode-se estar interessado no conteúdo desses objetos ou

na estrutura do conjunto, ou seja, nas relações entre os objetos. A teoria dos grafos permite que sejam concentradas as atenções no segundo objetivo, criando uma abstração do primeiro ponto de vista.

Os grafos constituem uma ferramenta básica para modelar fenômenos discretos e são fundamentais para a compreensão das estruturas de dados e análise de algoritmos. Na próxima seção será visto como utilizar grafos para se representar programas de computador e como utilizar esses grafos para depurar os programas.

6.1.2.1 – Grafos de Fluxo de Instrução e Grafos de Comunicação

Um dos aspectos da depuração de um programa é o estudo da seqüência de execução do programa. A maneira como um programa paralelo executa depende dos dados de entrada, a seqüência das instruções e a comunicação e sincronismo que existe entre as tarefas [Furtado, 1973].

Um programa de computador pode ser representado através de um grafo G = (N, A, f), onde:

- $N = \{n_1, n_2, ..., n_k\}$ é um conjunto não vazio de vértices (nós) que representa os comandos executáveis do programa ou um conjunto de instruções executadas seqüencialmente;
- A é um conjunto de arestas (arcos) que representa os caminhos do fluxo de instruções que o programa poderá percorrer;
- f é uma função que associa cada aresta a a um par não-ordenado x-y de vértices chamados de extremos de a.

Um caminho é uma seqüência finita de nós $(n_1, n_2, ..., n_k)$, $K \geq 2$, tal que existe um arco de n_i para n_{i+1} para i = 1,2,..., k-1, e representa uma seqüência de execução possível. Um caminho simples surge quando se passa apenas uma vez pelos arcos que o compõe. Quando s trata de um caminho simples diz-se que esse caminho é livre de laço.

Um caminho completo é um caminho onde o primeiro nó é o nó de entrada e o último nó é o nó de saída do grafo G. Seja IN(x) e OUT(x) o número de arcos que entram e que saem de x, respectivamente. Se IN(x)=0 então x é um nó de entrada, e se OUT(x)=0, então x é um nó de saída.

Quando se utiliza as estruturas de um grafo para fazer testes ou depurar um programa, é interessante que todos os caminhos possíveis do programa sejam executados. Entretanto, essa é uma tarefa bastante difícil, devido ao grande número de possíveis caminhos que um programa pode fornecer.

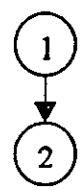
A representação de um programa P como um grafo de fluxo de controle consiste em

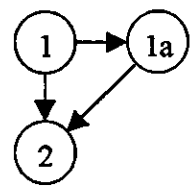
estabelecer uma correspondência entre nós e blocos e em indicar possíveis fluxos de controle entre blocos através dos arcos. É, portanto, um dígrafo com um único nó de entrada $n_1 \in N$ e um único nó de saída n_k, no qual cada vértice representa um bloco indivisível de comandos e cada aresta representa um possível desvio de um bloco para o outro.

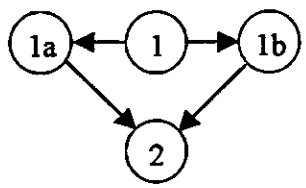
A tabela 6.1 apresenta os grafos que representam as estruturas básicas de um programa. As estruturas são apresentadas através de algoritmos para não se prender a sintaxe de uma linguagem específica.

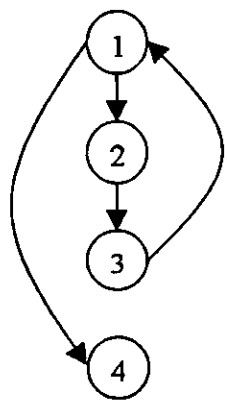
Através do grafo do fluxo de instruções podem ser escolhidos os componentes que devem ser executados, isto caracteriza um teste de caixa branca ou teste estrutural.

Tabela 6.1 – Estruturas de Programas e a Representação Através de Grafos

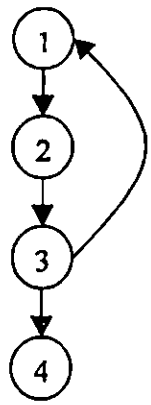
a)
```
1   X ← Y
2   Y ← 10
...
```


b)
```
1   se Condição
    |  então Comando
    Fim-se
2   ...
```


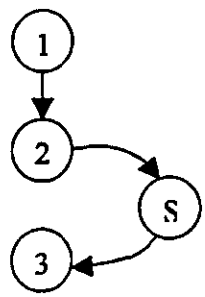
c)
```
1   se Condição
    |  então Comando     (1a)
    |  senão Comando     (1b)
    fim-se
    ...
```


d)
```
    enquanto Condição faça
    |  Comando
    fim-enquanto
    ...
```


e)
```
repita
|  Comando
até Condição
...
```



f)
```
...
SubRotina (Parâmetros)
...
```



Para exemplificar a criação de um grafo de fluxo de instruções que represente um programa, será utilizado o código fonte apresentado na listagem 6.1. Esse programa usa a biblioteca de troca de mensagens PVM (*Paralell Virtual Machine*).

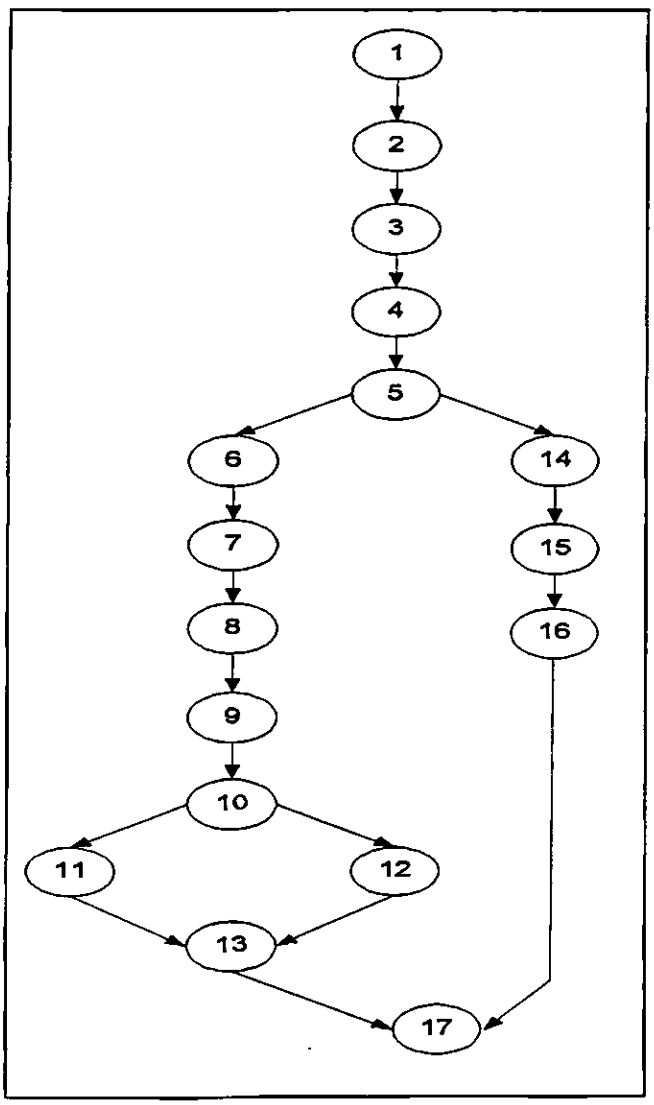
Listagem 6.1 – Programa Exemplo para criação de Grafo

```
        #include <stdlib.h>
        #include <stdio.h>
        #include <pvm3.h>
-01-    void main() {
-02-      int mytid, parent;
-03-      mytid = pvm_mytid();
-04-      parent = pvm_parent();;
-05-      if (parent == PvmNoParent) {
-06-         int tids[2], numt;
-07-         pvm_catchout(stdout);
-08-         numt = pvm_spawn("tarefas", NULL, PvmTaskDefault,
                              "", s, tids);
-09-        printf("Abriu %d processos\n", numt);
-10-        if (numt < 0)
-11-            printf("Nao abriu os processos\n");
            else
-12-            printf("Eu sou o mestre.\n");
-13-        pvm_exit();
          } else {
-14-         int mytid;
-15-         mytid = pvm_mytid();
-16-         printf("Eu sou o escravo %d\n", mytid); }
-17-   }
```

Inicialmente, as linhas do programa fonte são numeradas para facilitar a associação dos nós do grafo resultante com o programa. Cada comando do programa que está sendo depurado, corresponde a um vértice do grafo. A ferramenta analisa o código procurando pelas estruturas básicas apresentadas na tabela 6.1. Parte-se da primeira instrução do programa. Toda as declarações anteriores não são utilizadas para a montagem do grafo de fluxo de instruções. O interesse com a criação desse grafo são os diversos caminhos que o programa pode percorrer durante sua execução. Durante esta análise, a ferramenta vai definindo as relações de dependência entre os nós, ou seja, os fluxos possíveis de execução do programa.

A figura 6.5 apresenta o grafo resultante gerado a partir do programa da listagem 6.1 Nessa representação é possível notar que todos os comandos possuem um nó correspondente no grafo. Porém, representar todos os comandos do programa, produz um grafo de grandes dimensões.

Figura 6.5 – Grafo do Fluxo de Instruções da Listagem 6.1



Alguns comandos que não representam instruções paralelas podem ter seus nós excluídos do grafo, pois essa eliminação não altera a análise que será realizada para localizar os problemas de comunicação e sincronismo dos processos.

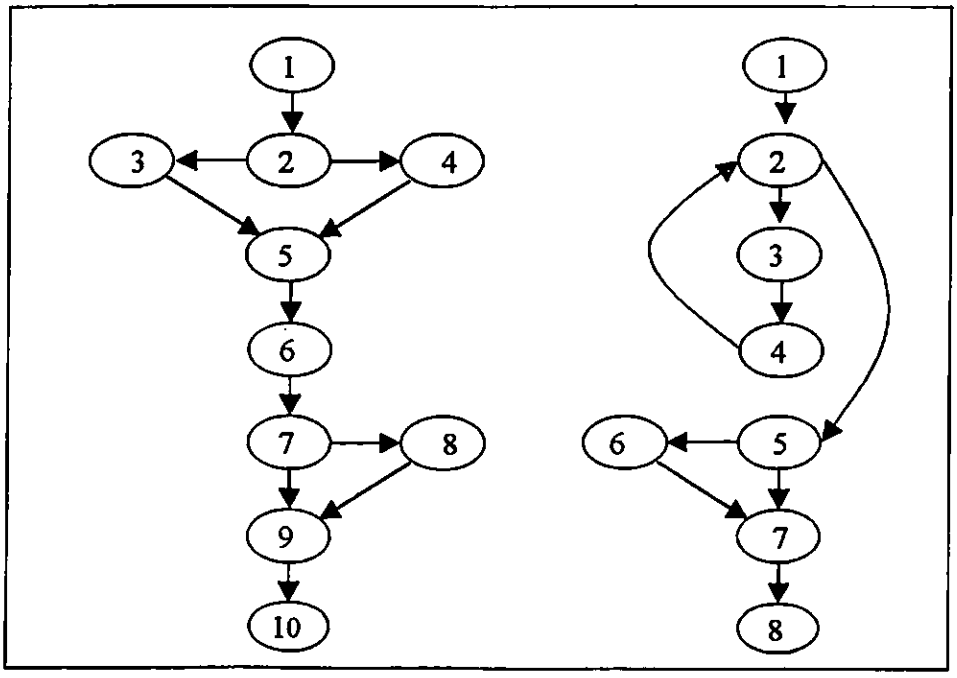
Listagem 6.2 – Exemplo de Troca de Mensagens Entre Processos

```
Processo A                          Processo B

1 inicio                            1 inicio
2     se condição                   2     enquanto condição faça
3        então comando              3        comando
4        senão comando              4     fim-enquanto
5     fim-se                        5     se condição
6     comando                       6        então RECEIVE (Mensagem)
7     se condição                   7     fim-se
8        então SEND (Mensagem)      8 fim
9     fim-se
10    fim
```

O grafo que será utilizado para a depuração é denominado grafo das comunicações. Para montar o grafo das comunicações, deve-se preservar todos os nós que representam instruções paralelas e os nós que representam mudanças no fluxo do programa, como as estruturas condicionais e as estruturas de repetição, quando existirem instruções paralelas dentro dessas estruturas, ou seja, quando houver caminhos alternativos que não passem pelas instruções paralelas.

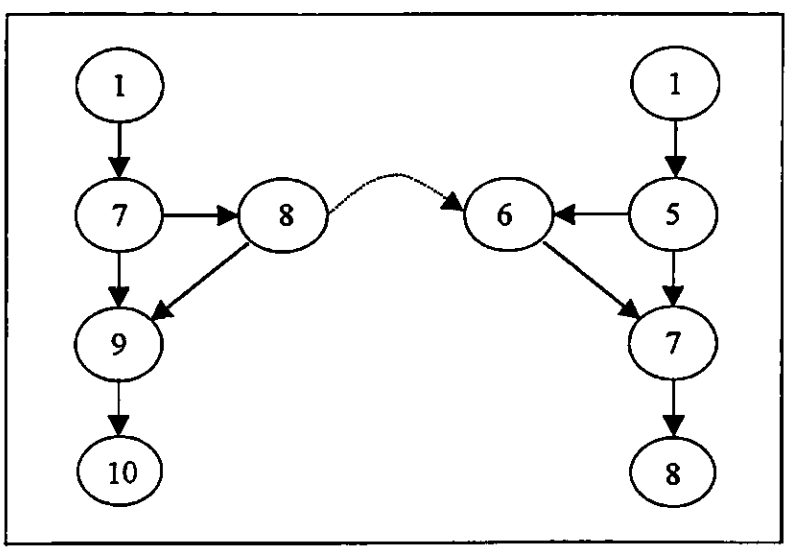
Figura 6.6 – Grafo do Fluxo de Instruções da Listagem 6.2



Para se construir o grafo das comunicações entre os processos, utiliza-se os grafos originais do fluxo de instruções de cada tarefa do programa, eliminando os nós que não representam as instruções de comunicação e sincronismo entre os processos e acrescentando os arcos de dependência entre essas instruções. A figura 6.6 apresenta os grafos de instruções que representam os programas da listagem 6.2.

Na figura 6.7 pode ser visto o grafo de comunicação entre os processos A e B da listagem 6.2 depois das modificações nos grafos de fluxo de instruções da figura 6.7. No processo A, os nós 2, 3, 4, 5 e 6 foram eliminados pois não existe comando paralelo nesse trecho. O nó 7 foi preservado pois representa um desvio no fluxo de execução do programa e um dos caminhos a partir desse nó possui um comando de envio de mensagem (nó 8). A transformação ocorrida no processo A foi semelhante, os nós 2, 3 e 4 não foram acrescentados no grafo de comunicações por motivos idênticos aos do processo A.

Figura 6.7 – Grafo do Fluxo das Comunicações dos Processos da Listagem 6.2



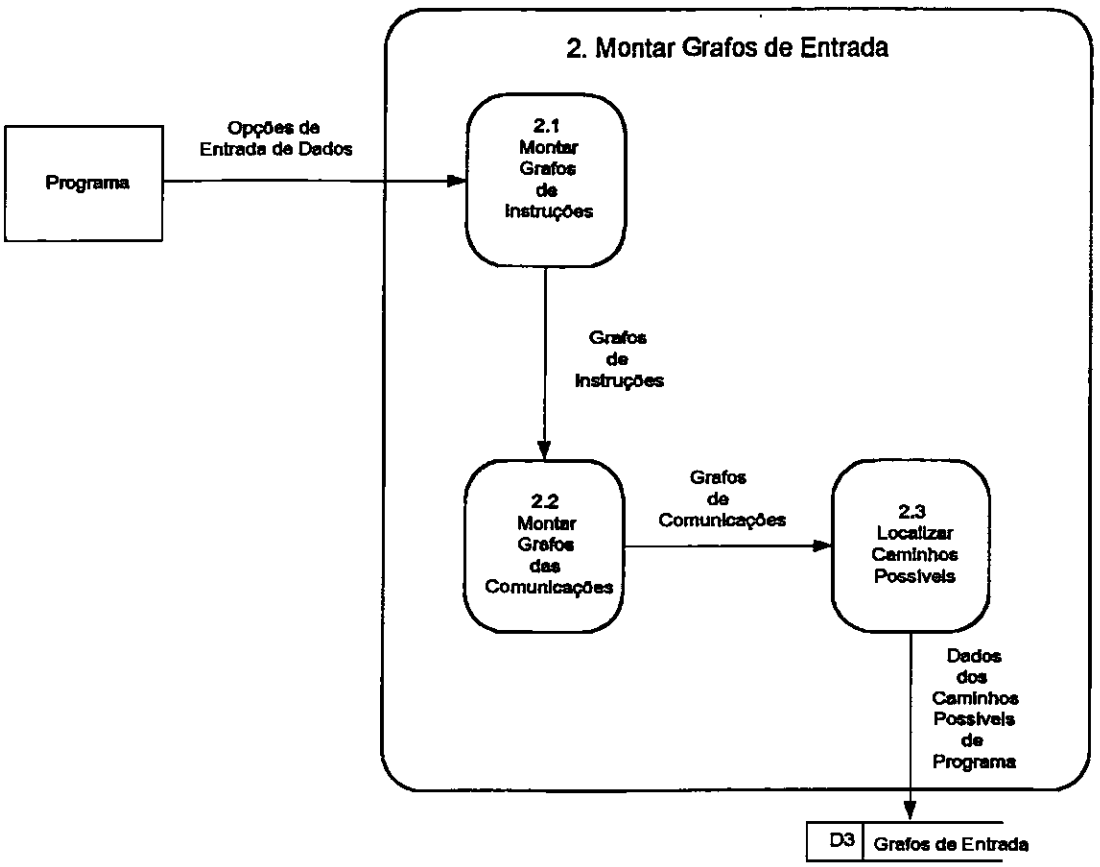
6.1.2.2 – Descrição do Processo 2

Esse processo é responsável em criar os dois tipos de grafos apresentados na seção anterior: grafos do fluxo de instruções (ou de controle) e grafos de sincronização e o detalhamento de suas funções está representado na figura 6.8.

O processo *2.1 – Montar os Grafos de Instruções* realiza uma análise semântica no programa do usuário, gerando os grafos do fluxo de instruções que representam os caminhos possíveis de serem executados pelo programa. Cada grafo representa a visão do comportamento funcional de uma tarefa. O processo *2.2 – Montar Grafos das Comunicações*, produz os grafos de sincronização que apresentam as alternativas dos caminhos existentes nas

comunicações não determinísticas dos processos. Após a geração desses grafos, o processo *2.3 – Localizar Caminhos Possíveis*, analisa os grafos para encontrar os caminhos que o programa pode percorrer durante a sua execução. Através desses caminhos, os processos 3, 4 e 5, podem depurar o programa, simulá-lo e interagir com o usuário.

Figura 6.8 – Explosão do Processo 2 – Montar Grafos de Entrada.



6.1.3 - Processo 3 – Depurar o Programa

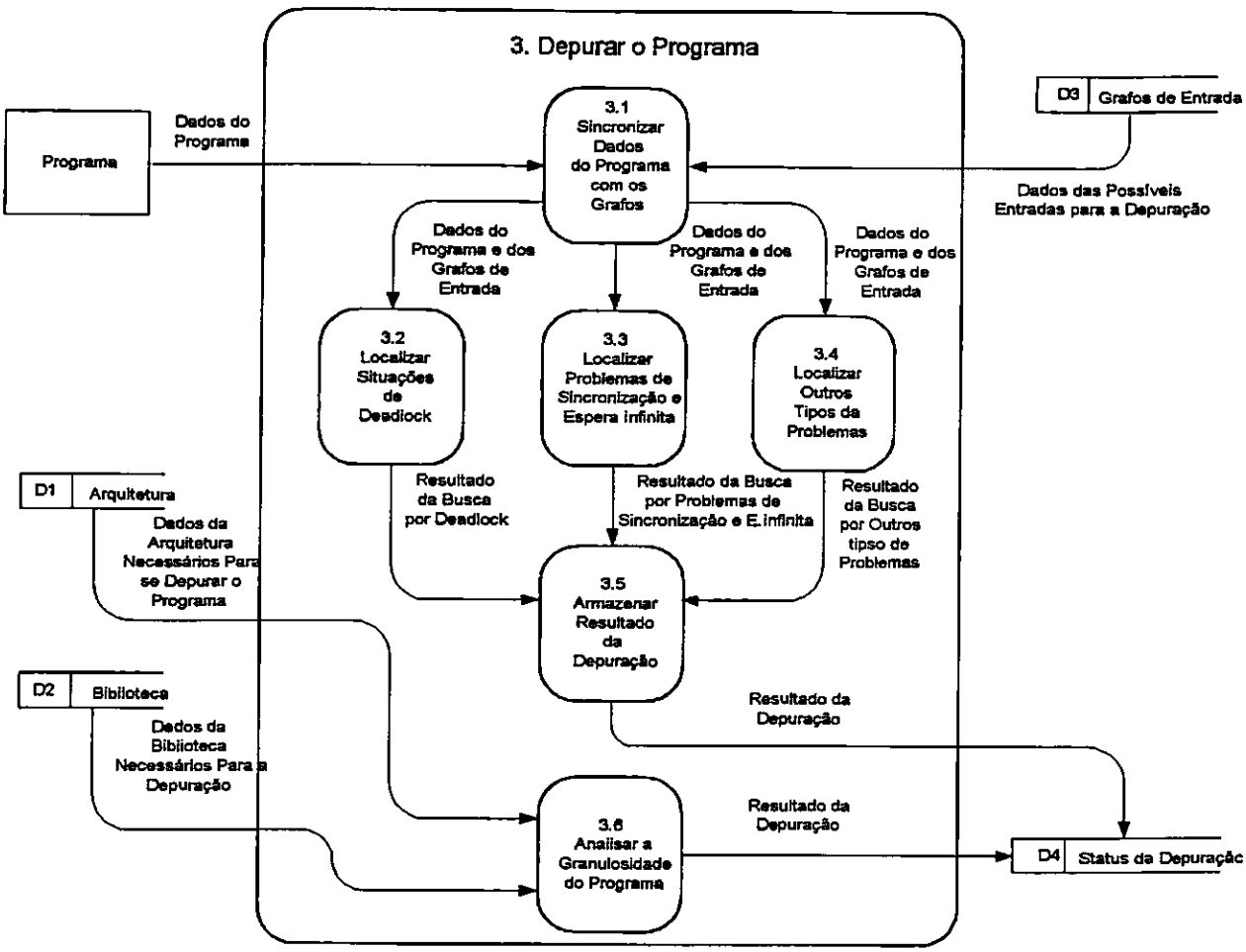
Esse processo é composto por vários módulos que possuem o objetivo de resolver problemas específicos.

Após a definição dos grafos no processo 2, o processo de depuração caminha pelas rotas possíveis, procurando por problemas que podem estar provocando os erros. Todos os resultados da depuração são armazenados para a utilização das camadas Tutor e de Apresentação.

O detalhamento do processo 3 pode ser visto na figura 6.9. Ele possui um processo para resolver problemas de *deadlock* (*3.2 – Localizar Situações de Deadlock*), um processo para resolver problemas de sincronização e espera infinita (*3.3 – Localizar Problemas de Sincronização e Espera Infinita*) e um processo que resolve outros tipos de erro como, por exemplo, as análises das partes seqüenciais do programa. O processo *3.1 – Sincronizar Dados*

do Programa Com os Grafos faz o vínculo de cada nó dos grafos criados com as respectivas instruções nas tarefas do programa paralelo. O processo *3.5 – Armazenar Resultado da Depuração*, grava as saídas dos subprocessos responsáveis pela localização de erros e o processo *3.6 – Analisar a Granulosidade do Programa,* procura verificar se a granulosidade de programa paralelo está adequada com a arquitetura.

Figura 6.9 – Explosão do Processo 3 – Depurar o Programa.



6.1.3.1 - O Problema da Espera Infinita

Entre os problemas que surgem com os programas paralelos, a espera infinita é um dos mais comuns. Não chega a caracterizar um problema de *deadlock*, pois não existe um laço contendo vários processos bloqueados, mas a existência de um processo esperando por uma mensagem que nunca ocorrerá independente dos outros processos estarem bloqueados ou não.

Para localizar a existência de caminhos que levem um ou mais processos de um programa paralelo a um estado de espera infinita é necessário verificar nos grafos de comunicação criados no processo 2, a existência de caminhos alternativos que não passem pela comunicação. É importante lembrar que os grafos de comunicação não possuem a mesma

quantidade de nós que os grafos originais do programa (grafos de fluxo de instruções) e mantém apenas os nós que definem as comunicações entre os processos. A existência de caminhos que não passem pelos arcos da comunicação indica a existência de caminhos que podem provocar o problema da espera infinita.

A figura 6.7 apresenta um grafo de comunicação e sincronismo entre os processos, com a possibilidade de ocorrer o problema de espera infinita, durante a execução do programa. Se o processo A seguir o caminho 1-7-9-10 e o processo B seguir o caminho 1-5-6, esse processo ficará esperando por um evento que nunca ocorrerá. Outra situação indesejável pode acontecer se o processo A seguir o caminho 1-7-8-9-10 e enviar uma mensagem e o processo B seguir o caminho 1-5-7-8 que não espera por mensagem. Se o envio for bloqueante o processo A entrará em espera infinita, caso contrário uma mensagem será perdida.

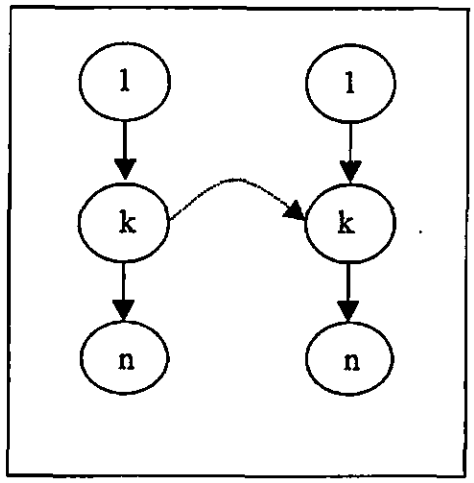
Listagem 6.3 – Exemplo de Processos Livres do Problema de Espera Infinita

Processo A	Processo B
1 inicio 2 ... {comandos} k SEND (Mensagem) k+1 ... {comandos} n fim	1 inicio 2 ... {comandos} k RECEIVE (Mensagem) k+1 ... {comandos} n fim

A dificuldade nessa análise é verificar o resultado das condições representadas pelos nós 7 no processo A e 5 no processo B. Para isso é preciso realizar uma análise do histórico das variáveis contidas na condição de seleção. É possível verificar a probabilidade dos processos entrarem em espera infinita através desse histórico, transformando todas as condições que compõem o caminho que se deseja analisar em equações [Gupta et al., 1998]. O conjunto dessas equações pode ser resolvido através de um sistema de equações. Se o sistema possuir solução, existirão valores que as variáveis poderão receber e que forçará a passagem pelo caminho com problema. Mas, ainda assim, é muito difícil encontrar o conjunto completo de valores das entradas do programa que levam a um resultado indesejado.

O grafo da figura 6.10 representa o programa da listagem 6.3. Nele pode-se verificar a impossibilidade de ocorrer o problema da espera infinita. Essa situação só irá ocorrer na prática se a mensagem enviada for perdida no meio de comunicação, situação que não será tratada por esta ferramenta.

Figura 6.10 – Grafo do Fluxo das Comunicações dos Processos da Listagem 6.3



6.1.3.2 - *Deadlock*

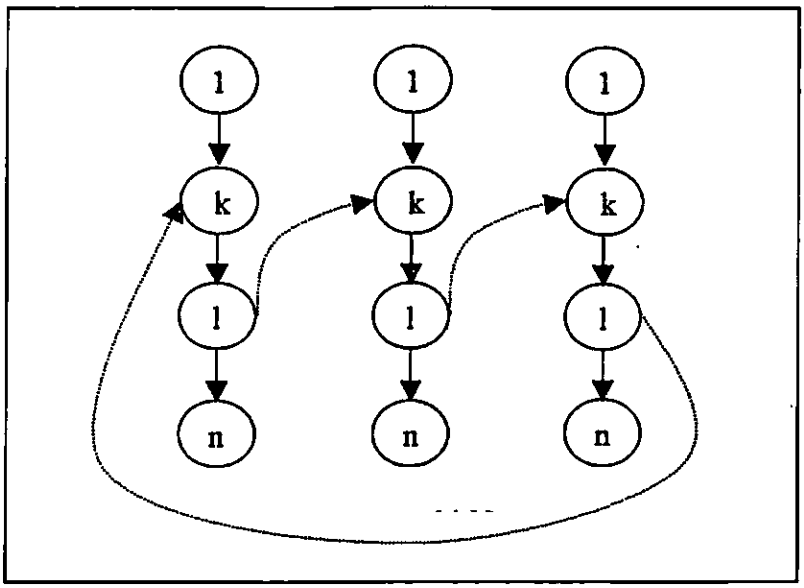
Para encontrar a ocorrência de *deadlock* é necessário percorrer os grafos de comunicação que compõem o programa paralelo procurando pela existência de ciclos. A existência de ciclos define a dependência das mensagens que trafegam entre as tarefas.

O objetivo da ferramenta é localizar a ocorrência de *deadlock* interno. A ferramenta pode através do processo 5, sugerir mudanças no programa do usuário para gerenciar a alocação de vários recursos, utilizando métodos como o apresentado em [Tanenbaum, 1992] denominado algoritmo do banqueiro, que libera os recursos somente quando esses recursos não causarem *deadlock*, como em um empréstimo bancário. A listagem 6.4 apresenta três tarefas que se encontram em *deadlock* como pode ser visto na figura 6.11, através do ciclo representado pelos arcos da comunicação. Durante a interação com o usuário, a ferramenta deverá propor mudanças no código para resolver o problema como, por exemplo, a inversão do nó *k* com o nó *l* na tarefa A, eliminando o ciclo e conseqüentemente o *deadlock*. Se essa mudança não for possível, devido à lógica do programa, então a ferramenta irá sugerir a troca do comando para um *receive* não bloqueante.

Listagem 6.4 – Processos em *Deadlock*

Tarefa A	Tarefa B	Tarefa C
l inicio	l inicio	l inicio
... {comandos}	... {comandos}	... {comandos}
k RECEIVE (C, Msg)	k RECEIVE (A, Msg)	k RECEIVE (B, Msg)
... {comandos}	... {comandos}	... {comandos}
l SEND (B, Msg)	l SEND (C, Msg)	l SEND (A, Msg)
... {comandos}	... {comandos}	... {comandos}
n fim	n fim	n fim

Figura 6.11 – Grafo do Fluxo das Comunicações das Tarefas da Listagem 6.4



6.1.4 - Processo 4 – Criar Ambiente Tutor

O processo *4 – Criar Ambiente Tutor,* através das informações que trafegam no sistema, permite o desenvolvimento de um grande número de aplicações. É possível a criação de um ambiente tutor inteligente interagindo com uma fonte de conhecimento que poderá "aprender" o estilo de programação do usuário para possibilitar uma programação cada vez mais eficiente.

A criação de um ambiente tutor na ferramenta privilegia a mudança de paradigma no processo de ensino-aprendizagem: o modelo tradicional, que centraliza a atenção no professor, para o modelo centrado no aluno e com base em "aprender a aprender". Com a análise dos próprios erros e com o amparo do ambiente tutor, o usuário passa a dominar as técnicas e entende o processo a fim de que possa generalizar um aprendizado para novas situações.

É importante notar que a entidade externa "Fonte de Conhecimento" representa um sistema à parte. Trata dos aspectos inerentes aos Sistemas Tutores, ou seja, os módulos do domínio do conhecimento, o módulo tutor e o módulo de controle. Esse processo (Criar Ambiente Tutor) é o responsável pela comunicação entre o usuário, às informações que trafegam neste sistema (Depurador) e pelo sistema tutor, representando parte do módulo da interface com o usuário.

O motivo de se localizar a arquitetura do Tutor fora da fronteira de definição do sistema ocorre porque o Processo 4 realiza a mediação das informações que trafegam no sistema de depuração com a entidade externa *Conhecimento*. Essa independência torna

flexível a evolução dos dois sistemas (ambiente de depuração e sistema tutor) sem um afetar o outro. Como o modelo do sistema tutor irá possuir a sua maior parte implementada fora desta ferramenta, esse processo não será explodido.

6.1.5 - Processo 5 – Montar Grafos de Simulação

A simulação será utilizada para criar um ambiente onde o usuário percebe o comportamento do programa sem a necessidade de executá-lo. Existem algumas vantagens em simular o comportamento de um programa paralelo em um ambiente de depuração. Além da visão global do programa, a simulação permite verificar processos em estado de espera infinita ou a presença de *deadlock*. É muito difícil para um usuário iniciante da programação paralela visualizar o comportamento de um programa paralelo que possui tarefas que estão executando em processadores diferentes e que em muitos casos (por exemplo, em sistemas distribuídos) estão fisicamente distantes. É comum durante o aprendizado da programação paralela, a criação de processos que não executam da maneira esperada e que entram em espera infinita. Várias vezes, quando ocorrem esses erros durante a execução do programa, é difícil perceber qual processo está parado e em que ponto.

A simulação gerada por esse processo permite ao usuário controlar o comportamento do programa, percebendo a dependência que existe entre as tarefas. Além disso, é possível modificar o "desenho" do grafo no próprio ambiente para verificar o comportamento do sistema, ou seja, é possível propor mudanças no comportamento do programa para verificar o resultado antes mesmo da correção do programa. Outra relevância é a possibilidade de análise de um problema sem a existência de um programa paralelo implementado. Se o grafo construído diretamente no ambiente comportar-se corretamente é possível a utilização de ferramentas que geram o *arcabouço* da aplicação [Branco, 1999].

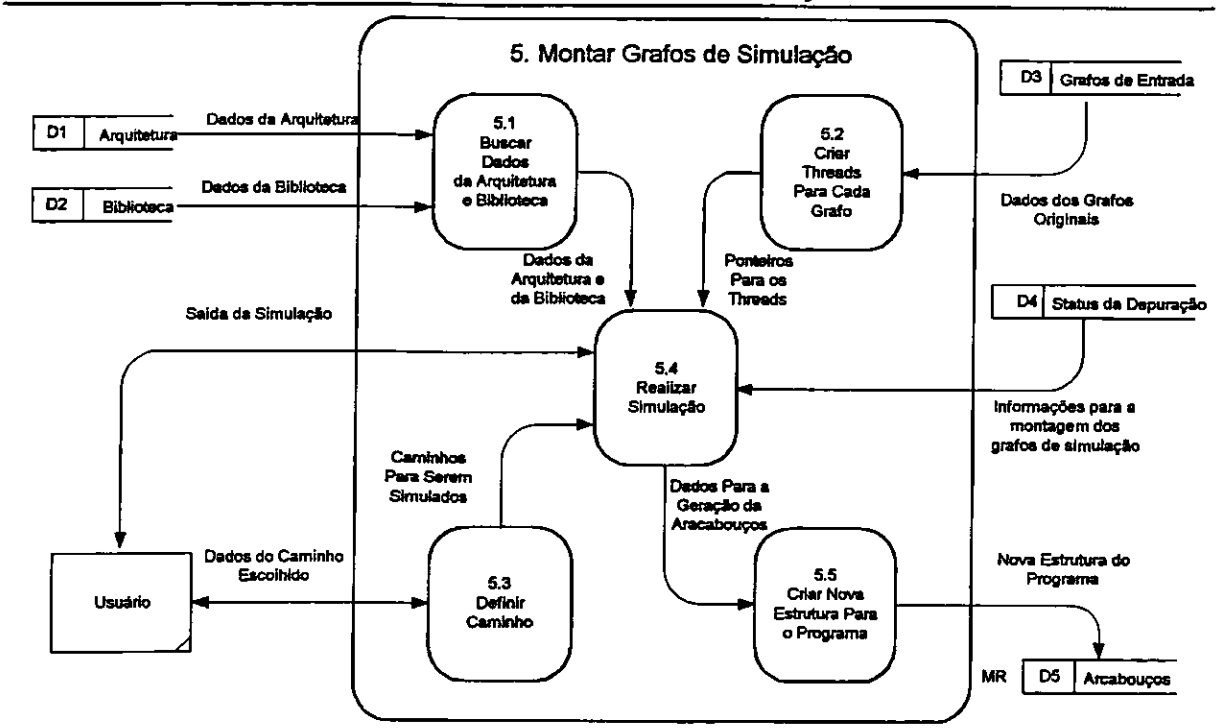
Durante a simulação, cada tarefa que compõe o programa paralelo é representada por um grafo. Na simulação é possível criar linhas de controle (*threads*) para a simulação de cada grafo, isto possibilita a representação da diferença de desempenho dos processadores em ambientes heterogêneos através da atribuição de prioridades às linhas de controle proporcionais as diferenças reais do poder de processamento das CPUs que compõem o ambiente onde o programa paralelo será executado.

A estrutura da ferramenta possui uma característica interessante que é a possibilidade de analisar o nível de paralelização utilizado pelo usuário, através dos grafos de fluxos do programa. É possível analisar as precedências das instruções e verificar o que pode ser melhorado no código do usuário. Através da análise de como os dados do programa se

relacionam e como as estruturas do programa estão encadeadas é possível identificar os trechos possíveis de paralelização. O processo permite após a paralelização comparar a eficiência alcançada.

O detalhamento desse processo pode ser visto através da figura 6.12. O processo *5.1 Buscar Dados da Arquitetura e Biblioteca* é o responsável de configurar o ambiente para simular os grafos com parâmetros próximos da arquitetura real. O processo *5.2 – Criar Threads Para Cada Grafo* inicializa os subprocessos que irão simular o fluxo de execução. O processo *5.3 – Definir Caminho* interage com o usuário para a escolha de um dos caminhos encontrados pelo processo 2.

Figura 6.12 – Explosão do Processo 5 – Montar Grafos de Simulação



Após essas definições, o processo *5.4 – Realizar a Simulação* coloca cada linha de controle (*thread*) para executar. Se o usuário desejar, é possível ainda, gerar o protocolo de saída para a montagem do arcabouço de uma nova aplicação.

O funcionamento desse processo pode ser melhor compreendido através das explicações contidas no próximo capítulo que descreve um protótipo da ferramenta de depuração de programas paralelos.

6.2 – Considerações Finais

Na especificação do projeto dessa ferramenta teve-se o cuidado de não analisar casos particulares, ou seja, as estruturas não pertencem a uma determinada linguagem. Foi objetivo criar uma especificação formal de como deve ser uma ferramenta para auxílio na depuração de programas paralelos contendo um ambiente de aprendizagem de programação paralela. Esse projeto possibilita a implementação em diversas linguagens para programação paralela voltada a ambientes em que a troca de mensagem seja usada como mecanismo de comunicação e sincronização de processos.

Capítulo 7
O Desenvolvimento de Um Protótipo

O objetivo na criação do protótipo descrito nesse capítulo é demonstrar a aplicabilidade do projeto. Uma vez desenvolvido, o protótipo permite visualizar as tarefas que compõem a ferramenta. Além disso, é possível visualizar as dificuldades que serão enfrentadas durante a construção da ferramenta vinculada a uma linguagem ou ambiente de programação paralela específico.

A maior vantagem proporcionada pela criação de um protótipo é avaliar o potencial do sistema e permitir mudanças durante o projeto da ferramenta. Posteriormente, durante a implementação de uma versão final, em uma determinada linguagem, o protótipo pode ser reutilizado.

Outro aspecto importante é a utilização do protótipo como projeto de *interface* da ferramenta. É possível verificar a facilidade com que os usuários interagem com o ambiente.

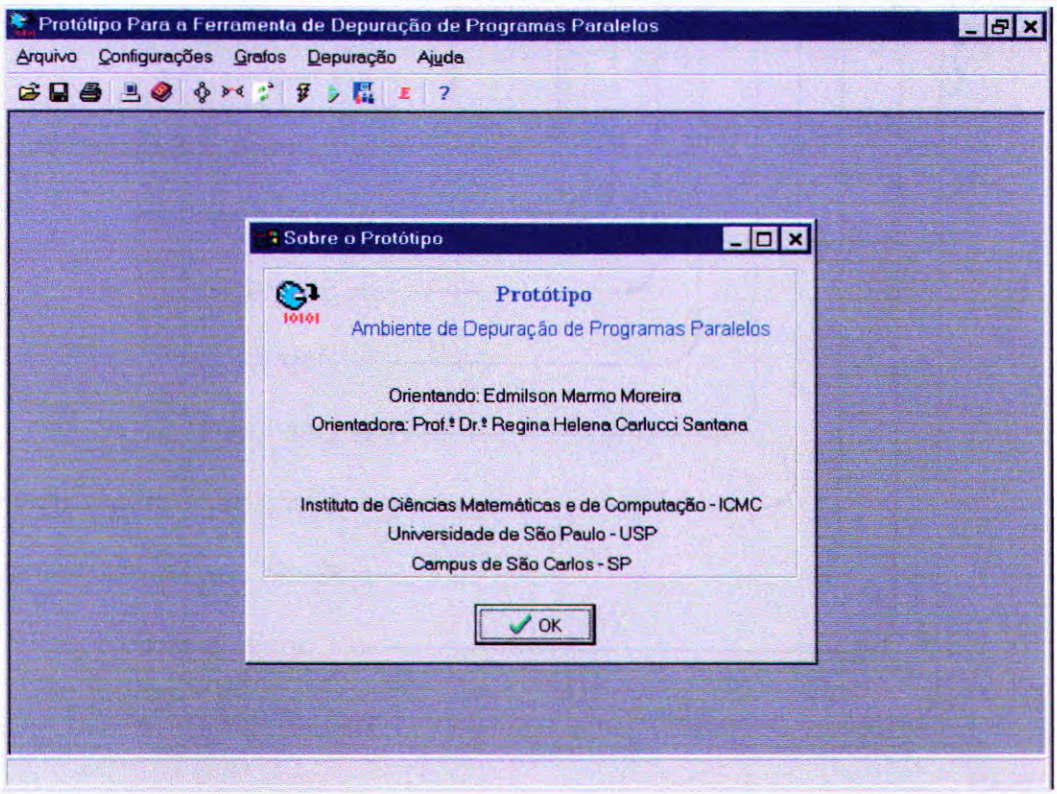
A *interface* com o usuário é o mecanismo por meio do qual se estabelece um diálogo entre o programa e o ser humano [Pressman, 1992]. A *interface* proposta nesse trabalho é bastante intuitiva, como pode ser visto na figura 7.1.

Nas seções desse capítulo serão abordados os assuntos pertinentes à criação desse protótipo e as possíveis dificuldades de implementação que serão encontradas no desenvolvimento de uma versão final da ferramenta.

O protótipo foi desenvolvido com a utilização da linguagem de programação Delphi. Apesar de não ser uma linguagem explicitamente paralela, o Delphi atende aos objetivos propostos para esse protótipo. Em primeiro lugar, o Delphi é uma linguagem orientada a objetos, o que permite uma boa modularização do código, além de ser uma versão RAD (*Rapid Application Development* – Desenvolvimento Rápido de Aplicativos) do Turbo Pascal

para *Windows*. Em ambientes RAD, o projetista define o que quer ver e o ambiente de programação cria o código necessário para realizar isso. Outro fator importante é a possibilidade de se criar *threads* (linhas de controle). Durante a execução, o *thread* primário de um processo pode criar outros *threads* (e também outros processos), os quais, por sua vez, também podem criar mais *threads* ou mais processos. Dessa forma, o sistema operacional consegue escalonar o uso do processador entre os diversos *threads*, permitindo a concorrência entre os *threads* pertencentes a um ou vários processos. Mais informações sobre essa linguagem podem ser encontradas em [Carvalho, 1998], [Norton e Mueller, 1997] e [Cantú, 1999].

Figura 7.1 – Tela Principal do Ambiente de Depuração de Programas Paralelos



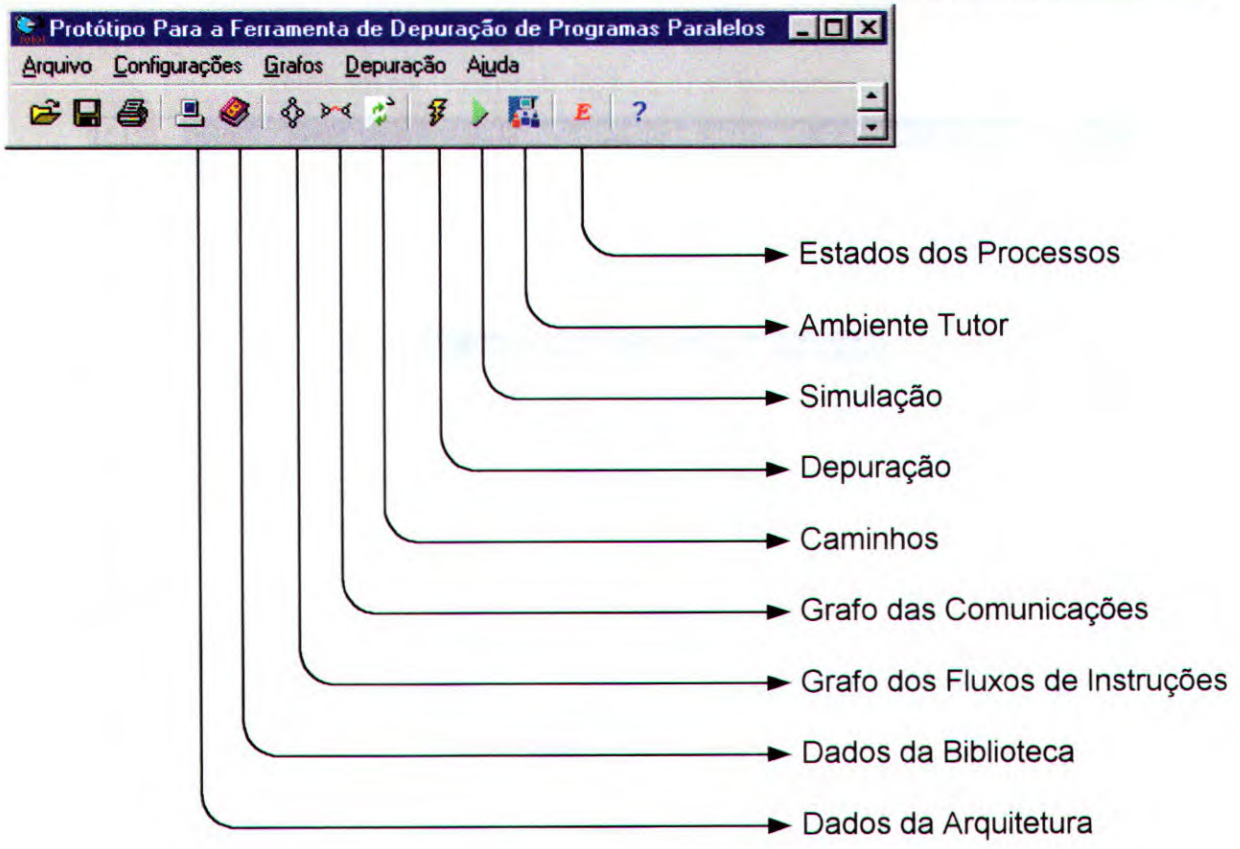
7.1 – O Ambiente

A *Interface* básica do protótipo é dividida em duas partes: o *Menu* de opções da ferramenta e a área de trabalho que irá apresentar as diversas janelas para as simulações além de permitir a edição dos algoritmos paralelos.

A ferramenta desenvolvida deve expressar com clareza as camadas especificadas em seu projeto e os processos que surgiram no modelo apresentado no capítulo anterior. Isto pode ser facilmente identificado nas opções que o protótipo oferece (figura 7.2). No item de Configurações é possível atualizar os dados correspondentes às camadas de *Arquitetura* e da

Biblioteca. O item Grafos está dividido em três opções que permitem criar os grafos de fluxo de instruções, os grafos de comunicação e gerar os caminhos possíveis de serem percorridos pelo programa, além de visualizar os estados dos processos que estiverem sendo simulados. O item Depuração possui as opções para se depurar o programa, simulá-lo ou acionar o ambiente tutor. Através da opção de Arquivo é possível abrir um programa ou editá-lo, além das opções de salvar e imprimir os arquivos de programa, os grafos e os caminhos correspondentes.

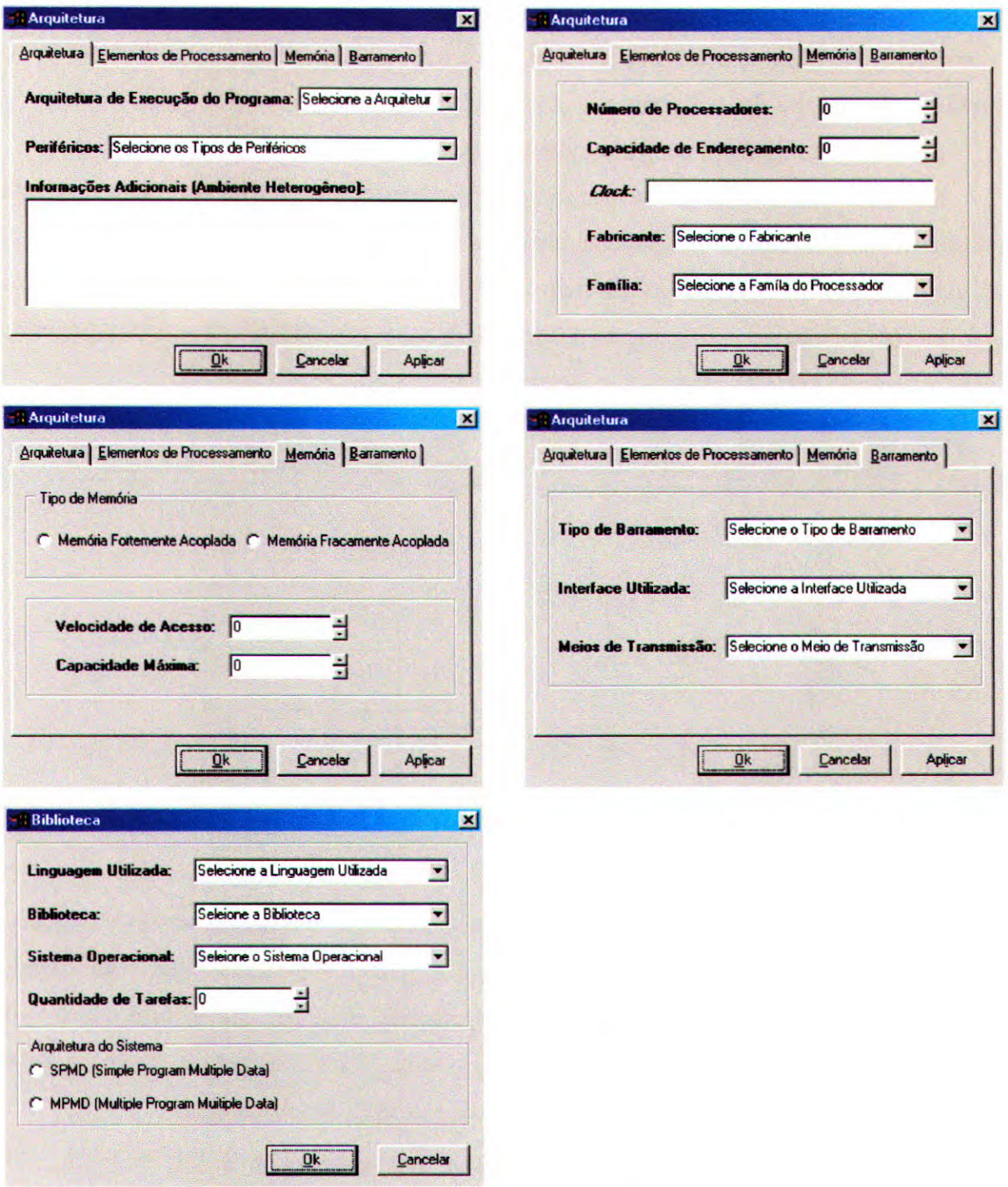
Figura 7.2 – As principais Funções do Ambiente de Depuração de Programas Paralelos.



As telas para entrada de dados das informações definidas no capítulo 6, referente às camadas de Arquitetura e Biblioteca, podem ser vistas na figura 7.3.

A próxima seção descreve as características das estruturas responsáveis pela definição dos grafos utilizados no protótipo.

Figura 7.3 – Entrada dos Dados da Arquitetura e da Biblioteca.



7.2 – Representando um Algoritmo Através de Um Grafo

Existem várias formas de armazenar um grafo em um computador. Uma representação bastante comum é através da utilização de matrizes.

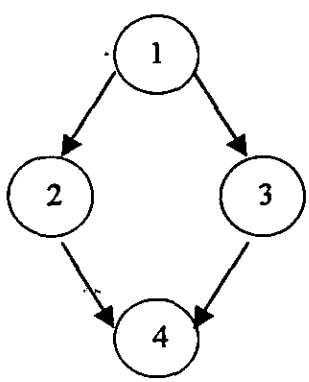
Supondo que um grafo possui n vértices numerados n_1, n_2, ..., n_n. Essa numeração define uma ordenação arbitrária no conjunto de vértices. É importante observar que esse tipo de conjunto é chamado de um conjunto ordenado. No entanto, isso é feito com o único intuito

de identificar os vértices – não há qualquer relevância no fato de um vértice aparecer antes de outro na ordenação. A partir dos vértices ordenados, é possível formar uma matriz $n \times n$ onde o elemento i, j é o número de arestas entre os vértices n_i e n_j. Essa matriz é chamada de **matriz de adjacências** do grafo com relação à ordenação. Portanto,

$a_{ij} = p$ onde existem p arcos partindo do vértice n_i para o vértice n_j

Estão sendo considerados apenas os grafos dirigidos (dígrafos) devido a sua relevância para este trabalho. A figura 7.4 apresenta um grafo e a matriz de adjacência correspondente. É uma matriz 4 x 4, pois o grafo possui quatro nós. O elemento 1,1 é 0 devido ao fato de não haver um laço no vértice 1. O elemento 1,2 (primeira linha, segunda coluna) é 1 porque existe apenas uma aresta entre os vértices 1 e 2, e assim por diante.

Figura 7.4 – Exemplo de Grafo e Sua Representação através de uma Matriz de Adjacência.



	1	2	3	4
1	0	1	1	0
2	0	0	0	1
3	0	0	0	1
4	0	0	0	0

O grafo apresentado na figura 7.4 produziu uma matriz esparsa, devido aos poucos arcos que existem no grafo. Os grafos que representam um programa têm relativamente poucas arestas. Esses grafos têm matrizes de adjacências *esparsas;* isto é, suas matrizes de adjacências contêm muito zeros. Supondo um programa com n instruções, produzindo um grafo com n vértices, serão necessários n^2 elementos da matriz de adjacências.

Um grafo com relativamente poucas arestas pode ser representado mais eficientemente, armazenando apenas os elementos não-nulos de sua matriz de adjacências. Essa representação consiste em uma lista para cada vértice de todos os vértices adjacentes a ele. Usam-se ponteiros para permitir que se caminhe de um elemento da lista para o seguinte. Esse tipo de estrutura de dados é chamado de *lista encadeada.* [Gersting, 1993]

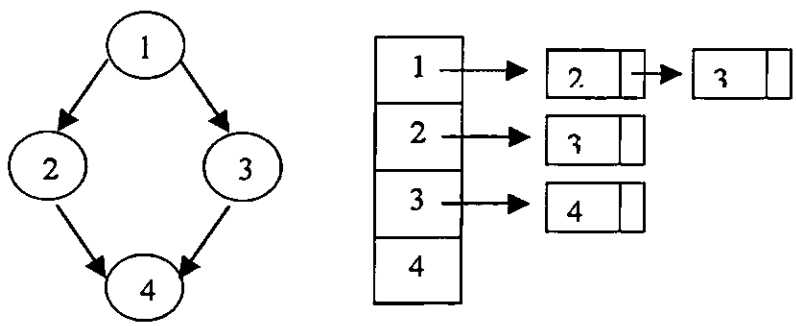
Para encontrar-se todos os vértices adjacentes a n_i é preciso varrer a lista referente a n_i, que deve ter menos elementos que os n que seriam examinados na matriz de adjacências.

A maior dificuldade encontrada nessa representação está relacionada com a necessidade de determinar se um vértice n_j em particular é adjacente ao vértice n_i. Para isso,

será preciso percorrer toda a lista encadeada de n_i.

Como exemplo, a figura 7.5 apresenta o mesmo grafo da figura 7.4 e a respectiva lista de adjacências contendo um vetor de quatro elementos de ponteiros, um para cada vértice. O ponteiro de cada vértice aponta para um vértice adjacente, que aponta para outro vértice adjacente, e assim por diante, até o último nó que não aponta para nenhum nó. O nó 4 não aponta para nenhum outro nó.

Figura 7.5 – Exemplo de Grafo e Lista de Adjacência correspondente.



Na listagem 7.1 pode-se observar as estruturas existentes no protótipo, responsáveis pela representação dos grafos. Inicialmente é definido um ponteiro para um nó (`PonteiroNo`). O nó é um registro que armazena um campo de informação, ou seja, o número da linha que o nó representa e um campo de ligação que é o ponteiro para o próximo nó adjacente. O tipo enumerado `Estado` foi definido para representar o estado em que se encontra um programa em simulação. As opções são: `Exec` para quando o processo estiver em execução, `Send` quando o processo estiver bloqueado para enviar uma mensagem, `Receiv` quando o processo estiver bloqueado aguardando por uma mensagem e `Dead` quando o processo encerrou a sua execução. Cada objeto da *classe de objetos* `TLista` pode manipular uma lista encadeada. Essa classe possui os atributos `Comeco`, que é um ponteiro para o primeiro nó da lista, `Tamanho`, que armazena a quantidade de nós da lista, além de `nBloq`, `nPBloq` e `Tipo`. Cada lista representa um vértice do grafo e, como os nós que compõem a lista representam os nós adjacentes, o atributo `nBloq` é usado para armazenar o estado do nó representado pela lista durante a simulação do programa. O atributo `nPBloq` é utilizado durante um bloqueio para envio ou recebimento de uma mensagem. Ele sinaliza qual processo irá receber a mensagem, quando se tratar de um comando para envio de mensagem, ou de onde a mensagem está chegando, quando se tratar de um comando para recebimento de mensagem. O atributo `Tipo` é um campo auxiliar para desenhar o grafo. Ele representa o tipo

de comando (condicional, repetição, comando simples, etc.), pois cada estrutura proporciona um arranjo visual diferente como foi visto na tabela 6.1, no capítulo anterior. Os métodos declarados nessa classe são utilizados para a inserção, remoção e consulta dos elementos do objeto lista.

Listagem 7.1 – Estrutura da Lista de Adjacências Implementada no Protótipo.

```
type PonteiroNo = ^No;
No = record
        Informacao: integer;
        Ligacao   : PonteiroNo;
      end;
Estado = (Exec, Send, Receiv,
          Dead);
TLista = class
private
 Comeco : PonteiroNo;
 Tamanho: integer;
 Bloq   : Estado;
 nPBloq : integer;
 Tipo   : String;
public
 constructor Inicia;
 destructor Encerra;
 procedure Insere (Dado: integer);
 procedure Retira (Dado: Integer);
 function Consulta(Dado: Integer):
Integer;
 function ConsultaTipo  : String;
 function ConsultaEstado: Estado;
 function ConsultaPBloq: byte;
end;
TGrafo = class
private
 Nos: array of TLista;
 Tamanho: integer;
public
 constructor Inicia (Quant_Nos:
integer);
 function ConsultaPosicao(Posicao:
integer) : Tlista;
 function ConsultaTamanho :
integer;
 procedure MontaGrafo (Programa :
Text);
 destructor Encerra;
end;
```

A classe `TGrafo` define um objeto vetor de `TLista`. O seu construtor `Inicia` é responsável em criar, dinamicamente, o vetor para conter todos os nós do grafo. O método `ConsultaPosicao` retorna um objeto `TLista` do vetor, especificado através do argumento `Posicao`. O método `ConsultaTamanho` retorna a quantidade de nós do grafo. O método `MontaGrafo`, a partir de um arquivo texto (o programa, nesse caso) monta a estrutura do grafo que representa o programa.

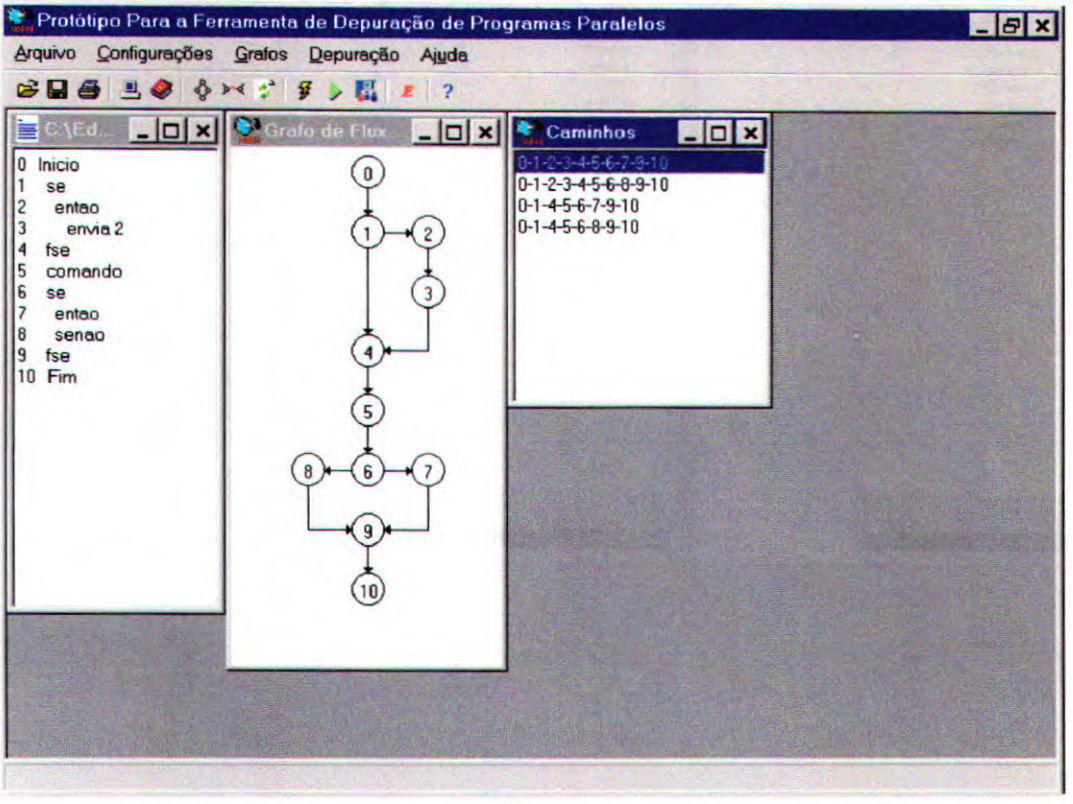
Para se realizar a depuração ou a simulação, é preciso abrir todos os arquivos com os algoritmos que compõe a aplicação que se deseja depurar, gerar os grafos correspondentes e todos os caminhos. A figura 7.7 apresenta um exemplo de aplicação paralela onde uma tarefa realiza um processamento e envia uma mensagem para a segunda tarefa que recebe a mensagem para poder processá-la, tipicamente uma situação do problema clássico denominado produtor-consumidor. Nesse exemplo só existe um único caminho associado a cada tarefa.

7.2.1 – Simulando Um Programa

Após aberto, ou editado um algoritmo, que vai representar uma das tarefas da aplicação paralela, o protótipo pode gerar o grafo e seus caminhos correspondentes (figura

7.6).

Figura 7.6 – Algoritmo Aberto Pelo Protótipo.



Após a abertura de todas as tarefas é possível produzir uma simulação através da criação de uma linha de controle (*thread*) para cada grafo. Cada linha de controle irá colorindo os nós dos seus respectivos grafos seqüencialmente, com um intervalo de tempo correspondente com a estimativa da diferença de performance de cada processador. Quando a linha de controle encontrar um comando para enviar uma mensagem *Send* (ENVIA) ou de recebimento de uma mensagem *Receive* (RECEBE), a sua execução será suspensa e só voltara à nova execução quando o usuário pedir explicitamente para a mensagem ser enviada. Nesse caso a ferramenta verifica a consistência da mensagem, ou seja, se o processo destino realmente está esperando por uma mensagem e em caso positivo libera os dois processos. A figura 7.8 apresenta os dois processos da figura 7.7 bloqueados durante a simulação.

Durante a simulação o usuário poderá verificar a situação de todas as tarefas, inclusive daquelas cujas janelas estão minimizadas, através da opção que informa os *Estados dos Processos*, (Os estados possíveis são: *Exec* – o processo está em execução; *Bloq* – quando o processo estiver bloqueado por um comando de envio ou de recebimento de mensagens e *Morto* – quando a tarefa encerrou a sua execução). A figura 7.9 apresenta os estados das tarefas da figura 7.8 no instante em que aguardam a liberação da mensagem pelo usuário.

Figura 7.7 – Exemplo de Aplicação Paralela Aberta no Protótipo.

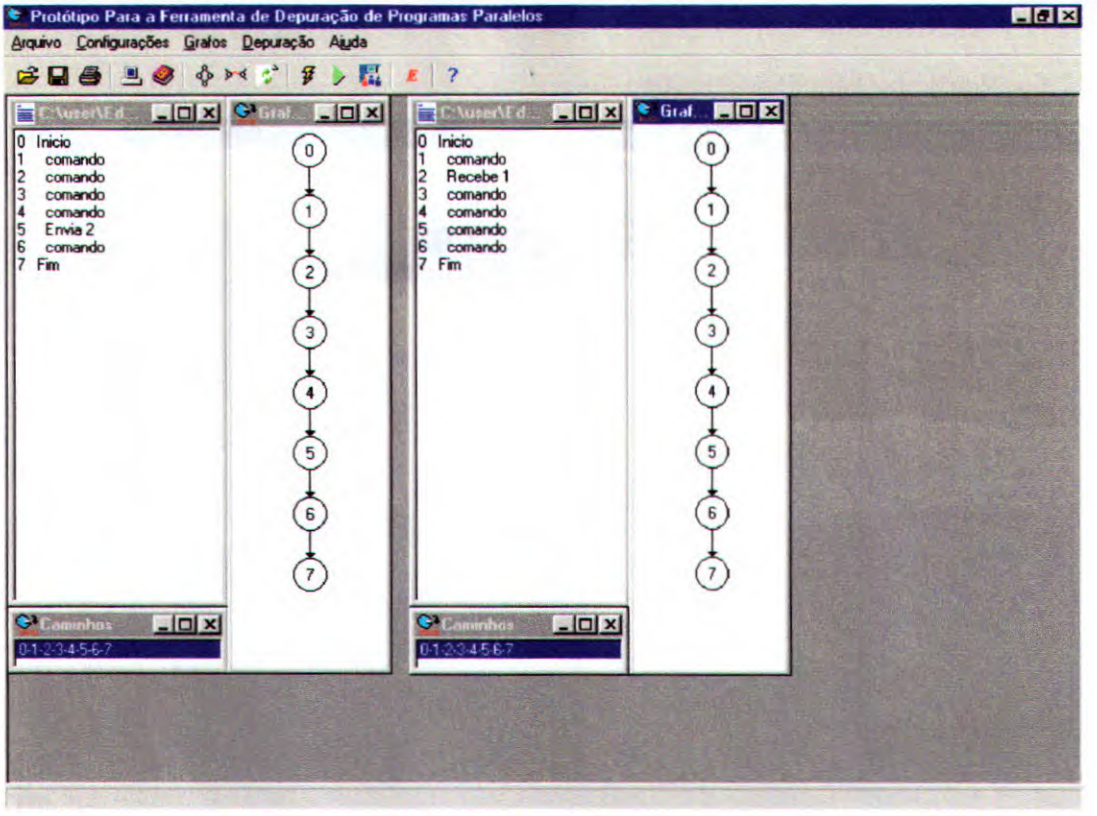
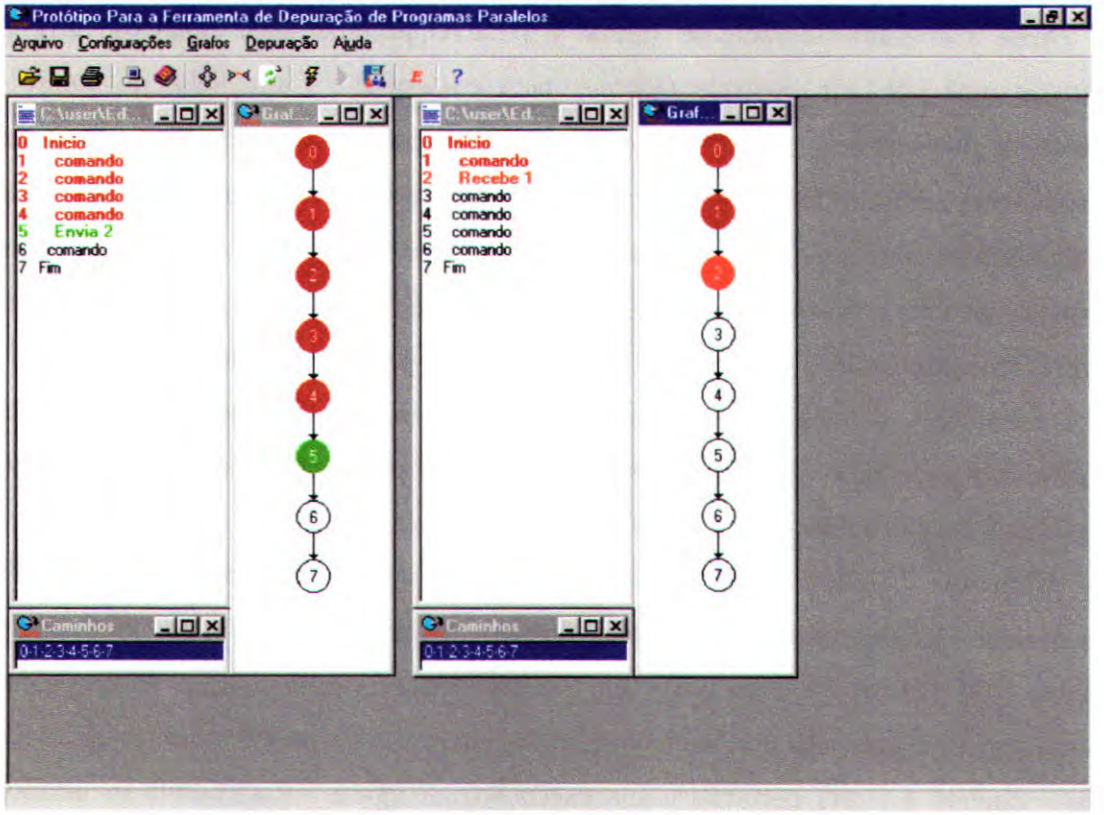
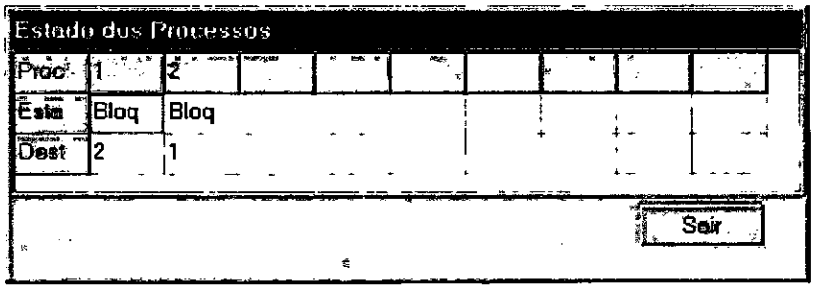


Figura 7.8 – Tarefas Bloqueadas Aguardando Liberação Pelo Usuário.



Quando o grafo gerado possuir vários caminhos o usuário irá selecionar a seqüência que deseja que seja percorrida para cada grafo, após o reconhecimento dos caminhos pela ferramenta.

Figura 7.9 – Estado das Tarefas da Figura 7.8.



Em se tratando da depuração, o procedimento interno é semelhante. Porém, a ferramenta faz a busca nas estruturas dos grafos utilizando todos os caminhos localizados, apresentando o resultado para o usuário.

7.3 – Estrutura Interna do Protótipo

Algumas características desse protótipo merecem especial atenção, pois permitirão o desenvolvimento de uma versão final mais consistente.

7.3.1 - *Threads*

Com o objetivo de facilitar a implementação de aplicações com múltiplos *threads*, o Delphi provê a classe *TThread*. Utilizando essa classe, pode-se criar linhas de controle sem ser necessário o acesso direto às funções para manipulação de *threads* do sistema. Cada objeto *TThread* (ou de alguma classe derivada dela) corresponde a um *thread*.

A classe *TThread* é uma classe abstrata. Uma *Classe Abstrata* é uma classe que não tem instâncias diretas, mas cujas classes descendentes têm instâncias diretas. Uma classe abstrata pode definir o protocolo para uma operação sem fornecer o código correspondente. Essa operação é chamada de *abstrata*. Uma operação abstrata define a forma de uma operação para qual cada subclasse concreta deve providenciar sua própria implementação. A classe *TThread* possui um método abstrato que deve ser sobrescrito em uma classe descendente chamado `Execute`. Esse método é o código principal do *thread* e que será executado durante o ciclo de vida do objeto *thread*.

No protótipo foi definido a classe `ThProcesso` que é descendente direta da classe *TThread*. Os objetos instanciados a partir dessa classe controlam a simulação dos grafos do programa. Essa classe é responsável pela simulação dos grafos colorindo os caminhos que

forem selecionados. Existe um objeto `ThProcesso` para cada tarefa do programa paralelo que está sendo depurado.

A listagem 7.2 apresenta a descrição da classe `ThProcesso`. Os objetos dessa classe são utilizados para simular o comportamento do programa através do grafo. Ela possui dois atributos privados: `No` e `Ant`. `No` representa o vértice atual durante a simulação e `Ant` representa o vértice onde fluxo se encontrava em um passo anterior. O atributo público `Eu` armazena o número do *thread*, ou seja, o número de identificação da tarefa dentro da aplicação paralela, a ferramenta atribui um número único para cada tarefa no instante em que ela é aberta ou criada. Os atributos `frmG`, `frmP`, `frmC` e `frmE` são, respectivamente, ponteiros para as janelas do grafo, do programa, dos caminhos e da janela de estados das tarefas. Com exceção da janela de estado das tarefas, que é uma janela compartilhada por todos os *threads* que fazem parte do processo em depuração, cada *thread* possui as suas próprias janelas para apresentação do grafo, do programa e dos caminhos possíveis. O método `Execute` é relativamente simples. Inicialmente uma variável (`Caminho`) recebe o caminho escolhido pelo usuário. Uma função (`Caminha`) é sempre chamada para retornar o próximo nó a ser percorrido no caminho escolhido. Essa função retorna –1 quando todos os nós já tiverem sido percorridos. Após definido o próximo nó a ser percorrido o método `Desenha` é chamado com o auxílio do método definido na classe *TThread* `Synchronize`, que é responsável pela exclusão mútua no acesso ao vídeo (que é um recurso compartilhado pelos *threads*). O método `Desenha` colore o próximo nó a ser percorrido de azul, em caso de comando que não envolve paralelismo, verde em caso de um comando de envio de mensagem e vermelho em caso de comando que espera por uma mensagem. Além disso, os nós que já foram percorridos são coloridos de marrom, para que o usuário possa perceber o caminho que foi realizado até então. Esse método também colore com as mesmas cores os comandos na janela do programa correspondente. Um procedimento denominado `Clock` é chamado para temporizar a execução criando uma imagem do ciclo de *clock* da CPU. Essa informação é calculada a partir dos dados da arquitetura, podendo ser diferente para cada *thread*, quando se tratar de arquiteturas heterogêneas. O método `Desenha` ainda suspende a execução dos *threads* que estiverem apontando para nós que representam comandos de *envio* e *recebimento* de mensagens. Esses *threads* só voltarão a executar quando o usuário liberar as mensagens, se isso for possível.

7.3.2 – Utilizando Seções Críticas

É importante notar que esse protótipo, por criar linhas de controle concorrentes,

necessita resolver problemas de sincronização e comunicação entre elas, como nos processos que pretende depurar. Dessa forma é necessário utilizar políticas de sincronização. Como o protótipo foi desenvolvido sob a plataforma *Windows*, é necessário utilizar as funções de sua API que são responsáveis pela sincronização de *threads* e processos (disponíveis nas plataformas Win32).

Os mecanismos de sincronização no *Windows* são:

- **Seções Críticas:** as seções críticas só podem ser usadas dentro de um único processo, um único aplicativo.
- **Mutexes:** são objetos globais que podem ser usados para serializar o acesso a um recurso.
- **Semáforos:** são semelhantes aos mutexes, mas são contadores, permitindo mais de um acesso a um dado recurso ao mesmo tempo. Um mutex é equivalente a um semáforo com contagem máxima de 1.
- **Eventos:** podem ser usados como um meio de sincronizar um processo com os eventos do sistema, como nas operações com arquivos.

Listagem 7.2 – Definição da Classe *ThProcesso* Implementada no Protótipo.

```
type
  ThProcesso = class(TThread)
  private
    Ant,
    No : integer;
  protected
    procedure Desenha;
    procedure Execute; override;
  public
    Eu : byte;
    frmG : TfrmGrafo;
    frmP : TfrmProg;
    frmC : TfrmCaminho;
    frmE : TfrmEstado;
  end;
```

```
procedure ThProcesso.Execute;
var Contador : LongInt;
    Caminho : string;
begin
  {obtendo o caminho selecionado}
  if frmC.lbCam.ItemIndex <> -1
then
    Caminho := frmC.lbCam.Items
          [frmC.lbCam.ItemIndex]
  else
    Caminho := '';
  Ant := 0;
  No := Caminha(Caminho);
  while No <> -1 do
  begin
    Synchronize (Desenha);
    Clock;
    Ant := No;
    No := Caminha(Caminho);
  end;
  EnterCriticalSection (Critica);
   PCB[Eu].EstProc := Dead;
   PCB[Eu].Destino := 0;
   frmE.sgPCB.Cells[Eu,1] :=
'Morto';
   frmE.sgPCB.Cells[Eu,2] := '0';
  LeaveCriticalSection (Critica);
end
```

No protótipo desenvolvido, foi utilizada a técnica de seção crítica, pois é necessário controlar o acesso a uma variável compartilhada que informa os estados de todos processos denominada PCB (*Process Control Block*). Os métodos `Desenha` e `Execute` da classe `ThProcesso` utilizam os procedimentos `EnterCriticalSection` e `LeaveCriticalSection` para entrar e sair da seção crítica respectivamente. A declaração da seção crítica (`Critica`) e do vetor `PCB` de estados das tarefas, além do comando de inicialização da variável `Critica` podem ser vistos na listagem 7.3.

Listagem 7.3 – Declaração e Inicialização da Seção Crítica e da Variável PCB.

```
var
 Critica: TRTLCriticalSection;
 PCB : array [1..nProc] of record
   EstProc : Estado;
   Destino : byte;
 end;

...
  {inicializando uma seção
crítica}
   InitializeCriticalSection
(Critica);
...
  {encerrando uma seção crítica}
   DeleteCriticalSection
(Critica);
...
{Inicialização da variável PCB)
var Contador : byte;
begin
   EnterCriticalSection (Critica);
    for Contador := 1 to nProc do
    begin
      PCB[Contador].Estado  :=
Exec;
      PCB[Contador].Destino := 0;
    end;
   LeaveCriticalSection (Critica);
end;
```

7.4 – Considerações Finais

O objetivo desse capítulo não foi criar um texto sobre programação; assim, não foram discutidos problemas da codificação em detalhes. O objetivo foi discutir a *interface* da ferramenta de auxílio na depuração de programas paralelos e levantar algumas considerações sobre a implementação da ferramenta.

De forma resumida, os benefícios na construção de um protótipo podem ser definidos como sendo:

- Mostrar claramente os objetivos e funcionalidades do ambiente proposto;
- Desenvolver alternativas para os problemas e detalhes relativos a implementação da ferramenta que surgem no desenvolvimento do protótipo, para que, na versão final essas situações possam ser mais facilmente resolvidas.

Na implementação da versão final os problemas de implementação necessitarão de uma análise mais refinada. Por exemplo, foi criada uma versão relativamente simples do algoritmo para geração dos caminhos em um grafo. Na versão final será necessário um estudo da eficiência dos algoritmos existentes, para a escolha do método mais adequado.

Antes de uma implementação final da ferramenta ainda é necessário explorar o potencial do protótipo colocando-o a disposição de alunos de programação paralela, para uma análise da *interface* e de sua funcionalidade. Esse passo é o cerne da abordagem de prototipação, uma vez que o usuário poderá examinar uma representação real do projeto e sugerir modificações que façam com que esta ferramenta atenda melhor as necessidades reais.

Conclusões e Trabalhos Futuros

Começar um trabalho é geralmente uma tarefa fácil. Chegar ao seu término, porém, exige um esforço maior. No início são as dúvidas, os questionamentos, a busca do melhor caminho. E para concluir o trabalho é necessário, antes de tudo, reconhecer que ele chegou ao fim. Nesse momento é deve-se avaliar o que foi feito, quais foram as contribuições e o que ainda é possível desenvolver em trabalhos futuros.

Esse projeto enfocou duas áreas relevantes da programação paralela: o seu aprendizado e a depuração de programas paralelos.

A depuração de programas, como foi exposto durante toda a dissertação, é uma tarefa árdua e é importante abrir fronteiras para que as soluções possam ir aos poucos se tornando mais simples e mais presentes nas ferramentas de desenvolvimento de *software* paralelo.

A aprendizagem está presente em todas as áreas do conhecimento humano. Por mais técnica ou sofisticada que possa ser um ramo do conhecimento, são necessárias o desenvolvimento de metodologias de aprendizagem. Durante toda a execução deste trabalho estava implícito que o *saber pensar* se fortalece a partir do *saber fazer*. Uma ferramenta de depuração aliada a uma metodologia de aprendizagem permite aprimorar o senso crítico do usuário, criando oportunidades de questionamento e, conseqüentemente, aumentando o processo de ensino-aprendizagem.

A unificação dessas duas áreas, ou seja, a depuração de programas paralelos com a aprendizagem de programação paralela é a maior contribuição deste trabalho.

Além disso, o desenvolvimento do protótipo proporcionou, além do objetivo inicial, que era o projeto de um ambiente para depuração de programas paralelos e a utilização dos

erros encontrados na aprendizagem dos usuários inexperientes, uma vantagem não esperada: a possibilidade de utilizar o próprio protótipo como ferramenta de aprendizagem de programação paralela, independentemente da necessidade de se depurar um programa, através da demonstração dos problemas e dificuldades encontradas na programação e do comportamento de várias tarefas sendo executadas em paralelo.

Dessa forma, as contribuições deste trabalho podem ser resumidas como sendo:

- O projeto de um ambiente de depuração paralela que permite ao usuário verificar o comportamento das tarefas durante a depuração, através da visualização dos caminhos que o programa pode percorrer durante a execução real do programa.
- Especificação de um ambiente amigável ao usuário iniciante, que permite motivar um número maior de usuários para a utilização da programação paralela.
- A independência de plataforma na especificação do projeto, o que permite a implementação de versões para vários ambientes de troca de mensagem.
- A possibilidade de se depurar o código do programa sem a necessidade da plataforma em que o programa foi implementado, facilitando a aprendizagem de um grande número de linguagens e bibliotecas paralelas, permitindo que cursos de programação concorrente usem a ferramenta como instrumento de aprendizagem.
- A flexibilidade de se conectar esta ferramenta com outras ferramentas de auxílio no desenvolvimento de programas paralelos [Calonêgo, 1997] [Branco, 1999], estendendo o seu uso e aplicabilidade.
- A possibilidade de servir como base para novas pesquisas sobre o tema: depuração e testes de programas paralelos no grupo de Programação Concorrente e Sistemas Distribuídos.

Após o término deste projeto foi possível definir vários outros trabalhos como, por exemplo:

- **Ambiente Para Testes de Programas Paralelos:** testar e depurar programas são tarefas que estão intimamente relacionadas. A ferramenta possui flexibilidade para se testar programas paralelos. Entretanto, desenvolver os casos de teste para colocá-los à prova e avaliar os resultados apresenta alguns problemas que precisam ser resolvidos. Mesmo para pequenos programas, o número de caminhos lógicos possíveis pode ser muito grande.
- **Ambiente Para Simulação de Programas Paralelos:** o protótipo desenvolvido neste trabalho permite simular os algoritmos paralelos. O que aconteceria se as tarefas possuíssem 100 linhas de código (tarefas pequenas), ou 500 ou 1000...? É

necessário uma avaliação da melhor maneira de simular tarefas cujos grafos correspondentes dificultem a visualização. É importante fazer um estudo de como criar supergrafos e como eles seriam visualizados pelos usuários.

- **Paralelizador Para Programas de Granulosidade Grossa:** As ferramentas de paralelização automática são geralmente destinadas a tarefas de granulosidade fina. A possibilidade de verificar o grau de paralelismo alcançado pelo usuário depende de formas de paralelização de procedimentos ou de tarefas inteiras.
- **Criação de Versões Para Linguagens Paralelas Específicas:** Agora que o projeto está pronto é preciso implementar versões para linguagens existentes. Quando isto for realizado, certamente irão surgir problemas específicos de cada linguagem que precisarão ser sanados.
- **Comparação de Performance com os Depuradores Existentes:** Apesar do objetivo desta ferramenta ser os usuários inexperientes, que geralmente desenvolvem projetos pequenos, é interessante verificar a extensibilidade desta ferramenta com a utilização de programas que produzam grafos com grande quantidade de nós.
- **Tradutor Para Linguagens Paralelas:** Após a implementação de várias versões, será possível converter um programa desenvolvido em uma linguagem ou biblioteca de troca de mensagens, para outra com características semelhantes, como por exemplo, PVM e MPI.
- **Criação de Uma Versão Distribuída:** Depurar grandes programas paralelos é uma tarefa quase impossível. Muitos caminhos que os programas podem percorrer são descartados nos casos de testes. Uma possibilidade para se estender esta ferramenta será transformar essa versão seqüencial em uma versão de depuração distribuída. Para isso é preciso uma análise de toda a problemática envolvendo a paralelização do depurador.

Certamente outros trabalhos surgirão além desses.

Referências Bibliográficas

[Aho, et. al., 1995] AHO, A.V., SETHI, R. e ULLMAN, J.D. *Compiladores Princípios, Técnicas e Ferramentas.* Rio de Janeiro : Livros Técnicos e Científicos, 1995.

[Almasi e Gottlieb, 1994] ALMASI, G. S. e GOTTLIEB, A. *Highly Parallel Computing.* 2. ed. Redwood City : The Benjamin Cummings, 1994.

[Axford, 1989] AXFORD, T. *Concurrent Programming – Fundamental Tecniques for Real-Time and Parallel Software Design.* Chichester : John Wiley & Sons, 1989.

[Baiardi et al., 1986] BAIARDI, F.B., NICOLETTA, F., VAGLINI, G. Development of a Debugger for a Concurrent Language. *IEEE Transactions On Software Engineering*, v. 12, n. 4, p. 547-553, April, 1986.

[Ben-Ari, 1990] BEN-ARI, M. *Principles of Concurrent and Distributed Programming.* New York : British Library Cataloguing in Publication Data, 1990.

[Branco, 1999] BRANCO, K.R.L.J.C. *Extensão da Ferramenta de Apoio a Programação Paralela (F.A.P.P.) para Ambientes Paralelos Virtuais.* São Carlos, 1999. Dissertação (Mestrado em Ciência da Computação e Matemática Computacional) - Instituto de Ciências Matemáticas e de Computação – Universidade de São Paulo.

[Brawer, 1989] BRAWER, S. *Introduction to Parallel Programming.* Boston : Academic Press, 1989.

[Browne et al., 1995] BROWNE, J.C., HYDER, S.I., DONGARRA, J., MOORE, K., NEWTON, P. Visual Programming and Debugging for Parallel Computing. *IEEE Parallel & Distributed Technology*, p. 75-83, 1995.

[Calônego, 1997] CALÔNEGO JR., N. *Uma Abordagem Orientada a Objetos de Uma Ferramenta de Auxílio a Programação Paralela,* Tese (Doutorado), Instituto de Física e Química da Universidade de São Paulo (USP), Outubro, 1997.

[Cantú, 1998] CANTÚ, M. *Dominando o Delphi 4.* São Paulo : Makron Books, 1998.

[Carvalho, 1998] CARVALHO, F. F. *Programação Orientada a Objetos Usando o Delphi 3.* São Paulo : Érica, 1998.

[Cheung et al., 1990] CHEUNG, W.H., BLACK, J.P., MANNING, E. A Framework for Distributed Debugging. *IEEE Software*, p. 106-115, Jan., 1990.

[Chi, 1985] CHI, U.H. Formal Specification of User Interfaces: A Comparison and Evaluation of Four Axiomatic Approaches. *IEEE Transactions on Software Engineering*, v. SE-11, n. 8, p. 671-685, Aug., 1985.

[Craveiro, 1998] CRAVEIRO, G.S. *Detecção de Propriedades Estáveis em Sistemas Distribuídos.* Campinas, 1998. Dissertação (Mestrado em Computação) – Instituto de Computação, Universidade Estadual de Campinas.

[Damitio e Turner, 1995] DAMITIO, M., TURNER, S. Debugging Concurrent Programs With Dynamic Slicing and Backtracking. *EPSRC PPECC Workshop 'Distributed vs Parallel: convergence or divergence'*, April, 1995, *Proceedings*, p. 45-49.

[Davis, 1994] DAVIS, W.S. *Análise e Projeto de Sistemas: Uma Abordagem Estruturada.* Rio de Janeiro : Livros Técnicos e Científicos, 1994.

[Duncan, 1990] DUNCAN, R. A Survey of Parallel Computer Architectures. *IEEE Computer*, p. 5-16, Feb., 1990.

[Flynn, 1972] FLYNN, M.J. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, v. C-21, p. 948-960, 1972.

[Freire, 1998] FREIRE, M.E.P. *O Sistema Tutor de um Ambiente Inteligente para Treinamento e Ensino.* São Carlos, 1998. Dissertação (Mestrado em Ciência da Computação e Matemática Computacional) - Instituto de Ciências Matemáticas e de Computação – Universidade de São Paulo.

[Furtado, 1973] FURTADO, A.L. *Teoria dos Grafos: Algoritmos.* Rio de Janeiro : Livros Técnicos e Científicos, 1973.

[Gane, 1988] GANE, C. *Desenvolvimento Rápido de Sistemas.* Rio de Janeiro : Livros Técnicos e Científicos, 1988.

[Gane e Sarson, 1983] GANE, C. e SARSON, T. *Análise Estruturada de Sistemas.* Rio de Janeiro : Livros Técnicos e Científicos, 1983.

[Gersting, 1993] GERSTING, J.L. *Fundamentos Matemáticos Para a Ciência da Computação.* 3ª. ed., Rio de Janeiro : Livros Técnicos e Científicos, 1983.

[Gupta et al., 1998] GUPTA, N., MATHUR, A.P., SOFFA, M. L. Automated Test Data Generation Using An Iterative Relaxation Method. In: *ACM SIGSOFT Sixth International Symposium on Foundations of Software Engineering*, Orlando, 1998, *Proceedings*, p. 231-244.

[Habermann, 1991] HABERMANN, F. Giving Real Meaning to 'Easy-to-use' Interfaces. *IEEE Software*, v. 8, n. 4, p. 90-91, July, 1991.

[Machado e Abreu, 1995] MACHADO, F.N.R., ABREU, M. *Projeto de Banco de Dados: Un*
Visão Prática. São Paulo : Érica, 1995.

[McDowell e Helmbold, 1989] MCDOWELL, C.E., HELMBOLD, D.P. Debugging Concurr
Programs. *ACM Computing Surveys*, v. 21, n. 4, p. 593-622, Dec., 1989.

[Norton e Mueller] NORTON, P., MUELLER, J. P. *Perter Norton: Guia Para o Delphi 2,* S
Paulo : Makon Books, 1997.

[Pimentel e Direne, 1998] PIMENTEL, A.R., DIRENE, A.I. *Medidas Cognitivas no Ensin*
de Programação de Computadores com Sistemas Tutores Inteligentes. [online].
[10/04/1999]. Disponível na Internet:
http://janus.inf.ufsc.br:1998/materiais/sbie98/anais/artigos/art36.html.

[Pressman, 1992] PRESSMAN, R.S. *Software Engineering: A Practitioner's Approach*. 3.
London : McGraw-Hill, 1992.

[Seleções, 1975] História do Homem: Nos Últimos Dois Milhões de Anos. *In: Seleções*
Reader's Digest, 1975.

[Shay, 1996] SHAY, W.A. *Sistemas Operacionais*. São Paulo : Makron Books, 1996.

[Stewart e Gentleman, 1997] STEWART, D.A., GENTLEMAN, W.N. Non-Stop Monitoring a
Debugging on Shared-Memory Multiprocessors. *IEEE Computer Society*, p. 263-26
May, 1997.

[Tanenbaum, 1992] TANENBAUM, A.S. *Modern Operation System*. Engewood Cliff
Prentice-Hall, 1992.