UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE POS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Um Ambiente Para Exploração de Paralelismo na Programação em Lógica

por

Adenauer Corrêa Yamin

Dissertação submetida como requisito parcial para
a obtenção do grau de
Mestre em Ciência da Computação

Prof. Cláudio Fernando R. Geyer
Orientador

Prof. Philippe Olivier A. Navaux
Co-orientador

Porto Alegre, janeiro de 1994

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Yamim, Adenauer Corrêa

Um Ambiente Para Exploração de Paralelismo na Programação em Lógica / Adenauer Corrêa Yamim.- Porto Alegre: CPGCC da UFRGS, 1994.

204p.: il.

Dissertação (mestrado)-Universidade Federal do Rio Grande do Sul, Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, 1994. Orientador: Geyer, Cláudio F.R.. Co-orientador: Navaux, Philippe O.A.

Dissertação: Processamento Paralelo, Prolog Paralelo, Paralelismo E, Paralelismo OU, Arquitetura de Computadores

Seis homens honestos me servem. Eles me ensinaram tudo que sei. Seus nomes são: *O Que, Por Que, Quando, Como, Onde* e *Quem.*

- RUDYARD KIPLING

AGRADECIMENTOS

Gostaria de agradecer aos professores Cláudio Geyer e Philippe Navaux, pelo estímulo, pela intensa participação nesta fase da minha formação, e pela orientação dedicada a todos os trabalhos.

Aos auxiliares de pesquisa do projeto OPERA; Patrícia, Vinícius, Marcos e Roberto, pela inestimável colaboração.

Aos colegas de mestrado, todos agora tão especiais para mim, pelas amplas e inúmeras discussões técnicas, por todo coleguismo que tiveram (a lista de nomes seria imensa!).

À colega e parceira de OPERA Otilia, pela paciência e grande dedicação, nas inúmeras vezes que avaliamos alternativas para nossos trabalhos, por todo companherismo.

Aos professores e funcionários do CPGCC, pelos ensinamentos e receptividade, que tanto contribuíram para os meus estudos.

A toda minha família pelo apoio, em especial a Paula, esposa e companheira, que mais uma vez acreditou em mim, e cujo estímulo e dedicação viabilizaram este trabalho.

Às Universidades Católica e Federal de Pelotas, em particular aos colegas do Centro de Ciências Exatas e Centro de informática, por toda colaboração.

À CAPES pelo apoio financeiro, sem o qual também este trabalho não teria acontecido.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

ACT	Tabela Característica da Aplicação (Application Characteristic Table)
CGE	Conditional Graph Expression
ECP	Estrutura de Chamadas Paralelas
GS	Goal Stack Pointer
IGM	Input Goal Marker
IWS	Pilha de Trabalhadores Desocupados (Idle Workers Stack)
LGM	Local Goal Marker
MAP	Máquina Abstrata Prolog
OWS	Pilha de Trabalhadores Sobrecarregados (Overloaded Worker Stack)
PDL	Push Down List
PF	Parcall Frame Pointer
PIP	Put Instructions Pointer
POP	Pilha de Objetivos Paralelos
RAP	Restricted AND Parallelism
SRT	Tabela de Recursos do Sistema (System Resource Table)
TWAM	Transputer Warren Abstract Machine
WAM	Warren Abstract Machine
WIT	Tabela de Identificação de Trabalhadores (Workers Indentification Table)
WM	Wait Marker

RESUMO

Este trabalho é dedicado ao estudo da exploração de paralelismo na Programação em Lógica. O aspecto declarativo das linguagens de Programação em Lógica permite uma exploração eficiente do paralelismo implícito no código, de forma mais simples que as linguagens imperativas. Ao mesmo tempo, o paralelismo tem-se mostrado uma forte opção para procura de aumentos significativos do desempenho dos computadores. Como conseqüência, nos últimos anos, diversas máquinas paralelas tem surgido no mercado. No entanto, a sua efetiva utilização ainda ressente-se de uma dificuldade de programação maior que a das máquinas seqüenciais.

Por outro lado, o alto nível das linguagens de Programação em Lógica permite o desenvolvimento de programas de forma mais rápida e concisa do que as linguagens tradicionais (imperativas). Porém, apesar dos importantes progressos nas técnicas de compilação destas linguagens, elas permanecem menos eficientes que as linguagens imperativas. O aumento na eficiência de execução da Programação em Lógica, com o uso do paralelismo, certamente estenderá o seu emprego.

Em função disto, a união da Programação em Lógica e máquinas paralelas tem sido proposta como uma alternativa para facilitar a programação das máquinas paralelas, bem como para aumentar o desempenho na Programação em Lógica.

O ponto central do trabalho é a concepção de um modelo para exploração do paralelismo E Restrito na execução de Prolog, voltado para arquiteturas multiprocessadoras sem memória comum. Como ponto de partida foi utilizado o modelo já definido para exploração do paralelismo OU do projeto OPERA, do Instituto de Informática da UFRGS, de maneira que o

modelo de paralelismo E proposto possa vir a compor, com aquele, uma plataforma que integre a exploração simultânea dos paralelismos E e OU.

O modelo concebido compreende uma proposta de compilação e um ambiente de execução. A detecção e o controle do paralelismo é iniciado na compilação. Nesta fase, é gerada uma Expressão Condicional de Execução para cada cláusula do programa Prolog, cuja avaliação em tempo de processamento determina a execução, em paralelo ou não, dos literais que compõem a cláusula.

A Máquina Abstrata Prolog, projetada para o emulador paralelo, é baseada na WAM (*Warren Abstract Machine*), uma das mais eficientes e difundidas técnicas para compilação Prolog. Isto, dentre outros aspectos, confere uma boa portabilidade ao modelo.

O ambiente de execução compreende a concepção de uma arquitetura de processos formada por trabalhadores OPERA, uma filosofia de escalonamento de serviço entre estes trabalhadores, uma política para gerência de sua memória e uma estratégia para as comunicações.

Para validar o modelo proposto para exploração do paralelismo E, o mesmo foi implementado em rede local de estações Unix, obtendo bons resultados.

PALAVRAS-CHAVE: Processamento Paralelo, Paralelismo na Programação em Lógica, Paralelismo E, Paralelismo OU, Arquitetura de Computadores.

TITLE: "A ENVIRONMENT TO EXPLOTATION OF PARALLELISM IN THE LOGIC PROGRAMMING"

ABSTRACT

This work is devoted to the study of the exploration of parallelism in Logic Programming. The declarative aspect of the Logic Programming languages allows an efficient exploration of the implicit parallelism in the code, in a simpler form than the imperative languages. At the same time, parallelism has been shown as a strong option to the search for significant increases in the performance of the computers. As a consequence, in the last years, several parallel machines have been sprung up into the market. Nevertheless, their effective usefulness still undergoes some difficulties in programming which are greater than those of the sequential machines.

On the other hand, the high level of Logic Programming languages allows programs development to be faster and concise than in the traditional languages (imperatives). However, despite the important progress in compiling techniques for these languages, they remain less efficient than the imperatives languages. The increase in execution efficiency of logic programs, with the use of parallelism, will probabily extend their use.

Having this in mind, the union of the Logic Programming and parallel machines has been proposed as an alternative to make programming of the parallel machines easier, as well as to increase the performance of Logic Programming.

The central aspect of the work is the conception of a model to explore the Restricted AND Parallelism in the execution of Prolog, turned to multiprocessing architectures without a common memory. As a starting point, the already defined model for exploring OR parallelism of the OPERA project, from the Instituto de Informática da UFRGS was used. This

happened so that the proposed model of AND parallelism can make up a plataform with that one to integrate the simultaneous exploration of the AND and OR parallelisms.

The conceived model holds a proposal of compilation and execution environment. The detection and the control of the parallelism is started in the compilation. A Conditional Expression of Execution to each clause of the Prolog program is generated on this phase. Its evaluation, during the time of processing, determines the execution, whether or not in parallel, of the literals that constitute the clause.

The Abstract Prolog Machine, projected for the parallel emulator, is based on the WAM (Warren Abstract Machine) which is one of the most efficient and spread techniques for Prolog compilation. This aspects, among others, gives a good portability to the model.

The environmente of execution comprises the conception of an architecture of processes formed by OPERA workers and a philosophy of scheduling service among these workers; it also comprise a policy to manage its memory and a strategy for the communications.

So that the proposed model for the exploitation of AND parallelism got validated, it was implemented on a local net of Unix workstations, obtaining good results.

KEYWORDS: Parallel Processing, Parallelism in Logic Programming, AND Parallelism, OR Parallelism, Parallel Prolog, Computers Architecture.

1 INTRODUÇÃO

1.1 O Tema

O tema desta dissertação é o estudo e a avaliação de uma implementação da linguagem Prolog em arquiteturas paralelas. Neste sentido, é proposto um modelo para exploração do paralelismo E implícito na linguagem, voltado para multiprocessadores sem memória comum.

1.2 Motivação

Como resposta aos avanços da tecnologia em microeletrônica (VLSI), tem surgido comercialmente computadores com capacidade de processamento paralelo. Rapidamente este tipo de equipamento está se tornando uma das fontes de poder computacional com melhor relação custo/benefício. Porém, esta tecnologia exige que o *software* seja capaz de canalizar o paralelismo inerente a cada problema, de forma adequada à organização da máquina paralela a ser empregada, e que deste modo aproveite, de forma eficiente, todos recursos (processadores) que esta oferece.

Existem duas grandes propostas para obtenção deste tipo de *software*:

- o uso de indicações explícitas por parte do programador, anotando o possível paralelismo existente no programa. Para esta proposta, devem ser empregadas linguagens que

disponham de construções paralelas, ou ambientes de programação paralela, isto é, linguagens seqüenciais agregadadas à bibliotecas de primitivas para construções paralelas.

- o emprego de compiladores especiais que, em conjunto com o ambiente de execução, realizem uma exploração automática do paralelismo existente no problema. Normalmente, neste segundo caso também é dada opção ao programador de especificar a execução paralela ou não de trechos do programa.

A primeira proposta deixa ao encargo do programador determinar as dependências entre as diferentes partes do programa, seu seqüenciamento e sincronização. Esta tarefa, muito delicada e sujeita a inúmeros erros, provoca um incremento na complexidade de algo que intrinsecamente é complexo: o projeto e a depuração de programas.

A segunda proposta, ao contrário, libera o programador destas tarefas. Por outro lado o projeto do conjunto compilador/ambiente de execução, que consiga explorar, de forma automática e eficiente, toda possibilidade de paralelismo, é bastante complexo, sobretudo quando voltados para linguagens de programação com semântica imperativa. Nestas linguagens o programador especifica **como** deve ser resolvido o problema, determinando completamente a seqüência de instruções.

A dificuldade de se obter sistemas com exploração do paralelismo implícito na linguagem é reduzida quando a programação é feita com linguagens declarativas. Estas linguagens especificam (principalmente) **o que** deve ser feito para resolver o problema, sem impor uma ordem

particular de execução, e isto possibilita melhor eficiência na detecção e na gerência automática do paralelismo.

Por sua vez, as linguagens de Programação em Lógica, inseridas no escopo das linguagens declarativas, são muito oportunas para construir aplicações em certos domínios, dentre estes: tratamento de linguagens formais ou naturais, expressão de grandes problemas combinatórios e construção de sistemas especialistas. Uma característica destas aplicações é a exigência de muito processamento.

A linguagem Prolog, inclusa entre as linguagens de Programação em Lógica, caracteriza-se por apresentar um elevado poder de expressão ([KOW 74]), porém aliado a uma ineficiência de execução. Por outro lado, ela possui diversas fontes de paralelismo implícito ([HER 90]).

Assim, a união de Pogramação em Lógica e arquiteturas paralelas tem sido proposta com dois objetivos:

- ser uma alternativa para simplificar a programação de máquinas paralelas;
- aumentar o desempenho da programação em lógica.

1.3 O Contexto Local

O projeto OPERA, do Instituto de Informática da UFRGS, tem por objetivo investigar de forma ampla o tema paralelismo na Programação em Lógica.

O projeto já conta com as especificações de um modelo que explora o paralelismo OU na execução de programas Prolog [GEY 91]. A arquitetura destino deste modelo é constituída por multiprocessadores com memória distribuída.

O projeto OPERA, quando da definição dos trabalhos desta dissertação contemplava as seguintes frentes de trabalho:

- a proposta de um escalonador mais preciso (distribuído ou hierárquico) para o modelo do paralelismo OU Multiseqüencial;

- o projeto, ainda em fase inicial, de um novo compilador, que incorporasse as técnicas mais recentes para compilação Prolog. Sendo um novo produto, permitiria que novos requisitos de compilação, surgidos a partir dos estudos relativos ao Prolog, sejam melhor atendidos;

- estudos sobre avaliação automática de grão para tarefas a serem paralelizadas. Um trabalho contemplaria o estudo abrangente de várias propostas e técnicas, enquanto outro investigaria aspectos de implementação a partir de uma proposta específica;

- a definição de um modelo para exploração do paralelismo E na Programação em Lógica;

- a implementação de um primeiro protótipo para exploração do paralelismo E na Programação em Lógica, voltado para arquiteturas paralelas sem memória comum;

- o projeto, ainda em fase inicial, de um modelo que explorasse de forma integrada os paralelismo E e OU na Programação em Lógica.

1.4 A Contribuição do Autor

Considerando os interesses de pesquisa do projeto, o autor concentrou sua contribuição em três objetivos:

- participou na definição do modelo para exploração do paralelismo E do projeto OPERA. Este trabalho englobou: a definição de uma técnica de compilação, a definição da respectiva Máquina Abstrata Prolog e a proposta de um modelo computacional para suporte à execução paralela. Todas as componentes foram projetadas considerando uma arquitetura paralela, cuja comunicação entre processadores é efetuada por troca de mensagens. Tendo em vista uma futura integração, estas definições para exploração do paralelismo E foram feitas considerando as já existentes para o paralelismo OU.

- definiu o ambiente de execução para o modelo de paralelismo E do projeto OPERA. Este ambiente compreende a arquitetura paralela de trabalhadores OPERA, a política de escalonamento para esta arquitetura, a gerência de memória dos trabalhadores OPERA, e a estratégia de comunicação entre os processos que formam o trabalhador OPERA (comunicação local), e entre estes trabalhadores (comunicação remota).

- trabalhou na implementação da proposta para exploração do paralelismo E do projeto OPERA, em rede local de estações Unix.

1.5 A Estrutura da Dissertação

A partir dos fundamentos da Programação em Lógica e das possibilidades que esta oferece para exploração do paralelismo, é apresentada uma ampla revisão dos modelos que integram estas duas propostas. Circunstanciado por esta revisão crítica, é apresentado o modelo proposto para exploração do paralelismo E. A descrição deste modelo é enriquecida pela discussão dos aspectos pertinentes a sua implementação.

A dissertação está dividida em duas partes:

A **parte 1- O Contexto** (capítulos 2 e 3) é uma introdução à Programação em Lógica e uma análise das alternativas que esta oferece para exploração do paralelismo.

No capítulo 2, é feita uma breve introdução aos conceitos da Programação em Lógica. O Prolog é apresentado como um exemplo de linguagem de Programação em Lógica.

No capítulo 3, são analisadas as fontes de paralelismo na Programação em Lógica, e as estratégias para sua exploração. Neste capítulo, é feita uma comparação crítica dos principais trabalhos na área.

A **parte 2 - A Proposta** (capítulos 4, 5 e 6) registra a contribuição central do autor. Nesta parte, é apresentado o modelo proposto

para exploração do paralelismo E, sua implementação e as conclusões do trabalho.

No capítulo 4, são descritas as estratégias já definidas para exploração do paralelismo OU, bem como as especificações do modelo para exploração do paralelismo E na Programação em Lógica proposto para o projeto OPERA, objeto central da dissertação.

No capítulo 5, são relatados os aspectos inerentes à implementação do protótipo do modelo proposto no capítulo 4, em rede local de estações Unix. Destacam-se neste capítulo, as considerações para operação em rede local, as ferramentas para comunicação com o usuário e a análise de desempenho do protótipo.

O capítulo 6, por sua vez, registra as conclusões do trabalho.

O anexo A-1, contém o manual de operação do protótipo, com exemplos de resultados.

O anexo A-2 apresenta o Prolog OPERA, e faz um paralelo com a sintaxe do C-Prolog.

O anexo A-3 contém a listagem dos programas dos quais foi feita a análise de desempenho no capítulo 5

2 PROGRAMAÇÃO EM LÓGICA E PROLOG

Este capítulo é uma breve revisão dos aspectos básicos da Programação em Lógica, da linguagem Prolog e dos fundamentos de uma das técnicas mais difundidas para compilação Prolog, a máquina abstrata de Warren ([WAR 83]). O seu objetivo é introduzir conceitos que serão empregados nos capítulos seguintes.

O estudo destes fundamentos, no âmbito deste trabalho de dissertação, teve como objetivo embasar a avaliação crítica das diferentes alternativas para exploração do paralelismo na Programação em Lógica.

Para um estudo dos conceitos básicos da Programação em Lógica (Lógica Computacional) é sugerido o clássico de Kowalski [KOW74]. Para uma abordagem, que englobe aspectos teóricos e de implementação da Programação em Lógica, o livro de Hogger ([HOG 84]) se mostra interessante. Os livros de Clocksin & Mellish ([CLO 81]) e Sterling & Shapiro ([STE 86]) são oportunos como tutoriais para programação Prolog.

2.1 Programação em Lógica

2.1.1 INTRODUÇÃO

A Programação em Lógica originou-se em grande parte na pesquisa sobre prova automática de teoremas, particularmente no desenvolvimento do *Princípio da Resolução* por Robinson (1965). Um dos primeiros trabalhos relacionando o Princípio da Resolução com a

programação de computadores deve-se a Green (1969), que mostrou que o mecanismo para a extração de respostas em sistemas de resolução poderia ser empregado para sintetizar programas convencionais.

O termo "Programação em Lógica" é devido a Kowalski (1974), e designa o uso da lógica como base teórica para linguagens de programação de computadores. Kowalski identificou, em um particular procedimento de prova de teoremas, um procedimento computacional, permitindo uma interpretação procedimental da lógica e estabelecendo as condições que permitem entendê-la como uma linguagem de programação de uso geral. Esse foi um avanço essencial, necessário para adaptar os conceitos relacionados com a prova de teoremas às técnicas computacionais já dominadas pelos programadores. Avanços nas técnicas de implementação também foram de grande importância para o emprego da lógica como linguagem de programação.

Uma das principais idéias da Programação em Lógica é que um algoritmo é constituído por dois elementos disjuntos: a *lógica* e o *controle*. O componente lógico corresponde à definição do que deve ser solucionado, enquanto que o componente de controle estabelece como a solução pode ser obtida. O programador precisa somente descrever o componente lógico de um algoritmo, deixando o controle da execução para ser exercido pelo sistema de Programação em Lógica utilizado. Em outras palavras, a tarefa do programador passa a ser simplesmente a **especificação** do problema que deve ser solucionado, razão pela qual as linguagens lógicas podem ser vistas simultaneamente como linguagens para especificação formal e linguagens para a programação de computadores.

O paradigma fundamental da Programação em Lógica é o da programação declarativa, em oposição à programação procedimental típica

das linguagens convencionais. A programação declarativa engloba, também, a programação funcional, cujo exemplo mais conhecido é a linguagem Lisp. A programação funcional é um estilo empregado há bastante tempo; a linguagem Lisp, por exemplo, data de 1960. A Programação em Lógica, por sua vez, só ganhou ímpeto a partir dos anos 80, quando foi escolhida como a linguagem básica do Projeto Japonês de Computadores de Quinta Geração ([MOT 81]). O ponto focal da Programação em Lógica consiste em identificar a noção de computação com a noção de dedução. Mais precisamente, os sistemas de Programação em Lógica reduzem a execução de programas à pesquisa da refutação das sentenças do programa em conjunto com a negação da sentença que expressa a consulta.

2.1.2 FUNDAMENTOS

As estruturas de dados básicas da Programação em Lógica são os **termos**. Um termo é uma constante, uma variável, ou um termo funcional. As constantes denotam indivíduos particulares, tais como números e átomos, enquanto que as variáveis denotam um indivíduo ainda não especificado. O símbolo correspondente a um átomo pode ser qualquer seqüência de caracteres, a qual pode ser incluída entre apóstrofos (') caso haja possibilidade de confusão com outros símbolos, como variáveis ou números.

Um **termo funcional** é formado por um functor e por uma seqüência de um ou mais termos denominados *argumentos*. Um **functor** é caracterizado pelo seu nome (um átomo) e por sua aridade ou número de argumentos. As constantes podem ser consideradas functores de aridade

zero. Sintaticamente os termos funcionais possuem a forma **f(t1,t2,...tn)**, onde **f** é o nome do functor e **n** a sua aridade, enquanto que os **ti's** são os seus argumentos. Um functor **f** de aridade **n** é denotado por **f/n**. Functores com o mesmo nome e diferente aridade são considerados distintos. Os termos são denominados **básicos** se não contiverem nenhuma variável, caso contrário são ditos **não-básicos**.

Uma **substituição** é um conjunto finito (possivelmente vazio) de pares na forma X = t, onde **X** é uma variável e **t** é um termo, não havendo dois pares com a mesma variável no lado esquerdo da igualdade. Para toda substituição $\theta = \{X_1 = t_1, X_2 = t_2, ..., X_n = t_n\}$ e para todo termo **s**, o termo $s\theta$ denota o resultado da substituição simultânea em **s** de cada ocorrência da variável **Xi** pelo termo **ti** correspondente em θ, $1 \leq i \leq n$. O termo $s\theta$ é denominado uma **instância** de **s**.

Um **literal** é um átomo ou a negação de um átomo. Um **literal positivo** é um átomo, enquanto que um **literal negativo** é a negação de um átomo.

Um programa em lógica é um conjunto finito de cláusulas definidas. Sendo A e B literais, uma **cláusula** é uma sentença lógica universalmente quantificada na forma:

$$A \leftarrow B_1, B_2, ..., B_n \quad n \geq 0$$

onde os $Bi's$ são *objetivos*. Tal sentença pode ser lida declarativamente como ``conjunção dos $Bi's$ implica $A$'', e interpretada operacionalmente como ``para responder a consulta A, responda a conjunção de consultas $B1, B2, ..., Bn$''. A é denominada a **cabeça**, e a conjunção de $Bi's$ o **corpo da cláusula**. Se $n = 0$, a cláusula é denominada

um **fato** ou **clausal unitária**, e é incondicionalmente verdadeira no contexto do programa.

Uma **consulta** é uma conjunção na forma:

$$\leftarrow B_1, B_2, \ldots, B_n \quad n > 0$$

onde os $Bi's$ são objetivos. As variáveis em uma consulta devem ser entendidas como se fossem existencialmente quantificadas.

Uma **computação** de um programa em lógica P corresponde à instanciação de uma dada consulta, logicamente deduzida a partir de P. Um objetivo G pode ser deduzido de um programa P se existe uma instância A de G em P onde $A \leftarrow B_1, B_2, \ldots, B_n$, $n \geq 0$, é uma instância básica de uma cláusula em P, e os $Bi's$ podem ser deduzidos em P.

O **significado** de um programa em lógica P é representado por $S(P)$, sendo indutivamente definido através de dedução lógica: o conjunto de instâncias básicas dos **fatos** está incluído no significado. Um objetivo básico G está no significado se há uma instância básica $G \leftarrow B_1, B_2, \ldots, B_n$ de uma cláusula em P tal que $B_1, B_2, \ldots, B_n$ estão no significado. O significado é, portanto, constituído por todas as instâncias básicas que podem ser deduzidas a partir do programa.

O **significado pretendido** S de um programa P é também um conjunto de objetivos unitários básicos. Um programa P é dito correto com respeito a um significado pretendido S se o seu significado $S(P)$ é um subconjunto de S. Por outro lado, o programa P é completo em relação a S se S é um subconjunto de $S(P)$.

Portanto um programa P será correto e completo em relação ao seu significado pretendido, situação esta desejada, se $S = S(P)$.

A lógica de predicados de primeira ordem é uma linguagem de especificação de grande expressividade, com um mecanismo dedutivo automático capaz de solucionar praticamente qualquer problema corretamente especificado. A propriedade de ser simultaneamente uma linguagem de especificação formal de grande expressividade e uma linguagem de programação de nível muito alto, inclui a lógica clausal entre as mais poderosas linguagens disponíveis para programação de computadores.

2.2 A Linguagem Prolog

2.2.1 INTRODUÇÃO

Prolog ([STE 86]) é uma linguagem de programação desenvolvida a partir do modelo de Programação em Lógica com cláusulas de Horn, proposto por Kowalski no início dos anos 70. O primeiro interpretador experimental foi implementado por Colmerauer, Roussel e outros na Universidade de Aix-Marseille (1972) com o nome de Prolog, um acrônimo para "Programmation en Logique".

O advento da linguagem Prolog reforçou a tese de que a lógica é um formalismo conveniente para representar e processar conhecimento. Seu uso evita que o programador descreva os procedimentos necessários para a solução de um problema, permitindo que ele expresse

declarativamente apenas a sua estrutura lógica, através de fatos, regras e consultas. Algumas das principais características da linguagem Prolog são:

- É uma linguagem orientada para processamento simbólico;
- Apresenta uma semântica declarativa inerente à lógica;
- Permite a definição de programas reversíveis, isto é, programas que não distinguem entre os argumentos de entrada e os de saída;
- Permite a obtenção de respostas alternativas;
- Suporta código recursivo e iterativo para a descrição de processos e problemas, dispensando os mecanismos tradicionais de controle, tais como *while, repeat,* etc;
- Permite associar o processo de especificação ao processo de codificação de programas;
- Representa programas e dados através do mesmo formalismo;
- Incorpora facilidades computacionais extralógicas e metalógicas.

2.2.2 FUNDAMENTOS

Em Prolog as cláusulas de Horn são definidas como:

k(X,Y) :- f(X), g(X,Y), f(Y).

onde o símbolo ":-" representa o operador de implicação ($\leftarrow$) e o símbolo "," o operador de conjunção ($\wedge$).

Como na Programação em Lógica, as constantes são iniciadas por letras minúsculas, e as variáveis iniciadas por letras maiúsculas.

2.2.2.1 Fatos

Considere a árvore genealógica mostrada na Figura 2.1, a seguir:

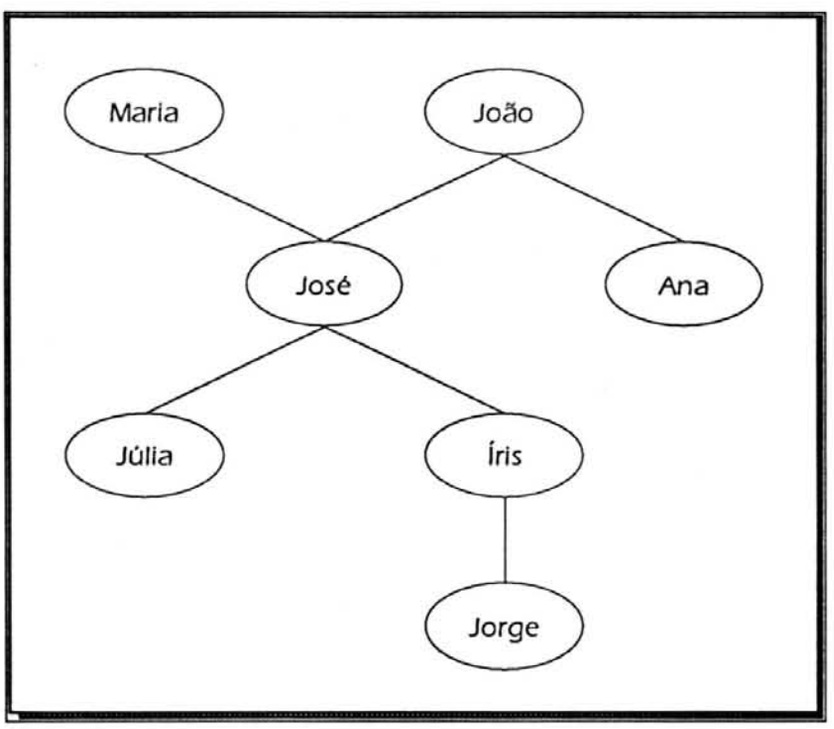

Figura 2.1 Uma árvore genealógica

É possível definir, entre os objetos (indivíduos) mostrados na Figura 2.1 uma relação denominada "progenitor" que associa um indivíduo a um dos seus progenitores. Por exemplo, o **fato** de que João é um dos progenitores de José pode ser denotado por:

progenitor(joão, josé).

onde progenitor é o nome da relação e joão e josé são os seus argumentos. A relação progenitor completa, como representada na figura 2.1 pode ser definida pelo programa Prolog apresentado na figura 2.2:

```
progenitor(maria, josé).
progenitor(joão, josé).
progenitor(joão, ana).
progenitor(josé, julia).
progenitor(josé, íris).
progenitor(íris, jorge).
```

Figura 2.2 Programa Prolog "Árvore Genealógica"

O programa da figura 2.2 compõe-se de seis cláusulas, cada uma das quais denota um **fato** acerca da relação progenitor.

2.2.2.2 Regras

Enquanto um fato é sempre verdadeiro, as regras especificam algo que "pode ser verdadeiro se algumas condições forem satisfeitas". As regras tem (figura 2.3):

- Uma conclusão (o lado esquerdo da cláusula), e
- Uma lista de condição (o lado direito da cláusula).

O símbolo ":-" significa "se" e separa a cláusula em conclusão ou cabeça da cláusula, da condição ou corpo da cláusula, como é mostrado no exemplo abaixo. Se a condição expressa pelo corpo da cláusula - progenitor (X, Y) - é verdadeira, então segue, como conseqüência lógica, que a cabeça -

filho(Y, X) - também o é. Por outro lado, se não for possível demonstrar que o corpo da cláusula é verdadeiro, o mesmo irá se aplicar à cabeça.

Um exemplo do uso de regras no Prolog seria:

filho(Y,X) :- progenitor(X,Y).

que também pode ser lida como: "para todo X e Y, se X é progenitor de Y, então Y é filho de X".

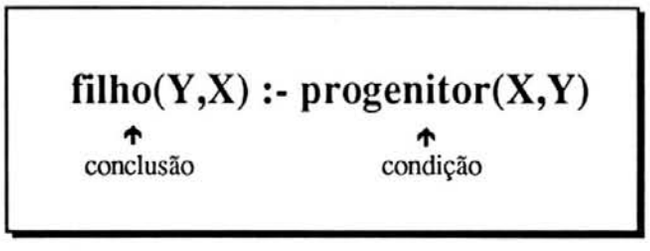

Figura 2.3 Regra Prolog

A utilização das regras pelo sistema Prolog pode ser ilustrada pelo seguinte exemplo: vamos perguntar ao programa se josé é filho de maria. Não há qualquer fato a esse respeito no programa, portanto, a única forma de considerar esta questão é aplicando a regra correspondente. A regra é genérica, no sentido de ser aplicável a quaisquer objetos X e Y. Logo, pode ser aplicada a objetos particulares, como josé e maria. Para aplicar a regra, Y será substituído por josé e X por maria, isto é, as variáveis X e Y se tornaram instanciadas para:

X=maria e Y=josé

A parte de condição se transformou então no objetivo progenitor(maria, josé). Em seguida o sistema tenta verificar se essa condição é verdadeira. Assim, o objetivo inicial, filho(josé, maria), foi

substituído pelo subobjetivo progenitor(maria, josé). Esse novo objetivo apresenta-se como trivial, uma vez que há um fato no programa estabelecendo exatamente que Maria é um dos progenitores de José. Isso significa que a parte de condição da regra é verdadeira, portanto, a parte de conclusão também é verdadeira e o sistema responde "sim".

2.2.2.3 Predicados

Em Prolog, um **predicado** é composto por todas as cláusulas cuja cabeça é definida pelo mesmo átomo e com o mesmo número de argumentos. O número de argumentos é chamado de **aridade**.

```
p(a,b).
p(c,d).
p(X,Y) :- g(X,Z) , h(Y).
```

No exemplo acima, o predicado **p** está definido por dois **fatos** e uma **regra**.

2.2.2.4 Consultas

Uma **consulta** em Prolog é sempre uma seqüência composta por um ou mais objetivos. Para obter a resposta, o sistema Prolog tenta satisfazer todos os objetivos que compõem a consulta, interpretando-os como uma conjunção. Satisfazer um objetivo significa demonstrar que este objetivo é verdadeiro, assumindo que as relações que o implicam são verdadeiras no contexto do programa. Se a questão também contém variáveis, o sistema Prolog deverá encontrar, ainda, os objetos particulares que, atribuídos às variáveis, satisfazem a todos os subobjetivos propostos na consulta. A particular instanciação das variáveis com os objetos que tornam

o objetivo verdadeiro, é então apresentada ao usuário. Se não for possível encontrar, no contexto do programa, uma instanciação comum de suas variáveis que permita derivar algum dos subobjetivos propostos, então a resposta será "não".

Se o programa da figura 2.2 for submetido a um sistema Prolog, este será capaz de responder algumas questões sobre a relação ali representada. Por exemplo: "José é o progenitor de Íris?". Como há um fato no programa declarando, explicitamente, que José é o progenitor de Íris, o sistema responde "sim".

```
?-progenitor(josé, íris).
sim
```

Perguntas mais interessantes podem, também, ser formuladas, como por exemplo: "Quem é progenitor de Íris?". Para fazer isso, introduz-se uma variável, por exemplo "X", na posição do argumento correspondente ao progenitor de Íris. Desta feita o sistema não se limitará a responder "sim" ou "não", mas irá procurar (e informar caso for encontrado) um valor de X que torne a assertiva "X é progenitor de Íris" verdadeira.

```
?-progenitor(X, íris).
X=josé
```

Da mesma forma a questão "Quem são os filhos de José?" pode ser formulada com a introdução de uma variável na posição do argumento correspondente ao filho de José. Note que, neste caso, mais de uma resposta verdadeira pode ser encontrada. O sistema irá fornecer a primeira que encontrar e aguardará manifestação por parte do usuário, que determinará se deve ser buscada ou não outra solução.

```
?-progenitor(josé, X).
X=júlia;
```

X=íris;
não

Aqui, a última resposta obtida foi "não" significando que não há mais soluções.

2.2.2.5 Unificação

Dentre as operações básicas da Programação em Lógica temos a **Unificação**. Como conseqüência, esta operação é também fundamental para o Prolog.

Dois termos unificam se:

- eles são idênticos, ou
- duas substituições θ_1 e θ_2 aplicadas a t_1 e t_2, os tornam idênticos e iguais a um termo t ($t_1\theta_1 = t_2\theta_2 = t$) sendo t o termo comum mais geral, isto é, qualquer outro t' (onde $t_1\theta_1 = t_2\theta_2 = t'$) é uma instância de t.

Por exemplo, os termos **data(D,M,1990)** e **data(X,fevereiro,A)**, unificam. Uma instanciação que torna os dois termos idênticos é:

1. D é instanciado com X;
2. M é instanciado com fevereiro;
3. A é instanciado com 1990.

Onde o termo t é data(X, fevereiro, 1990).

A unificação em Prolog sempre resulta na instanciação mais geral, isto é, a que limita o mínimo possível o escopo de valores das

variáveis, deixando a maior liberdade possível às instanciações posteriores. Por exemplo, no caso acima, D é instaciado a X e não D e X a um dia específico.

2.2.2.6 Retrocesso (*Backtracking*)

À medida que a complexidade de um programa Prolog aumenta, de modo geral também aumenta o grau de não determinismo das possíveis respostas, isto é, temos um número maior de possíveis soluções para uma consulta.

Prolog utiliza o mecanismo de retrocesso (*backtracking*) para percorrer a "árvore" cujos nodos (*choice-points*) são os pontos de confluência de duas ou mais alternativas que podem satisfazer um objetivo. O algoritmo de caminhamento empregado em Prolog (vide figura 2.4), é de cima para baixo até atingir um fim de alternativa, e da esquerda para direita (*depth-first left-to-right*).

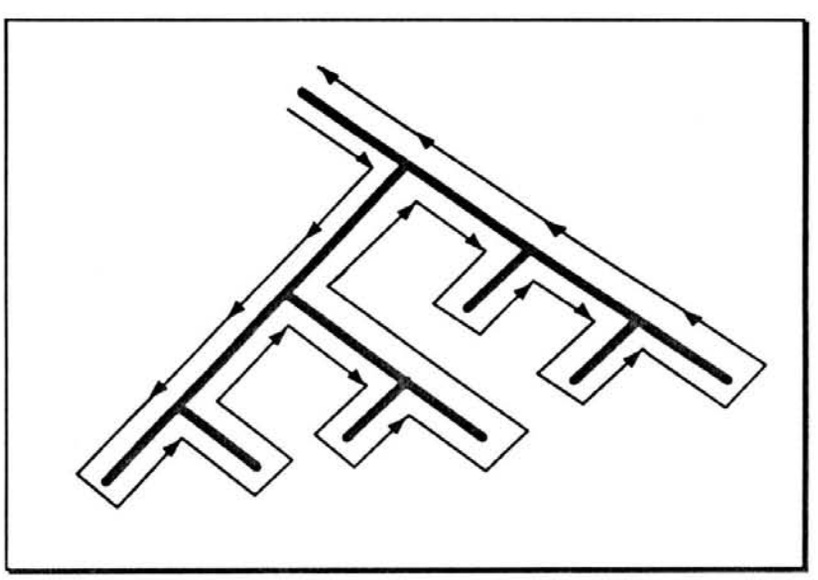

Figura 2.4 Árvore de Busca Prolog

2.2.2.7 Cut

Linguagens de programação "reais" como Prolog, incluem diversas extensões ao modelo de Programação em Lógica "puro". Entre estas extensões temos: read (leitura de dados fora do mundo externo); write (escrita para o mundo externo) e cut. Estas extensões afetam fortemente a visão declarativa do programa lógico.

O mecanismo de retrocesso é totalmente automático. Algumas vezes o programador deseja evitar a pesquisa de determinadas alternativas; isto pode ser feito com o uso do *cut* (corte). Este é representado por um "!" e é inserido entre os objetivos como uma espécie de pseudo-objetivo, que sempre é bem-sucedido, e que inibe a busca de outras alternativas através do retrocesso. É como se fosse "cortado" um ramo da árvore de metas.

O uso do **cut** aumenta a eficiência da execução do programa, pois evita a pesquisa em alternativas que não interessam para solução, porém reduz o aspecto declarativo do programa, o que se traduz em aumento do sincronismo quando da paralelização do processo de busca na árvore de metas.

2.3 Máquina Abstrata Prolog - WAM

2.3.1 Introdução

As implementações de Prolog entraram em uma nova era com a veloz implementação feita por D.H.D. Warren ([WAR 83]) para o DEC-10. O principal resultado deste trabalho foi um compilador eficiente, com

desempenho próximo ao das linguagens procedimentais. As técnicas de compilação, propostas por Warren, levaram a uma estratégia padrão de implementação, usualmente chamada WAM (*Warren Abstract Machine*). Em implementações baseadas na WAM, o código fonte em Prolog é compilado na linguagem de uma máquina abstrata baseada em pilhas. Um emulador portável desta máquina abstrata (típicamente escrito em C), permite um rápido e portável sistema Prolog. A maioria das implementações paralelas de Prolog tem como base a paralelização de versões deste emulador.

A busca de maior eficiência nas implementações seqüenciais permanece, a despeito do crescimento das implementações paralelas, pois um dos fatores que afeta o desempenho de uma versão paralela é a velocidade dos diversos módulos com implementação seqüencial que a compõem. Por outro lado, o desempenho das implementações seqüenciais serve de referência para o desempenho das paralelas, pois a comparação somente entre sistemas paralelos carece de significado prático, se existirem alternativas seqüenciais que são ou poderiam ser mais rápidas. "Ganhos de velocidade de execução são relativamente fáceis de se obter se a velocidade de referência é baixa" ([LUS 88]).

2.3.2 Estrutura Básica

A máquina abstrata WAM, é definida por um conjunto de estruturas de dados, um conjunto de registradores, um conjunto de instruções e uma organização de memória. São apresentadas a seguir a organização da memória e a representação das variáveis. Para maiores detalhes consultar [AIT 90].

2.3.2.1 A Organização da Memória e o Retrocesso

A memória está organizada em 3 pilhas (conforme a figura 2.5)

- A pilha ***Local***: utilizada para controlar a chamada dos predicados e do retrocesso, e para a alocação de variáveis locais a uma cláusula;
- A pilha ***Global***: utilizada para criação dos termos estruturados;
- A pilha ***Trail***: utilizada para atualizar as variáveis no momento do retrocesso.

A pilha *Local* contém dois tipos de estruturas de dados:

1. As **Áreas de Dados Locais**: Uma área de dados local (*environment*) contém as variáveis locais de uma cláusula em execução e o ponto de retorno à cláusula mãe;
2. Os **Nodos OU**: Um nodo OU (ou *choice-point*) contém as informações do estado da máquina, permitindo o retrocesso.

A área de dados locais corresponde à noção de bloco de ativação de um procedimento (*procedure*) das linguagens clássicas, e contém, entre outras, as informações necessárias para retornar à cláusula chamadora.

Um nodo OU é criado se o predicado chamado por um objeto é não-determinista (mais de uma cláusula). Os diversos nodos OU são empilhados, sendo o mais recente empregado em caso de falha (retrocesso), de maneira a restaurar o estado da máquina para execução da próxima cláusula do predicado. Um nodo OU é destruído (liberado) se esta próxima cláusula é a última do predicado (a liberação ocorre no início da execução desta cláusula).

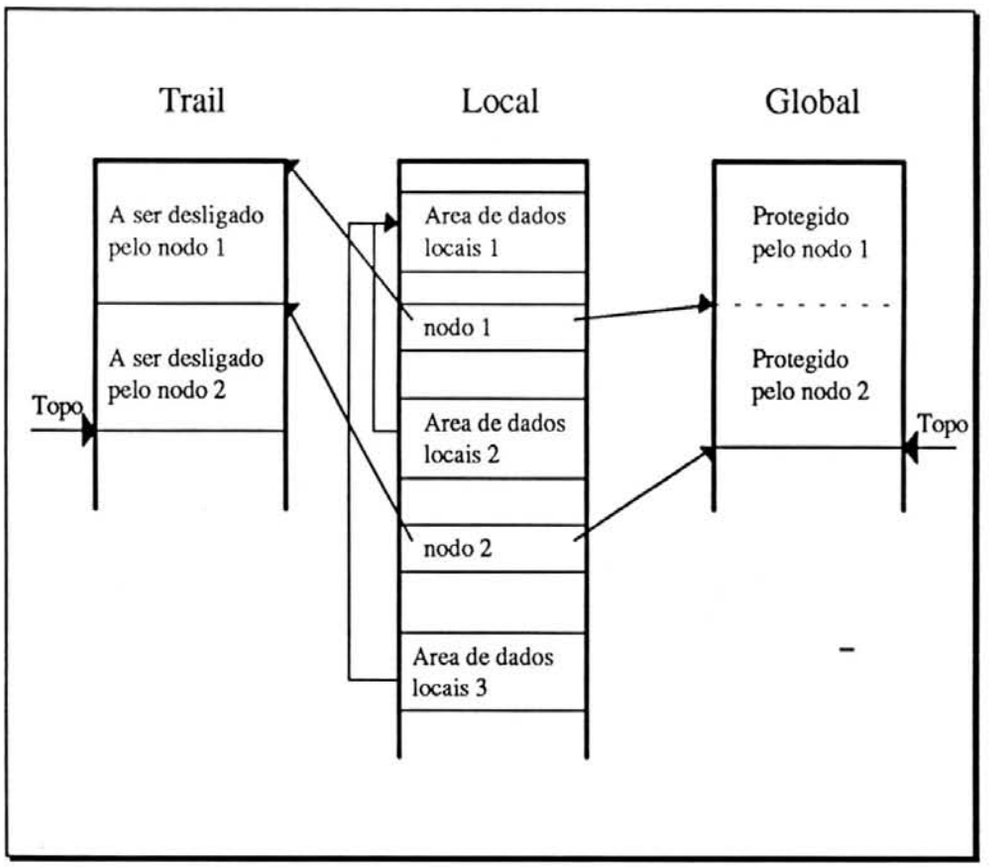

Figura 2.5 Memória Simplificada da WAM

Um nodo OU protege, automaticamente, as áreas de dados locais necessárias ao retrocesso, porque as duas estruturas são alojadas na mesma pilha física. O espaço de uma área de dados local não pode ser liberado (ao final da cláusula) se algum nodo OU foi criado durante sua execução. Em outras palavras, um nova área de dados local é sempre alocada no topo da pilha (considerando-se as duas estruturas). O ambiente protegido por um nodo OU só é recuperado quando do retorno a um nodo OU mais velho. Este mecanismo de proteção e recuperação de memória, assim como os mecanismos equivalentes para as pilhas *Global* e *Trail*, exigem o salvamento dos topos dessas pilhas (estado da máquina) nos nodos OU.

Os termos estruturados são construídos na pilha *Global*. O espaço alocado só é recuperado quando de um retrocesso a um nodo OU criado antes do termo, isto é, em caso de falha ou procura de outra solução.

A função da pilha *Trail* será examinada após a apresentação do esquema de implementação das variáveis.

2.3.2.2 As Variáveis: Representação e Estado

Uma variável lógica é representada por uma célula de memória da pilha *Local*, se ela for local a uma cláusula. O valor inicial de uma célula, após uma substituição, é representado por um ponteiro ao termo que representa o valor da variável. Esta operação é chamada **ligação**.

Algumas variáveis podem ter sido criadas antes de um nodo OU e ligadas por uma substituição após a criação do nodo OU. No caso de retrocesso a este nodo essas ligações devem ser desfeitas sem destruição das variáveis. Esta operação é chamada **desligamento**.

O grafo de estados das variáveis é mostrado na figura 2.6.

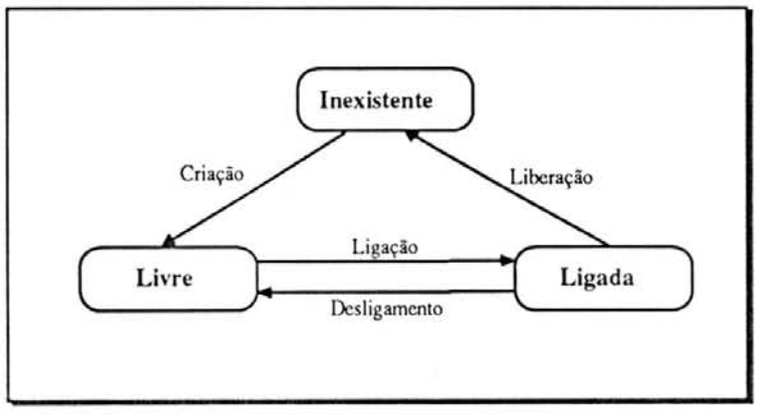

Figura 2.6 Grafo de Estados de Uma Variável

As ligações condicionais (incondicionais) são aquelas que são (não são) separadas da criação da variável pela criação de um nodo OU (figura 2.7). Uma ligação incondicional nunca é desligada (desfeita).

2.3.2.3 A Pilha Trail

Os desligamentos podem ser efetuados a partir de um registro de cada ligação condicional. Em geral estes registros são mantidos em uma pilha chamada pilha *Trail*.

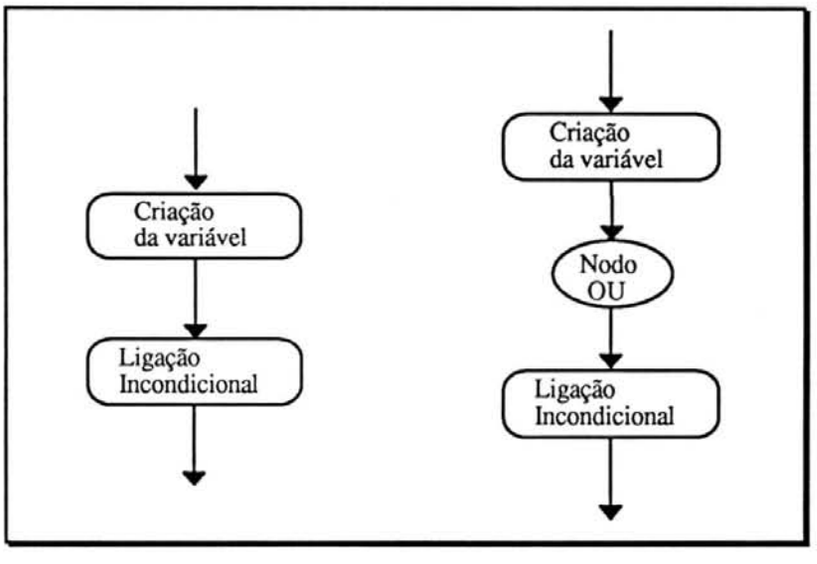

Figura 2.7 Ligações Condicionais e Incondicionais

Quando de um retrocesso a um nodo OU, esta pilha é percorrida entre o topo atual e o topo corrente quando da criação do nodo OU. As ligações das variáveis registradas neste trecho da pilha são desfeitas.

2.3.3 CONJUNTO DE INSTRUÇÕES

O conjunto de instruções da WAM é dividido em seis grupos, a saber: *Put* (correspondente aos argumentos do corpo da cláusula), *Get* (correspondente aos argumentos da cabeça da cláusula), *Unify* (correspondente aos argumentos de listas e estruturas), *Control* (controla as chamadas e o retorno), *Choice* (manipula os nodos OU) e *Indexing* (efetua a indexação de cláusulas). Esse conjunto de instruções é descrito em detalhes em [WAR 83] e [AIT 90].

2.4 Conclusões

Este capítulo apresentou os fundamentos da Programação em Lógica, sendo caracterizado o aspecto declarativo do programa lógico, no qual existe uma separação entre a semântica da linguagem e o controle da execução.

Foi também apresentada a WAM, uma das mais eficientes e difundidas técnicas para compilação Prolog.

Prolog foi introduzido como um exemplo prático de linguagem de Programação em Lógica, sendo revista sua estratégia particular de controle. Outras estratégias de controle para suporte a execução paralela de programas lógicos serão discutidas no próximo capítulo.

3 PARALELISMO NA PROGRAMAÇÃO EM LÓGICA

Neste capítulo serão apresentadas, de forma breve, as diferentes fontes de paralelismo na Programação em Lógica, bem como as principais estratégias para sua exploração.

O estudo que originou este capítulo, fez parte dos trabalhos para definição do modelo para paralelismo E do projeto OPERA. Sua inclusão nesta dissertação objetiva, além de apresentar conceitos, situar a opção feita no projeto OPERA em relação ao contexto internacional.

3.1 Fontes de Paralelismo na Programação em Lógica

A predominância do paradigma declarativo, na construção de um programa lógico, introduz características oportunas para exploração do paralelismo. Dentre estas destaca-se: a possibilidade de execução das cláusulas e literais de forma não-seqüencial, o não-determinismo na escolha de caminhos que levem à solução e o comportamento bidirecional da variável lógica.

Os tipos de paralelismo passíveis de serem explorados na execução de programas lógicos são:

- **paralelismo OU**: execução em paralelo das cláusulas de um predicado;
- **paralelismo E**: execução paralela dos literais que constituem o corpo de uma cláusula;

- **paralelismo de baixo nível**: execução paralela na unificação de predicados, o paralelismo E de fluxo (utilizando o conceito produtor-consumidor), e o paralelismo a nível de busca obtido pela distribuição da base de dados do programa entre vários processadores.

3.2 Paralelismo Explícito e Implícito

Uma outra classificação relativa ao paralelismo na Programação em Lógica, diz respeito a sua detecção a nível de linguagem. Neste sentido existem duas possibilidades:

- **paralelismo implícito**: nesta abordagem, a linguagem não dispõe de mecanismos de controle do paralelismo. O paralelismo é descoberto e gerenciado automaticamente pelo sistema computacional, utilizando análises em tempo de compilação e/ou execução.

- **paralelismo explícito**: a linguagem, neste caso, contém operadores para controle do paralelismo. Neste enfoque temos duas situações:

    1. linguagem estendida: consiste no acréscimo de predicados pré-definidos aos já existentes na linguagem. Através destes o programador pode controlar a execução paralela. Um exemplo é o uso de uma anotação que especifique a execução paralela (ou não) de alguns predicados.

    2. nova linguagem: neste caso são introduzidos mecanismos de controle que caracterizam uma

mudança substancial na semântica original da linguagem. As Linguagens Lógicas com Guardas ([SHA 89]) são um exemplo típico.

A principal vantagem do paralelismo implícito é liberar o programador das tarefas inerentes à gerência do paralelismo (sincronização de tarefas, concorrência, etc...), conseguindo melhoria de desempenho em relação à plataforma seqüencial de maneira transparente ao usuário. Outras vantagens desta proposta são: o aumento da portabilidade da aplicação do usuário entre as diferentes plataformas paralelas, e no caso de arquiteturas escalonáveis, a total independência entre o código da aplicação e o número de processadores que estão alocados.

Por sua vez o paralelismo explícito, também apresenta algumas características oportunas. A custos pequenos (tempos pequenos) de compilação, permite a obtenção de códigos paralelos eficientes. Um exemplo é a avaliação da granulosidade das tarefas a serem executadas em paralelo; este tipo de análise acrescenta um custo computacional ao compilador. Na abordagem do paralelismo explícito este problema é resolvido no momento da escrita do código, tendo por base o conhecimento do problema por parte do programador. Por outro lado, como o que deve ser executado em paralelo está explicitado no código, as rotinas para detecção e gerência do paralelismo durante a execução apresentam, também, pequeno custo computacional.

3.3 O Paralelismo OU

Este tipo de paralelismo refere-se a uma estratégia de busca paralela na árvore de metas do programa lógico. Assim, quando o processo

de busca encontra uma ramificação na árvore de metas, ele pode iniciar processos paralelos de busca em cada ramo descendente (vide figura 3.1). O nome paralelismo OU deriva do fato que em programas lógicos não determinísticos, existem várias respostas (vários caminhos) que satisfazem o objetivo.

3.3.1 O USO DO PARALELISMO OU

Existem diversas razões que justificam o fato das pesquisas terem focado, em um primeiro passo, o paralelismo OU ([LUS 88]). De maneira genérica, a exploração transparente do Paralelismo OU é mais simples de ser feita que a do paralelismo E. A seguir são apresentadas algumas razões para esta afirmação. Contudo, é importante ressaltar que nenhuma destas é impeditiva para a integração com o paralelismo E:

- **Generalidade**: é possível explorar o paralelismo OU sem restringir o poder da linguagem de Programação em Lógica empregada;

- **Simplicidade**: é possível explorar o paralelismo OU sem quaisquer anotações especiais por parte do programador, e também sem tornar o processo de compilação muito complexo;

- **Similaridade ao Prolog**: é possível explorar o paralelismo OU com um modelo de execução bastante próximo ao do Prolog seqüencial. Esta é uma maneira de tirar proveito de toda tecnologia existente (e experiência) para obter a melhor velocidade absoluta por processador. Esta similaridade também ajuda a preservar a semântica do Prolog;

- **Granulosidade**: o paralelismo OU tem o potencial, pelo menos para uma boa faixa de problemas em Prolog, de permitir paralelismo com elevada granulosidade. O tamanho do grão de uma computação paralela refere-se à quantidade de serviço que pode ser executado sem a necessidade de interação com outras "peças de trabalho" que estejam sendo executadas em paralelo. É mais fácil explorar com eficiência o paralelismo quando a granulosidade é grande;

- **Aplicações**: um significativo potencial de paralelismo OU é inerente a um grande grupo de aplicações, especialmente na área de inteligência artificial. Ele está presente em qualquer processo de busca, seja verificando as regras de um sistema especialista, provando um teorema, interpretando linguagem natural, ou respondendo a uma consulta em bases de dados.

É importante salientar dois aspectos na exploração do paralelismo OU, quais sejam: a explosão combinatória de processos e o ambiente de ligação das variáveis.

Programas lógicos não determinísticos têm, intrinsecamente, uma potencialidade maior de paralelismo. Assim, a maioria das aplicações em Inteligência Artificial pode gerar um grande número de pequenos processos ativos, explorando todas as possibilidades da árvore de metas. Isto iria implicar, para um número fixo de processadores (caso real), um alto consumo de memória e um elevado custo de gerenciamento. Daí vem a forte influência da granulosidade dos processos e do escalonador de serviços no desempenho de um sistema que explora o paralelismo OU.

No que diz respeito ao ambiente de ligação das variáveis, existe a questão de como podem ser melhor armazenadas e tratadas as diferentes ligações da mesma variável, geradas pela pesquisa simultânea nos diferentes ramos da árvore de metas. O objetivo das diferentes propostas de tratamento das ligações é implementar uma estratégia de gerência do paralelismo, que introduza o menor custo (*overhead*) possível.

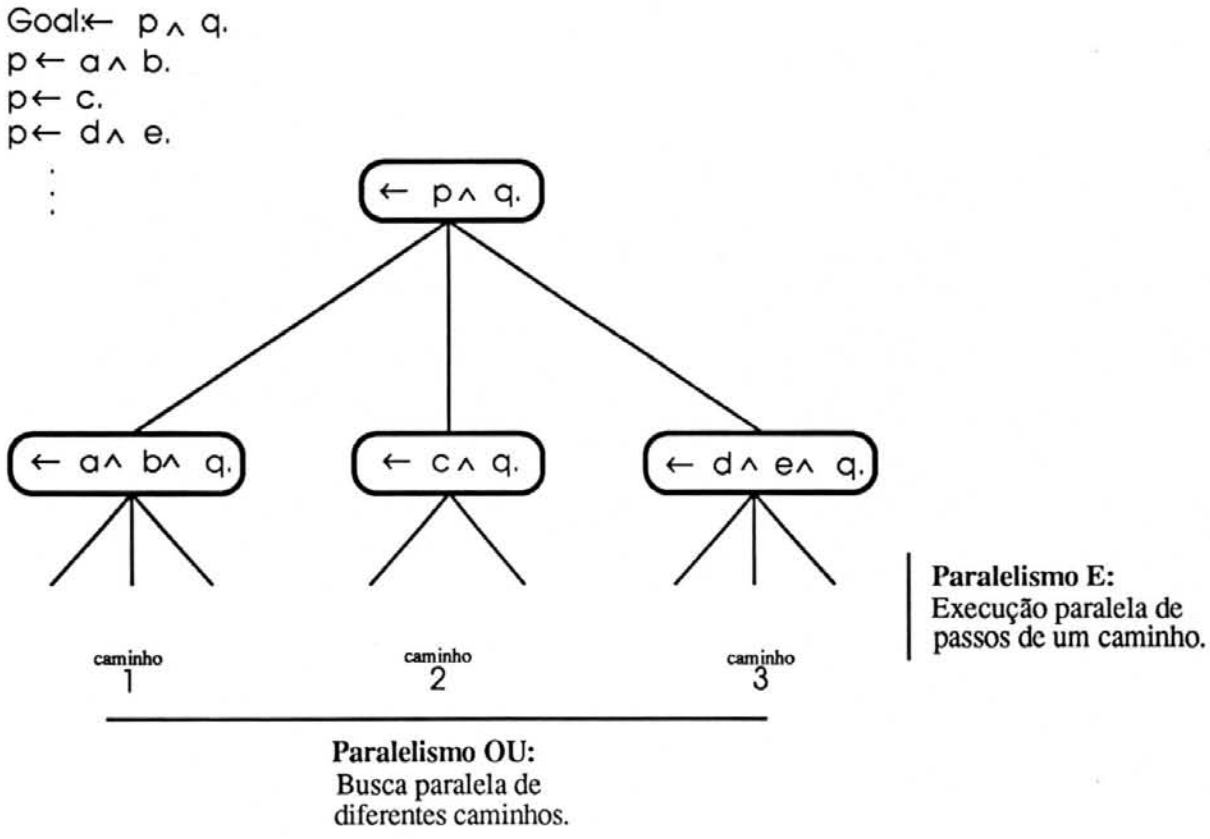

Figura 3.1 Paralelismo E & Paralelismo OU na Árvore de Objetivos

3.4 O Paralelismo E

Em termos da árvore de objetivos (vide figura 3.1), o paralelismo E corresponde à construção paralela de uma ramificação. Neste caso, quando o interpretador Prolog reconhece que um número de passos deve ser efetuado para completar um ramo, ele pode iniciar processos paralelos para avaliar estes passos.

3.4.1 O Uso do paralelismo E

A grande motivação para o paralelismo E, é a sua disponibilidade em quase todo programa lógico. Este tipo de paralelismo, ao contrário do OU, pode ser vantajoso para programas fortemente determinísticos.

Uma das maiores dificuldades na implementação do paralelismo E, é a gerência das variáveis comuns aos vários *literais* de uma cláusula. Esta interdependência pode gerar conflitos de atribuições de valores às variáveis, pois cada literal pode tentar atribuir valores diferentes às variáveis compartilhadas.

Por exemplo: a partir do programa apresentado na figura 3.2 (uma versão do clássico problema da genealogia), pode ser feita a consulta: quem é o avô de Jorge?

```
avo(X, Z) :- progenitor(X, Y), progenitor(Y, Z).
progenitor(maria, jose).
progenitor(joao, jose).
progenitor(joao, ana).
progenitor(jose, julia).
progenitor(jose, íris).
progenitor(iris, jorge).
```

Figura 3.2 Programa da "Árvore Genealógica"

consulta: ← avo(Y, jorge).

Considerando mais especificamente o predicado avo o qual tem potencial para o paralelismo E, temos a seguinte objetivo a ser satisfeito:

←progenitor(X,Y), progenitor(Y,jorge).

a execução em paralelo dos dois literais que constituem o objetivo resulta nos seguintes conjuntos solução:

Ramo 1: :- progenitor(X,Y)

Conjunto de ligações à variáveis: {X=maria, Y=jose}, {X=joao, Y=jose}, {X=joao, Y=ana}, {X=jose, Y=julia}, {X=jose, Y=iris}, {X=iris, Y=jorge}.

Ramo 2: :- progenitor(Y, jorge)

Conjunto de ligações à variáveis: {Y=iris}

Obviamente somente o penúltimo conjunto de ligações do ramo 1 ({X=jose, Y=iris}) é compatível com a única ligação produzida pelo segundo ramo. O sistema, após executar a intersecção dos dois conjuntos iria

responder que o avô de jorge é jose. É exatamente para fugir desta intersecção, que apresenta um elevado custo computacional sobretudo para programas fortemente não-determinísticos (realizam grande número de ligações por subobjetivo), que são empregadas as técnicas para implementação do paralelismo E descritas na seção 3.7.3.

3.5 Paralelismo de Baixo Nível

Além dos paralelismo E e OU, existem outros tipos de paralelismo que são considerados de mais baixo nível.

3.5.1 Paralelismo E de Fluxo

Nesta abordagem é também possível a avaliação concorrente de literais que compartilhem variáveis, trabalhando sobre uma mesma solução.

Vários processos podem avaliar estruturas de dados complexas, de forma incremental, em paralelo com o processo que as está produzindo. Assim, cada vez que o processo *A* produz um valor para uma variável compartilhada pelo processo *B*, o primeiro envia este valor ao segundo e continua a computação, da mesma maneira que o processo *B* segue calculando depois de receber o valor. Podemos supor este tipo de comunicação como sendo um esquema *produtor-consumidor*, em que alguns literais produzem valores para variáveis e outros os consomem. Este é o paralelismo típico das linguagens lógicas concorrentes. Na seção 3.6, considerando a sua importância, é apresentada uma descrição das linguagens lógicas concorrentes.

Esta forma de exploração do paralelismo é fortemente adequada aos programas lógicos determinísticos. Quando este não for o caso, a fim de que se obtenha eficiência, se faz necessário restringir o não-determinismo.

A maneira de restringir o não-determinismo é permitir que cada literal somente calcule uma solução. Esta é a desvantagem do paralelismo E de Fluxo, em relação às outras formas de paralelismo.

Existem várias implementações que empregam esta abordagem, dentre estas: a proposta de Somogyi et al ([SOM 88]), Parlog ([GRE 87]), Concurrent Prolog ([SAR 86]), P-Prolog ([YAN 86]), e GHC ([UED 85]).

3.5.2 PARALELISMO DE BUSCA

A razão para esta proposta ser considerada "de mais baixo nível" [CON 87] vem da necessidade de ser considerada a topologia de ligação e a distribuição da memória dos processadores.

Nesta proposta, o problema é quebrado em programas menores, que são armazenados em memórias locais, de ambientes paralelos ou distribuídos. A busca é realizada através da distribuição, por um solucionador de problemas, do objetivo a ser resolvido aos nós processadores, os quais realizam unificações e retornam as que tiverem sucesso. O maior inconveniente deste sistema é a quantidade de comunicações necessária.

O projeto mais conhecido que emprega esta forma de paralelismo é o PRISM [KAS 83].

3.5.3 PARALELISMO DE UNIFICAÇÃO

Quando ocorre a unificação de um objetivo com a cabeça de uma cláusula, vários pares de termos (argumentos) podem ser unificados em paralelo. Nesta proposta, a idéia é melhorar o desempenho dos interpretadores, sem necessariamente modificar a estratégia de controle da linguagem de Programação em Lógica empregada.

A possibilidade da unificação paralela é uma questão controvertida. Alguns pesquisadores como Dwork, Kanelakis e Mitchell ([DWO 84]) alegam esta não ser paralelizável, por sua vez, outros, como Lindistron ([LIN 84]), afirmam que para a média das situações é possível ter a unificação em paralelo.

Como exemplo de equipamentos processadores, cujas unidades para execução seqüencial incorporam otimizações em muito baixo nível, temos: a PLM ([DOR 85]), a PEK ([TAM 84]) e a PSI ([TAK 84])).

3.6 Linguagens Lógicas Concorrentes (Paralelismo E de Fluxo)

O ambiente usual da Programação em Lógica é limitado a sistemas *que trabalham por transformação*, isto é, sistemas que encerram o processamento após obter algum resultado para uma dada entrada. Esta proposta não é recomendada para sistemas *reativos*, ou seja, sistemas *abertos* capazes de reagir a uma seqüência de entradas provenientes do "mundo real".

Para dotar a Programação em Lógica de agentes interativos, foram empregadas as técnicas de comunicação e sincronismo, provenientes da teoria da concorrência ([DIJ 75]). As noções básicas para integração da concorrência em programas lógicos tem origem na Linguagem Relacional ([CLA 81]), antecessora de linguagens lógicas concorrentes como Parlog ([CLA 86]), GHC e KL1 ([UED 90]), Concurrent Prolog ([SHA 83]) e FCP ([SHA 89]).

Os principais conceitos inerentes a estas linguagens são:

- Uma **interpretação** *processual* dos programas lógicos substitui a tradicional interpretação procedimental. Cada objetivo é visto como um processo distinto. Literais relacionados por conjunção (AND) processam concorrentemente. Esta forma de paralelismo E é chamada dependente. No paralelismo E dependente, literais que compartilham variáveis também podem ser processados em paralelo.

- A **comunicação** acontece através de *variáveis lógicas*: toda variável compartilhada por diversos literais/processos atua como um canal entre estes. Esta variável constitui um poderoso e elegante mecanismo de comunicação.

- A **sincronização** entre processos é obtida pelo uso da relação produtor/consumidor. Um processo pode ser bloqueado enquanto as variáveis que ele *consome* são manipuladas por outros processos.

- O uso do conceito de **guardas** introduzido por Dijkstra ([DIJ 75] cujo emprego inicial se deu em linguagens imperativas

como Ada e Occam). Uma guarda é um construtor formado pelo par condições | ação, onde a ação é retardada até que uma das condições seja verdadeira. Este conceito, transladado para a Programação em Lógica, significa que cada cláusula recebe uma *guarda* que aparece entre a cabeça e o corpo. A guarda, neste caso, é uma seqüência de literais cuja avaliação deve ter sucesso, antes que o corpo da cláusula seja avaliado. Aspectos de implementação levaram a última geração de linguagens, tais como KL1 (Kernel Language 1) ou FCP (Flat Concurrent Prolog) a utilizarem apenas guardas "planas" (*flat guards*), isto é, guardas compostas somente por predicados pré-definidos (*built-in*). Isto assegura que a estrutura de guardas nunca será hierárquica, e a sua rotina de verificação de estado será razoavelmente rápida.

Nas linguagens lógicas concorrentes, o não-determinismo "*don't care*" substitui o não-determinismo "*don't know*", tradicional das linguagens lógicas. Isto significa que, a cada passo da resolução, somente uma alternativa possível é perseguida (e não importa qual - don't care). Esta abordagem é contrária à realizada pelas linguagens lógicas como o Prolog, as quais perseguem todas alternativas possíveis. O Prolog cria um nodo OU (*choice-point*) sempre que mais de uma cláusula unifica com o objetivo em questão, e no caso de falha da alternativa escolhida, busca outra não tentada, utilizando o mecanismo de retrocesso (*backtracking*) ao nodo OU criado. No não-determinismo *don't care* não são criados nodos OU e não existe o mecanismo de retrocesso, resultando um sistema determinístico. O não-determinismo permanece somente na verificação das guardas, uma vez que todas são avaliadas em paralelo. Entre as cláusulas com guardas satisfeitas, uma delas é escolhida, e a computação é restrita (*committed*) a

esta, sendo todas as outras abandonadas. Este mecanismo é, sintaticamente, expresso pela presença de um operador de restrição entre a guarda e o corpo de cada cláusula. O operador de restrição pode ser comparado com o operador "cut" do Prolog (vide item 2.2.2.7).

3.7 Técnicas Para Implementação de Paralelismo na Programação em Lógica

Esta seção compreende uma discussão sobre as principais técnicas empregadas para implementação de sistemas, para exploração do paralelismo na Programação em Lógica. Seu objetivo é caracterizar, de forma breve, as estratégias adotadas nos modelos mais importantes.

3.7.1 Modelos Teóricos & Modelos Multiseqüenciais

3.7.1.1 Modelos Teóricos

O principal objetivo das propostas teóricas, é a exploração de todo paralelismo existente em um programa Prolog padrão (paralelismo implícito). A origem do nome "teóricos" está na dificuldade de obter-se uma implementação eficiente. Estes modelos apresentam as seguintes características:

- **processos criados**: a quantidade de processos criados, tende a extrapolar os recursos disponíveis para cálculo, e/ou a memória da arquitetura multiprocessadora;

- **comunicações necessárias**: o grande tráfego de mensagens entre os processos (para sincronismo, passagem de parâmetros ou retorno de soluções), associado à existência destes em grande número, conduz à saturação os mecanismos de comunicação atualmente disponíveis;

- **granulosidade**: a não existência de controle da granulosidade do paralelismo explorado, permite a criação de processos, cujo custo de instalação pode exceder a duração da sua execução.

Um exemplo da complexidade destas propostas é o modelo E/OU de Conery ([CON 85]); neste modelo, processos AND são criados para computar o corpo das cláusulas, enquanto processos OU são criados para cada literal dos corpos das cláusulas. Cada um destes processos, por si mesmo, cria processos filhos, obtendo-se desta forma, uma árvore de processos que trocam mensagens. Os processos AND utilizam complexos algoritmos para, dinamicamente, reordenar os literais do corpo da cláusula que gerenciam, de forma tal, que dois processos OU, que poderiam atribuir valores conflitantes às variáveis, não estejam ativos ao mesmo tempo.

Outras propostas teóricas são: COALA ([SIM 86]), MIMAP ([KHA 88]).

3.7.1.2 Modelos Multiseqüenciais

O objetivo dos sistemas multiseqüenciais é, exatamente, limitar o custo computacional proveniente da criação de processos, ou de complexas rotinas para suporte ao paralelismo, em tempo de execução. Nestes sistemas, o montante de paralelismo empregado é, automaticamente,

adaptado aos recursos disponíveis. A computação é realizada por um determinado número de trabalhadores, cujo total está associado ao número de elementos de processamento disponíveis para tal, no momento da execução. Durante o processamento, os trabalhadores podem estar ativos, computando parte do programa, ou inativos, aguardando serviço.

Os sistemas multiseqüenciais podem incorporar a maioria das técnicas de otimização, disponíveis para os sistemas Prolog seqüenciais. Quase todos são baseados na compilação dos programas fonte Prolog para a Máquina Abstrata de Warren (WAM - vide seção 2.2.2.8). A sua proposta, porém, é também compatível com técnicas de implementação mais atuais, como a BAM (*Berkeley Abstract Machine*) apresentada em [ROY 92].

A maioria dos sistemas implementados são baseados na abordagem multiseqüencial.

3.7.2 TÉCNICAS DE IMPLEMENTAÇÃO DO PARALELISMO OU

Nos sistemas que exploram o paralelismo OU, a busca de todas as soluções para uma consulta pode empregar computações concorrentes, ao invés de computações seqüenciais separadas por retrocesso (*backtracking*). Diversas *resolvantes*, formadas pelo conteúdo das pilhas da Máquina Abstrata Prolog empregada (normalmente a WAM) e seus registradores, podem ser avaliadas concorrentemente por diversos trabalhadores.

Diversas técnicas tem sido propostas para administrar, de forma eficiente, o processamento simultâneo de várias *resolvantes*; uma família de soluções é baseada na cópia de pilhas, outra família no compartilhamento de pilhas, e uma terceira, emprega a reconstrução de pilhas por recomputação.

3.7.2.1 Cópia de Pilhas

Na busca de trabalho, um trabalhador desocupado copia a porção de pilha com o estado da computação anterior ao nodo OU (*choice-point*), que contém a alternativa não explorada.

Ocorre um problema com este esquema: o trabalhador que está buscando trabalho (importador) precisa restaurar sua cópia de pilhas ao estado em que estavam, quando o nodo OU (com a alternativa não explorada) foi criado. Duas soluções foram propostas para tratar esta situação:

- **uso de datas**: esta proposta, apresentada no modelo **Kabu-Wake** ([MAZ 86]), associa uma data lógica a toda atribuição de valor à variáveis. Neste método, toda ligação condicional (vide seção 2.2.2.8), cuja data for posterior a criação do nodo onde está o trabalho (alternativa) a ser importado, é descartada pelo trabalhador importador;

- **uso da pilha *Trail***: esta solução exige sincronismo entre trabalhador-exportador e importador. Neste caso, o trabalhador-importador usa sua cópia da pilha *Trail* para desligar (*unbind*), em suas pilhas, todas as ligações condicionais realizadas pelo trabalhador-exportador e que não são válidas para ele, reproduzindo parte das operações feitas durante um retrocesso (*backtracking*).

A proposta empregando datas lógicas gera um gasto de memória e tempo, associando datas lógicas a cada ligação de variável, enquanto que, na segunda proposta, o trabalhador-exportador (trabalhador já ativo) fica preso em função do sincronismo necessário durante a operação de cópia (no

mínimo durante a cópia da *Trail*).

Uma visão mais detalhada deste esquema está no item 4.1.2.3, onde é apresentada a proposta de paralelismo OU do projeto OPERA.

3.7.2.2 Compartilhamento de Pilhas

No esquema de pilhas compartilhadas, partes das pilhas *Local* e *Global* são compartilhadas e partes são privadas: os trabalhadores compartilham as partes das pilhas que correspondem às porções da árvore de busca que eles tem em comum. Em adição, estruturas de dados privadas são usadas para armazenar as diversas ligações condicionais às variáveis das partes compartilhadas das pilhas. Dois tipos de estruturas de dados privadas foram propostas: *binding arrays* ([WAR 87]) e *hash-windows* ([BOR 84]).

Um *binding array* é uma tabela, privada de cada trabalhador, que contém tais ligações condicionais conflituosas. As variáveis da parte compartilhada, as quais podem ser ligadas simultaneamente com valores conflitantes pelos vários trabalhadores, não armazenam valores (ligações), como em um sistema Prolog, mas inteiros. Estes inteiros são indexadores das ligações das variáveis dentro do *binding array* de cada trabalhador. Um sistema que emprega esta técnica é o **Aurora** ([WAR 87]).

Na segunda alternativa, as ligações realizadas para as mesmas variáveis são armazenadas em estruturas chamadas *hash-windows* e conectadas a cada nodo da árvore de busca. Todas as ligações realizadas ao longo do caminho, desde a raiz da árvore de busca até o nodo OU da

computação são válidos para o trabalhador calculando este nodo. Por esta razão, as *hash-windows* são ligadas formando uma cadeia.

Diversas implementações utilizam o conceito de *hash-windows* ([BUT 86], [KER 89a]).

O esquema de *binding array* adiciona um custo constante à operação de busca de variável. Outro custo deste esquema é a atualização do *binding array* de um trabalhador, quando este importa trabalho de outro. Por sua vez, de forma contrária, iniciar um novo nodo tem um baixo custo no esquema *hash-windows*, uma vez que é somente necessário criar uma nova *hash-window* e ligá-la a uma anterior. Contudo, o custo da busca de uma variável não é fixo, uma vez que pode implicar na busca em uma cadeia de *hash-window* de tamanho indeterminado.

3.7.2.3 Recomputação de Pilhas

Para evitar os custos associados às soluções de cópia ou compartilhamento de pilhas, é possível ter trabalhadores computando, independentemente, caminhos completos da árvore de busca. As partes da árvore de busca correspondendo às porções de pilhas, as quais seriam copiadas ou compartilhadas nos esquemas anteriores, são recomputadas por cada um dos trabalhadores ativos. A cada trabalhador é dado um caminho pré-determinado da árvore de busca, descrito por um *oráculo* alocado a um trabalhador especializado, chamado *controlador*. Nesta proposta, os programas são reescritos para obter uma árvore de busca com aridade 2, de forma a obter *oráculos* eficientes. Uma descrição detalhada desta proposta pode ser encontrada em ([CLO 88]).

3.7.3 TÉCNICAS DE IMPLEMENTAÇÃO DO PARALELISMO E

Nos sistemas que exploram o paralelismo E, vários trabalhadores podem ligar variáveis lógicas com valores conflitantes. Dividindo em dois grandes grupos, temos: os sistemas que utilizam o paralelismo **E Independente**, os quais exploram o paralelismo somente entre literais independentes (sem possibilidade de conflito na atribuição de valores a variáveis), e aqueles que utilizam o paralelismo **E Dependente**, que por sua vez, exploram o paralelismo entre quaisquer literais (às custas de um mecanismo de detecção e gerência do paralelismo mais complexo).

3.7.3.1 Paralelismo E Independente

Nesta abordagem, o ponto focal da gerência do paralelismo é a determinação da independência entre os literais que formam a cláusula Prolog.

Detecção de Independência em Tempo de Compilação

Para este enfoque, é realizada uma análise em tempo de compilação das dependências de dados no programa, a fim de caracterizar os literais independentes, bem como determinar a seqüência de execução dos literais não-independentes.

Uma das poucas propostas, neste sentido, é a de Chang ([CHA 85]). A idéia geral do método é derivar grafos de dependência em tempo de compilação, a partir de informações provenientes do usuário. O programador fornece ao compilador o *modo de ativação* da consulta, ou seja, quais procedimentos serão chamados, e os estado dos argumentos (*fechados,*

independentes ou dependentes). A saída do Analisador de Dependência Estática de Dados do modelo (*Static Data Dependency Analysis* - SDDA) é um grafo que determina quais literais no corpo da cláusula podem ser executados em paralelo (independentes) e a ordem de ativação dos seus literais não-independentes.

A principal vantagem desta abordagem é não necessitar de suporte, durante o tempo de execução, para detectar conflitos de ligação de variáveis. Como desvantagem, temos:

- ser permitido somente um tipo de consulta para cada procedimento; outras consultas, que não estejam de acordo com o *modo de ativação* anteriormente declarado, não serão executadas em paralelo;

- a análise de dependências estáticas ser baseada no pior caso, pois, em tempo de compilação, não é conhecido tudo a respeito das possíveis ligações das variáveis e, deste modo, pode não ser explorado todo paralelismo disponível.

Um aspecto que também influencia no desempenho é a complexidade do algoritmo de retrocesso semi-inteligente que este método utiliza. Este algoritmo, suportado em tempo de execução, introduz considerável custo computacional extra. É possível empregar o mecanismo tradicional de retrocesso do Prolog, porém, nesta proposta, este subtrairia eficiência na gerência da execução paralela.

Detecção de Independência Em Tempo de Execução

Nesta abordagem, os conflitos de atribuição de valores a variáveis são administrados somente em tempo de execução. O problema desta solução está no custo computacional da sua estratégia para gerenciar o paralelismo, a ponto de, na maioria das arquiteturas, ter uma baixa produtividade.

Uma proposta com estas características é o modelo E/OU de Conery ([CON 85]). Este modelo foi o primeiro a propor uma solução completa para gerência do paralelismo E, e objetiva uma execução paralela não-determinística *don't Know*, baseada em troca de mensagens. O suporte à gerência do paralelismo é feito através de *grafos de dependência de dados*, que são gerados e manipulados por diversos algoritmos durante a execução.

Sua eficiência é baixa, exatamente pelo grande número de operações necessárias para controle do paralelismo, uma vez que os *grafos de dependência de dados* precisam ser recomputados a cada invocação de cláusula, bem como na ocorrência de retrocesso.

Paralelismo E Restrito

No paralelismo E restrito (*Restricted And Parallelism - RAP*), a gerência e o controle do paralelismo estão distribuídos entre a compilação e a execução ([DEG 84]). No RAP, a compilação gera, para cada cláusula do programa Prolog, uma CGE (*Conditional Graph Expression*). No momento do seu processamento, cada CGE, em função do estado das variáveis, elegerá um grafo de ativação para a cláusula em questão. Esta verificação do estado das variáveis determina quais variáveis são independentes. O grafo selecionado será um entre aqueles que iniciam a execução paralela

com os literais que envolvem somente estas variáveis. A geração de CGEs, bem como a verificação do estado das variáveis, apresentam um pequeno custo computacional.

3.7.3.2 Paralelismo E Dependente

Na exploração do paralelismo E Dependente, mesmo os literais que compartilham variáveis podem ser executados em paralelo. Tais literais são sincronizados por uma relação produtor/consumidor através de variáveis comuns.

Esta forma de paralelismo é adequada para programas lógicos determinísticos. Quando este não for o caso, é necessário uma forma de restringir o não-determinismo para que seja alcançada eficiência. A maneira de restringir o não-determinismo é fazer com que seja computada somente uma solução para cada objetivo (a conseqüência desta restrição é análoga à utilização do operador de corte em Prolog -*CUT*-). E esta característica é a sua maior desvantagem em relação a outras propostas.

Maiores detalhes desta abordagem estão descritos na seção 3.6, onde são apresentadas as características das linguagens lógicas concorrentes.

3.8 Principais Sistemas Propostos

O objetivo desta seção é apresentar os modelos de exploração do paralelismo na Programação em Lógica que tem uma proposta consolidada e, em alguns casos, uma implementação eficiente.

3.8.1 SISTEMAS QUE EXPLORAM O PARALELISMO OU

Nos últimos anos, diversos sistemas tem sido propostos para exploração do paralelismo OU na Programação em Lógica. Nesta seção descreveremos, de forma resumida, as principais implementações em arquiteturas multiprocessadoras. Em [YAM 92] (trabalho apresentado ao CPGCC/UFRGS), pode ser encontrada uma discussão mais ampla sobre os modelos de exploração do paralelismo OU.

3.8.1.1 KABU-WAKE

O sistema **Kabu-Wake** ([MAZ 86]) foi o primeiro a empregar cópia de pilhas, quando da troca de serviço entre os processadores, bem como em utilizar datas lógicas para descartar ligações não válidas. Uma implementação do **Kabu-Wake**, em um *hardware* paralelo dedicado, conseguiu, para problemas grandes (árvore de busca com ramos de elevada granulosidade), um ganho de velocidade na execução paralela, quase linear ao número de processadores. Contudo, é importante observar que este sistema é baseado em um emulador (multiseqüencial) lento.

3.8.1.2 ANL-WAM

O sistema **ANL-WAM** do *Argonne National Laboratories* ([BUT 86], [DIS 87]) foi o primeiro sistema Prolog paralelo baseado em técnicas de compilação e implementado em multiprocessador de memória compartilhada. Todas as variáveis lógicas trabalhadas são armazenadas em *hash-windows*, mesmo quando não ocorre computação paralela. Tendo sido a primeira implementação eficiente de Prolog paralelo, a **ANL-WAM** foi amplamente utilizada para fins experimentais. Apesar de apresentar um

bom ganho na velocidade de execução por processador (bom *speedup*), o desempenho global do sistema **ANL-WAM** é limitado pela baixa eficiência da implementação de WAM empregada e pelo custo adicional imposto pela proposta de *hash-window*.

3.8.1.2 PEPSys

No sistema **PEPSys** ([BAR 88b]), as *hash-windows* são utilizadas somente se necessário, isto é, quando variáveis lógicas, tendo a mesma identificação WAM, são ligadas simultaneamente por vários trabalhadores. A maioria dos acessos aos valores das variáveis são tão eficientes como em uma implementação seqüencial, porém alguns acessos podem envolver pesquisa em uma cadeia de *hash-window* de tamanho indefinido. O custo de instalação é independente da distância entre o trabalho e o trabalhador inativo na árvore de busca. Apesar do uso de compartilhamento de pilhas, **PEPSys** não exige um espaço de endereçamento global, e tem sido simulado em arquiteturas escalonáveis, combinando o uso de memória compartilhada em grupos (*clusters*) de processadores e troca de mensagens entre grupos ([BAR 88a]). O **PEPSys** também tem sido eficientemente implementado em multiprocessadores de memória compartilhada. Em um único processador, a implementação do **PEPSys**, baseada na WAM, processa de 30% a 40% mais lentamente que o SICStus Prolog[1]. Em paralelo, para programas com um grande espaço de busca, **PEPSys** tem ganhos quase lineares na execução ([BAR 88b]). Outros resultados experimentais ([KER 89b]) mostram que

---

[1] Uma das mais eficientes implementações acadêmicas de Prolog seqüencial. Foi desenvolvida pelo Swedish Institute of Computer Science. Hoje dispõe de versão comercial.

longas cadeias de *hash-windows* são raras, e por isso não comprometem a eficiência da implementação.

3.8.1.3 Aurora

O sistema **Aurora** é baseado no modelo SRI ([WAR 87]), onde as ligações das variáveis lógicas que compartilham a mesma identificação são feitas utilizando *binding arrays*. Um protótipo do **Aurora**, baseado no SICStus Prolog, foi implementado em diversos multiprocessadores comerciais com memória compartilhada. Quatro diferentes escalonadores foram implementados: três deles ([BUT 88], [CAL 88] e [BRA 88]) empregam várias técnicas para o trabalhador inativo (*idle*) buscar serviço do nodo OU mais próximo que esteja acima na árvore de metas. O objetivo de procurar serviço no nodo mais próximo possível é minimizar o custo da cópia incremental de pilhas (vide 3.7.2.1). Uma vez que resultados experimentais ([SZE 89]) mostraram que o custo de instalação de trabalha permanece razoavelmente limitado (custo da cópia de pilhas), o escalonador Bristol ([BEA 91]) executa o compartilhamento de trabalho tendo como principal critério a maximização da granulosidade das tarefas paralelas. Todos os escalonadores suportam a linguagem Prolog completa, incluindo *side-efects*. O sistema Aurora processou, com sucesso, um grande número de programas Prolog, sendo que o desempenho de algumas execuções está reportada em [SZE 89] e [BEA 91]. Em um único processador, **Aurora** é um pouco mais eficiente que **PEPSys** e, em média, para execução paralela, ele também se mostra um pouco mais eficiente.

3.8.1.4 Muse

Para o modelo **Muse,** é assumida uma arquitetura cujos processadores dispõem de um espaço de endereçamento privado, e um espaço de endereçamento comum. Sistemas operacionais distribuídos como o DYNIX e o MACH apresentam estas características.

No modelo **Muse** ([ALI 90]), trabalhadores ativos compartilham com trabalhadores inativos diversos nodos contendo alternativas não processadas. O compartilhamento é realizado por um trabalhador ativo, o qual cria uma imagem de uma parte de sua pilha de nodos, em um espaço de trabalho comum com um trabalhador inativo. As pilhas do trabalhador ativo são então copiadas de sua memória local, através da região compartilhada, para a memória local do trabalhador inativo. Nesta operação, os trabalhadores empregam a técnica da cópia incremental.

A solução do conflito de ligações na região compartilhada utiliza a pilha *Trail* ao invés de datas lógicas. O **Muse** foi implementado em diversos multiprocessadores comerciais, com endereçamento de memória uniforme e não uniforme. Uma implementação do **Muse** foi realizada, também, sobre o protótipo da *BC-Machine* construído no *Swedish Institute of Computer Science* (SICS), o qual oferece endereçamento de memória privado e compartilhado a nível de *hardware* ([ALI 91]). A implementação do **Muse** é baseada no SICStus Prolog. O desempenho em um processador é bastante próximo ao do SICStus seqüencial (custo adicional de 5% pelas funções de gerência do paralelismo). Por sua vez, os ganhos na velocidade de execução paralela são bastante similares aos do modelo **Aurora**.

3.8.1.5 K-LEAF

O sistema **K-LEAF** ([BOS 90]), em contraste com outros modelos multiseqüenciais, cria todas as possíveis tarefas OU paralelas. A explosão combinatória do número de processos é evitada por construtores na linguagem, os quais garantem uma granulosidade mínima para as tarefas. **K-LEAF** foi implementado em uma arquitetura multiprocessadora, baseada em Transputers, que apresenta um espaço virtual de endereçamento global. Esta implementação é baseada na WAM, utilizando o esquema de *binding array*. Um aspecto interessante neste sistema é a possibilidade do código WAM poder ser emulado ou expandido em C e então compilado. A versão compilada chega a ser cinco vezes mais eficiente que a primeira.

3.8.2 Sistemas Que Exploram o Paralelismo E Independente

3.8.2.1 &-Prolog

O sistema **&-Prolog** ([HER 90]) é um dos mais maduros sistemas explorando paralelismo E independente. Ele combina uma detecção de independência a nível de compilação com um eficiente ambiente de execução implementado em multiprocessadores com memória compartilhada. Sua proposta é fundamentada no Paralelismo E Restrito ([DEG 84]), e utiliza como técnica de compilação uma variante da WAM.

O paralelismo pode ser registrado explicitamente pelo usuário com a linguagem &-Prolog. **&-Prolog** é muito similar ao Prolog, com a adição do operador de conjunção paralela "&" e as primitivas utilizadas para formação das CGEs (vide item 4.2.2.3). Por exemplo o programa Prolog:

p(X) :- q(X), r(X).

Pode ser escrito em &-Prolog como:

p(X) :- (ground(X) ⇒ q(X) & r(X)).

O paralelismo também pode ser explorado automaticamente, a partir de um programa em Prolog padrão. O compilador do modelo executa a transformação de programas Prolog para &-Prolog. A análise estática do compilador detecta a independência entre literais, mesmo na presença de predicados com *side-effects* ([MUT 89], [MUT 89a]).

Os programas &-Prolog são compilados em uma extensão da WAM denominada PWAM, cuja principal diferença em relação a WAM, é a adição de uma *pilha de literais paralelizáveis*, onde processadores inativos retiram trabalho.

Para ajustar os recursos computacionais para o volume de trabalho existente, o escalonador **&-Prolog** organiza as máquinas abstratas PWAM em anel. Os agentes de processamento do modelo, denominados trabalhadores, procuram por uma PWAM inativa no anel. Se não for encontrada uma PWAM inativa e existir memória suficiente, o trabalhador cria uma nova PWAM, liga esta nova máquina ao anel, e procura por trabalho na *pilha de literais paralelizáveis* de outras PWAMs ativas.

O ambiente de execução do **&-Prolog** é baseado no SICStus. Executando em um processador, o **&-Prolog** tem um desempenho muito próximo do SICStus seqüencial. A perda é menor que 5% e se deve principalmente ao manuseio da *pilha de literais paralelizáveis*. Os ganhos

de velocidade na execução paralela são muito bons e, naturalmente, dependentes do problema em questão.

3.8.3 SISTEMAS QUE EXPLORAM O PARALELISMO E DEPENDENTE

Nesta seção, a título de exemplo, são apresentadas apenas algumas das várias implementações que exploram o Paralelismo E Dependente. Em [SHA 89] pode ser encontrada uma abordagem bastante completa do tema.

3.8.3.1 Parlog

O sistema **Parlog** ([CLA 86]) emprega uma Máquina Abstrata Prolog denominada JAM. Esta máquina, baseada na WAM, foi projetada para suportar Linguagens Lógicas Concorrentes e em especial a própria linguagem Parlog. O **Parlog** foi implementado em multiprocessador com memória compartilhada ([CRA 88]). Computando em um único processador atinge a metade do desempenho do SICStus Prolog seqüencial. Para os programas que apresentam volume significativo de paralelismo E dependente, os ganhos de velocidade na execução paralela, são entre 12 e 15 para 20 processadores. Por outro lado, programas, que não contenham paralelismo E dependente (por exemplo o conhecido *queens*), processam praticamente à mesma velocidade, tanto em paralelo como seqüencialmente (em somente 1 processador).

3.8.3.2 GHC e KL1

Um grande número de implementações das Linguagens Lógicas Concorrentes **GHC** (*Guarded Horn Clause*) e **KL1** (*Kernel Language 1*) ([UED 90]) foram realizadas no ambiente do Projeto Japonês de Computadores de Quinta Geração, coordenadas pelo ICOT ([MOT 81]). A proposta para suporte a **KL1** foi implementada em um multiprocessador com memória compartilhada, disponível comercialmente ([SAT 88]). Esta implementação conseguiu um ganho de execução paralela de 2 a 9 vezes menor que o do sistema **Aurora.** A comparação foi feita utilizando programas com a mesma finalidade. Os trabalhos mais significativos para exploração do paralelismo E Dependente, no ICOT, envolveram o desenvolvimento de *software* e *hardware*, que resultaram no projeto de arquiteturas dedicadas para execução da Linguagem Lógica Concorrente **KL1**. Dentre as arquiteturas propostas, há o multiprocessador PIM, que atinge até 512 processadores (especializados). Sua arquitetura pode acomodar até 64 grupos de processadores (*clusters*), sendo cada um similar a um multiprocessador de 8 elementos com memória compartilhada ([TAK 92], ([KUM 92]).

3.8.3.3 Strand

O **Strand** (STReam AND-parallelism) ([FOS 89]) é um produto comercial que foi derivado do **Parlog** e do FCP (Flat Concurrent Prolog). Na linguagem **Strand**, não há unificação mas casamento da cabeça de uma cláusula com um objetivo.

A implementação do **Strand** é baseada na *Strand Abstract Machine* (SAM), projetada com o objetivo de minimizar alterações necessárias para portar o **Strand** para diferentes arquiteturas. A SAM é

dividida em uma componente de redução (versão simplificada de uma implementação de linguagem lógica concorrente), e uma componente de comunicação que trata da entrada/saída. Todos os aspectos da implementação dependentes da máquina destino estão localizados na componente de comunicação. O **Strand** implementado em um só processador executa duas vezes mais rápido que o **Parlog**. O **Strand** foi implementado em diversos multiprocessadores, tanto com memória compartilhada como distribuída.

3.8.4 Implementações que Exploram Simultaneamente os Paralelismos E e OU

Como o objetivo do projeto OPERA é chegar a um modelo que contemple exploração combinada dos paralelismo E e OU na Programação em Lógica, uma avaliação dos principais modelos propostos com esta filosofia foi realizada com o propósito de obter parâmetros para a atual fase do projeto, que investiga a exploração do paralelismo E. Nos últimos anos, foram feitas algumas propostas neste sentido. Neste capítulo são tratadas apenas as mais maduras.

Apesar dos sistemas que exploram somente um tipo de paralelismo terem demonstrado sua possibilidade de ganho na velocidade de execução sobre os sistemas Prolog seqüenciais mais eficientes, alguns programas não têm sua execução substancialmente acelerada nestes sistemas. Nesta situação, estão os programas determinísticos, quando processados em sistemas que exploram paralelismo OU, e os programas com grandes árvores de busca (fortemente não-determinísticos), quando processados pelos sistemas que exploram o paralelismo E. Também neste sentido, enquanto os sistemas que exploram o paralelismo OU e o

paralelismo E Independente conseguem explorar, de forma eficiente, principalmente o paralelismo de elevada granulosidade, os sistemas que trabalham com o paralelismo E Dependente, empregam técnicas que permitem explorar o paralelismo de muito baixa granulosidade, presente em muitas aplicações.

A premissa é a de que o modelo, que fizer uma exploração combinada das diferentes estratégias para o paralelismo na Programação em Lógica, terá maior desempenho do que aquele que empregar somente uma, apesar dos custos de gerenciamento inerentes à integração.

3.8.4.1 Exploração Combinada dos Paralelismos E Independente e OU

Rolog

O sistema **Rolog** implementa o *Reduce OR Process Model* (ROPM) ([KAL 87]), o qual combina paralelismo E Independente com paralelismo OU. Os programas são compilados em uma Máquina Abstrata Prolog baseada na WAM. O sistema é formado por dois entes: o compilador Prolog propriamente dito, e um núcleo (denominado *Chare Kernel*). O compilador é visto como uma aplicação que processa sobre este núcleo. O *Chare Kernel,* por sua vez, é dependente do hardware destino, e engloba todas as funções de mais baixo nível para gerência do paralalelismo, tais como: criação de processos, gerenciamento da memória e distribuição de carga. Esta estratégia permitiu que o **Rolog** fosse portado para diversas arquiteturas paralelas, tanto com memória comum, como distribuída (Encore Multimax, Sequent Symmetry, Intel iPSC/2 Hypercube, Alliant FX/8, dentre outras).

Devido à complexidade do seu modelo de execução, a eficiência do **Rolog** em um único processador (eficiência seqüencial) é várias vezes inferior a de sistemas Prolog paralelos que exploram um só tipo de paralelismo na mesma condição. Porém, quando executando em paralelo, ele apresenta um ganho de velocidade praticamente linear ao número de processadores. Resultados experimentais indicaram que, para diversos programas, significativas reduções do espaço de busca (um dos objetivos do modelo) podem ser obtidas, evitando a recomputação dos ramos E paralelos, quando do retrocesso (*backtracking*).

ACE

O objetivo do modelo **ACE** ([HER 91]) é o de combinar a proposta de exploração do paralelismo E Independente do sistema **&-Prolog**, com a proposta de exploração do paralelismo OU do sistema **Muse**. No **ACE,** nenhum esforço é feito no sentido de reutilizar os resultados das subcomputações E Independentes, que ocorrem nos diferentes ramos OU (como no sistema **Rolog**); seu objetivo é empregar técnicas simples, cuja eficiência já foi demonstrada. Apesar da consistência de sua definição, não existe registro de uma implementação para o **ACE**.

3.8.4.2 Exploração Combinada dos Paralelismos E Dependente e OU

Existem duas linhas principais de pesquisa neste sentido, dependendo da linguagem a ser suportada: uma primeira, baseada no *Modelo Andorra Básico,* suporta a linguagem Prolog, enquanto uma outra, baseada no *Andorra Kernel Language,* integra ao Prolog construtores das linguagens lógicas concorrentes.

Andorra-I

O sistema **Andorra-I** ([YAN 93]) é uma implementação do *Modelo Andorra Básico* em multiprocessadores com memória compartilhada. Este sistema consiste de um compilador, uma Máquina Abstrata Prolog e um escalonador.

O compilador executa uma análise do programa, baseada em técnicas de interpretação abstrata, com o objetivo de determinar, para cada procedimento, o modo dos seus argumentos, isto é, os possíveis tipos de atribuições de valores (fechados, não-fechados, variáveis, etc...). Esta informação é empregada, junto com a análise dos operadores de restrição e *side-effects* (seção 2.2.2.7), para gerar uma seqüência de instruções, a qual é empregada pelo ambiente de execução.

O compilador produz código para a *Andorra-I Abstract Machine*, que é uma extensão da máquina abstrata proposta por Crammond (JAM - seção 3.8.3.1, Parlog).

O ambiente de execução suporta paralelismo OU, empregando técnicas derivadas do sistema **Aurora** (*binding arrays*) (seção 3.8.1.3), e explora paralelismo E Dependente, empregando técnicas provenientes da implementação do **Parlog** (JAM).

A avaliação de desempenho do **Andorra-I** em um multiprocessador com memória compartilhada (Sequent Symmetry) mostrou que o ganho relativo na velocidade de execução explorando somente o paralelismo OU é semelhante ao do sistema **Aurora**, e que o ganho relativo com a exploração do paralelismo E Dependente é semelhante ao do **Parlog**.

Em um só processador, **Andorra-I** processa tão rapidamente quanto a JAM (Máquina Abstrata empregada no **Parlog**) e se mostra, em média, 2,5 vezes mais lento que SICStus Prolog (técnica de compilação seqüencial em que se baseia o sistema **Aurora**). Portanto, em termos absolutos, é aproximadamente duas vezes mais lento que o próprio sistema **Aurora**. Esta perda de desempenho ocorre em função dos custos maiores introduzidos pelos mecanismos para gerenciar a exploração combinada de ambos paralelismos. Quando o paralelismo E Dependente e o paralelismo OU são explorados simultaneamente, o ganho geral na velocidade de execução, dependendo do problema, pode ser bem maior que o obtido com somente um dos tipos de paralelismo.

Extensões do Modelo Andorra Básico

D. H. D. Warren, recentemente, propôs um novo modelo de execução ([WAR 90]) que estende o *Modelo Andorra Básico*, permitindo a execução paralela entre literais não-determinísticos, enquanto eles executam uma computação local, isto é, enquanto não necessitam de variáveis não-locais. Deste modo, literais não-determinísticos são sincronizados (bloqueados), somente quando eles tentam obter o valor de alguma variável externa. Este paralelismo extra contém paralelismo E Independente como subcaso. A extenção do *Modelo Andorra Básico* (*Extended Andorra Model*) combina, portanto, todos os três tipos de paralelismo: paralelismo OU, E Dependente e E Independente.

O modelo **IDIOM** ([GUP 91]), apresenta outra combinação de três tipos de paralelismo. Ele emprega *Conditional Graph Expressions-CGEs* (seção 4.2.2.3), para expressar o paralelismo E Independente como no sistema **&-Prolog**, e a execução acontece como descrito a seguir. Primeiro

ocorre a fase de exploração do paralelismo E Dependente, quando todos os literais determinísticos são avaliados em paralelo. Quando não existirem mais literais determinísticos, o literal mais à esquerda é selecionado para avaliação, se este for uma CGE, então a exploração do paralelismo E Independente é disparada, caso contrário um nodo OU é criado como no *Modelo Andorra Básico*.

Estas duas propostas estão com os trabalhos de implementação em andamento.

O Andorra Kernel Language

O *Andorra Kernel Language*-AKL ([JAN 91]), estende o *Modelo Andorra Básico* pelo acréscimo de alguns construtores das linguagens lógicas concorrentes. A maioria dos programas escritos em Prolog, GHC e Parlog, podem ser automaticamente convertidos para AKL. A semântica do AKL é apresentada em ([HAR 90]), e suas características exigem uma máquina abstrata relativamente simples. Sua implementação paralela está em desenvolvimento, e emprega um mecanismo semelhante ao do sistema **Muse** para administrar o paralelismo OU.

Outros trabalhos, além daquele dos autores, envolvem o AKL. Um primeiro é a proposta de uma nova máquina abstrata, inspirada na JAM (semelhante a implementação do **Andorra-I**), para acomodar as novas construções da AKL ([PAL 91]); um segundo, é a definição de um modelo de execução ([MOO 91]) que explora principalmente o paralelismo OU e emprega *hash-windows* de maneira similar ao **PEPSys**. Acreditam os proponentes, ser o esquema *hash-windows* mais adequado à execução do paralelismo OU no AKL do que o *binding-arrays* (empregado no **Andorra-I**).

3.9 Conclusões

Na seção anterior, as diferentes fontes de paralelismo na Programação em Lógica foram introduzidas, e alguns aspectos inerentes a sua exploração foram analisados.

Foram também discutidas, as principais técnicas para implementação dos paralelismos E e OU.

E por último, foi registrado de forma suscinta, os resultados de um estudo comparativo entre as principais propostas para exploração de paralelismo na Programação em Lógica.

No capítulo seguinte, será apresentada a opção feita para exploração do paralelismo E no projeto OPERA.

4 EXPLORAÇÃO DO PARALELISMO NO PROJETO OPERA

Este capítulo apresenta a proposta de exploração do paralelismo na Programação em Lógica do projeto OPERA . Esta proposta foi apresentada, de forma resumida, no Seminário de Software e Hardware do XII Congresso da Sociedade Brasileira de Computação ([GEY 92]).

Tendo como objetivo uma ampla investigação das alternativas para exploração do paralelismo na Programação em Lógica, e estando consolidada a proposta de exploração do paralalelismo OU, a atual fase do projeto OPERA investiga a exploração do paralelismo E.

Na proposta do paralelismo E foram consideradas as premissas já definidas no projeto, com o objetivo de chegar a um modelo que contemple a exploração combinada dos paralelismos E e OU.

A organização do capítulo contempla primeiramente uma descrição sucinta do paralelismo OU do projeto OPERA, e a seguir a apresentação dos componentes da proposta para exploração do paralelismo E.

4.1 O Paralelismo OU

O objetivo do projeto OPERA com a exploração do paralelismo OU é utilizar o significativo poder computacional, disponível nos multiprocessadores escalonáveis com memória distribuída, para executar, com rapidez, programas Prolog, de forma diferente do proposto na linha de trabalho das linguagens Lógicas Concorrentes ([SHA 89]), onde é

contemplado o desenvolvimento de novas linguagens voltadas ao processamento paralelo.

A proposta de exploração do paralelismo OU Multiseqüencial, adotada pelo projeto OPERA, foi validada com uma implementação no **Supernode**, um multiprocessador com memória distribuída baseado em Transputers ([BRI 90], [GEY 91]).

4.1.1 Características da Linguagem

A linguagem suportada pelo paralelismo OU Multiseqüencial OPERA é o Prolog *padrão*, incluindo *cut* e alguns predicados de entrada/saída. Os predicados *data-base*, que modificam o programa, não foram considerados nesta versão. Não são utilizadas declarações especiais para distinguir predicados paralelos, e as mesmas cláusulas alternativas que promovem o retrocesso (*backtracking*) são geradoras de eventuais paralelismos. Assim, qualquer nodo OU (*choice-point*) pode ser utilizado para criar um novo processo.

4.1.2 O Paralelismo OU Multiseqüencial

4.1.2.1 Descrição

No modelo de paralelismo OU Multiseqüencial do OPERA, cada trabalhador é uma TWAM (*Transputer Warren Abstract Machine*), executando a sua cópia local do programa Prolog em questão. O OPERA, sob paralelismo OU multiseqüencial, executa programas em um nível de

eficiência seqüencial determinado pela implementação da TWAM empregada. Este nível está bastante próximo das melhores implementações seqüenciais de Prolog, baseadas na WAM (seção 2.2.2.8). No melhor dos casos, o ganho na velocidade de execução paralela em relação à seqüencial é quase linear ao número de processadores empregados.

Nenhum paralelismo é disparado pelos trabalhadores ativos. Os trabalhadores inativos solicitam trabalho aos ativos, que dispõem de alternativas de exploração da árvore de metas, ainda não tentadas nos seus nodos OU (*choice-points*). Um trabalhador inativo torna-se ativo, quando recebe uma alternativa para processar.

Quando recebe trabalho de um trabalhador ativo, o inativo copia o estado da TWAM ativa quando da criação do nodo OU que gerou a alternativa. Variáveis criadas antes deste nodo OU mas *ligadas* depois (e por conseqüência instanciadas), precisam ser *desligadas* (vide figura 2.7). Para evitar de recorrer à pilha de instanciações para desligar estas variáveis, todas as ligações são etiquetadas com uma data (vide seção 3.7.2.1).

4.1.2.2 As Estruturas de Dados da TWAM

As pilhas *Local* e *Global* da WAM foram reorganizadas resultando 4 pilhas na TWAM. Os nodos OU têm uma pilha específica na TWAM, ao invés de ficarem junto com os registros de ativação de cláusulas como na pilha *Local* da WAM. A principal vantagem desta solução é reduzir a sincronização entre o processo que faz a exportação e o processo responsável pela interpretação do programa Prolog, a qual aconteceria toda vez que uma alternativa inexplorada fosse exportada. O uso de mecanismo de DMA

(*direct memory access*), disponível no Transputer, possibilita que o processo exportador aproveite os ciclos que a TWAM não acessa a memória, resultando em um paralelismo de baixo nível entre os dois processos, o que aumenta o desempenho do trabalhador como um todo.

Outra estrutura de dados adicional é a *pilha de variáveis*, utilizada para armazenar todas as variáveis Prolog. Cada variável da *pilha de variáveis* é etiquetada com uma "data", a qual é determinada pela atual profundidade do corrente nodo OU da árvore de busca. O agrupamento das variáveis em uma pilha separada aumenta a eficiência da operação de cópia de contexto.

4.1.2.3 A cópia de Pilhas

Quando um trabalhador inativo W_2 (importador) recebe serviço de um ativo W_1 (exportador), acontecem as seguintes operações:

1. A porção da *pilha de variáveis* existente na criação do nodo OU da alternativa não explorada é copiada da pilha de W_1 para a pilha de W_2.

2. Partes das pilhas *Local*, *Global* e de *Trail* de W_1 são copiadas para W_2. Enquanto isto, W_2 testa a data de cada variável copiada na fase 1, para verificar se ela pertence ao ambiente da alternativa que está sendo importada. Variáveis *ligadas* após a criação do nodo OU da alternativa em questão em W_1 são *desligadas* por W_2. O uso de datas permite a W_1 permanecer ativo durante a operação de cópia, não exigindo nenhuma sincronização quando ligando uma variável a um termo. Para

evitar inconsistências, a data inicial de variáveis livres é inicializada com valor infinito. Assim, se uma variável livre é copiada depois de W_1 ter inicializado sua data e antes de ser atualizado seu valor, ela será corretamente *desligada* por W_2.

4.1.2.4 A Cópia Incremental das Pilhas

A maior parte das vezes, quando um trabalhador inativo W_2 torna-se ativo, não é necessário que seja feita uma cópia completa das pilhas do trabalhador W_1, uma vez que W_1 e W_2 compartilham parte da árvore de metas do programa. Neste caso, W_2 retorna ao último nodo OU (*choice-point*) em comum (*Common Choice Point* - CCP), antes de copiar os segmentos das pilhas de W_1 que são mais *jovens* que CCP, e mais *velhos* que o nodo OU WCP (*Work Choice Point*), envolvido nesta distribuição de trabalho (ver figura 4.1).

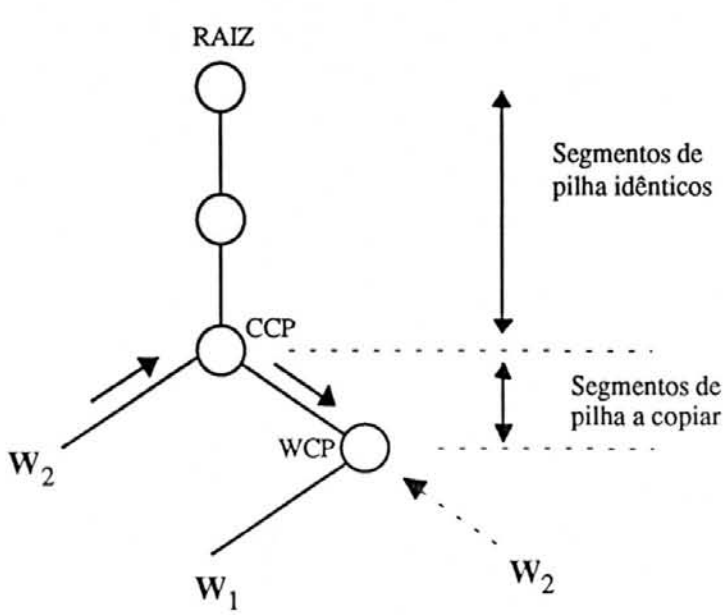

Figura 4.1 Cópia Incremental de Pilhas no OPERA OU

4.1.2.5 O Escalonamento do Trabalho

O objetivo principal de um escalonador é garantir o melhor desempenho possível do sistema paralelo, e para tanto:

- deve manter os trabalhadores tão ocupados quanto possível;
- deve estar atento para a necessidade de limitar o custo computacional adicional (*overhead*), introduzido pela gerência do paralelismo.

A primeira asserção implica em promover ao máximo a execução em paralelo, enquanto que a segunda, condiciona a granulosidade (volume de processamento) de cada nova atividade paralela a um valor que "compense" o custo que ela introduz. O escalonamento em arquiteturas com memória distribuída, é mais complexo que em arquiteturas com memória compartilhada, por diversos fatores, dentre estes: a ausência de um estado global preciso da arquitetura, o custo mais elevado para comunicação entre processadores e a conseqüente necessidade de trabalhar com granulosidades elevadas. Assim, é difícil no OPERA a partir de um certo ponto (o ponto depende do problema), aumentar o número de trabalhadores ativos no multiprocessador, por que a possibilidade de paralelismo da aplicação Prolog com "boa" granulosidade deverá acompanhar esse aumento.

Critérios para o Escalonamento

Devido à complexidade do escalonamento no modelo de paralelismo OU Multiseqüencial do OPERA, significativos recursos computacionais são destinados a esta função. Um trabalhador dedicado ao escalonamento coleta informações sobre a atividade dos outros

trabalhadores, enquanto estes efetuam a computação do programa Prolog. As informações coletadas são relativas às alternativas não processadas da árvore de metas, as quais são utilizadas na distribuição de trabalho aos trabalhadores inativos. Um problema adicional, inerente às arquiteturas com memória distribuída, é que somente um estado aproximado da computação pode ser mantido pelo escalonador, uma vez que a manutenção de um estado exato implica o uso de um mecanismo distribuído de sincronização, com elevado consumo de recursos computacionais (canais de comunicação, processador). Trabalhadores inativos obtém, a partir do escalonador, o endereço de um nodo OU de um trabalhador ativo, que contém uma alternativa não tentada. Dois critérios são utilizados pelo escalonador para selecionar trabalho na árvore de metas:

1. Maximizar o benefício inerente à seleção de um trabalho, o que é equivalente a maximizar a granulosidade deste trabalho. Para atingir esta meta, o escalonador do paralelismo OU multiseqüencial seleciona o trabalho mais alto possível na árvore de busca, segundo a heurística de que o *trabalho mais próximo ao topo tem a maior granulosidade.*

2. Minimizar o *overhead* gerado pela inicialização de um novo trabalho. Isto pode ser obtido selecionando, entre os possíveis trabalhos, aquele que minimize o tamanho dos segmentos de pilha a serem copiados. Assim, por este critério, o trabalho selecionado na árvore de metas do programa, estará tão próximo quanto possível, do estado atual do trabalhador inativo a receber a alternativa (ver figura 4.1).

Estes dois critérios podem ser muitas vezes contraditórios durante a execução , e a definição do melhor compromisso entre os dois é um tema de pesquisa bastante atual no contexto internacional.

4.1.2.6 Situação do Protótipo

O protótipo do OPERA, quando da elaboração do artigo [BRI 90a], se mostrou uma das mais eficientes implementações de Prolog disponíveis para Transputers, particularmente no Supernode, conseguindo um significativo ganho na velocidade de processamento, sobretudo para os problemas com grandes volumes de processamento.

4.2 A Proposta do Paralelismo E

A premissa básica empregada na definição do paralelismo E do projeto OPERA é a sua compatibilidade com a já definida proposta de paralelismo OU Multiseqüencial, visando sua futura integração em um modelo que faça uma exploração combinada de ambos paralelismos. Em linhas gerais isto representa:

- explorar o paralelismo implícito na linguagem Prolog;
- processar em arquiteturas multiprocessadoras sem memória comum;
- empregar uma Máquina Abstrata Prolog (técnica de compilação) passível de integração com a TWAM.

Nesta seção, são apresentadas e discutidas as opções feitas na definição do modelo para exploração do paralelismo E no projeto OPERA.

4.2.1 CARACTERÍSTICAS DA LINGUAGEM

Dentro da premissa de compatibilidade com o modelo já definido para o paralelismo OU Multiseqüencial, na definição do paralelismo E é mantida a proposta de aumentar o desempenho da linguagem Prolog padrão, explorando o paralelismo implícito da linguagem. O fato de manter a linguagem Prolog inalterada possibilita a execução de programas já existentes, sem necessidade de modificações, e também não introduz complexidade no desenvolvimento de programas novos.

Um sistema paralelo ideal deve garantir que não ocorra degradação de desempenho na execução de programas paralelos, em relação aos seus correspondentes seqüenciais. Como, em um primeiro momento, a análise de granulosidade das tarefas durante a execução de programas Prolog é realizada por heurísticas (a análise automática da granulosidade é uma frente de pesquisa em andamento no projeto OPERA), ficam disponíveis ao programador primitivas para a anotação de programas. O objetivo das primitivas é inibir a execução paralela de tarefas com granulosidade muito pequenas e/ou garantir a execução paralela de outras.

Estas primitivas são de uso opcional, estando previstas, inicialmente, somente as construções *SEQ* e PAR. No modelo proposto para paralelismo E, parte-se do pressuposto que uma dada expressão poderá ser processada em paralelo.

O programador, a partir da sua experiência, quando souber ser ineficiente paralelizar determinados subobjetivos, empregará a primitiva SEQ, como por exemplo:

p:- q, SEQ(r,s),t., garante a execução seqüencial de *r* e *s*, *enquanto q* poderá ser executado em paralelo com *SEQ(r,s)* e *t*.

Por outro lado, quando o programador souber da possibilidade de execução paralela de dois objetivos, empregará a primitiva PAR, subtraindo do sistema o custo de fazer esta avaliação, por exemplo:

p :- PAR(q,r),s,l., garante a execução paralela de *q* e *r*.

Outras primitivas poderão ser incluídas no modelo, destinadas a classes específicas de programas, sempre com o objetivo de obter um desempenho melhor.

Pelo fato de as primitivas de anotação de programas serem de uso opcional, mantém-se as vantagens citadas anteriormente, ou seja, a execução de programas já existentes continua sendo possível sem alterações, e o desenvolvimento de novos programas não se torna mais complexo.

4.2.2 PARALELISMO E RESTRITO (RAP)

O modelo do paralelismo E do projeto OPERA é fundamentado no *Restricted And-Paralelism-RAP* ([DEG 84], [DEG 87]). As razões desta opção são as seguintes características do RAP:

- permitir a exploração do paralelismo implícito na linguagem Prolog;

- ser passível de exploração mesmo na presença do não-determinismo *don't-know;*

- ter sua proposta de detecção e gerência do paralelismo dividida entre a compilação e a execução, o que minimiza as comunicações entre processadores na administração da execução paralela. Este aspecto tem muito significado no âmbito do projeto OPERA, uma vez que este trabalha com arquiteturas multiprocessadoras sem memória comum.

Para a avaliação de dependências de dados entre objetivos no corpo de uma cláusula, DeGroot preocupou-se em encontrar uma solução entre as abordagens onde a avaliação é somente feita em tempo de execução (ex.: [CON 85]), e aquelas em que esta ocorre exclusivamente em tempo de compilação (ex.: [CHA 85]).

Como explora o paralelismo E Restrito, o modelo de paralelismo E do projeto OPERA, executa em paralelo os literais de uma cláusula que não apresentem variáveis comuns, bem como aqueles que contenham estruturas e/ou listas sem variáveis.

Quando os literais compartilharem variáveis, estes poderão ser executados em paralelo somente se, no momento da execução, as variáveis comuns estiverem instanciadas com valor definido.

Os literais que contiverem estruturas e/ou listas com variáveis são executados seqüencialmente, em função do custo (que poderá ser elevado,

dependendo do tamanho da estrutura e/ou lista) para verificação de eventuais dependências. A execução paralela destes literais somente ocorrerá se o usuário especificar que assim deseja com o uso da primitiva *PAR*.

A detecção do paralelismo E Restrito no projeto OPERA ocorre a partir de três mecanismos básicos: uma rotina para atribuição de tipos, uma rotina para determinação de independência e as expressões condicionais.

4.2.2.1 Rotina Para Atribuição de Tipos

Esta rotina (*typing algorithm*) monitora o estado dos termos durante a execução, atribuindo-lhes um tipo de acordo com os seguintes critérios:

- **tipo G**: se o termo não contiver variáveis;
- **tipo V**: se o termo for uma variável livre;
- **tipo N**: se o termo não for uma variável, mas contiver ao menos uma variável.

Durante a execução, termos do tipo V e N podem ser alterados, pois as variáveis podem ser instanciadas com valores. Termos do tipo V podem passar a ser do tipo N ou G; termos do tipo N podem assumir o tipo G; e os termos do tipo G não variam.

A rotina de atribuição de tipos está concebida com a preocupação de introduzir o menor custo computacional (*overhead*), por isto realiza uma avaliação parcial dos conteúdos dos termos. Deste modo, pode acontecer de

ser atribuído um tipo não apropriado a um termo, por exemplo, pode ser atribuído tipo N a um termo que na verdade poderia ser do tipo G. Naturalmente, o ideal seria uma rotina de atribuição de tipos precisa, porém o seu custo computacional a torna impraticável. A precisão na atribuição de tipos é um compromisso entre o custo computacional para obte-la, e o ganho com melhor exploração do paralelismo.

4.2.2.2 Rotina Para Determinação de Independência

Analisando somente a cláusula f(X) ← g(X), h(X), k(X)., um compilador assumiria que os literais que a compõem são interdependentes. Contudo se, em tempo de execução, for verificado pela rotina para determinação de independência (*Independence algorithm*) que o parâmetro de **f** é fechado (tipo G), **g**, **h** e **k** poderão ser executados em paralelo, caso contrário é assumida sua execução seqüencial. Contudo, após a execução de g(X) pode ser novamente verificado o tipo de X; se for fechado **h** e **k**, poderão ser executados em paralelo.

Por sua vez, para a cláusula f(X,Y) ← g(X), h(Y)., o compilador entenderá g(X) e h(Y) como literais potencialmente independentes. Porém as variáveis diferentes (X e Y) podem ser instanciadas com uma variável comum; assim sendo, a nível de execução é preciso uma avaliação desta independência.

O algoritmo utilizado para testar a independência de dois parâmetros é mostrado na figura 4.2. A avaliação de independência para mais que duas variáveis ocorre de forma similar, mas envolve $O(n^2)$ testes, onde **n** é o número de argumentos que aparece na cabeça da cláusula. Como normalmente **n** é pequeno e o algoritmo é bastante simples, o custo

computacional da rotina para determinação de independência é bastante baixo.

```
Se ( tipo(ARG1) = G ) ou ( tipo(ARG2) = G )
    ENTAO Independentes
    SENÃO Se ( tipo(ARG1) = tipo(ARG2) = V ) e
                ( Endereço(ARG1) <> Endereço(ARG2) )
                ENTAO Independentes
                SENÃO Dependentes;
```

Figura 4.2 Rotina Para Determinação de Independência

4.2.2.3 Expressões Condicionais

Formação

As expressões condicionais (*Conditional Graph Expressions - CGEs*), são montadas na fase de geração de código, e definidas recursivamente a partir de seis tipos de primitivas, definidas como segue:

- **G** - chamada de procedimento;

- **(SEQ E_1 ... E_n)** - indica que as expressões E_1 ... E_n são executadas seqüencialmente, nesta ordem. *SEQ* é empregado quando algum argumento da expressão é uma estrutura com variáveis ou no caso de uma anotação explícita do usuário;

- **(PAR E_1 ... E_n)** - indica que as expressões E_1 ... E_n são executadas em paralelo. É usada quando as E_1 ... E_n

envolverem *strings*, inteiros ou estruturas sem variáveis (fechadas);

- **(GPAR(X_1 ... X_k) E_1 ... E_n)** - indica que, se todos os argumentos X_1 ... X_k da chamada a GPAR são termos fechados, então as expressões E_1 ... E_n são executadas em paralelo; caso contrário, a execução ocorre de forma seqüencial. Esta primitiva é usada quando há dependência de variáveis entre literais, observando-se que estas variáveis podem estar fechadas em tempo de execução. Exemplo: *p(X) :- q(X), r(X).*;

- **(IPAR(X_1 ... X_k) E_1 ... E_n)** - indica que, se todos os argumentos X_1 ... X_k da chamada a IPAR são mutuamente independentes, então as expressões E_1 ... E_n são executadas em paralelo, caso contrário, a execução ocorre de forma seqüencial. Esta primitiva é empregada quando não for possível concluir, em tempo de compilação, se as variáveis dos literais são dependentes ou independentes, pois os literais envolvidos não têm variáveis comuns. Exemplo: *p(X,Y):- q(X),r(Y).*;

- **(IF E_1 E_2 E_3)** - a expressão E_1 é avaliada e, se esta for verdadeira, E_2 é executada; caso contrário, E_3 é executada. Esta primitiva tem como função evitar testes redundantes. Por exemplo, considerando a cláusula *f(X,Y):- p(X),q(Y),s(X,Y),t(Y).*, o compilador poderia produzir a seguinte CGE:

```
( GPAR(X,Y)
    (IPAR(X,Y) p(X) q(Y))
    (GPAR(Y) s(X,Y) t(Y)) ).
```

Se *GPAR(X,Y)* for verdade, o teste *GPAR(Y)* é redundante. Para evitar esta situação, utilizando a primitiva IF, poderia ser construída a CGE apresentada na figura 4.3:

```
( IF (GPAR(X,Y)
    (PAR(p(X) q(Y) s(X,Y) t(Y)))
    (SEQ((IPAR(X,Y) p(X) q(Y)) (GPAR(Y) s(X,Y) t(Y)))) ).
```

Figura 4.3 Uso da Primitiva IF

A combinação destas primitivas para a representação das CGEs forma um programa, sendo que é construída uma CGE por cláusula. Os testes que as primitivas realizam em tempo de execução são muito simples, introduzindo um custo computacional mínimo.

Considerações sobre a execução

Considere a cláusula f(X) ← g(X), h(X), k(X)., cuja possibilidade de execução paralela já foi avaliada em 4.2.2.2. A compilação desta cláusula irá gerar a seguinte *CGE*:

```
f(X) :- (GPAR(X) g(X)
         (GPAR(X) h(X) k(X))).
```

A execução desta *CGE* pode ocorrer como em qualquer dos três grafos da figura 4.4 (este grafos expressam as considerações feitas em 4.2.2.2).

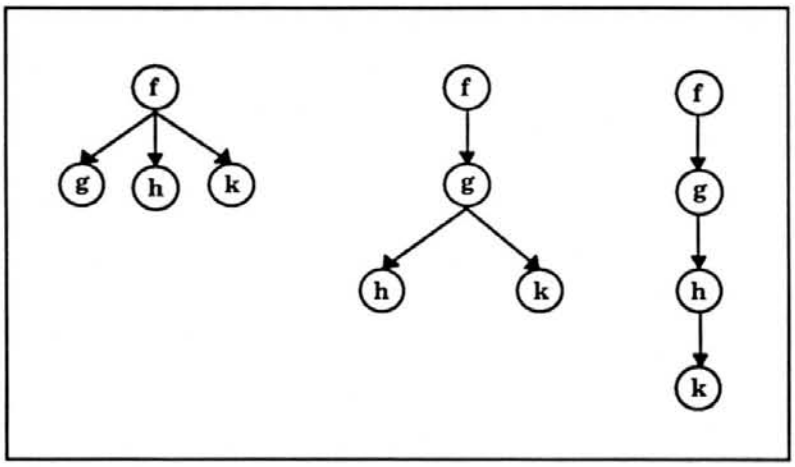

Figura 4.4 Grafos de Execução para f(x) :- g(x), h(x), k(x).

Outro exemplo, caracterizando limitações emprega a CGE da figura 4.3.

Para a CGE da figura 4.3 se X e Y são independentes no momento da execução, mas não *fechados* (não instanciados com valor constante), executando a CGE teremos o grafo da figura 4.7. Se compararmos com a proposta de Conery (seção 3.7.3.1), ela produzirá este mesmo grafo, porém, tão logo q(Y) seja avaliado, t(Y) pode começar independentemente de p(X) já estar avaliado ou não. Para a CGE gerada, t(Y) somente será executada após estar terminada a avaliação de q(Y) e p(X). Esta perda de paralelismo é devida à limitação da CGE em expressar a relação entre os objetivos, e não como resultado da técnica de aproximação empregada pelo algoritmo de tipagem (seção 4.2.2.1). Conseqüentemente, há dois meios pelos quais o paralelismo E de uma cláusula pode ser restringido: pela imprecisão do algoritmo de tipagem e pela limitação das CGEs. Porém, mesmo com a perda da máxima detecção de paralelismo, devido ao algoritmo de *tipagem* empregado, ou como resultado da limitação das primitivas disponíveis para formação das CGEs, é suposto ser o paralelismo detectado, suficiente para manter um sistema de processamento paralelo de tamanho médio (25 processadores) ocupado (uma evidência empírica neste sentido é o trabalho

de Carlton e Van Roy que implementaram uma versão distribuída do emulador C-Prolog [CAR 88]. Registre-se também que esta não detecção de toda possibilidade de paralelismo, tem como retorno um compilador rápido e rotinas de detecção e controle do paralelismo para tempo de execução eficientes.

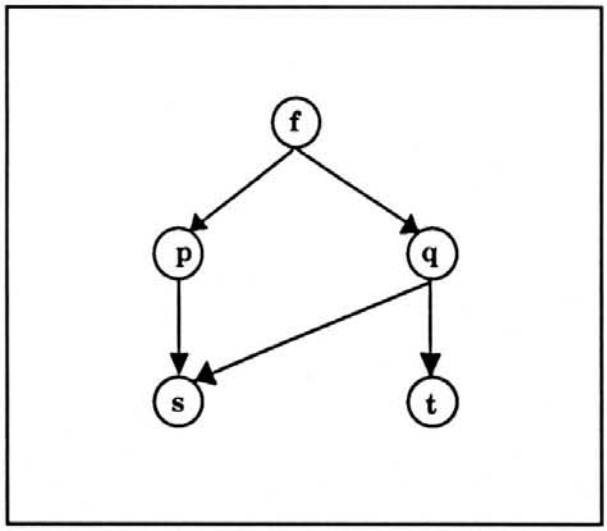

Figura 4.7 Exemplo de limitação das CGEs

4.2.3 O Modelo Computacional

No modelo computacional do paralelismo E do projeto OPERA, cada trabalhador tem um acesso rápido ao código do programa. Na atual proposta, voltada para multiprocessadores sem memória compartilhada, o programa está disponível na memória local de cada processador onde ocorrerá a execução paralela, ou seja, em todo processador que alojar um trabalhador OPERA.

A proposta de modelo é baseada em quatro preceitos básicos:

- o processamento começa com uma consulta do usuário sendo atendida por um trabalhador escolhido como principal;

- quando todos os trabalhadores do sistema estiverem ocupados não ocorrerá distribuição de trabalho;

- um trabalhador receberá literais (trabalho) para execução somente quando se alistar no escalonador como disponível. Isto ocorrerá quando este trabalhador estiver sem trabalho (*idle*) ou seus trabalhos estiverem suspensos (aguardando um resultado);

- será evitada a distribuição de objetivos que exijam pouco processamento (baixa granulosidade - vide seção 4.2.1).

Considerando que na proposta de paralelismo E do modelo OPERA não existe avaliação de granulosidade das tarefas, e com a premissa de não ser produtivo em função disto distribuir trabalho para trabalhadores que estejam ocupados, acredita-se não ser comprometedor se alguma possibilidade de paralelismo não for detectada.

Cada trabalhador mantém duas pilhas com expressões: uma pilha seqüencial e uma pilha paralela. A pilha seqüencial contém expressões que devem ser (ou que foi decidido serem), executadas seqüencialmente; por sua vez, a pilha paralela contém expressões que devem ser executadas em paralelo. Quando a consulta do usuário é lida ela é convertida em uma expressão condicional de execução (CGE) e colocada na pilha seqüencial do trabalhador definido como principal. Após receber trabalho, cada trabalhador verifica o topo da sua pilha seqüencial para obter sua próxima peça de trabalho. Se a expressão no topo da pilha seqüencial é um objetivo,

este é executado do mesmo modo que no Prolog seqüencial. Se for uma expressão IPAR ou GPAR, o respectivo teste é executado, e se o resultado for positivo, os respectivos literais são colocados na pilha paralela. Se o resultado é falso, os objetivos são todos colocados na pilha seqüencial. Uma expressão SEQ coloca sua lista de objetivos diretamente na pilha seqüencial, enquanto a expressão PAR coloca sua lista de objetivos na pilha paralela.

Se a pilha seqüencial esvaziar, o trabalhador retirará expressões de sua pilha paralela para execução. Se a pilha paralela também estiver vazia, após tratar os resultados obtidos, o trabalhador se alistará no escalonador como disponível (*idle*) para ajudar outro trabalhador.

Se existirem trabalhadores com serviço paralelo, em volume que justifique exportação de trabalho (vide seção 4.2.6.2) e trabalhadores sem trabalho, o escalonador agenciará a comunicação entre estes. Uma das entradas da pilha paralela de um trabalhador ocupado será transferida para um desocupado. A execução dos objetivos paralelos, como são independentes, pode ocorrer em qualquer ordem.

4.2.4 A Máquina Abstrata Prolog do Modelo E

Esta seção tem por objetivo apresentar, em linhas gerais, a Máquina Abstrata Prolog projetada para exploração do paralelismo E no Projeto OPERA, a WAM-E. São descritas suas estruturas, o seu conjunto de instruções e as ações que tratam as mensagens inerentes à exploração do paralelismo. Outros aspectos pertinentes a WAM-E, podem ser encontrados em [WER 94], onde é tratado, com mais especificidade, o tema compilação na proposta de paralelismo E para o projeto OPERA.

A proposta desta Máquina Abstrata Prolog considerou sua futura integração com a já definida para o paralelismo OU multiseqüencial. A WAM-E, assim como a TWAM, também é baseada na WAM. Esta mesma "origem" de ambas é fator determinante na integração futura das duas propostas. A figura 4.6 apresenta a estrutura da WAM-E.

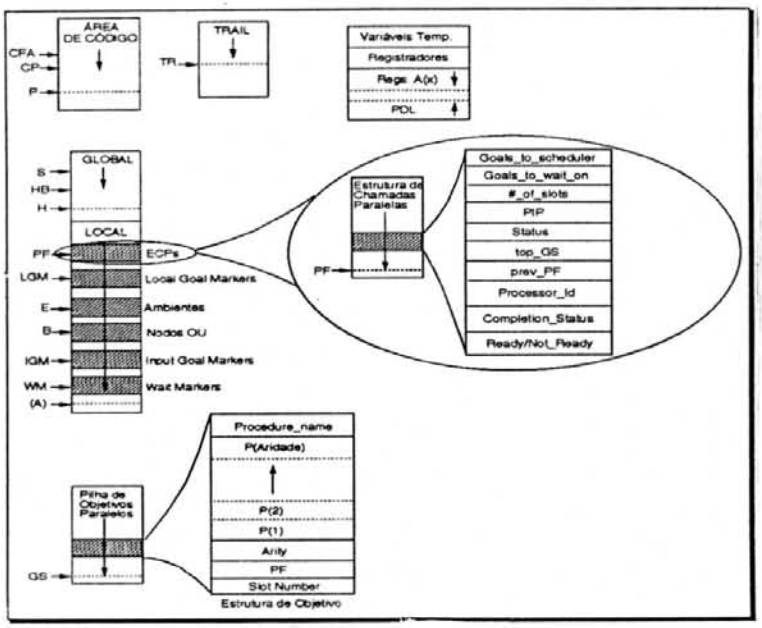

Figura 4.6 Área de Dados e Registradores da WAM-E

4.2.4.1 As Novas Estruturas

Tendo como referência a WAM (seção 2.2.2.8), a seguir estão descritas as duas novas estruturas da Máquina Abstrata Prolog proposta para exploração do paralelismo E.

Pilha de Objetivos Paralelos (*Goal Stack*)

A Pilha de Objetivos Paralelos (POP), contém os objetivos prontos para serem executados em paralelo. Cada entrada nesta pilha é denominada Estrutura de Objetivos e é composta pelas seguintes informações:

- **Procedure_name**: referencia a primeira instrução do procedimento a ser executado, sendo este endereço necessário ao início da execução do dado procedimento;

- **Registradores** *P1, ..., Pn*: são uma cópia dos *n* registradores de argumentos do procedimento a ser executado, sendo que estes dados são necessários quando da chamada de um procedimento;

- **Número de Parâmetros**: contém a aridade do procedimento;

- **Parcall Frame Pointer (PF)**: referencia a Estrutura de Chamadas Paralelas (vide especificação a seguir) a qual o objetivo em questão corresponde;

- **Número do Slot**: identifica o *slot* da Estrutura de Chamadas Paralelas que corresponde ao objetivo em questão.

A WAM-E também apresenta um novo registrador, chamado *GS*. Este registrador aponta sempre para o topo da POP, ou seja, aponta para o primeiro objetivo candidato à exportação. Todos os objetivos de uma chamada paralela são empilhados nesta estrutura, podendo ser desempilhados através de uma requisição local (caso o trabalhador tenha esgotado seu trabalho seqüencial) ou remota (correspondente a um pedido de exportação).

Para a execução destes objetivos, são necessárias somente a cópia dos registradores de parâmetros para os registradores de argumentos e a carga do registrador *P* (este registrador referência a próxima instrução a ser executada - vide figura 4.6), com o valor do campo *Procedure_name* da POP.

O registrador *GS* tem, ainda, outra função. Ele é o valor informado ao escalonador para a estimativa de trabalho paralelo na MAP em questão. Esta estimativa é importante na escolha da MAP exportadora, ou seja, o trabalhador que irá ceder trabalho a um trabalhador desocupado, sendo que um dos critérios é o número de entradas na POP.

Quando ocorrer uma exportação de serviço, o trabalhador que cede o literal para execução remota fica na condição de trabalhador-pai daquele que importa, o qual, a partir de então, é visto como trabalhador-filho.

Estrutura de Chamadas Paralelas (*Parcall Frame*)

A Estrutura de Chamadas Paralelas (ECP) está inserida na pilha *Local* e controla a execução paralela de objetivos e as ações tomadas. Ela é necessária devido à ocorrência de retrocesso, pois as entradas na POP, correspondentes a esses objetivos, desaparecem assim que algum trabalhador-filho os importa.

Como é necessário manter algumas informações sobre estes objetivos, é criada uma entrada na ECP a cada chamada paralela, sendo esta formada por uma parte fixa, que diz respeito à chamada paralela como um todo, e uma parte variável, que contém informações individuais sobre cada objetivo desta chamada. O registrador *PF* aponta para o topo da ECP criada.

A parte fixa apresenta as seguintes informações:

- **Número de objetivos a serem escalonados (Goals_to_schedule)**: este campo é inicializado com o número de objetivos a serem executados em paralelo e é decrementado cada vez que um processador (local ou remoto) toma um objetivo da Pilha de Objetivos Paralelos;

- **Número de objetivos sendo executados (Goals_to_wait_on)**: este campo informa o número de objetivos que estão executando e não retornaram resposta ainda, para que o trabalhador saiba quando todos os objetivos da chamada paralela, ou seja, da CGE, terminaram de executar. Seu valor é incrementado quando um processador rouba um objetivo da POP e é decrementado quando um processador envia uma mensagem que identifica o final da execução do objetivo;

- **Número total de slots (#_of_slots)**: informa o número de slots que existem nesta entrada da ECP, determinando o total de objetivos da CGE;

- **Put Instructions Pointer (PIP)**: este campo contém o endereço da primeira instrução do primeiro objetivo da chamada paralela, sendo usado para reiniciar o empilhamento de objetivos na Pilha de Objetivos Paralelos depois do retrocesso;

- **Status**: determina se a execução da chamada paralela correspondente a esta ECP já foi completada uma vez (*Status* =

outside) ou se esta é a primeira passagem (*Status* = *inside*). Esta informação é utilizada no processamento do retrocesso. A instrução *allocate_pcall* inicializa este campo com o valor INSIDE e a instrução *wait_on_siblings* modifica-o para OUTSIDE;

- **top_GS**: contém o topo da POP no momento da chamada paralela. Tal valor é também utilizado na operação de retrocesso;

- **prev_PF**: contém o valor de PF antes da criação da entrada atual na ECP. Este valor é usado pela instrução *wait_on_siblings* para reinicializar o registrador PF depois de sair da chamada paralela.

- Por sua vez a parte variável contém as seguintes informações por *slot*:

- **Processor_id (P_id)**: contém a identificação do processador que está executando ou já executou o objetivo correspondente. Tal informação é necessária ao trabalhador pai, no momento deste enviar uma mensagem ao filho;

- **Completion Status (Comp_Status)**: este campo é inicializado com valor nulo, sendo alterado quando do retorno da execução de um objetivo, informando um de três estados: falha, sucesso sem alternativas, sucesso com alternativas;

- **Ready/Not_Ready**: é um campo de um bit, usado para selecionar os objetivos que serão empilhados na POP. Quando é

adicionada uma entrada à Estrutura de Chamadas Paralelas, todos os *slots* assumem o estado *Ready*, mudando para *Not_Ready* assim que forem escalonados. Este campo é verificado quando da retomada da execução depois da ocorrência de um retrocesso, pois somente alguns dos objetivos da chamada paralela necessitarão ser novamente escalonados;

Marcadores

A WAM modificada para o paralelismo E acrescenta à pilha *Local* três tipos de marcadores, que são descritos a seguir.

- **Local Goal Marker (LGM)**

  Um *Local Goal Marker* é empilhado na *Local,* sempre que um trabalhador tomar um objetivo da sua própria pilha de objetivos paralelos (POP), usando a instrução *pop_pending_goal*, descrita mais adiante. Este marcador assinala que um objetivo da POP foi executado localmente, e é empregado para atualizar as informações pertinentes ao mesmo quando do término da sua execução. O registrador LGM aponta para o final da cadeia de LGM's.

- **Input Goal Marker (IGM)**

  Um *Input Goal Marker* é empilhado na *Local* da máquina-filha, sempre que um trabalhador roubar um objetivo da POP de outro trabalhador. Assim, IGM's marcam a separação entre diferentes seções de pilha, correspondentes à execução de

diferentes objetivos roubados de outros trabalhadores. O registrador IGM aponta para o final da cadeia de IGM's.

- **Wait Markers** (WM)

  Um *Wait Marker* é empilhado na *Local* quando ocorrer uma saída com sucesso da chamada paralela. Um registrador chamado WM aponta para o último *Wait Marker*.

4.2.4.2 As Novas Instruções

O conjunto estendido de instruções definido para a MAP do modelo E divide-se em três grupos: Instruções de Verificação de Estado, Instruções de Escalonamento de Objetivos e Instruções de Controle.

Instruções de Verificação de Estado

As Instruções de Verificação de Estado, são usadas para codificar as condições de uma CGE, sendo executados dois tipos de verificações: se termos são *fechados* e se são *independentes*.

- **check_me_else** ***Label***

  Esta instrução tem a função de carregar o registrador *Check Failure Address* (CFA) com o endereço identificado por *Label*. Este endereço (que aponta para o trecho de código seqüencial), é empregado quando os testes em tempo de execução especificados na CGE não obtiverem sucesso, sendo necessário, então, a execução do código seqüencial correspondente.

- **check_ground** V_n

  Esta instrução verifica se o conteúdo do registrador V_n é fechado. Se assim for, continua com a próxima instrução; caso contrário, desvia para o endereço contido no registrador CFA, que corresponde ao código seqüencial.

- **check_independent** V_n, V_m

  Se os registradores V_n e V_m forem independentes (segundo o algoritmo descrito na seção 4.2.2.2), é executada a próxima instrução; caso contrário, ocorre um desvio para o endereço referenciado pelo registrador CFA.

Instruções de Escalonamento de Objetivos

As Instruções de Escalonamento de Objetivos são empregadas para empilhar objetivos e seus argumentos na POP, e se necessário, para desempilhá-los no trabalhador local.

- **push_call Procedure_name / Arity, Slot #**

  Esta instrução é usada para empilhar na POP objetivos que podem ser executados em paralelo. O algoritmo executado é o seguinte:

```
Inibe as exportações de objetivos (nãopodem ocorrer
    exportações enquanto estão sendo empilhados
    objetivos)
Empilha na POP:
    Procedure_name,
    registradores Aarity,      Aarity-1, ..., A1,
    Slot_#,
```

PF atual
Adiciona 1 ao total de objetivos paralelizáveis do trabalhador
Libera exportações de objetivos

- **pop_pending_goal**

Esta instrução é usada pelo trabalhador local para desempilhar objetivos da sua própria POP, visando executá-los localmente. O algoritmo executado é o seguinte:

Se (Número de objetivos a serem escalonados na ECP ≠ 0)
então:
Cria um LGM na *Local*
Atualiza ponteiro LGM
Marca slot da PF (correspondente ao objetivo desempilhado) como LOCAL
Decrementa Número de objetivos a serem escalonados
Transfere os Registradores de argumento do objetivo
P = Procedure_name
CP = LGM

Quando o objetivo tiver sucesso, a instrução *proceed* detectará o sucesso de um objetivo local (CP apontando para fora da *Program Area*), e atualizará o slot da PF (um ponteiro para este é armazenado no LGM). Assim, outros objetivos pendentes serão também desempilhados da POP. Esse processo continua até que não haja mais objetivos (Número de objetivos a serem escalonados = 0). A próxima instrução é, então, executada.

Instruções de Controle

As Instruções de Controle são utilizadas para controlar uma chamada paralela: criação e deleção de *ECPs*, seleção de objetivos a serem escalonados e espera por resultados dos filhos.

- **allocate_pcall** ***#_of_slots, M***

  Esta instrução cria uma entrada na *ECP* e inicializa-a. O parâmetro *M*, que corresponde ao número de variáveis permanentes ainda necessárias no ambiente, é usado para melhorar a ocupação de espaço na *Local*. O algoritmo executado é o seguinte:

```
CPF = PF
Empilha na PF: uma célula para cada objetivo
        (P_id = NULL, CompStatus = NULL,
        Ready/Not_Ready = Ready)
        Goals_to_schedule = #_of_slots
        Goals_to_wait_on = 0
        PIP (aponta para primeira instrução da seqüência
                "check/put/push_call")
        Status = INSIDE
        GS
```

- **check_ready Slot_#, Label**

  Esta instrução é útil durante a operação de retrocesso, quando somente alguns dos objetivos de uma chamada paralela precisarão ser reinicializados depois de uma falha. Esta instrução não permite que os objetivos cujos *slots* estiverem marcados como *NotReady* sejam empilhados na Pilha de

Objetivos Paralelos. O parâmetro *slot_#* identifica o número do *slot* correspondente ao objetivo em questão, por sua vez o parâmetro *Label* indica o endereço da instrução *pop_pending_goal*, a qual é seguida de uma instrução *wait_on_siblings*.

O algoritmo executado é o seguinte:

```
Verifica o slot indicado por Slot_# na ECP atual
Se (Ready/Not_Ready == NOT_READY)
        então desvie para Label
          senão assinala slot para NOT_READY, e executa
            a próxima instrução (que executa o
            empilhamento de objetivos na POP)
```

- **wait_on_siblings**

Esta instrução é executada na saída com sucesso de uma chamada paralela, e é utilizada para detectar o momento em que todos os objetivos da CGE terminaram de executar, bem como para salvar informações importantes, que podem ser necessárias quando da ocorrência de um retrocesso. O algoritmo executado é o seguinte:

```
Detecta quando o momento em que o número de objetivos
      em execução = 0
Empilha um WM na Local, salvando PF atual em BPF
Restaura PF (PF = Prev_PF)
Se Status = INSIDE
      então STATUS = OUTSIDE
```

Instruções Modificadas

- **proceed**

A instrução *proceed* do conjunto de instruções da WAM precisa ser modificada para detectar sucesso na execução de objetivos e, para avisar o pai, deste acontecimento. O algoritmo executado é o seguinte:

```
Se CP aponta para Area de Código
então P = CP
senão Se (CP = IGM) (objetivo remoto obteve sucesso)
        então Lê PF e Slot # do IGM
                Envia mensagem de sucesso ao pai
        senão Se (CP = LGM) (objetivo local obteve
sucesso)
                então Lê PF e Slot # do LGM
                Atualiza este slot com sucesso (com ou sem
                        alternativas)
                Lê PIP do PF
                P = PIP (execução continua na seqüência

        "put/push_call/pop_pending_goal")
```

Nota: Um objetivo não tem alternativas quando (PF < IGM) e (B < IGM).

4.2.4.3 Ações

As ações constituem o tratamento dado às mensagens que chegam de trabalhadores remotos e precisam ser atendidas com prioridade.

- **failure**

Esta ação corresponde ao tratamento dado a uma mensagem de falha local (originada no trabalhador local) ou remota

(originada em outro trabalhador e recebida pelo trabalhador, que é pai deste). Tanto para a falha local como remota, o trabalhador recupera o contexto antes da execução do objetivo. No caso de falha remota o trabalhador-pai enviará mensagem de *kill* a todos os filhos que estiverem trabalhando na cláusula do objetivo que falhou.

- **success**

Quando um trabalho que foi importado termina sua execução com sucesso, o trabalhador que o resolveu envia uma mensagem de sucesso ao trabalhador exportador. Quando um trabalhador recebe esta mensagem, a ação de *success* decrementa o número de respostas que ele está aguardando dos filhos e muda o estado do objetivo, em questão, para "sucesso com alternativas" ou "sucesso sem alternativas", caso o objetivo tenha ou não alternativas a serem tentadas. Um trabalhador somente envia o resultado de seu trabalho ao pai, quando este solicitar (através de uma mensagem de *solution_request* ou *final_solution_request* - definidas a seguir-)

- **kill**

Esta ação é disparada por uma mensagem de mesmo nome. Esta mensagem parte de um trabalhador-pai com destino a um filho, e indica que o objetivo que está sendo executado no *processador-filho* não é mais útil e deve ser descartado. O procedimento executado é o seguinte:

A cadeia de *ECPs*/Wait Markers é seguida enviando-se mensagens de *unwind* (ou *kill*, se o objetivo que não é mais útil estiver sendo executado) a todos os trabalhadores-filhos.
É efetuada uma operação de *Unwind* na pilha *Trail* até que não haja nenhuma *ECP* acima do IGM atual.
Os ponteiros para áreas de dados são restaurados (do IGM) e o trabalhador retorna ao estado de inatividade.

- **unwind**

Quando um trabalhador-pai estiver realizando retrocesso, uma mensagem de *unwind* é enviada por ele aos seus filhos. Esta mensagem dispara uma ação de mesmo nome, equivalente a uma ação *kill*, exceto que, neste momento, o trabalhador não está executando um objetivo que não é mais útil.

Todos os trabalhadores-filho são descartados até, e inclusive, o IGM atual.

- **redo**

Uma mensagem de *redo* é enviada pelo *processador-pai* ao filho, depois que este reportou estado de solução com alternativas. A ação de *redo* disparada por esta mensagem, é equivalente ao tratamento de uma falha local, descrito anteriormente.

- **import**

A ação *import* é equivalente à instrução *pop_pending_goal*, mas aplicado a um objetivo remoto (desempilha um objetivo da

POP da máquina do trabalhador exportador). Ela efetua o tratamento de uma autorização (mensagem) de importação enviada pelo escalonador do sistema ao trabalhador importador. O algoritmo executado é o seguinte:

Recebe o trabalho a ser executado
Cria um IGM na *Local*
Carrega registradores com argumentos importados (valores para argumentos *ground* ou endereços para argumentos livres)
Armazena os ponteiros para as áreas de dados no IGM criado
P = "procedure_name"
CP = IGM

Quando este objetivo obtiver sucesso, a instrução *proceed* detecta que o valor no ponteiro de próxima instrução (CP) é um IGM e reporta sucesso ao trabalhador-pai. O trabalhador-pai ao receber uma mensagem de sucesso atualiza a sua *ECP*.

- **export**

A mensagem de exportação é enviada pelo escalonador ao trabalhador exportador provocando o disparo da ação export na máquina deste. O algoritmo executado está descrito a seguir.

Atualiza o *slot* correspondente ao trabalho exportado com o *Pid* do trabalhador importador (máquina-importadora)
Atualiza o *status* do trabalho em questão, (campo Comp_Status) denotando que este está em execução
Decrementa o número de objetivos a serem escalonados
Incrementa o número de objetivos que a trab. exportador (máquina-pai) precisa aguardar
Decrementa um (1) do total de objetivos paralelizáveis do trabalhador

Lê na POP os argumentos necessários à execução do objetivo
Lê o deslocamento no código, correspondente a primeira instrução a ser executada na máquina-importadora
Envia à máquina-importadora os parâmetros do objetivo a ser executado

- **close**

Esta ação ocorre quando o usuário determina o encerramento da arqutitetura de processos. A sua execução acontece, a partir de mensagem proveniente do processo *mestre* (vide item 4.2.6.1) a todos os trabalhadores da arquitetura. É, neste instante, que o trabalhador libera os semáforos e as áreas de memória compartilhada alocados durante o processamento, e também. encerra sua execução.

- **new_application**

Esta ação é disparada por uma mensagem de mesmo nome, proveniente do processo que gerencia a execução paralela (processo *mestre* - vide item 4.2.6.1). Esta ação prepara o trabalhador para a execução de uma nova aplicação, ou seja, para o processamento de um novo programa Prolog. A partir desta mensagem, o trabalhador aguarda a chegada dos parâmetros pertinentes a execução da nova aplicação. Estes parâmetros tambem regulam a dinâmica da execução paralela (vide ACT - item 5.5.1.1).

- **solution_request**

A ação *solution_request* é responsável por responder à requisição de resultados originada no trabalhador pai, sendo executada no trabalhador filho.

A instrução *wait_on_siblings*, além de guardar informações necessárias quando da ocorrência de um retrocesso, também detecta o momento em que todos os filhos retornaram sucesso ao pai (goals_to_wait_on = 0). A partir deste momento, o trabalhador pai inicia a busca de soluções dos trabalhadores filhos (através do procedimento *get_children_results*). O trabalhador pai envia uma mensagem de *solution_request* a cada filho e aguarda pela resposta dos mesmos. O trabalhador-filho através da ação *solution_request* enviará ao pai os endereços de destino e os valores das variáveis que foram por ele ligadas.

- **final_solution_request**

Da mesma forma que a *solution_request*, a ação *final_solution_request* tem a função de atender à solicitação de resultados. A diferença está no fato de que, neste caso, a requisição parte do processo que gerencia a execução paralela (processo *mestre*) e é destinada ao trabalhador principal. Se trata do pedido do resultado final do processamento, enquanto que a *solution_request* é uma requisição de resultados parciais, feita por um trabalhador-pai para seus filhos.

4.2.5 A ARQUITETURA DE PROCESSOS

A arquitetura proposta para exploração do paralelismo E no projeto OPERA tem dois tipos básicos de agentes: ***Mestre*** e ***Trabalhador***, (vide figura 4.7). O processo *Mestre* é responsável pela gerência da arquitetura. Uma vez instalado em um dos processadores, ele controla o ambiente de execução, criando trabalhadores e escalonando serviços. Cada trabalhador, por sua vez, é formado por três processos, a saber:

- ***Solver***: processo responsável pela computação dos programas Prolog; é este processo que contém a Máquina Abstrata Prolog, WAM-E;

- ***Spy***: processo reponsável por manter o agente *Mestre* informado sobre o estado dos trabalhadores;

- ***Communicator***: processo que efetua todo o tratamento das comunicações assíncronas, recebidas pelo trabalhador.

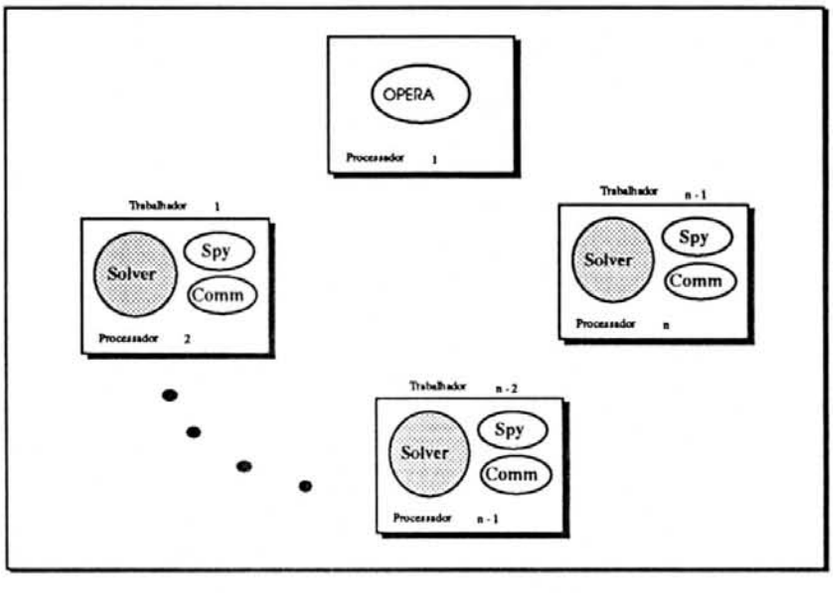

Figura 4.7 A Arquitetura de Processos para o Paralelismo E

Ao ser disparado, o processo *Mestre* verifica se a arquitetura já está ativa. Caso não esteja, os trabalhadores são instalados nos processadores. A arquitetura de processos, uma vez operacional, fornece suporte à uma seqüência de aplicações, até que o usuário delibere seu encerramento.

Os processadores onde serão instalados os trabalhadores estão indicados em uma tabela (Tabela de Recursos do Sistema - vide item 5.5.1.1), que é definida pelo usuário do sistema. Considerando não serem homogêneos, a SRT contém informações sobre o poder computacional dos processadores, as quais serão empregadas durante o escalonamento de serviço.

Cada programa Prolog utiliza um determinado número de processadores (que deve ser um subconjunto daqueles que formam a arquitetura a arquitetura de processos). Estando a arquitetura operacional, o processo *Mestre* lê os parâmetros sobre os quais será executada a aplicação e os distribui, através de uma mensagem, a todos os processadores envolvidos. Os parâmetros de execução são específicos para cada aplicação e definidos também pelo usuário em uma tabela (Tabela Característica da Aplicação - ACT - vide item 5.5.1.1).

4.2.6 Comportamento Dinâmico do Sistema

Para todo programa Prolog, uma vez concluído o trabalho de geração de código paralelo, este é entregue ao processo *Mestre*, que gerenciará seu processamento, a partir de parâmetros do usuário. O processo *Mestre* desdobra sua atuação em cinco fases bem distintas.

4.2.6.1 Fases do Processo *Mestre*:

- **instalação**: caso a arquitetura de processos não esteja montada, ou precise ser modificada, é nesta fase que os trabalhadores serão instalados em alguns processadores e encerrados em outros. O procedimento de instalação é analisado no item 5.3.1.2;

- **distribuição**: nesta fase, são distribuídos os parâmetros necessários para execução da aplicação Prolog aos trabalhadores envolvidos.

- **escalonamento**: nesta fase, o processo *Mestre* se transforma em escalonador e permanece distribuindo trabalho até que a execução esteja encerrada em todos os trabalhadores;

- **encerramento**: esta fase consiste em registar o resultado final da execução do programa Prolog e enviar sinal de sincronismo aos trabalhadores, preparando o sistema para outra execução;

- **remoção**: nesta fase, são encerrados nos processadores indicados pela Tabela de Recursos do Sistema, os processos que compõem o trabalhador. A remoção é disparada por determinação do usuário.

4.2.6.2 Estados do processo Solver

Na exploração do paralelismo E, a avaliação da carga de trabalho paralelizável em determinado trabalhador, é feita dinamicamente. É utilizada a heurística de associar carga ao número de literais na pilha

paralela (POP). Tendo como critério sua carga paralelizável, os trabalhadores são enquadrados em um de três estados. O limite entre os estados são específicos de cada aplicação:

- **idle**: neste estado, não há trabalho para ser executado. O trabalhador está aguardando uma autorização para busca de uma tarefa em outro trabalhador (importação);

- **quiet**: neste estado, o trabalhador está ativo, mas não dispõe de trabalho em volume suficiente, que justifique a cessão de parte deste para outro trabalhador;

- **overloaded**: neste estado, o trabalhador está ativo e o volume de trabalho que dispõe viabiliza uma exportação.

4.2.7 A Política de Escalonamento e a Gerência de Memória

No paralelismo E, os literais que serão executados se comportam como procedimentos, não existindo, obrigatoriamente, uma seqüência ou relação entre os literais que são buscados pelas Máquinas Abstratas Prolog, isto é, na execução paralela, podemos ter diferentes cláusulas sendo processadas nos diversos trabalhadores. Por este motivo, o literal que um trabalhador vai executar não depende somente da semântica procedimental da linguagem, mas também de outros parâmetros pertinentes à implementação.

A seguir, é avaliada a relação entre escalonamento e gerenciamento de memória, tendo em vista os seguintes objetivos: minimizar o tempo inativo dos trabalhadores, otimizar o uso da memória,

evitar ao máximo o uso de rotinas para *garbage collection* e garantir uma distribuição de carga proporcional à capacidade de resposta de cada trabalhador.

4.2.7.1 Administrando Resultados

A seguir, de modo sucinto, temos os procedimentos possíveis quando da execução de um literal.

- **Falha**: neste caso, o trabalhador envia ao pai uma mensagem de falha e aos trabalhadores-filhos uma mensagem que encerra nestes, a execução corrente (mensagem de *Kill*). Os trabalhadores, ao receberem esta mensagem, procedem a liberação de área nas pilhas de suas Máquinas Abstratas Prolog.

- **Sucesso**: o trabalhador envia uma mensagem de sucesso ao trabalhador-pai e fica aguardando deste:

  1. Uma mensagem solicitando os resultados obtidos, ou;

  2. Uma mensagem de *Kill* encerrando a execução do literal pendente e autorizando liberação de área nas pilhas de dados utilizadas pelo seu processo *solver*, ou

  3. Uma mensagem de *Redo* fazendo com que seja explorada outra cláusula-alternativa para o literal já processado.

4.2.7.2 A Situação da Memória no Trabalhador Filho

Para minimizar o tempo inativo, enquanto aguarda determinações (mensagens) do trabalhador-pai após ter obtido sucesso no processamento de um literal, o trabalhador pode buscar outro literal para execução. Disto podem decorrer as seguintes situações:

- O processamento do novo literal armazenado nas pilhas da Máquina Abstrata Prolog (WAM-E) é terminado antes do literal que está aguardando, receba sua mensagem de *Kill* ou *Redo*. Esta é a melhor situação e o mecanismo normal de retrocesso (*backtracking*) consegue recuperar toda a memória alocada;

- O processamento do novo literal não é esgotado antes do anteriormente empilhado (também fica aguardando mensagem). Desta situação decorrem duas possibilidades:

  - É recebida mensagem de *Kill* para o primeiro literal pendente: neste caso o processamento irá continuar com uma faixa sem uso nas pilhas da Máquina Abstrata Prolog. Esta faixa será liberada quando ocorrer retrocesso no literal empilhado por último;

  - É recebida mensagem de *Redo* para o primeiro literal pendente: neste caso, poderá faltar área de trabalho (contígua) para o *Redo*, o que é uma situação de gerenciamento de memória de difícil administração.

4.2.7.3 Alternativas Para Otimizar o uso da Memória do Trabalhador

As alternativas a seguir tem o objetivo de evitar os problemas de alocação de memória citados:

- Buscar novos literais somente do trabalhador-pai. Isto garante que o último literal importado terá seu processamento esgotado, antes que seja solicitado *redo* para algum literal pendente;

- Não havendo mais possibilidade de importar literais no trabalhador-pai, verificar se o literal pendente tem nodo OU (possibilidade de *redo*). Caso não possua, pode ser buscado literal em qualquer trabalhador. Com este procedimento, na pior hipótese, ter-se-á parte do processamento com uma faixa de memória que não pode ser liberada;

- Não existindo mais literais disponíveis no trabalhador-pai, e sendo o literal empilhado primeiro passível de *Redo*, a alternativa é criar um novo conjunto de pilhas, nas quais poderá ser processado qualquer outro literal disponível para execução paralela no sistema, sem risco de sobreposição à área contígua.

Na atual fase do protótipo está implementada a primeira alternativa. Esta alternativa é a que apresenta menor custo de gerenciamento e se mostra vantajosa desde que se disponha de vários elementos processadores na arquitetura paralela.

4.3 Conclusões

Este capítulo resumiu os principais conceitos pertinentes ao paralelismo OU Multiseqüencial do projeto OPERA definidos em [GEY 91], e apresentou o modelo para exploração do paralelismo E proposto para o projeto OPERA, questão central desta dissertação.

Partindo dos fundamentos do paralelismo E Restrito, são apresentadas as opções feitas para exploração do paralelismo E, passando pela proposta do seu modelo computacional, pela definição da Máquina Abstrata Prolog empregada, pela composição da arquitetura de processos para execução paralela e dos seus aspectos dinâmicos (estados de processos, escalonamento de trabalho e gerência de memória).

A definição das componentes do modelo de paralelismo E, foram feitas considerando sua futura integração com as do paralelismo OU.

No próximo capítulo, serão tratados aspectos pertinentes a implementação deste modelo em uma arquitetura paralela sem memória comum.

5 O PROTÓTIPO DO PARALELISMO E DO PROJETO OPERA

Esta seção tem por objetivo descrever o protótipo para exploração do paralelismo E do projeto OPERA. O objetivo do protótipo é validar o modelo de exploração do paralelismo E, apresentado no capítulo 4. Duas premissas nortearam a construção do protótipo:

- ser **passível de integração** com o protótipo do paralelismo OU multiseqüencial, tendo em vista o objetivo de chegar a uma proposta que contemple uma exploração combinada dos paralelismos E e OU da Programação em Lógica;

- ser **portável** para outras arquiteturas paralelas. Para tanto, os trechos de código com relação direta com a arquitetura destino (*hardware*), estão confinados em alguns módulos, restringindo, deste modo, o custo de conversão de *software*.

A codificação de todos os programas que formam o protótipo foi feita empregando a linguagem C. Na tabela A-1.1 (Anexo A-1) é apresentada a nominata destes arquivos.

As linhas gerais do protótipo do paralelismo E do projeto OPERA foram defendidas no V Simpósio Brasileiro de Arquitetura de Computadores - Processamento de Alto Desempenho, ocorrido no XII Congresso da Sociedade Brasileira de Computação ([YAM 93]).

5.1 O Ambiente de Execução

A atual versão do protótipo do paralelismo E do projeto OPERA é processada sobre rede local de computadores. As estações de trabalho da rede local empregam o sistema operacional Unix. O padrão físico de rede empregado é o Ethernet ([TAN 89]), com uma taxa nominal de transferência de 10 M bits por segundo, sendo utilizado o protocolo TCP/IP (*Transmission Control Protocol/Internet Protocol*) ([STE 90]). A configuração empregada é formada por estações SUN empregando o Sistema Operacional SunOS 4.1.1. O número máximo de estações a serem utilizadas na exploração do paralelismo é determinado pelo usuário, através da **Tabela de Recursos do Sistema** (SRT - *System Resource Table* - item 5.5.1.1). Este *hardware* constitui uma arquitetura multiprocessadora sem memória comum (vide figura 5.1), onde a comunicação entre processadores é feita somente por troca de mensagens (*MIMD - fluxo Múltiplo de Instruções para fluxo Múltiplo de Dados; tipo NORMA - Sem Acesso a Memória Remota*).

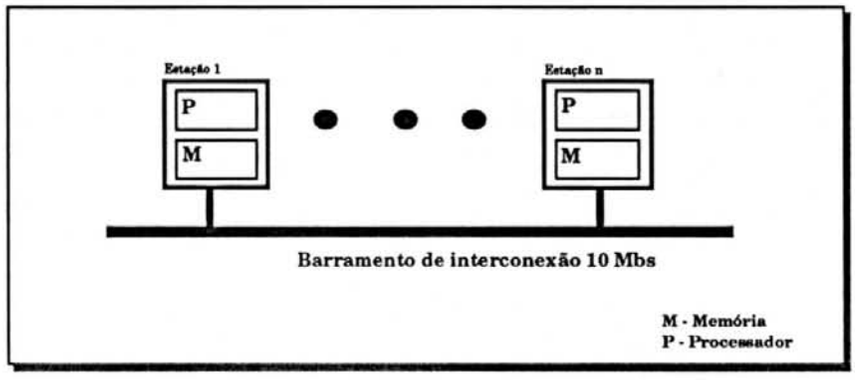

Figura 5.1 Arquitetura básica do Ambiente de Execução

Dois são os principais fatores que estimularam o uso de uma rede local na implementação do protótipo de paralelismo E do modelo Opera E/OU. O primeiro é da ordem de aproveitamento de recursos. Este fator leva

em consideração a existência, no ambiente acadêmico, de um bom número de máquinas parcialmente ociosas [DOU 91]. Tal ociosidade, se deve a trabalhos com elevada interação com o usuário (leitura e escrita de textos), característicos neste ambiente. Existem casos pesquisados que revelam situações mais extremas, onde um total correspondente a um terço do montante de equipamentos, em determinados períodos, está sem qualquer uso, mesmo durante horas úteis do dia [THE 89]. Propostas como o protótipo para exploração do paralelismo E do projeto OPERA constituem uma alternativa para o aproveitamento destas estações total ou parcialmente livres, como "servidores de processamento".

O segundo fator está ligado à disponibilidade, e leva em conta o fato das redes de computadores serem hoje a "arquitetura paralela" mais difundida na comunidade acadêmica, e isto, dentre outros aspectos, facilita o intercâmbio de experiências entre instituições.

No caso do Instituto de Informática da UFRGS, quando da construção do protótipo do paralelismo E, a rede local de computadores era a única opção disponível para sua implementação.

5.2 Composição das Comunicações

O protótipo do paralelismo E, combina recursos da programação concorrente, com os da programação distribuída. Neste tipo de sistema, os mecanismos de comunicação empregados são determinantes da eficiência de execução.

Esta seção apresenta as opções feitas, tanto para as comunicações entre processos alojados em um mesmo processador (estação de trabalho), como para aqueles localizados em processadores diferentes.

5.2.1 COMUNICAÇÕES NO TRABALHADOR (LOCAIS)

Para a troca de informações, os processos que constituem o trabalhador OPERA empregam áreas de memória compartilhada. Existem 2 áreas de memória compartilhada (vide figura 5.2):

1. ***Worker_Load*:** através desta área de memória comum, o processo *Spy* captura o indicador da quantidade de literais que o trabalhador pode exportar. Este indicador da carga paralelizável é gerenciado pelo processo Solver, conforme descrito no item 4.2.4.1(Pilha de Objetivos Paralelos).

2. ***Assync_Mesg*:** o processo *Communicator,* ao receber mensagens assíncronas de outros trabalhadores, emprega esta área de memória comum para repassá-las ao processo *Solver*.

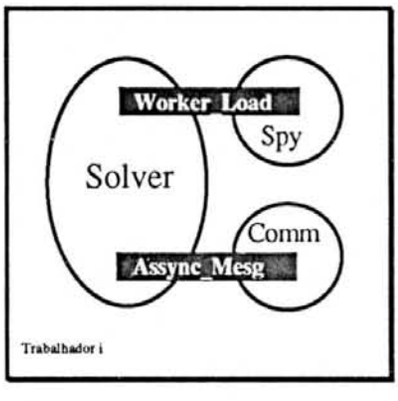

Figura 5.2 Áreas de Memória Compartilhada no Trabalhador

A opção pelo uso de memória compartilhada torna mais complexa a gerência da comunicação entre os módulos, porém constitui uma possibilidade não onerosa de comunicação assíncrona entre processos Unix. Esta característica é fundamental para minimizar o custo (*overhead*) introduzido pela gerência do paralelismo.

Toda interação entre os processos Spy e Solver é feito atráves da área de memória compartihada, não sendo necessário nenhum outro tipo de sincronismo. Por sua vez, entre os processos *Communicator* e *Solver* também são empregados semáforos para sincronização.

5.2.2 COMUNICAÇÃO ENTRE TRABALHADORES

A escolha da plataforma de *software,* sobre a qual foi construído o subsistema de comunicações entre processadores (estações), levou em consideração prioritariamente dois critérios: o primeiro é o do **desempenho** global da plataforma (ver tabela 5.1), e o segundo é o de manter um bom nível de **portabilidade** para outras arquiteturas paralelas.

Tabela 5.1 Desempenho de Ferramentas & Granulosidade

	Sockets		**RPC**		**HetNOS**		**PVM**	
Gran.	Tempo	Desvio	Tempo	Desvio	Tempo	Desvio	Tempo	Desvio
1	12.345	0.119	262.37	21.673	13106.418	0.020	864.068	1.635
128	0.209	0.025	3.525	0.436	102.376	0.020	7.125	0.069
256	0.145	0.005	2.760	0.260	51.194	0.016	3.657	0.092
512	0.111	0.004	2.011	0.011	25.589	0.028	1.848	0.071
1024	0.098	0.003	12.494	0.504	12.776	0.090	1.006	0.073
2048	0.098	0.003	6.182	0.475	6.396	0.020	0.615	0.073
4096	0.096	0.005	3.390	0.435	3.190	0.030	0.480	0.087
16384	0.092	0.002	1.429	0.031	1.611	0.033	0.293	0.180

1. Tempo, Desvio em segundos.
2. Grão em *bytes*

A tabela 5.1 registra o desempenho em função da granulosidade da mensagem, para algumas ferramentas empregadas na construção de rotinas de comunicação em sistemas distribuídos/paralelos. Os valores foram obtidos com a transmissão de uma mensagem de tamanho fixo de 64 Kb entre dois processos.Unix. Os processos foram localizados em estações de trabalho SUN 4/20 conectadas por uma rede local padrão ethernet (com taxa de transferência de 10 Mbits por segundo). A coluna **Gran.** apresenta a granulosidade empregada em *bytes* para envio da mensagem; assim sendo, para uma granulosidade de 1024 *bytes* são necessários 64 envios para um processo repassar toda a mensagem para outro. As diferenças no desempenho são devidas, principalmente, aos tempos característicos de cada ferramenta, para disparar/concluir cada envio. A coluna **Tempo** registra o tempo em segundos gasto com a transmissão. Esta estimativa de tempo foi obtida pela média de 50 medições. A coluna **Desvio,** por sua vez, apresenta o desvio padrão em segundos destas medições em relação a média. São comparados os desempenhos para quatro ferramentas, a saber:

- *Sockets TCP* [STE 90]; os sockets são uma abstração que caracteriza um canal de ligação entre dois processos, através da rede de computadores. Suas características são apresentadas na seção 5.2.2.1.

- *RPC (Remote Procedure Call)* [COR 91]; esta biblioteca de rotinas viabiliza a chamada de procedimentos remotos como se fossem locais, simplificando a programação de aplicações distribuídas. A partir de definições do usuário, uma ferramenta denominada rpcgen se encarrega de gerar os trechos de código para suporte das comunicações entre processos. Este código, gerado automaticamente, deve ser compilado junto com o código da aplicação distribuída, construído pelo usuário.

- *HetNOS (Heterogeneous Network Operating System)* [BAR 92]; sistema operacional de rede heterogêneo, que facilita o desenvolvimento de aplicações distribuídas em redes de estações homogêneas ou heterogêneas. O HetNOS permite ao usuário concentrar-se apenas nas questões inerentes à distribuição dos seus algoritmos, abstraindo os detalhes dos mecanismos de comunicação entre processos.

- *PVM (Parallel Virtual Machine)* [GEI 93]; esta plataforma, de ampla aceitação internacional, de forma análoga ao HetNOS, retira do usuário o encargo de gerenciar os aspectos da comunicação entre processadores, introduzindo total transparência na localidade física dos processos, aumentando, deste modo, o nível de abstração da programação distribuída/paralela, e simplificando, em muito, o seu desenvolvimento.

A tabela 5.1 evidencia a melhor eficiência do Sockets sobre as outras ferramentas, sobretudo para envios de pequeno tamanho. Considerando que a maioria das mensagens na exploração do paralelismo E são pequenas, e considerando também a intenção de controle, no nível mais baixo possível, sobre o código fonte, a comunicação remota entre processos no OPERA é feita utilizando primitivas construídas com Sockets.

5.2.2.1 Sockets BSD

Os Sockets foram introduzidos em 1983 com o Unix BSD versão 4.1c. Atualmente, diversos sistemas operacionais os incorporam, inclusive sistemas Unix derivados da família System V, dentre estes temos: o HP-UX

[HEL 88] [HEL 89] e o IRIX [SIL 91] [SIL 91a]. A disponibilidade da interface Sockets, em diferentes plataformas, contribui para a portabilidade dos sistemas que a utilizam.

A interface de *sockets* integrante dos sistemas BSD suporta três protocolos: *domínio Unix*, *domínio Internet* e *domínio XNS*. Os *Sockets* no *domínio Unix* constituem uma forma de comunicação **local** entre processos (IPC local). Apesar de implementados na forma de arquivos, estes mecanismos fazem parte do sistema de rede BSD, como os demais protocolos.

Os outros dois domínios, *Internet* e *XNS*, fornecem comunicação local e remota entre processos. Operações com *sockets* utilizam uma estrutura para identificar endereços de *hosts*, sendo que tal estrutura possui dois campos: a família (o domínio) do endereço, e uma identificação específica de cada protocolo de até 14 *bytes*. Como exemplo, no *domínio Internet* (AF_INET no campo família), essa estrutura é redefinida para conter a identidade do domínio, um número de porta (16 bits) e a identidade do *host*/rede (32 bits). Estas estruturas estão definidas nos arquivos <sys/socket.h> e <netinet/in.h> do Unix.

Há vários tipos de *sockets* (*stream*, *datagrama*, *puro*, *pacotes seqüenciais*, etc.), sendo a disponibilidade de um determinado tipo dependente do domínio considerado. Exemplificando, a versão 4.1.1 do sistema SunOS suporta três tipos de *sockets*:

- ***puro***: esta interface fornece aos usuários acesso direto aos protocolos de comunicação que suportam a abstração de *sockets;* serve, por exemplo, para a implementação de um novo protocolo de comunicação;

- ***datagrama***: interface sem conexão, transmite dados de forma não confiável através de mensagens. Não há controle da seqüência envio/recebimento, ou da perda de pacotes;

- ***stream***: interface de transmissão de dados confiável (seqüenciamento, controle de fluxo e erros), orientada a conexão e cuja transmissão se dá por um fluxo de *bytes*.

A interface pura não é apropriada para escrever um software distribuído/paralelo de uso específico, pois é de muito baixo nível. As vantagens do esquema por *datagrama* são a preservação de fronteira de mensagens, a ausência de uma fase de conexão e a não limitação quanto ao número de conexões abertas. A maior vantagem de *sockets* tipo *stream* é a sua confiabilidade, sendo, usualmente, este o fator decisório em favor desta opção [SUN88, p.223].

Os mecanismos de comunicação empregados pelo OPERA precisam ser confiáveis, portanto empregam *stream sockets*. Esta opção evita ter de agregar à implementação paralela/distribuída um mecanismo de conferência de pacotes perdidos, de seqüenciamento (para evitar embaralhamento de pacotes) e de detecção de pacotes errados. Estes mecanismos são relativamente complexos, e exigem controle assíncrono de *timeout*.

A seqüência típica de utilização de *sockets* para comunicação entre dois trabalhadores OPERA, está ilustrada na figura 5.3. É importante observar que o procedimento não é simétrico, uma vez que um dos processos, temporariamente na condição de servidor, espera que outro, como cliente, efetive a conexão.

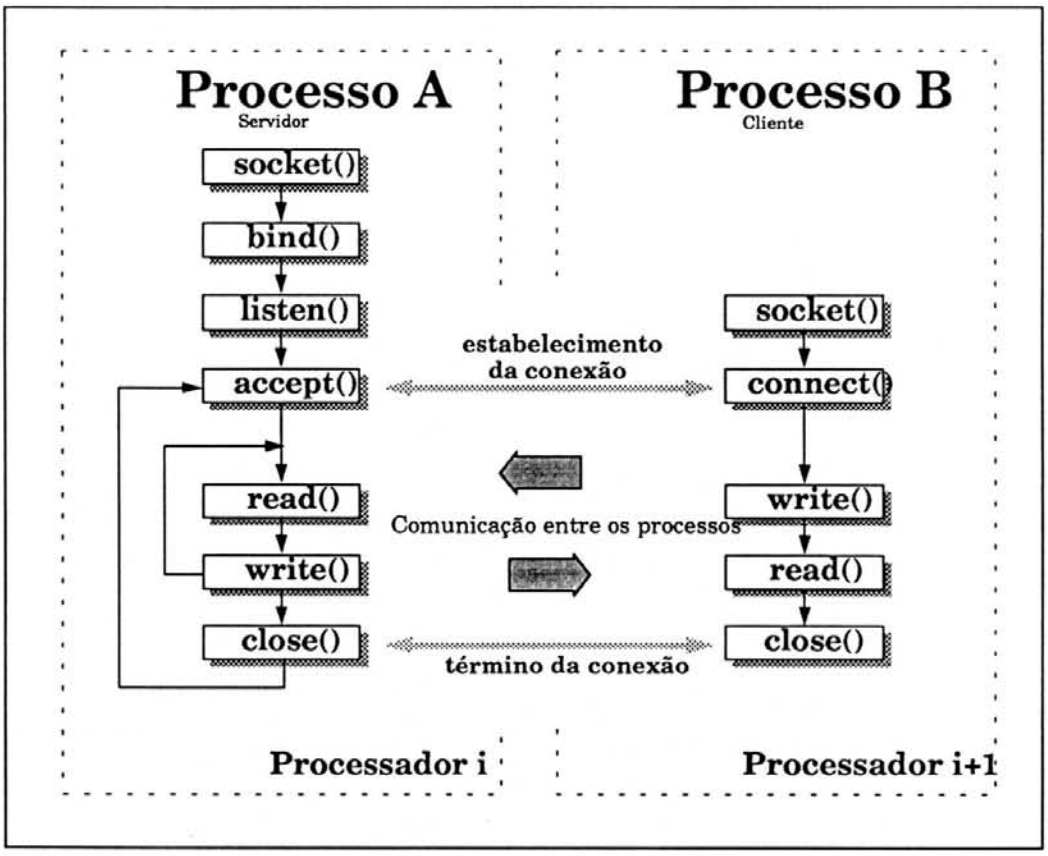

Figura 5.3 Protocolo Para Comunicação Entre Trabalhadores OPERA

5.3 Considerações Para Implementação Em Rede Local

5.3.1 Criação da Arquitetura de Processos

5.3.1.1 O Uso de Mensagens

O trabalhador OPERA, uma vez instalado em determinado processador (estação), assim permanece até que o usuário determine seu encerramento. A ativação de um trabalhador para tratar aplicações, bem como seu encerramento, é feito através de mensagens. Em se tratando a

arquitetura paralela empregada de uma rede de estações UNIX, o uso de mensagens trouxe uma redução significativa no tempo necessário para preparação dos trabalhadores para cada nova aplicação Prolog. Este ganho de tempo decorre de não ser necessário, para cada aplicação Prolog, utilizar comandos UNIX para disparo remoto de processos (rlogin, rsh) para compor os trabalhadores, os quais mostraram um desempenho bastante menor do que o obtido com o uso de mensagens.

A mensagem, com os parâmetros da aplicação que será processada, exige um servidor de comunicações sempre ativo no trabalhador, sendo empregado para esta função o próprio processo *Communicator*, o qual já é responsável pela recepção de mensagens assíncronas, durante a execução paralela da aplicação Prolog.

5.3.1.2 Procedimento de Instalação

Com o objetivo de minimizar o custo de instalação de trabalhadores, quando o processo *Mestre* é disparado em uma das estações, ele verifica o estado da arquitetura de processos OPERA na rede onde ocorrerá a execução, e a partir desta verificação decorrem três possibilidades:

- nenhum trabalhador está instalado; neste caso, o processo *Mestre* instalará trabalhadores em todas estações definidas pelo usuário na Tabela de Recursos do Sistema (SRT - *System Resource Table* - item 5.5.1.1) pertinente à aplicação.

- existem trabalhadores instalados, porém a SRT da aplicação a ser executada contempla estações diferentes daquelas onde já

estão instalados trabalhadores; neste caso, o processo *Mestre* promoverá uma adequação à SRT da atual aplicação, sendo disparados trabalhadores em algumas estações, e encerrados em outras, ou;

- os trabalhadores instalados correspondem aos definidos pelo usuário na SRT da aplicação; neste caso, o processo Mestre passará imediatamente à etapa seguinte, que é a distribuição dos parâmetros de execução da aplicação.

5.3.2 ESCALONAMENTO DE TRABALHO

Com a arquitetura de processos já instalada nos processadores, e tendo sido concluída a distribuição dos parâmetros característicos da aplicação, o processo *Mestre* assume as funções de escalonamento, e é iniciada a execução paralela do programa Prolog.

5.3.2.1 A Política do Escalonador

A atual versão do protótipo contempla uma filosofia de escalonamento centralizada. Isto significa que todos os trabalhadores da arquitetura se reportam diretamente a um único escalonador, o qual mantém as informações sobre a distribuição de trabalho na arquitetura.

A execução Prolog começa no trabalhador definido pelo usuário na Tabela Característica da Aplicação (ACT - *Application Characteristic Table* - item 5.5.1.1) como principal. É a partir da exportação de serviço deste trabalhador que se desdobra a exploração do paralelismo E. Também é este

trabalhador que devolverá a resposta final da consulta do usuário, ao processo *Mestre*

O fluxo de trabalho, conforme o modelo computacional (seção 4.2.3), é controlado por demanda, ficando a iniciativa de busca de serviço aos trabalhadores inativos.

Na alocação de serviço aos trabalhadores são adotados os seguintes critérios:

- como **importador** é escolhido o trabalhador desocupado (estado *idle*) que apresente maior *Indice de Importação*. Como critérios para seleção são empregados: o poder computacional da estação que aloja o trabalhador, e o seu número de usuários no momento da importação. A relação entre estes critérios é a seguinte:

$$\text{Índice Imp.} = \frac{\text{Poder Computacional}}{1 + \sqrt{\text{Outros Usuários}}}$$

O perfil de uso da rede local de estações UNIX do Instituto de Informática da UFRGS, empregada para execução do protótipo, é fortemente influenciada por aplicações com elevada interação com o usuário, como é característico em redes de uso geral no ambiente universitário. A raiz quadrada no critério número de usuários, é empregada para ajudar a compensar a não linearidade da relação *usuários & ocupação do processador*.

A relação que define o Índice de Importação é empírica, tendo sido obtida por experimentação, o objetivo com o seu uso é ter um balanço entre a filosofia de entrega de serviço aos

trabalhadores mais poderosos e a de priorizar o uso de estações com menor número de usuários. A principal intenção desta estratégia é minimizar o número de usuários da rede, que terão de compartilhar os recursos computacionais da estação que utilizam, com o trabalhador OPERA.

- como **exportador** é escolhido o trabalhador sobrecarregado (estado *overloaded*) com maior *Índice de Exportação*. Neste caso, são utilizados como critérios: o número de literais exportáveis que dispõem o trabalhador no momento da exportação, e o poder computacional da estação que o aloja. A relação entre os critérios é:

$$\text{Índice Exp.} = \frac{\text{Número de Literais Exportáveis x } (1 + \sqrt{\text{Outros Usuários}})}{\text{Poder Computacional}}$$

Esta relação também é empírica, e seu objetivo é ter na escolha do trabalhador exportador, um balanço entre o critério volume de carga paralelizável no momento da exportação, e os critérios poder computacional e número de usuários. Estes dois últimos, implicam, de forma direta, na capacidade do trabalhador de processar o serviço que tem acumulado. O emprego da raiz quadrada no critério *outros usuários*, se deve às razões já citadas.

5.3.2.2 Manutenção do Estado Global do Sistema

Na proposta de um escalonador, para uma arquitetura paralela sem memória comum destaca-se a escolha de indicadores para avaliação de carga nos processadores. A correta tomada de decisão por parte de um escalonador depende deste ter informações as mais precisas possíveis, sobre o estado global da carga de trabalho na arquitetura.

Nos tópicos a seguir está resumida a estratégia empregada na coleta de informações para escalonamento no protótipo para exploração do paralelismo E. Ela busca um compromisso entre uma coleta de informações suficientemente precisa, e o seu custo computacional, proporcional ao grau de precisão:

- somente quando o trabalhador estiver no estado *overloaded*, o processo *Spy* periodicamente enviará a situação da sua carga ao escalonador (o período é determinado pelo usuário na ACT de cada aplicação). Para que se efetive este envio é necessário uma variação na carga, cujo mínimo é, também, estabelecido pelo usuário;

- quando o trabalhador entrar no estado *quiet* ou *idle*, o processo *Spy* informa esta transição de estado, e interrompe o envio periódico de informações sobre a carga do trabalhador ao escalonador. O propósito é evitar o congestionamento da rede, minimizando o tráfego de comunicações gerenciais.

- no momento em que o escalonador tiver nas *Pilhas de Estado do Sistema* (vide seção 5.5.1) um trabalhador assinalado como *idle* e outro como *overloaded*, são disparadas as respectivas autorizações de importação/exportação de trabalho;

- enquanto durar o processo exportação/importação, os trabalhadores tem seus estados bloqueados no escalonador, para evitar múltiplas autorizações;

- se, no momento da efetivação da operação de exportação/importação, o atual estado do trabalhador exportador não for mais *overloaded*, será enviado ao trabalhador importador um sinal de NOWORK. Esta situação decorre da imprecisão das informações que o escalonador dispõe sobre a carga de trabalho da arquitetura;

- ao término da operação de exportação/importação (mesmo com final NOWORK), os trabalhadores envolvidos enviam seus estados ao escalonador para minimizar inconsistências nas *Tabelas de Estado do Sistema*, e caracterizar-se como livres para outras transações.

Os objetivos desta política de manutenção do estado global do sistema, são minimizar o custo da criação de informações gerenciais no trabalhador, bem como evitar o congestionamento do canal de comunicação entre trabalhadores (rede local), provocado por um fluxo muito intenso destas mesmas informações.

5.4 Mensagens e Estados da WAM-E

Este resumo não aponta as inúmeras situações que podem ocorrer durante a emulação paralela, limitando-se apenas as mais importantes. (vide figura 5.4).

A emulação da WAM-E inicia com um teste que verifica se a estação de trabalho na qual está sendo executado o trabalhador OPERA o configura como principal da arquitetura de processos. Se a estação em questão não for a principal, o trabalhador fica no estado, denominado *case-idle* aguardando uma mensagem externa. O estado *case-idle* se repete cada vez que o trabalhador voltar ao estado de inatividade. A mensagem externa capaz ser atendida neste estado pode caracterizar uma importação de trabalho, uma execução de nova alternativa para o mesmo literal recém executado na máquina em questão, uma sinalização de que outro programa Prolog deve ser executado, ou seja, uma nova aplicação será executada; ou ainda pode caracterizar o encerramento dos processos que compõem o trabalhador. O nome das quatro mensagens possíveis são *import*, *redo*, *new_application* e *close*, respectivamente.

Por outro lado, se o trabalhador for o principal, este começa imediatamente a interpretação do programa Prolog. (como já visto, o empilhamento de objetivos na POP desta máquina é que desencadeia a execução paralela).

Um trabalhador não-principal durante a execução, lerá e emulará uma instrução de cada vez até que chegue uma mensagem do mundo externo ou termine a execução do literal corrente. Neste ponto, as mensagens possíveis podem sinalizar uma falha (*failure*) ou um sucesso (*success*) de um filho, uma autorização do escalonador para exportação (*export*) de trabalho para um trabalhador inativo, uma mensagem de *unwind* ou de *kill* enviada pelo trabalhador-pai. Se o trabalhador chegou ao final da execução do objetivo, este é o momento de avisar o pai da ocorrência de seu sucesso ou de sua falha. Supondo que tenha tido sucesso, este sinal será consumido pelo pai, que decrementará o número de filhos que está esperando terminarem a execução. A partir daí, o filho que sinalizou sucesso

fica esperando que lhe seja enviada uma requisição de resultados através da mensagem *solution_request*.. Depois de receber tal mensagem, o filho envia os resultados ao pai e volta ao estado *case idle*. No caso do trabalhador sinalizar uma falha para o seu pai, este se encarregará de sinalizar todos os outros trabalhadores filhos desta ocorrência, interrompendo nestes, pelo envio de uma mensagem de *kill,* o processamento do objetivo do qual já falhou um literal. Em função do objetivo que está sendo processado ter ou não nodos OU (não-determinístico-outras alternativas a serem tentadas) o trabalhador pai enviará ou não mensagem de *redo* aos trabalhadores filho..

Por sua vez, quando de um término de execução com sucesso, o trabalhador principal, fica esperando que o escalonador envie uma mensagem de *final_solution_request*. O tratamento dado a esta mensagem será o envio da solução final da consulta do usuário ao programa Prolog.. Depois deste procedimento, o trabalhador principal aguarda mensagem do processo Mestre determinando o início de uma nova aplicação (*new_application*) ou o encerramento de todos trabalhadores (*close*).

A seguir, na figura 5.4 é mostrado um diagrama do fluxo de mensagens e estados para o trabalhador OPERA.

5.5 Estruturas de Estado do Sistema

5.5.1 Pilhas de Estado do Sistema

Nas *Pilhas de Estado do Sistema* o escalonador mantém atualizados os estados dos trabalhadores da arquitetura, empregando a estratégia já descrita no item 5.3.2.2.

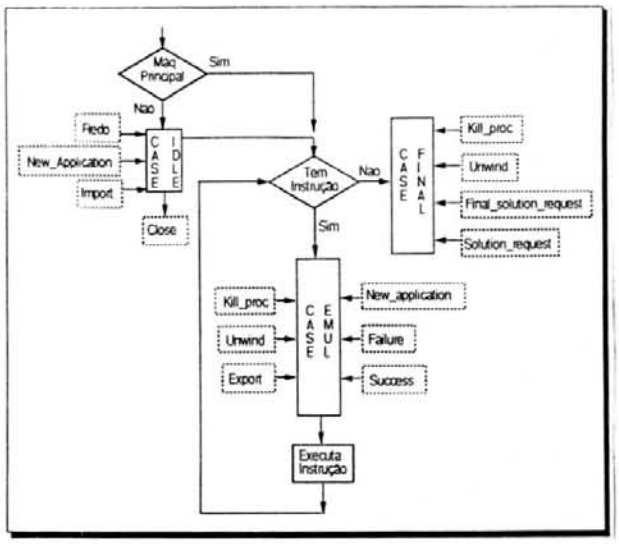

Figura 5.4 Estados e Mensagens no Trabalhador OPERA

Duas pilhas de estado são particularmente importantes: a Pilha de Trabalhadores Sobrecarregados (OWS - *Overloaded Worker Stack*) e a de Trabalhadores Desocupados (IWS - *Idle Worker Stack*).

A atualização destas pilhas é feita, dinâmicamente, à medida que o escalonador recebe informações sobre a carga dos trabalhadores. O trabalhador que apresentar maior Índice de Importação é colocado no topo da IWS, e, por sua vez, aquele que apresentar maior Índice de Exportação é colocado no topo da OWS.

5.5.2 Tabela de Identificação de Trabalhadores

Com o objetivo de tornar o sistema mais facilmente portável para outras arquiteturas paralelas o escalonador, a partir de indicações feitas em alto nível pelo usuário na SRT e na ACT, cria a Tabela de Identificação de

Trabalhadores (WIT - *Workers Indentification Table*), associando trabalhadores com uma identificação (*PId*), e a distribui, por mensagem, aos trabalhadores. Toda troca de mensagens entre trabalhadores é feita empregando a WIT. O seu uso objetiva evitar que o *software,* a nível de Máquina Abstrata Prolog (WAM-E), se comprometa com as características dos mecanismos de comunicação da arquitetura empregada. Entre estas características, destaca-se o mecanismo para roteamento de mensagens entre os processadores. Esta organização modular do software, entre outros aspectos, reduz o custo de portabilidade do sistema para outras arquiteturas paralelas.

5.6 Interface do Protótipo

5.6.1 Estruturas de Entrada e Saída

O objetivo com o uso das estruturas de entrada é facilitar a definição dos parâmetros de execução para as diferentes aplicações Prolog. Estes parâmetros determinam quais estações serão utilizadas no processamento, bem como os critérios de operação para o escalonador. Tal facilidade para configuração do sistema se faz importante, uma vez que o tema escalonamento na exploração do paralelismo na programação em lógica é um problema em aberto ([GUP89], [GUP90]) e, portanto, ainda objeto de diversas pesquisas.

As estruturas de saída, por sua vez, contemplam o registro do processamento efetuado, o que é indispensável para a avaliação do

comportamento dinâmico do modelo face às diferentes aplicações e seus parâmetros.

5.6.1.1 Estruturas de Entrada

Para configurar uma execução paralela, o protótipo do paralelismo E do projeto OPERA dispõe das seguintes estruturas de entrada:

- **Tabela de Recursos do Sistema** (SRT - *System Resource Table*): o objetivo desta tabela é definir quais e como são as máquinas da rede que formam a arquitetura do sistema. Ela define em que estações serão instalados trabalhadores, associando a cada uma o seu "poder computacional". O poder computacional da estação é uma informação básica na política de escalonamento de trabalho. Para uma descrição de como é formada a SRT ver item A-1.2.1.

- **Tabela Característica da Aplicação** (ACT - *Application Characteristic Table*): esta tabela, por sua vez, contém informações específicas de cada aplicação, dentre as quais destacamos: quais máquinas definidas na SRT serão utilizadas, em qual máquina será iniciado o processamento, qual a freqüência de atividade do processo *Spy* e qual o limite entre as faixas de carga *quiet/overloaded*. Uma descrição completa das informações contidas na ACT está no item A-1.2.2.

A interpretação destas tabelas é feita pelo processo *Mestre,* o qual também realiza uma consistência da sua composição. O uso destas tabelas,

manipulápeis por editor de texto, apresentam um elevado conforto para o usuário na definição da aplicação.

5.6.1.2 Estruturas de Saída

As propostas de execução paralela, introduzem um sério incremento na complexidade de uma atividade que por si mesma é complexa: o projeto e a depuração de programas. Como a arquitetura empregada pelo protótipo para o paralelismo E é uma rede de computadores, surgem alguns agravantes, dentres estes:

- os processadores (estações) que alojam os trabalhadores OPERA apresentam diferentes poderes computacionais;
- além do trabalhador OPERA, os processadores estão sujeitos a outras cargas de trabalho, de intensidade e duração aleatórias;
- aspectos dinâmicos do escalonamento e da operação do trabalhador podem variar de uma aplicação para outra, a critério do usuário.

Em função destes aspectos, para viabilizar uma compreensão de como aconteceu determinada execução, o protótipo é dotado de um sistema de *trace*. A ativação do *trace* é determinada pelo usuário.

O trace registra a atividade do escalonador, e abrange portanto todos os trabalhadores ativos. Os resultados da rotina de trace, estão descritos a seguir:

- **Resultados Textuais do Trace** (*Trace Text Results*): neste arquivo texto ficam registrados os parâmetros empregados na execução, bem como a situação da carga dos trabalhadores sempre que ocorrer uma transição de estado em qualquer dos processos *Solver* da arquitetura. A composição deste relatório está no item A-1.6.1.

- **Resultados Gráficos do Trace** (*Trace Graphic Results*): o sistema gera uma massa de dados que será processada por uma ferramenta de visualização. Dentre os diversos gráficos gerados, temos em função do tempo de execução: Número de Processadores Ativos, Carga no Processador e Períodos de Atividade do processador. Uma descrição de todos gráficos gerados pode ser encontrada no item A-1.6.2.

5.6.2 CHAMADA DO SISTEMA

5.6.2.1 Fases da Operação

A execução de um programa no protótipo para exploração do paralelismo E tem quatro etapas, conforme mostra a figura 5.5.

O programa Prolog é entregue ao Anotador. Este módulo do compilador irá anotar o código fonte, utilizando primitivas conforme descrito no item 4.2.2.3, gerando, para toda cláusula do programa, a correspondente CGE. A saída do Anotador é um arquivo com a extensão *ann*.

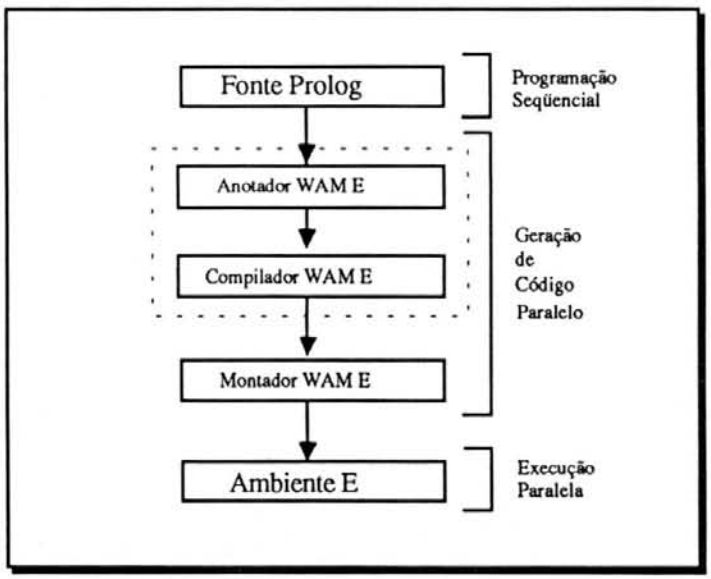

Figura 5.5 Etapas de Operação do Protótipo

Por sua vez, o Compilador, a partir do código Prolog anotado irá gerar código para a Máquina Abstrata Prolog WAM-E. Isto significa que, na geração de código, irá empregar as instruções para suporte ao paralelismo descritas na seção 4.2.4. O Compilador gera um arquivo com a extensão *plm*.

O Assemblador, a partir do código WAM-E irá gerar um código decimal. Este código é formado, principalmente, por *byte-codes* correspondendo às instruções WAM-E. O arquivo gerado pelo Assemblador tem a extensão *dec*.

Estando concluída a geração de cógido, este é entregue ao protótipo do ambiente de execução OPERA E, para que aconteça a execução paralela.

5.6.2.2 Comunicação Com o Usuário

A comunicação do usuário com o protótipo do paralelismo E do projeto OPERA pode acontecer de duas formas: por linha de comando Unix, ou através da interface XOPERA, como descrito no apêndice A-1.4

A interface XOPERA é um ambiente de janelas baseado no X_WINDOWS, a qual provê um ambiente bastante interativo com o usuário, facilitando sobremaneira a execução do sistema, bem como a visualização dos resultados gerados pelo trace da execução Prolog, nos trabalhadores OPERA.

5.7 Resultados Obtidos

Nesta seção estão registrados os dados de desempenho do protótipo para exploração do paralelismo E. Após uma breve descrição dos programas selecionados, são apresentados os resultados da execução seqüencial (em um processador) e os pertinentes à execução paralela.

5.7.1 Os Programas de Teste

A seguir são relacionados três dos programas que foram utilizados nos testes do protótipo, cada um com possibilidade diferenciada de exploração do paralelismo E. O código Prolog destes programas está no Anexo A-3.

- **fibo**: calcula o enésimo número da série de Fibonacci;

- **timings**: implementa laços independentes, sendo o número destes determinado pelo programador. O processamento nos laços se resume ao controle da iteração;

- **comb**: calcula a combinação de n elementos, tomando-os r a r.

5.7.2 Resultados em Seqüencial

Os programas de teste foram processados em uma estação de trabalho SUN 4/20 (modelo Sparc Station SLC). Os parâmetros empregados no processamento foram os seguintes:

- **fibo**: cálculo do quinto elemento da série de Fibonacci;

- **timings**: chamada de quatro laços de 750 iterações cada um;

- **comb**: cálculo da combinação de 10 elementos, 2 a 2.

A tabela 5.2 apresenta os resultados obtidos, sendo que a primeira coluna contém as aplicações. Já a segunda coluna informa o tempo de execução de cada aplicação, utilizando a Máquina Abstrata de Warren em sua versão seqüencial. A terceira coluna apresenta os tempos de execução das aplicações em um só processador para a WAM-E, empregando código anotado, isto é, código empregado na execução paralela.

A quarta coluna apresenta a relação entre o tempo de execução na WAM-E e o na WAM seqüencial, registrando a degradação decorrente dos mecanismos para gerência e controle do paralelismo.

Tabela 5.2 Resultados em Seqüencial

Aplicações	WAM-Seqüencial (ms)	WAM-E (ms)	Degradação
fibo	460,03	550,80	0,84
timings	3660,00	3663,10	1,00
comb	42,00	44,70	0,94

É importante ressaltar que os valores absolutos (2a. e 3a. colunas) são dependentes do poder do processador (estação de trabalho) empregado. Por sua vez a degradação é função da natureza do programa e dos seus parâmetros de execução.

5.7.3 RESULTADOS EM PARALELO

Para a execução paralela dos programas de teste foi utilizado um conjunto homogêneo de estações de trabalho SUN 4/20 SPARC SLC. As figuras 5.6, 5.7 e 5.8 mostram o comportamento na velocidade de execução em função do número de processadores para os programas *fibo*, *timings* e *comb*. Nestas figuras, o desempenho obtido pela execução de cada programa pode ser comparado com a curva que representa o desempenho ideal (linear ao número de processadores).

Os parâmetros dos programas, para execução paralela são os mesmos da execução seqüencial.

O programa *timings*, em função da elevada granulosidade com que permite a exploração do paralelismo E Restrito, apresenta o melhor desempenho, aproximando-se do ideal. Cada laço caracterizado pela chamada do predicado p(750) (vide anexo A-3) constitui um objetivo paralelo de elevada granulosidade.

O programa *fibo*, por sua vez, que também apresenta boa possibilidade de exploração do paralelismo, obteve ganho de desempenho. Observe-se que quando o número de processadores é menor a carga de trabalho que cada um recebe, compensa melhor os custos de gerência da execução paralela (detecção do paralelismo e comunicação). Isto se deve ao fato do fibo ser um programa recursivo e as chamadas paralelas serem decorrentes do empilhamento destas recursões, onde a cada nível de recursão o trabalho agregado é menor.

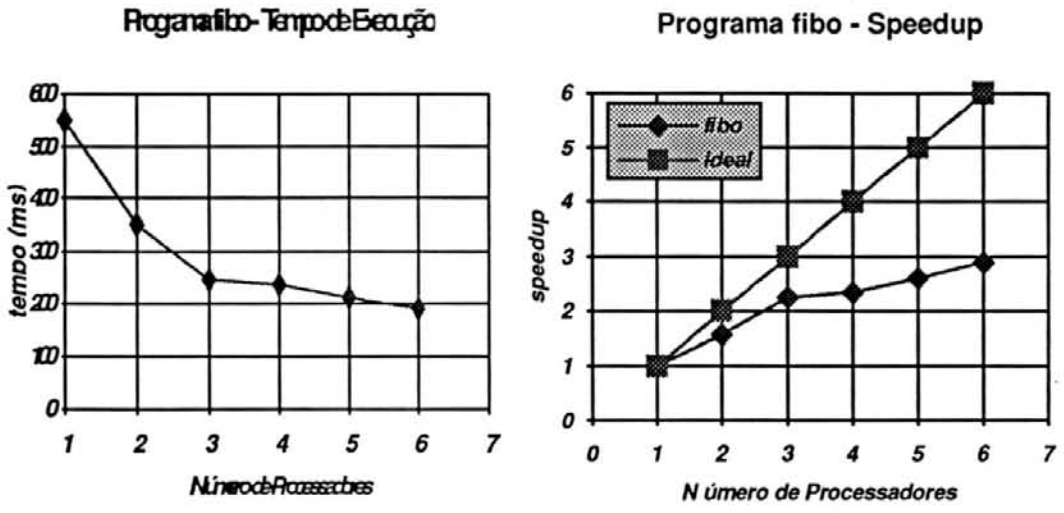

Figura 5.6 Comportamento do Programa fibo x Número de Processadores

O programa *comb*, em função da baixa granulosidade dos trechos paralelizáveis, obteve pouco ganho de desempenho. Os trechos paralelizáveis neste caso, são as chamadas dos fatoriais.

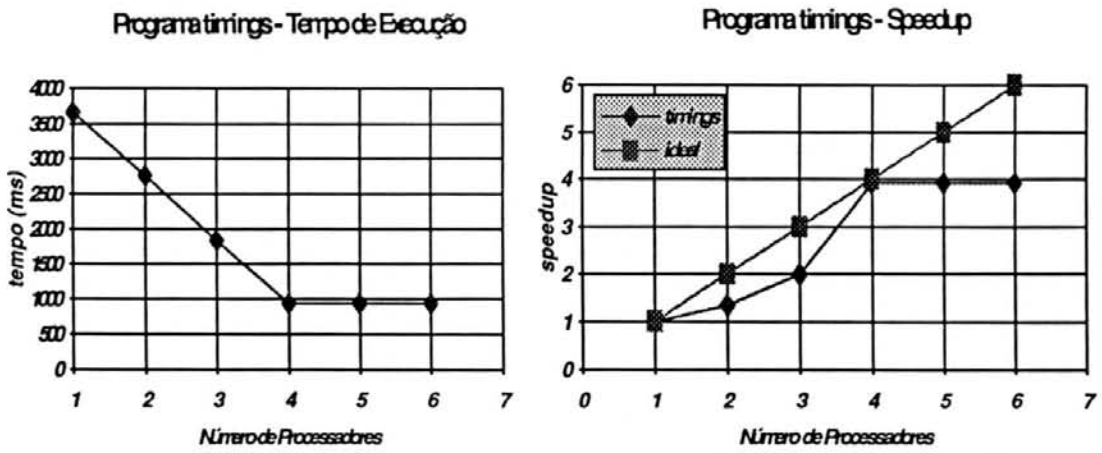

Figura 5.7 Comportamento do Programa timings x Número de Processadores

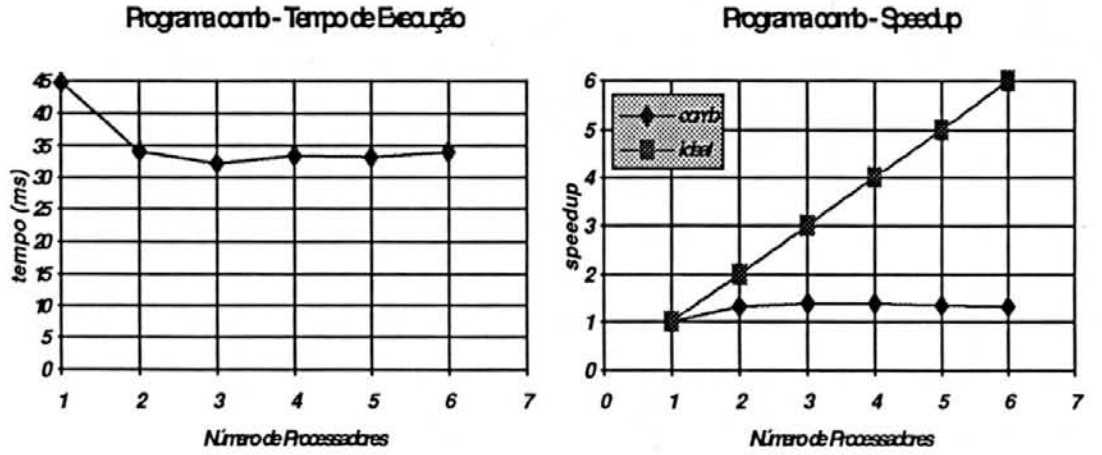

Figura 5.8 Comportamento do Programa comb x Número de Processadores

A execução seqüencial da WAM-E apresentou uma velocidade de execução de 3 a 4 vezes menor que o C-Prolog. Este desempenho se deve principalmente ao fato de seu código estar escrito integralmente em C, sem nenhuma otimização em baixo nível (linguagem assembly). Por outro lado, o código da WAM seqüencial do projeto OPERA não apresenta todas as otimizações atualmente disponíveis para interpretação Prolog. A pesquisa internacional aponta o SICStus Prolog como uma das plataformas

seqüenciais mais interessantes para o desenvolvimento de sistemas paralelos (como podemos ver no capítulo 3, PEPSys, Aurora, Muse, &-Prolog o utilizam), sendo que este também é derivado da WAM. Desta forma, uma das direções para aumentar o desempenho do modelo será a contrução de um protótipo empregando do SICStus Prolog.

Observe-se que o emprego de estações SPARC SLC (SUN 4/20) é conseqüência de, nesta fase de validação de resultados, existir um cuidado com a homogeneidade dos processadores empregados, e, neste momento, este é o modelo disponível em maior quantidade nos laboratórios do Instituto de Informática da UFRGS.

Os programas de teste foram processados com as máquinas sem usuários, com o objetivo de evitar a interferência de outros processos nos resultados. Mas é importante ressaltar a impossibilidade de controlar totalmente influências externas, uma vez que se trata de uma rede local conectada a INTERNET, onde surtos de tráfego ou o disparo remoto de processos são totalmente aleatórios.

5.8 Conclusões

Neste capítulo, foram apresentadas as linhas gerais empregadas na implementação do protótipo do para exploração do paralelismo E no projeto OPERA.

O prototótipo do paralelismo E (da mesma forma que o modelo) foi concebido tendo em vista sua futura integração com o protótipo do paralelismo OU. Sua codificação atual é toda em linguagem C.

Foram apresentadas as opções para comunicação entre processos locais (mesmo processador) e remotos (processadores diferentes), bem como a composição da política de escalonamento tendo em vista ser a arquitetura destino, da atual versão do protótipo, uma rede local de estações Unix.

Foram introduzidos os mecanismos que o protótipo oferece para comunicação com o usuário: tabelas de configuração da execução paralela, mecanismo de trace e interface gráfica. Uma descrição detalhada destes mecanismos consta no anexo A-1 (Manual do Usuário).

O protótipo executou com sucesso programas Prolog, legitimando o modelo proposto. Ao final do capítulo, é feita uma avaliação quantitativa do desempenho do protótipo

No próximo capítulo são apresentadas as conclusões pertinentes a todo o trabalho, e encaminhadas as futuras frentes de trabalho.

6 CONCLUSÕES

O ponto central deste trabalho foi o estudo da exploração de paralelismo na Programação em Lógica. Este estudo se concretizou com a proposta e a implementação de um modelo para exploração do paralelismo E.

6.1 Programação em Lógica e Paralelismo

O paralelismo OU somente consegue bom desempenho em problemas não determinísticos, e boa parte do trabalho realizado pelo conjunto de processos OU é desnecessário, quando é desejada uma solução em particular.

A exploração do paralelismo E, por sua vez, consegue ganhos no desempenho mesmo na ausência de não-determinismo no problema, e todo o trabalho realizado pelos processos E é util na computação de uma determinada solução. Porém, sua detecção e gerência é mais complexa, em função dos conflitos de ligações a variáveis bem como sua administração na presença do não-determinismo.

Um grande número de modelos computacionais tem sido proposto para execução paralela de programas lógicos, e alguns destes modelos foram eficientemente implementados em multiprocessadores. Atualmente, as propostas mais maduras exploram um só tipo de paralelismo ([BEA 91], [ALI 90] [HER 90]). Os sistemas implementados que combinam a exploração do paralelismo OU e E Independente ([KAL 87]), obtiveram ganhos quase lineares na velocidade de execução para problemas com paralelismo intrínseco, enquanto que, para programas com paralelismo não explorável

por esta proposta, seu desempenho é bem menor que o apresentado por bons sistemas seqüenciais. Situação análoga ocorre com as propostas que exploram o paralelismo OU e E Dependente ([YAN 93]), cujo desempenho quando da exploração individual de cada um dos tipos de paralelismo é menor do que no modelo do mesmo tipo. Estas propostas, porém, conseguem bons ganhos na velocidade de execução com a exploração simultânea de ambos paralelismos.

Os sistemas que exploram o paralelismo E Dependente, para muitas aplicações, apresentam um desempenho médio menor que as outras propostas, porém conseguem explorar melhor o paralelismo de granulosidade fina (presente na maioria das aplicações). As linguagens lógicas concorrentes, que exploram este tipo de paralelismo, provaram seu potencial para o desenvolvimento de diferentes tipos de programas. Como exemplo, temos o sistema operacional PIMOS ([FUR 92]) empregado nas máquinas multi-PSI ([TAK 84]) e PIM ([TAK 92]) formado por 100.000 linhas de código de KL1.

A avaliação teórica dos modelos que exploram 3 tipos de paralelismo ([WAR 90], [GUP 91]) mostra (de forma análoga aos modelos que exploram 2 tipos) que estas propostas ainda não atingiram o nível de eficiência daquelas que exploram um só tipo. Porém, promovem uma redução no espaço de busca na árvore de objetivos, evitando recomputações inúteis e, além disto, conseguem ganhos na velocidade de execução para um número bem maior de problemas.

A tendência da pesquisa internacional é a busca de sistemas que combinem, em uma plataforma eficiente, a exploração de diversas fontes de paralelismo.

6.2 Paralelismo E OPERA: Estado Atual

Os itens a seguir resumem as principais características atingidas para o modelo de exploração do paralelismo E proposto para o projeto OPERA:

- exploração do paralelismo implícito no código a partir do Prolog padrão (seqüencial). Porém é facultado ao programador o uso de anotações para garantir uma execução paralela ou seqüencial de determinados trechos do programa;

- suporte para exploração do paralelismo E Restrito na presença do não-determinismo *don't-know*. O conflito de ligações das variáveis é totalmente controlado pelas CGEs;

- o ambiente de execução para exploração do paralelismo E Restrito suporta as principais otimizações referentes ao consumo de memória e de desempenho da WAM seqüencial;

- transparência para o usuário da estratégia de controle do paralelismo. O escalonamento de trabalho é feito durante a execução. Toda comunicação e sincronismos necessários ficam a cargo do sistema (invisíveis para o usuário);

- controle distribuído. Os emuladores paralelos (WAM-E) são independentes; a única operação centralizada é a de escalonamento.

- degradação gradual do desempenho com a exaustão de recursos. O grafo de execução paralela, decorrente da avaliação de uma CGE, será processado automaticamente de forma

seqüencial se não existirem trabalhadores OPERA livres na arquitetura;

- compatibilidade com o código convencional WAM. O modelo suporta execução em um único trabalhador do código WAM seqüencial;

- desempenho seqüencial equivalente a WAM. Os programas, ou partes seqüenciais do código que não tenham sido anotados, são processados em velocidade semelhante à WAM seqüencial;

- ambiente de execução otimizado. A preocupação com o desempenho esteve presente durante a concepção do modelo. Neste sentido, uma vez que o modelo é voltado para arquiteturas com memória distribuída, foram reduzidas ao máximo as comunições e os sincronismos entre processadores bem como entre os processos que compõem o trabalhador;

- modelo validado. O protótipo do modelo implementado em rede de computadores processou, com sucesso, diferentes aplicações.

6.3 Trabalhos Futuros

Em decorrência dos trabalhos pertinentes a esta dissertação, surgiram novas frentes de investigação para o projeto OPERA, que se somam às apresentadas no item 1.3. Quais sejam:

- implementar aplicações de grande porte no prototótipo de exploração do paralelismo, como forma de caracterizar para a

comunidade usuária de Programação em Lógica a viabilidade do seu uso;

- investigar alternativas para compilação que ampliem a atual proposta de geração de uma CGE por cláusula do programa, para uma estratégia que contemple uma análise global do código. Esta análise global permitirá uma geração otimizada de CGEs, reduzindo o custo de detecção e gerência do paralelismo durante a execução. O ponto focal deste trabalho será propor uma estratégia de análise global que não eleve demasiadamente o custo (tempo) de compilação;

- estender o tratamento dado pelas rotinas para detecção e controle do paralelismo (rotinas para atribuição de tipos, para determinação de independência e geração de CGEs) às listas e estruturas. No modelo atual, a menos que seja especificado pelo usuário com uma anotação PAR, os predicados que empregam listas e estruturas serão executados seqüencialmente.

- integrar o modelo proposto para exploração do paralelismo E com o modelo para exploração do paralelismo OU Multiseqüencial. Inicialmente será contemplada uma exploração do paralelismo E Restrito abaixo do paralelismo OU Multiseqüencial, isto é, primeiro disparando o paralelismo OU em alguns processadores do sistema e, logo a seguir, o paralelismo E, como forma de acelerar a resolução dos ramos OU que estão sendo explorados de maneira multiseqüencial;

- portar o modelo para uma máquina paralela, onde os custos de comunicação entre processadores sejam menores. Neste

momento, serão também portadas as especificações da WAM-E para uma outra plataforma seqüencial, também baseada na WAM, que contemple as últimas atualizações em termos de Máquina Abstrata para compilação Prolog.

Por último, é importante ressaltar que o modelo de paralelismo E do projeto OPERA é uma das poucas propostas que contempla exploração do paralelismo E em arquiteturas paralelas com memória distribuída. Acredita-se que os relatórios técnicos decorrentes deste trabalho, somados a outros do projeto OPERA, constituem um consolidado ponto de partida para a busca de uma plataforma eficiente, que explore as diversas fontes de paralelismo na Programação em Lógica, e que processe em multiprocessadores escalonáveis sem memória comum.

ANEXO A-1 MANUAL DE OPERAÇÃO DO OPERA

Este anexo tem como objetivo apresentar os aspectos da operação do protótipo para a exploração do paralelismo E do projeto OPERA.

A-1.1 Características Básicas

O sistema possui três módulos com finalidades e modos de operação bem distintos. Conforme descrito no capítulo 5, os módulos são: **Compilador**, **Montador** e **Emulador**.

A operação do protótipo do projeto OPERA exije a disponibilidade de vários arquivos. Em função das suas finalidades, os arquivos estão agrupados em diretórios. A tabela A-1.1 (página seguinte) apresenta a estrutura de diretórios empregada.

A-1.2 Definindo uma Aplicação

Na proposta atual de execução do protótipo, cada aplicação é proprietária de um subdiretório. Este subdiretório contém os arquivos de entrada do sistema, bem como aqueles gerados durante a execução. A seguir serão apresentadas os itens que compõem uma aplicação no OPERA.

Tabela A-1.1 Descrição dos Diretórios do OPERA

Diretório	Descrição
OPERA	diretório raiz do OPERA
OPERA/bin/comm	arquivo executável do programa Communicator
OPERA/bin/scheduler	arquivo executável do programa Mestre (OPERA)
OPERA/bin/solver	arquivo executável do programa Solver
OPERA/bin/spy	arquivo executável do programa Spy
OPERA/bin/Xopera	arquivo executável da interface gráfica
OPERA/doc	arquivos de documentação do OPERA (*.txt, *.ps)
OPERA/src/prototype/comm	arquivo fonte do programa Communicator
OPERA/src/prototype/net	arquivo fonte com as rotinas de comunicação
OPERA/src/prototype/scheduler	arquivo fonte do programa Mestre (OPERA)
OPERA/src/prototype/solver	arquivo fonte do programa Solver
OPERA/src/prototype/spy	arquivo fonte do programa Spy
OPERA/src/prototype/Xopera	diretório com os prgms fontes da interface gráfica
OPERA/src/gshm	arquivos fontes das rotinas de suporte a IPC (semáforos, memória compartilhada e sinais)
OPERA/examples/fibo	conjunto de arquivos que compoe a aplicacao exemplo fibo

A-1.2.1 CONSTRUÇÃO DA TABELA DE RECURSOS DO SISTEMA (SRT)}

A-1.2.1.1 Objetivo

O objetivo da Tabela de Recursos do Sistema é caracterizar o *cluster* de processadores onde será explorado o paralelismo E. A SRT define em quais máquinas (estações de trabalho) do *cluster* (sub-rede) serão instalados os trabalhadores OPERA, e permite que o usuário atribua um poder computacional a cada um dos processadores. O poder computacional é um dos critérios empregados no escalonamento de trabalho. A determinação do poder computacional de um processador pelo usuário deve levar em consideração as características do *hardware* e a taxa de utilização do

processador em questão, para a execução de outras aplicações que não o trabalhador OPERA.

A-1.2.1.2 Composição

As linhas da Tabela de Recursos do Sistema são formadas pelos nomes de rede das máquinas que irão compor o cluster, seguidos do respectivo poder computacional. Para o poder computacional, é recomendada uma faixa de valores entre 1 e 100. O nome da estação deve ser separado do seu poder computacional por um caracter não numérico. As linhas de comentário devem ser iniciadas pelo caracter "#".

A-1.2.1.3 Exemplo

```
##############################################################################
#                     Tabela de Recursos do Sistema                          #
##############################################################################
# OPERA E/OU                                                                 #
# Projeto Prolog Paralelo                                                    #
# Coordenacao: Prof. Claudio Fernando Resin GEYER                            #
#                                                                            #
# Name    : fibo.srt                                                         #
# Version : 1.0                                                              #
# Date    : 01/10/93                                                         #
# Author  : Adenauer YAMIN                                                   #
#                                                                            #
# Copyright (c) 1993 - Curso de Pos-Graduacao em Ciencia da Computacao       #
#                      Universidade Federal do Rio Grande do Sul             #
#                      Brasil                                                #
##############################################################################

# Sparc Station 2  - 28 Mips
minuano - 90

# Sparc Station 1+ - 16 Mips
espora - 85

# Sparc Station 1  - 12 Mips
sinos - 70

# Sparc Station IPC - 16 Mips
```

```
cuia - 85
mate - 85
pala - 85

# Sparc Station SLC - 12 Mips
coxilha - 70
pampa - 70
pingo - 70

##########################################################################
```

A-1.2.2 CONSTRUÇÃO DA TABELA CARACTERÍSTICA DA APLICAÇÃO (ACT)

A-1.2.2.1 Objetivo

O objetivo da Tabela Característica da Aplicação (ACT) é definir os parâmetros que são empregados na execução de uma aplicação.

A-1.2.2.2 Composição

As linhas da ACT têm diferentes naturezas. O primeiro caracter da linha determina o parâmetro que está sendo especificado. A ordem em que são descritos os parâmetros é de escolha do usário. A especificação dos parâmetros é a seguinte:

- **d** <*path*/*file_name*>: especifica o arquivo executável (*byte_code*) da aplicação;

- **m** <*machine_name*>: assinala <*machine_name*> como uma das máquinas onde será ativado um trabalhador OPERA. Esta linha é necessária para todas as máquinas onde a aplicação será executada;

- **p** *<machine_name>*: o trabalhador OPERA principal será disparado em *<machine_name>*;

- **s** *<spy_sleep>*: determina o tempo, em microssegundos, de inatividade do processo spy. Este tempo de inatividade separa duas verificações consecutivas da carga do trabalhador e do número de usuários da máquina;

- **o** *<overloaded_level>*: estabelece o número de literais paralelizáveis a partir do qual o trabalhador é considerado sobrecarregado;

- **r** *<flutuation_range>*: estabelece a variação mínima que deve ocorrer na carga para justificar uma atualização do escalonador com o seu novo valor;

- **t** *<y* ou *n> <path/file_name>*: informa ao sistema se deve ser efetuado ou não o *trace*. O primeiro argumento confirma (*y*) ou não (*n*). O segundo argumento, por sua vez, determina o nome do arquivo onde serão armazenados os resultados numéricos.

A-1.2.2.3 Exemplo

```
##############################################################################
#                    Tabela Caracteristica da Aplicacao                      #
##############################################################################
# OPERA E/OU                                                                 #
# Projeto Prolog Paralelo                                                    #
# Coordenacao: Prof. Claudio Fernando Resin GEYER                            #
#                                                                            #
# Name    : fibo.act                                                         #
# Version : 1.0                                                              #
# Date    : 01/10/93                                                         #
# Author  : Adenauer Yamin                                                   #
# Remarks : This application calculates de fibonacci numbers                 #
```

```
#                                                                              #
# Copyright (c) 1993 - Curso de Pos-Graduacao em Ciencia da Computacao         #
#                      Universidade Federal do Rio Grande do Sul               #
#                      Brasil                                                  #
################################################################################

# Executable file name (dec):
d /home/minuano/werner/opera/tests/fibo/fibo_AND.dec

# Machines used in the execution:
m cuia
m mate

# Principal machine
p pala

# Spy sleep time (microseconds)
s 4000

# Overloaded level (goals)
o 15

# Range of load flutuation (goals)
r 3

# Trace option (yes with results to file fibo.trt)
T y fibo.trt

################################################################################
```

A-1.3 Resultados de uma Aplicação

Com o intuito de permitir a visualização de como se comportou uma determinada aplicação, ao longo do tempo, o sistema fornece *trace* da execução paralela. Este registro do processamento efetuado é indispensável para avaliação do comportamento dinâmico do sistema, face aos diferentes parâmetros possíveis para uma aplicação. Este registro é de duas naturezas:

- Resultados no Modo Texto: no arquivo *<application_name.trt>* constam os parâmetros empregados na execução, bem como um registro ao longo do tempo da situação dos trabalhadores. É

feito um registro para toda transição de estado em um dos processos *solver* da arquitetura.

- Resultados no Modo Gráfico: o sistema gera uma massa de dados, distribuída em vários arquivos (<application_name.g*>), que será processada por uma ferramenta de visualização (GNUPLOT). Para acesso aos gráficos gerados é recomendado o uso da interface gráfica do sistema (XOPERA).

A-1.4 A Operação do Sistema

A comunicação do usuário com o protótipo do paralelismo E do projeto OPERA pode acontecer de duas formas: por linha de comando ou através da interface gráfica XOPERA, como descrito a seguir.

A-1.4.1 Operação por Interface Gráfica (XOPERA)

O protótipo dispõe de uma interface de alto conforto para o controle de sua execução. Esta interface, denominada XOPERA, possui a seguinte estrutura:

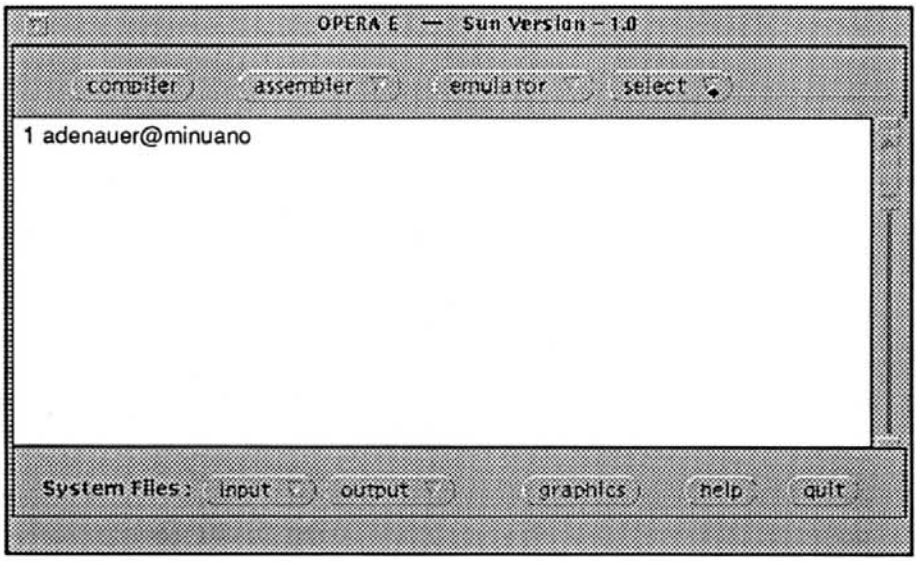

Figura A-1.1 Estrutura da Interface XOPERA

A interface XOPERA foi construída a partir da ferramenta *OpenWindows Developer's Guide*, versão 1.1, permitindo a execução das partes que compõem o OPERA, bem como a visualização das estruturas de entrada e saída do sistema. A janela principal da interface possui quatro botões dispostos na parte superior e cinco botões na parte inferior; uma área de mensagem, localizada na última linha; além de uma console, que permite ao usuário trabalhar unicamente no ambiente OPERA. Os botões superiores têm as seguintes funções:

Compiler aciona o módulo do compilador;

Assembler: aciona o módulo montador, usando o arquivo selecionado. Um menu de nível secundário pode ser invocado para selecionar um nome de arquivo ("*.plm"), pressionando-se o botão direito do *mouse*;

Emulator: aciona o módulo emulador, usando o arquivo selecionado. Um menu de nível secundário pode ser invocado para selecionar um

nome de arquivo ("*.dec"), pressionando-se o botão direito do *mouse*;

Select: é através desta opção que o usuário pode selecionar o arquivo de entrada a ser compilado ("*.pro"), o arquivo de entrada para o assembler ("*.plm") ou o arquivo de entrada para o emulador ("*.dec"), pressionando o botão direito do *mouse*. Depois do usuário escolher a máscara para exibição dos arquivos existentes no diretório corrente, é exibida uma janela de seleção de arquivo. Tal janela mostra os nomes de arquivos e permite a alteração do diretório de busca. A escolha do arquivo é feita a partir da lista de nomes de arquivos, usando-se o *mouse*.

Os botões inferiores têm as seguintes funções:

input: permite a manipulação de arquivos de entrada do sistema em um editor de textos, tendo-se a possibilidade de trabalhar com quatro tipos de arquivos ("*.pro", "*.act", "*.srt", "*.*");

output: permite a visualização de arquivos de saída do sistema em um editor de textos, tendo-se a possibilidade somente de leitura de cinco tipos de arquivos ("*.plm", "*.dec", "*.trt", "*.trg", "*.*");

graphics: permite a visualização, através de gráficos, de determinadas características da execução de um dado programa no protótipo do sistema OPERA. Tais características englobam as relações entre os processadores (carga e estado) e o tempo de execução. Como para cada execução pode ser gerado uma série de arquivos gráficos, ao iniciar-se a execução da ferramenta deve-se digitar o nome do arquivo que se quer visualizar. Cada gráfico está

associado a um arquivo, possuindo este o mesmo nome da aplicação. Os diferentes arquivos podem ter as seguintes extensões:

Tabela A-1.2 Extensões de Arquivos com Resultados Gráficos

EXTENSÃO	DESCRIÇÃO
gp	Número de Processadores X Tempo
ga	Processadores Ativos X Tempo
gag	Processadores Ativos X Tempo
gi	Processadores Idle X Tempo
gl	Carga de Cada Processador X Tempo

Feito isto, pode ser aberto um *menu* secundário com as opções de gráficos que podem ser gerados. Estas opções são as seguintes:

- *Number of Processors* X *Time* - mostra o número de processadores em função do tempo de execução. Nesse gráfico, pode-se avaliar quantos processadores chegaram a ficar ativos em cada momento da execução (vide figura A-1.3);

- *Processors Load* X *Time* - mostra a carga de cada processador durante a execução. Nestes gráficos (um para cada processador) tem-se a flutuação da carga de trabalho ao longo do tempo (vide figuras A-1.4, A-1.5 e A-1.6).

- *Idle Processors* X *Time* - mostra os intervalos de tempo em que cada processador ficou sem trabalho durante a execução da aplicação. O objetivo é aumentar os subsídios para definição dos parâmetros de operação do escalonador, de modo que possa

ser reduzido o tempo de inatividade de cada processador bem como o número de processadores necessários.

- *Active Processors* X *Time* - mostra os intervalos em que cada processador trabalhou durante a execução da aplicação, podendo-se avaliar a distribuição de trabalho durante o processamento.

- *Active Processors X Time* - mostra o tempo total de atividade de cada processador. Pode-se observar neste gráfico, por exemplo, qual processador esteve mais ocupado.

help: auxilia o usuário na operação do sistema. O XOPERA possui também outro nível de ajuda ao usuário, cuja ativação é feita através da tecla de função *Help*;

quit: permite que o usuário encerre a execução do sistema OPERA, removendo todas as janelas e aplicações.

A-1.4.2 Operação por Linha de Comando

Para cada aplicação é criado um diretório com seu nome, que contém os arquivos relativos à aplicação.

A-1.4.2.1 Compilação

O **compilador** do protótipo do projeto OPERA está escrito em Prolog e executa sobre o interpretador C-Prolog. Inicialmente, submete-se um arquivo com o programa fonte Prolog, empregando-se o seguinte

comando: Comp <arquivo Prolog>, onde <arquivo Prolog> corresponde ao arquivo fonte ("*.pro"). A saída do módulo Compilador é um arquivo com código WAM (*Warren Abstract Machine*) de mesmo nome do arquivo fonte, mas com extensão ".plm" , constituindo-se na entrada para o próximo módulo.

A-1.4.2.2 Montagem

O módulo seguinte a ser executado é o **Montador** e exige como entrada um arquivo com instruções WAM e com a terminação ".plm". Este módulo irá gerar um arquivo com código decimal (*byte code*) de mesmo nome do arquivo de entrada do módulo, mas com extensão ".dec". O usuário deve entrar com o nome do arquivo sem a extensão, conforme a sintaxe a seguir: Asm <arquivo WAM>. O arquivo de *byte-codes* é empregado pelo próximo módulo, o Emulador.

Como a versão atual do protótipo não contempla o Anotador de código, o arquivo de *byte-codes* deve ser modificado para que a execução do programa explore o paralelismo E presente no mesmo. Estas modificações são feitas utilizando-se os códigos das instruções WAM adicionadas ao conjunto de instruções para o suporte ao paralelismo E, como visto no capítulo 4. Tais códigos estão resumidos na tabela A-1.3.

Tabela A-1.3 Códigos de Instruções Paralelas

INSTRUÇÃO	CÓDIGO
check_me_else	90
check_ground	91
check_independent	92
push_call	93
pop_pending_goal	94
allocate_pcall	95
check_ready	96
wait_on_siblings	97

A última etapa consiste no módulo **Emulador**, que necessitará do arquivo decimal gerado pelo Montador. O processo **Mestre**, responsável pelo gerenciamento da execução paralela, deve ser disparado com a seguinte sintaxe: opera *<arquivo>*, onde *<arquivo>* corresponde ao byte-code (".dec") já anotado para a exploração do paralelismo E. O processo **Mestre** deve ser executado em qualquer máquina dentre as que não irão conter trabalhadores.

A-1.5 Exemplos de Código Paralelo

No capítulo 4 foram apresentados dois exemplos de código anotados com as primitivas de exploração do paralelismo E. Nas tabelas A-1.4 e A-1.5 são descritos, respectivamente, trechos dos códigos WAM e byte-code correspondentes a um programa .Prolog.

Tabela A-1.4 Código WAM Parcial de um Programa Prolog Anotado para Execução Paralela

INSTRUÇÕES WAM-E	COMENTÁRIOS
. . .	
cocall a/2,2	
check_me_else SEQ_CODE	guarda endereço do início do código seqüencial
check_ground x	verifica se as variáveis são *ground*
check_ground y	verifica se as variáveis são ground
allocate_pcall 2,2	aloca um *frame* na ECP
PIP: check_ready 2,POP	verifica se objetivo pode ser executado
put_y_value x,a1	put_y_value x,a1
put_y_value y,a2	
push_call b/2,1	empilha um objetivo paralelo
check_ready 1,POP	verifica se objetivo pode ser executado
put_y_value x,a1	
put_y_value y,a2	
push_call c/2,1	empilha um objetivo paralelo
POP: pop_pending_goal	desempilha um objetivo para execução local
wait_on_siblings	aguarda o fim da *CGE*
put_value x,a1	
put_value y,a2	
init_temp(x(3))	
deallocate	
coexecute d/3	
procedure(_a_2)	
get_integer(1,x(1))	
get_integer(0,x(2))	
. . .	

Tabela A-1.5 Código Decimal Parcial de um Programa Prolog Anotado para Execução Paralela

BYTE-CODE	INST. WAM-E (PARALELA)
...	
2 0 2 2 0 0 0 140	
90 228 0 0	check_me_else
91 1 0 0	check_ground
91 2 0 0	check_ground
95 2 2 1	allocate_pcall
96 108 2 0	check_ready
31 0 2 1	
31 0 1 2	
93 180 2 2	push_call
96 108 1 0	check_ready
31 0 2 1	31 0 2 1
31 0 1 2	31 0 1 2
93 160 2 1	push_call
94 0 0 1	pop_pending_goal
97 0 0 1	wait_on_siblings
31 0 2 1	
31 0 1 2	
19 3 0 0	
6 0 0 0	
3 0 3 0 0 0 0 200	
10 1 0 0 129 0 0 1	início do predicado **a**
10 2 0 0 129 0 0 0	
...	

Existe, no diretório opera/examples, um sub-diretório para cada aplicação-exemplo (como mostrado na tabela A-1.1). Em cada sub-diretório tem-se os arquivos correspondentes à aplicação em questão. Por exemplo, para a aplicação **fibo**, tem-se os arquivos das tabelas A-1.6 e A-1.7.

Tabela A-1.6 Arquivos Para a Versão Seqüencial

ARQUIVO	DESCRIÇÃO
fibo	arquivo fonte
fibo.plm	arquivo WAM
fibo.dec	arquivo de *byte-codes*

Tabela A-1.7 Arquivos Para a Versão Paralela

ARQUIVO	ARQUIVO DESCRIÇÃO
fibo_AND.plm	fibo_AND.plm arquivo WAM
fibo_AND.dec	fibo_AND.dec arquivo de byte-codes

A-1.6 Exemplos de Resultados

A-1.6.1 Resultados no Modo Texto

Caso o usuário especifique no arquivo ".act" da aplicação que deseja que o sistema realize o **trace** da execução do programa, é gerado o arquivo ".trt", correspondendo ao trace no modo texto das informações do escalonador. A figura A-1.2 apresenta um exemplo de tal arquivo.

A-1.6.2 Resultados no Modo Gráfico

A seguir são apresentados exemplos de resultados do *trace* gráfico do OPERA. Todos os gráficos referem-se a uma aplicação hipotética.

A figura A-1.3 mostra o gráfico com o número de processadores alocados ao longo da execução.

OPERA E/OU
Projeto Prolog Paralelo

Trace of: fibo
Date: 22/10/93
Time: 14:05

Parameters:

Total of Machines used: 3
Spy's Sleep: 4000
Spy's Range: 2
Principal Machine: 1
Overloaded Level: 9
Program File (DEC): fibo.dec

Machine Table:
1 - minuano
2 - mate
3 - pala

Scheduling Trace:

Time	EX	IM	Mach. Numb. 01	02	03
2.3908	01	03	OVER	IDLE	IDLE
4.7527			QUET	QUET	QUET
6.6308			OVER	QUET	QUET
7.9112	01	02	OVER	IDLE	QUET
10.0740			QUET	QUET	OVER
11.4505			OVER	QUET	OVER
12.5704	03	02	OVER	IDLE	OVER
14.1506	02	01	IDLE	QUET	QUET
16.0103			QUET	QUET	IDLE
19.0704			QUET	IDLE	IDLE
20.5693			IDLE	IDLE	IDLE

Total Time of execution: 20.56925

Total Time of execution at P1: 16.31876 (79.34 %)
Total Time of Execution at P2: 10.57482 (51.41 %)
Total Time of Execution at P3: 11.25755 (54.73 %)

Figura A-1.2 Exemplo de Resultado no Modo Texto

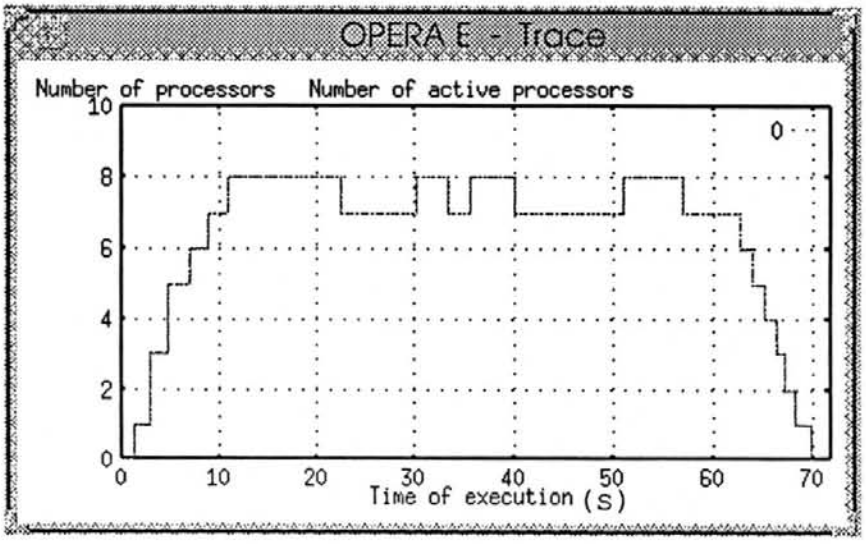

Figura A-1.3 Gráfico do Número de Processadores alocados X Tempo

A figura A-1.4 ilustra a variação da carga do processador 1 em relação ao tempo. As figuras A-1.5 e A-1.6 exibem a mesma relação, mas para os processadores 2 e 3, respectivamente.

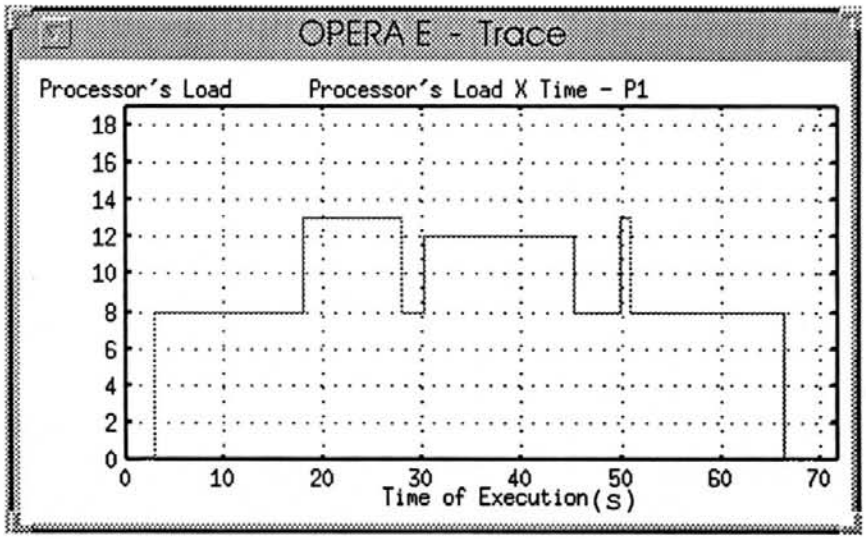

Figura A-1.4 Gráfico da Carga do Processador 1 X Tempo

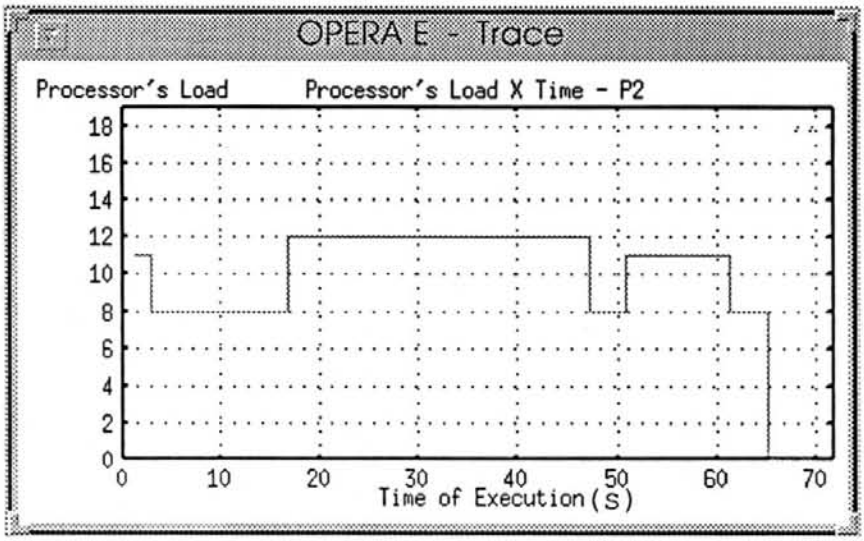

Figura A-1.5 Gráfico da Carga do Processador 2 X Tempo

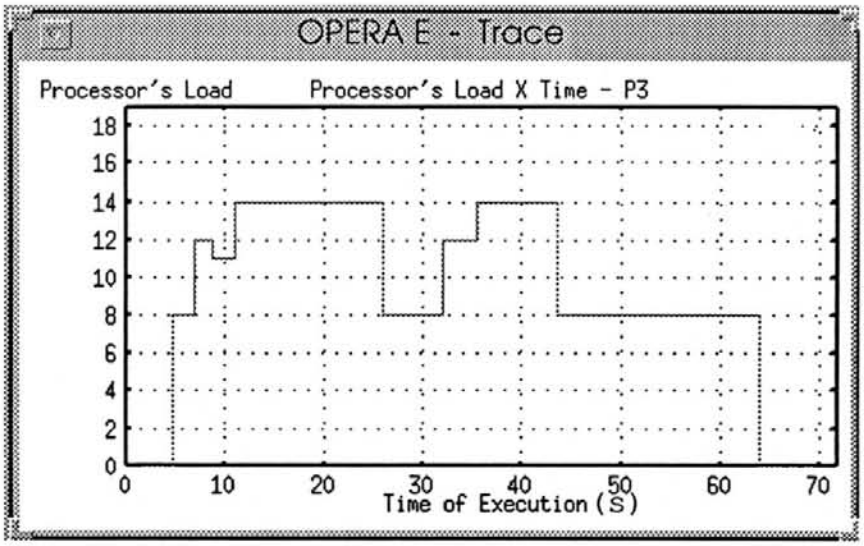

Figura A-1.6 Gráfico da Carga do Processador 3 X Tempo

Já a figura A-1.7 mostra o gráfico com os períodos de atividade dos processadores em relação ao tempo de execução, enquanto que a figura A-1.8 exibe os períodos de inatividade. Por último, a figura A-1.9 apresenta um gráfico de barras, que indica o total de tempo ativo de cada processador.

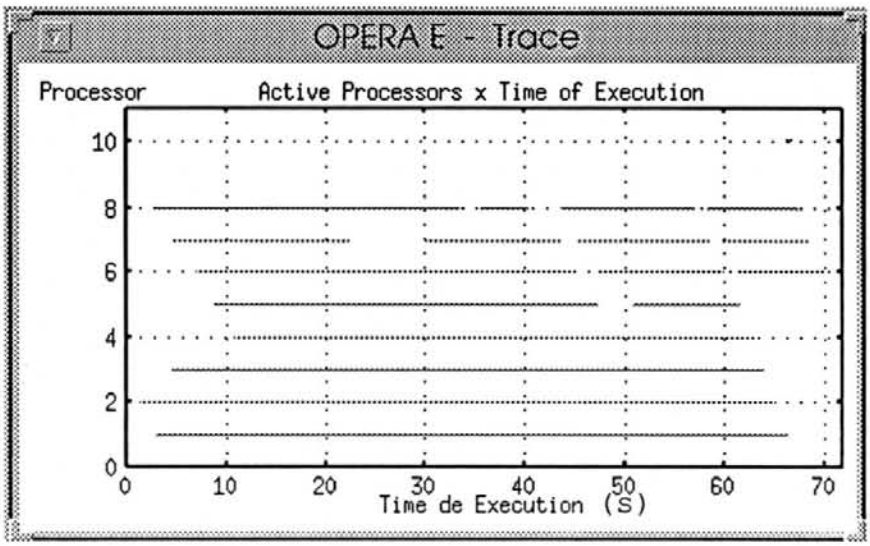

Figura A-1.7 Períodos Ativos dos Processadores X Tempo

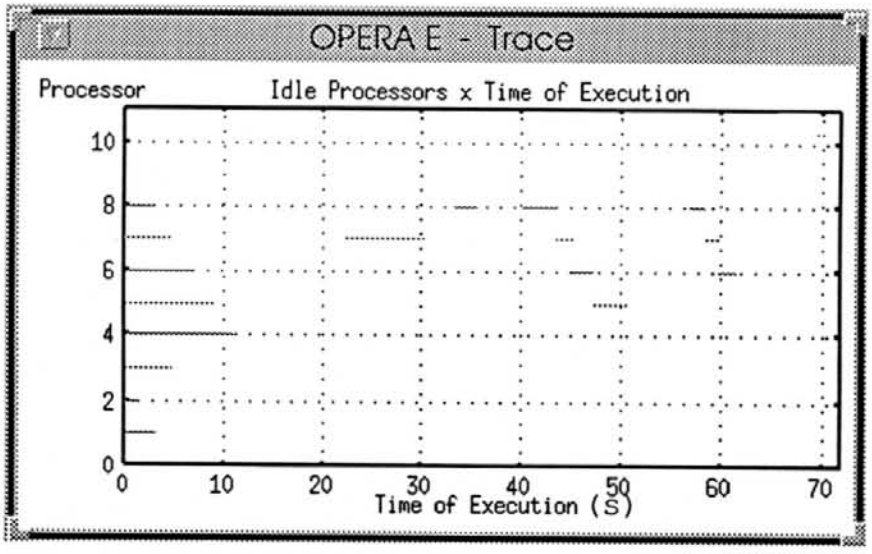

Figura A-1.8 Períodos Inativos dos Processadores X Tempo

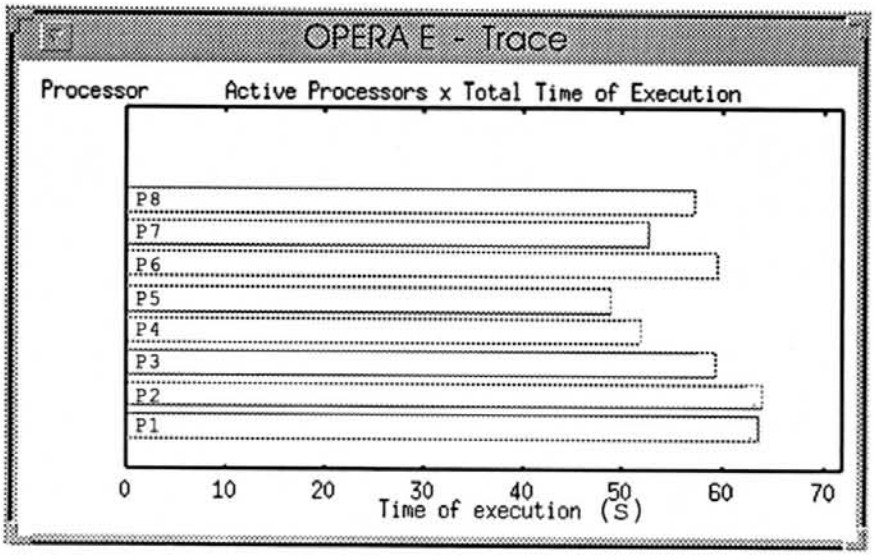

Figura A-1.9 Tempo Total Ativo de Cada Processador

ANEXO A-2 O PROLOG OPERA

Esta seção tem como objetivo apresentar ao usuário as características da linguagem Prolog fornecida pelo OPERA modelo E, no que diz respeito aos predicados pré-definidos suportados pelo mesmo (sua notação e aplicações).

Predicados pré-definidos são necessários por realizarem tarefas que não são possíveis no modelo puro de Prolog. Entradas e saídas são exemplos destes tipos de predicados. O modelo de Prolog não faz interação com o universo externo ao definido pela base de dados. Desta forma, qualquer ação além de casamento de padrões (*pattern-matching*) e instanciação de variáveis (*variable binding*) não é tratada pelo Prolog puro.

No Prolog OPERA, a sintaxe geral do C-Prolog é mantida [PER87]. No entanto, o sistema Opera possui predicados pré-definidos (*built-in*), que podem ser usados na construção dos programas.

Os predicados pré-definidos são usados exatamente como aqueles predicados projetados pelo programador, com a diferença de que os pré-definidos ficam disponíveis para serem usados sem especificação.

Muitos predicados pré-definidos existem para tornar a programação mais simples, como por exemplo o predicado *nl*, que redireciona a saída para a próxima linha. Ele poderia ser construído pelo usuário, mas, como é bastante usado, costuma estar pré-definido na implementação da maioria dos compiladores Prolog.

A-2.1 Predicados de Entrada e Saída

Considerando o modelo Prolog, a base de dados é a descrição do universo. Para explorar este universo ou derivar informações a partir dele, faz-se perguntas e obtém-se respostas. Se não forem especificados fonte e destino para as questões/respostas, a interação ocorre com a console do usuário. Mas, quando se quer modificar esta interação padrão, faz-se necessário o uso de predicados que manipulem a entrada e saída.

A primeira justificativa para a modificação da forma de interação é a de facilitar o uso da linguagem e a segunda é fazer com que as questões/respostas possam ser lidas/escritas de/em um arquivo. Na atual implementação do compilador Prolog do OPERA, são providos predicados para a saída de dados, estando estes descritos a seguir.

- **write(T)** : escreve o termo T;
- **nl** : nova linha.
- **statistics** : exibe dados estatísticos sobre a execução e a utilização das pilhas da WAM-E.

A-2.2 Predicados Aritméticos

Cálculos e manipulações de números podem ser feitos em todos os sistemas Prolog. Em alguns, pode-se apenas manipular números inteiros, mas, em outras versões de Prolog, pode-se realizar operações mais complexas. Existem predicados pré-definidos que efetuam operações

aritméticas e, no caso do Prolog OPERA, os predicados disponíveis não obedecem à sintaxe do C-Prolog, sendo descritos a seguir.

Os argumentos I1 e I2 devem estar ligados a valores inteiros, e R deve ser uma variável livre.

Tabela A-2.1 Predicados Aritméticos Prolog OPERA x C-Prolog

Prolog OPERA	C-Prolog	Significado
is(R,I1,+,I2)	R is I1 + I2	R←I1 + I2
is(R,I1,-,I2)	R is I1 - I2	R←I1 - I2
is(R,I1,*,I2)	R is I1 * I2	R←I1 * I2
is(R,I1,/,I2)	R is I1 / I2	R←I1 / I2

A-2.3 Predicados Lógicos

Da mesma forma, também estão disponíveis ao usuário alguns predicados lógicos pré-definidos, que são agora apresentados.

Tabela A-2.2 Predicados Lógicos Prolog OPERA

Sintaxe	Descrição
X,Y	X and Y
X;Y	X or Y
X = Y	X e Y são unificados
X < Y	o valor de X é menor que o valor de Y
X > Y	o valor de X é maior que o valor de Y
X =< Y	o valor de X é menor ou igual que o valor de Y
X >= Y	o valor de X é maior ou igual que o valor de Y
difnum(X,Y)	o valor de X é diferente do valor de Y
egnum(X,Y)	o valor de X é igual ao valor de Y

ANEXO A-3 PROGRAMAS EMPREGADOS NAS MEDIÇÕES

A 3.1 O programa fibo

```
%-------------------------------------------------------------
% Program :fibo
% Date    :06.12.93
%
% This program calculates the 12th number of fibonacci's series
%
% There is a loop to increase the processing time
%-------------------------------------------------------------

main :- statistics,
        exec(10),
        statistics.

exec(0):- !.
exec(N):-
         fibo(12, X),
         is(M,N,-,1),
         exec(M).

fibo(1,1).
fibo(2,1).

fibo(N,R) :-
            N > 2,
            is(N1, N, -, 1), is(N2, N, -, 2),
            fibo(N1,R1),
            fibo(N2,R2),
            is(R, R1, +, R2).
```

A 3.2 O programa timings

```
%-------------------------------------------------------------
% Program :timings
% Date    :06.12.93
%
% This program makes four p calls each one implementing a loop of
%  750 iterations
%
% There is a loop to increase the processing time
%-------------------------------------------------------------
```

```
main :- statistics,
        exec(1),
        statistics.

exec(0) :- !.
exec(R) :- p(750), p(750), p(750), p(750),
           is(T,R,-,1),
           exec(T).

p(0) :- !.
p(N) :- N > 0,
        is(Q,N,-,1), p(Q).
```

A 3.3 O programa comb

```
%-----------------------------------------------------------------
% Program :comb
% Date    :06.12.93
%
% This program calculates the combination of 10, taking 2 by 2
%
% There is a loop to increase the processing time
%-----------------------------------------------------------------

main :- statistics,
        exec(40),
        statistics.

exec(0) :- !.
exec(N) :- comb(10,2),
           is(M,N,-,1),
           exec(M).

comb(G,R) :- fat(G,X), fat(R,Y), is(Z,G,-,R), fat(Z,W),
             is(S,Y,*,W), is(T,X,/,S).

fat(0,1).
fat(N,R) :- N > 0,
            is(N1,N,-,1), fat(N1,R1), is(R,R1,*,N).
```

REFERÊNCIAS BIBLIOGRÁFICAS

[AIT 90] AIT-KACI, Hassan. **The WAM:** A (Real) Tutorial. Paris: Digital Equipment Corporation, Research Laboratory, Jan. 1990. 115p.

[ALI 90] ALI, Khayri; KARLSSON, Roland. **The Muse Or-Parallel Prolog Model and its Performance.** Kista: Swedish Institute of Computer Science, Mar, 1990. 20p. (Technical Report).

[ALI 91] ALI, Khayri; KARLSSON, Roland. Scheduling OR-Parallelism in Muse. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 8, Aug. 1991. **Proceedings...** Cambridge: MIT Press, 1991. p.807-821.

[BAR 88a] BARON, U.; ING, B.; RATCLIFFE, M. et al. A Distributed Architecture for the PEPSys Parallel Logic Programming System. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, Aug. 1988, Chicago. **Proceedings...** NewYork: IEEE Press, 1988. p.232-246.

[BAR 88b] BARON, U.; KERGOMMEAUX, J. et al. The Parallel ECRC Prolog System PEPSys:An Overview and Evaluation Results. In: INTERNATIONAL CONFERENCE ON FIFTH GENERATION COMPUTER SYSTEMS. May, 1988, Tokyo. **Proceedings...** Tokyo:ICOT Press, 1988. p.841-849.

[BAR 92] BARCELLOS, Antônio M. **O Sistema Operacional de Rede Heterogêneo HetNOS.** Porto Alegre: CPGCC-UFRGS, 1992, 239p. (Dissertação de Mestrado).

[BEA 91] BEAMONT, A.; RAMAN, S. M. et al. Flexible Scheduling of OR-Parallelism in Aurora: The Bristol Scheduler. In: PARLE 91 - PARALLEL ARCHITECTURE AND LANGUAGES EUROPE, June, 1991, Eindhoven. **Proceedings...** Berlin: Springer-Verlag, 1991. p.403-420. (Lecture Notes in Computer Science, 506).

[BOR 84] BORWARDT, Peter. Parallel Prolog Using Stacks Segments on Shared Memory Multiprocessors. In: INTERNATIONAL SYMPOSIUM ON LOGIC PROGRAMMING, Feb. 1984, San Francisco. **Proceedings...** New York: IEEE Press, 1984. p.2-11.

[BOS 90] BOSCO, P. G.; CECCHI, C. et al. Logic and Functional Programming on Distributed Memory Architectures. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 7., Sept. 1990. **Proceedings...** Cambridge: MIT Press, 1990. p.325-339.

[BRA 88] BRAND, P. **Wavefront Scheduling**. Kista: Swedish Institute of Computer Science, 1988. 19p. (Technical Report).

[BRI 90] BRIAT, J.; FAVRE, M.; GEYER, C. et al. **Opera:** a Parallel Prolog System and its Implementation on Supernode. Grenoble: Laboratoire de Genie Informatique de Grenoble /CAP-Gemini-Innovation, 1990. 19p. (Technical Report).

[BRI 90a] BRIAT, J.; FAVRE, M.; GEYER, C. et al. **Scheduling of OR-parallel Prolog on Scalable, Reconfigurable, Distributed-Memory Multiprocessor**. Grenoble: Laboratoire de Genie Informatique de Grenoble/CAP-Gemini-innovation, 1990. 18p. (Technical Report).

[BUT 86] BUTLER, R.; LUSK, E. L. et al. **ANLWAM**: A Parallel Implementation of the Warren Abstract Machine. Argonne: Argonne National Laboratories, 1986. 37p.

[BUT 88] BUTLER, Ralph et al. **Scheduling Or-Parallelism:** an Argonne perspective. Argonne: Argonne National Laboratory, 1988. 12p. (Technical Report).

[CAL 88] CALDERWOOD, Alan; SZEREDI, P. **Scheduling Or-Paralelism in Aurora - The Manchester Scheduler**. Manchester: Gigalips Project, 1988. 11p. (Technical Report).

[CAR 88] CARLTON, M.; VAN ROY, P. A Distributed Prolog System With AND Parallelism. **IEEE Software**, New York, n.1, p.43-51, Jan. 1988.

[CHA 85] CHANG, J.; DESPAIN, A. Semi-Intelligent Backtracking of Prolog Based on Static Data Dependency Analysis. **IEEE Software**, New York, v.1, n.6, p.10-21, Sept. 1985.

[CLA 81] CLARK, K.; GREGORY, S. A Relational Language for Parallel Programming. In: ACM CONFERENCE ON FUNCTIONAL LANGUAGES AND COMPUTER ACHITECTURE, 1981, New York. **Proceedings...** New York: ACM Press, 1981. p.189-208.

[CLA 86] CLARK, K.; GREGORY, S. PARLOG: Parallel Programming in Logic. **ACM TOPLAS**, New York, v.8, n.1, p.76-91. Jan. 1986.

[CLO 81] CLOCKSIN, W.F.; MELLISH, C.S. **Programming in Prolog.** Berlin: Springer-Verlag, 1981. 189p.

[CLO 88] CLOCKSIN, W. F.; ALSHAWI, H. A Method for Efficiently Executing Horn Clause Programs Using Multiple Processors. **New Generation Computing**, Berlin, v.3, n.5, p.361-376, 1988.

[CON 85] CONERY, J. S.; KIBLER, D.F. AND Parallellism and Nondeterminism in Logic Programs. **New Generation Computing**, Berlin, v.3, n.1, p.43-70, 1985.

[CON 87] CONERY, John S. **Parallel Execution of Logic Programs**. New York: Kluwer Academic Publishers, 1987. 234p.

[COR 91] CORBIN, John. **The Art of Distributed Applications**. Berlin: Springer-Verlag, 1991, 321p.

[CRA 88] CRAMMOND, J. **Implementation of Commited Choice Languages on Shared Memory Multiprocessors**. Edinburgh: Herriot-Watt University/Dept. of Computer Science, 1988. 237p. (PHD Thesis).

[DEG 84] DEGROOT, Doug. Restricted And-Parallelism. In: INTERNATIONAL CONFERENCE ON FIFTH GENERATION COMPUTER SYSTEMS. 1984, Tokyo. **Proceedings...** Tokyo: ICOT Press, 1984. p.471-478.

[DEG 87] DEGROOT, Doug. A Technique for compiling Execution Graph Expressions for Restricted And-Parallelism in Logic Programs. In: INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, June 1987, Athens. **Proceedings...** Berlin: Springer-Verlag, 1987. p.1074-1093 (Lecture Notes in Computer Science, v.297)

[DIJ 75] DIJKSTRA, Edsger W. Guarded Commands, Nondeterminancy and Formal Derivation of Programs. **Communications of the ACM**, New York, v.18, n.8, p.453-457, Aug. 1975.

[DIS 87] DISZ, T.; LUSK, E.; OVERBEEK, R. Experiments With OR-parallel Lógic Programs. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 4., May, 1987, Melbourne. **Proceedings...** Cambridge: MIT Press, 1987. p.576-599.

[DOR 85] DORBY, T., DESPAIN, A. M.; PATT, Y. Performance Studies of a Prolog Machines Architecture. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 12., 1985, Boston. **Proceedings...** Berlin: Springer-Verlag, 1985. p.481-495.

[DOU 91] DOUGLIS, Fred; OUSTERHOUT, John. Transparent Process Migration: Design Alternatives and Sprite Implementations. **Software - Practice and Experience**, New York, v.21, n.8, p.757-785, Aug. 1991.

[DWO 84] DWORK, C.; KANELLAKIS, P. C.; MITCHELL, J. C. On The Sequential Nature of Unification. **Journal of Logic Programming,** New York, v.1, n.1, p.9-15, June. 1984.

[FOS 89] FOSTER, I.; STEPHEN T. Strand: New Concepts in Parallel Programming. Englewood Cliffs: Prentice-Hall, 1989. 278p.

[FUR 92] FURUKAWA, Koichi. Logic Programming as the Integrator of the Fifth Generation Computer Systems Project. **Communications of the ACM,** New York, v.35, n.1, p.83-92, Mar. 1992.

[GEI 93] GEIST, Al; BUEGUELIN, Adam; DONGARRA, Jack et al. **PVM 3.0 User's Guide and Reference Manual**. Oak Ridge: Oak Ridge National Laboratory, 1993. 275p.

[GEY 91] GEYER, Cláudio F.R. **Une Contribution a L'Etude du Parallelisme OU en Prolog sur des Machines sans Mémoire Commune**. Grenoble: Université Joseph Fourier, 1991. 166p. (PHD Thesis).

[GEY 92] GEYER, Cláudio; YAMIN, Adenauer; WERNER, Otilia. Projeto OPERA: Um Modelo E/OU para Prolog. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, Rio de Janeiro, 1992. **Anais...** Rio de Janeiro: SBC, 1992. 390p. p.269-281.

[GRE 87] GREGORY, S. **Parallel Logic Programming in Parlog-The Language and its Implementation**. New York: Addison-Wesley, 1987. 267p.

[GUP 89] GUPTA, G.; JAYARAMAN, B. A Model for And-Or Parallel Execution of Logic Programs. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 10., Aug. 1989, Illinois. **Proceedings...** Berlin: Springer-Verlag, 1989. p132-149.

[GUP 90] GUPTA, G.; JAYARAMAN, B. Optimizing And-OR Parallel Implementation. In: NORTH-AMERICAN CONFERENCE ON LOGIC PROGRAMMING, Apr. 1990, Austin. **Proceedings...** New York: ACM Press, 1990. p.605-623.

[GUP 91] GUPTA, G.; COSTA, V. S. et al. IDIOM: Integrating Dependent AND-, Independent AND-, and OR-parallelism. In: INTERNATIONAL LOGIC PROGRAMMING SYMPOSIUM, 1991. **Proceedings...** Cambridge: MIT Press, 1991. p.152-166.

[HAR 90] HARIDI, S.; JANSON, S. Kernel Andorra Prolog and its Computation Model. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 7., Sept. 1990. **Proceedings...** Cambridge: MIT Press, 1990. p.31-46.

[HEL 88] HEWLETT PACKARD Co. **HP 9000 Computers Networking Reference**. Boston: Hewllett Packard co, 1988. 215p.

[HEL 89] HEWLETT PACKARD Co. **HP-UX Portability Guide**. Boston: Hewllett Packard co, 1989. 137p.

[HER 90] HERMENEGILDO, M.; ROSSI, F. &-Prolog and its Performance: Exploiting Independente And-Parallelism. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 7., Sept. 1990. **Proceedings...** Cambridge: MIT Press, 1990. p.325-339.

[HER 91] HERMENEGILDO, M.; GUPTA, G. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 8., Aug. 1991. **Proceedings...** Cambridge: MIT Press, 1991. p.146-158.

[HOG 84] HOGGER, C. J., **Introduction to Logic Programming.** London: Academic Press, 1984.

[JAN 91] JANSON, S.; HARIDI, S. Programming Paradigms of the Andorra Kernel Language. In: INTERNATIONAL LOGIC PROGRAMMING SYMPOSIUM, 1991. **Proceedings...** Cambridge: MIT Press, 1991. p.167-186.

[KAL 87] KALE, L. V. Parallel Execution of Logic Programs: The REDUCE-OR Process Model. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, May 1987, Melbourne. **Proceedings...** Berlin:Springer-Verlag, 1987. p.616-632. (Lecture Notes in Computer Science).

[KAS 83] KASIF, S.; KOHLI, M.; MINKER, J. PRISM: a Parallel Inference System for Problem Solving. Lisboa: Universidade de Nova Lisboa, 1983. 23p. (Relatório Técnico).

[KER 89a] KERGOMMEAUX, J.C. **Implémentation et Évaluation d'un Systeme Logique Parallele.** Grenoble: Universite Joseph Fourier-Grenoble I, 1989. 226p. (Docteur These).

[KER 89b] KERGOMMEAUX, J.C. **Measures of the PEPSys Implementation on the MX500.** Munchen: European Computer Research Center, 1989. 35p. (Technical Report CA-44).

[KHA 88] KHAROUNE, M. A Parallel Interpretation Model for Prolog. Paris: Université de Rennes I, 1988. 315p. (PHD Thesis).

[KOW 74] KOWALSKI, Robert. **Logic For Problem Solving**. New York: Elsevier, 1979.

[KUM 92] KUMON, K.; ASATO, A. et al. Architecture and Implementation of PIM/p. In: INTERNATIONAL CONFERENCE ON FIFTH GENERATION COMPUTER SYSTEMS, 1992, Tokyo. **Proceedings...** Tokyo: ICOT Press, 1981. p.50-72.

[LIN 84] LINDSTRON, G. Or Parallelism on Aplicative Architectures. In: INTERNATIONAL LOGIC PROGRAMMING CONFERENCE, 2., 1984. **Proceedings...** New York: ACM Press, 1984. p.31-47.

[LLO 84] LLOYD, J. W. **Foundations of Logic Programming.** Berlim: Springer-Verlag, 1984.

[LUS 88] LUSK, Ewing et al. **The Aurora Or-Parallel Prolog System.** Manchester: Department of Computer Science, May, 1988. 19p. (Technical Report-Gigalips Project).

[MAZ 86] MASUZAWA, Hideo; KUMON, Kouichi; ITASHIKI, Akihiro. Kabu-Wake: A New Parallel Inference Method and Its Evaluation. In: INTERNATIONAL CONFERENCE On FIFTH GENERATION COMPUTER SYSTEMS, Nov. 1986, New York. **Proceedings...** New York: IEEE Press, 1986. 386p. p.168-172.

[MOO 91] MOOLENAR, R.; HECKER, V. H.; DEMOEN, B. A Parallel Implementation of AKL. In: INTERNATIONAL LOGIC PROGRAMMING SYMPOSIUM, Aug. 1991, **Proceedings...** Cambridge: MIT Press, 1991. p.156-163.

[MOT 81] MOTO-OKA, T. Challenge for Knowledge Information Processing Systems. In: INTERNATIONAL CONFERENCE ON FIFTH GENERATION COMPUTER SYSTEMS, Sept. 1981, Tokyo. **Proceedings...** Tokyo: ICOT, 1981. p.3-89.

[MUT 89] MUTHUKUMAR, K.; HERMENEGILDO, M. V. Complete and Efficient Methods for Supporting Side-effects in Independent/Restricted AND-Parallelism. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 6., 1989. **Proceedings...** Cambridge: MIT Press, 1989. p.80-97.

[MUT 89a] MUTHUKUMAR, K.; HERMENEGILDO, M. V. Determination of Variable Dependence Information Through Abstract Interpretation. In: NORTH AMERICAN CONFERENCE ON LOGIC PROGRAMMING, Sept. 1989, Boston. **Proceedings...** New York: IEEE Press, 1989. p.166-.188.

[PAL 91] PALMER, D.; NAISH, L. NUA-Prolog: An Extension to the WAM for Parallel Andorra. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 8., Aug. 1991. **Proceedings...** Cambridge: MIT Press, 1991. p.429-442.

[PER 87] PEREIRA, Fernando et al. C-Prolog User's Manual - Version 1.5 and 1.5+. Edinburgh: University of Edinburgh-Department of Computer Science, 1987. 141p.

[ROY 92] ROY, PETER V.; DESPAIN, A. High-Performance Logic Programming With the Aquarius Prolog Compiler. **Computer,** New York, v.25, n.1, p13-30, Jan. 1992.

[SAR 86] SARASWAT, V. A. **Problems With Concurrent Prolog.** Atlanta: Carnegie-Mellon University, Jan. 1986. 87p. (Technical Report CS-86-100).

[SAT 88] SATO, M.; GOTO,A. Evaluation of the KL1 Parallel System on Shared Memory Multiprocessor. In: WORKING CONFERENCE ON PARALLEL PROCESSING, May. 1988, North-Holland. **Proceedings...** Berlin: Springer-Verlag, 1988. p.77-91.

[SHA 83] SHAPIRO, E. **A Subset of Concurrent Prolog and its Interpreter**. Rehovot: Weizmann Institute, 1983. 47p.

[SHA 89] SHAPIRO, E. The Family of Concurrent Logic Programming Languages. **ACM Computing Surveys**, New York, v.21, n.3, p12-18, Sept. 1989.

[SIL 91] SILICON GRAPHICS COMPUTERS SYSTEMS. **IRIX Programming Guide**. Mountain View: Silicon Graphics Co., 1991. v.1, 568p.

[SIL 91a] SILICON GRAPHICS COMPUTERS SYSTEMS. **IRIX Programming Guide**. Mountain View: Silicon Graphics Co., 1991. v.2, 536p.

[SIM 86] SIMON, C. Specification and Simulation of Prolog Multiprocessor Architecture. Toulouse: Université P. Sebatier, 1986. 287p. (PHD Thesis).

[SOM 88] SOMOGYI, Z.; RAMAMOHANARAO, K.; VAGHANI, J. A Stream AND-Parallel Execution Algorithm With Backtracking. In: INTERNATIONAL CONFERENCE AND SYMPOSIUM ON LOGIC PROGRAMMING, 5., 1987, Massachusts. **Proceedings ...** Cambridge: MIT Press, 1987 v.2, 425p. p.134-145.

[STE 86] STERLING, L; SHAPIRO, E. **The Art of Prolog**. Cambridge: MIT Press, 1986.

[STE 90] STEVENS, W. R. **Unix Network Programming.** Englewood Cliffs: Prentice-Hall, 1990. 772p.

[SUN 88] SUN MICROSYSTEMS INC. **Network Programming**. New York: Sun Microsystems Inc., 1988. 297p.

[SZE 89] SZEREDI, Peter. Performance analysis of the Aurora OR-Parallel Prolog System. In: NORTH AMERICAN CONFERENCE ON LOGIC PROGRAMMING, Sept. 1989, Boston. **Proceedings...** New York: IEEE Press, 1989. p371-486.

[TAK 84] TAKI, K.; YOKOTA, M. et al. Hardware Design and Implementation of the Personal Sequential Inference Machine (PSI). In: INTERNATIONAL CONFERENCE ON THE FIFTH GENERATION COMPUTER SYSTEMS, 1984, Tokyo. **Proceedings...** Tokyo: 1984. p.137-152.

[TAK 92] TAKI, Kazuo. Parallel Inference Machine. In: INTERNATIONAL CONFERENCE ON FIFTH GENERATION COMPUTER SYSTEMS, June, 1992, Tokyo. **Proceedings...** Tokyo: ICOT Press, 1981. p.50-72.

[TAM 84] TAMURA, N.; WADA, K. et al. Sequential Prolog Machine PEK. In: INTERNATIONAL CONFERENCE ON THE FIFTH GENERATION COMPUTER SYSTEMS, 1984, Tokyo. **Proceedings...** Tokyo: ICOT Press, 1984. p67-81.

[TAN 89] TANENBAUM, A. S. **Computer Networks.** 2nd. Englewood Cliffs: Prentice-Hall, 1989. 658p.

[THE 89] THEIMER, Marvin.; LANTZ, Keith. Finding Idle Machines in a Workstation-based Distributed System. **IEEE Transactions on Software Engineering**, New York, v.15, n.11, p.1444-1458, Sept. 1988.

[UED 85] UEDA, K. Guarded Horn Clauses. In: CONFERENCE ON LOGIC PROGRAMMING, 4., 1985, Tokyo. **Proceedings...** Tokyo: ICOT Press, 1985. 436p. p.65- 81.

[UED 90] UEDA, K.; CHIKAYAMA, T. Design of the Kernel Language for The Parallel Inference Machine. **Computer Journal,** New York, v.33, n.6, p.237-248, 1990.

[WAR 83] WARREN, David D.H. **An Abstract Prolog Instruction Set**. Manchester: SRI International, Oct. 1983. (Technical Note, 309).

[WAR 87] WARREN, David. H.D. The SRI Model for Or-Parallel Execution of Prolog - Abstract Design and Implementation Issues. In: SYMPOSIUM On LOGIC PROGRAMMING, Aug, 1987, San Francisco. **Proceedings...** New York:IEEE Press, 1987. p.92-102.

[WAR 90] WARREN, D.H.D. The Extended Andorra Model With Implicit Control. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 7., Sept. 1990. **Proceedings...** Cambridge: MIT Press, 1990. p.135-145.

[WER 94] WERNER, Otilia. **Uma Máquina Abstrata Estendida para o Paralelismo E na Programação em Lógica**. Porto Alegre: CPGCC-UFRGS, 1994. 145p. (Dissertação de mestrado).

[YAM 92] YAMIN, Adenauer C. **Modelos de Implementação do Paralelismo OU na Programação em Lógica**. Porto Alegre: CPGCC-UFRGS, 1992. 114p. (Trabalho Individual, 280).

[YAM 93] YAMIN, Adenauer C.; WERNER, Otilia; GEYER, Cláudio F.R. Prolog Paralelo em Rede de Computadores. In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES - PROCESSAMENTO DE ALTO DESEMPENHO, 5., Set. 1993, Florianópolis. **Anais...** Florianópolis: SBC, 1993. 775p. p.290-303.

[YAN 86] YANG, R; AISO, H. P-Prolog: A Parallel Logic Language Based on Exclusive Relation. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 3., 1986, London. **Proceedings...** Berlin: Springer-Verlag, 1986. 347p. p.237-251. (Lecture Notes in Computer Science).

[YAN 93] YANG, R.; BEAUMONT, T. et al. Performance of the Compiler-based Andorra-I System. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 10., 1993, **Proceedings...** Cambridge: MIT Press, 1993. p.150-166.

Informática
UFRGS

Um Ambiente para Exploração de Paralelismo na Programação em Lógica.

Dissertação apresentada aos Senhores:

Profa. Dra. Ana Maria de Alencar Price

Profa. Dra. Ingrid Eleonora Jansch Pôrto

Prof. Dr. Rogério Drummond P. de Mello Filho (DCC/UNICAMP)

Vista e permitida a impressão.
Porto Alegre, 31/8/94.

Prof. Dr. Cláudio Fernando Resin Geyer,
Orientador.

Prof. Dr. Philippe Olivier Alexandre Navaux,
Co-orientador.

Prof. Dr. José Palazzo Moreira de Oliveira,
Coordenador do Curso de Pós-Graduação
em Ciência da Computação.