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

Exploração de Paralelismo OU em uma Linguagem em Lógica com Restrições

por
Patrícia Kayser Vargas

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

Prof. Dr. Cláudio Fernando Resin Geyer
Orientador

Porto Alegre, agosto de 1998.

CIP – Catalogação na Publicação

Vargas, Patrícia Kayser

Exploração de Paralelismo OU em uma Linguagem em Lógica com Restrições / por Patrícia Kayser Vargas. – Porto Alegre: CPGCC da UFRGS, 1998.

95f.il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, BR-RS, 1998. Orientador: Geyer, Cláudio Fernando Resin.

1. Programação em lógica com restrições. 2. Processamento distribuído. 3. Paralelismo OU. 4. Escalonamento de tarefas. I. Geyer, Cláudio Fernando Resin. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitora: Profa. Wrana Panizzi

Pró-Reitor de Pós-Graduação: Prof. José Carlos Ferraz Hennemann

Diretor do Instituto de Informática: Prof. Philippe Olivier Alexandre Navaux

Coordenador do CPGCC: Profa. Carla Maria Dal Sasso Freitas

Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

Para ser grande, sê inteiro: nada
Teu exagera ou exclui.
Sê todo em cada coisa. Põe quanto és
No mínimo que fazes.
Assim em cada lago a lua toda
Brilha, porque alta vive.

FERNANDO PESSOA
(Odes de Ricardo Reis)

Agradecimentos

Gostaria de agradecer ao meu orientador Cláudio Geyer pelo apoio e estímulo que foram tão importantes nessa etapa, e pela orientação e ensinamentos transmitidos durante toda a minha formação acadêmica.

Aos colegas e amigos do projeto OPERA e APPELO, Denise Bandeira, Cristiano Costa, Luís Fernando Castro, Ana Paula Centeno, Débora Nice Ferrari e Jorge Barbosa, pelo companheirismo e pelas discussões tão importantes para o desenvolvimento deste trabalho. Aos auxiliares de pesquisa Maíra Wenzel, Bruno Roth e Paula Bronfman pelo apoio.

Ao colega de projeto APPELO Adenauer Yamim, por me iniciar à pesquisa com o seu exemplo e por todos os ensinamentos, carinho e força transmitidos.

Aos colegas de mestrado, por todos os momentos e dúvidas compartilhados.

A professora Inês de Castro Dutra pelas inúmeras contribuições.

Aos professores do Instituto de Informática pelos ensinamentos e aos funcionários pela receptividade.

Ao CNPq pelo apoio financeiro, sem o qual esse trabalho não teria sido realizado.

Ao Marco por todas as discussões sobre o meu trabalho, pelo incentivo constante, mas principalmente pela compreensão e amor e por ser tão especial.

À toda a minha família pelo apoio e compreensão, e de forma especial a minha mãe para a qual não existem palavras que expressem o quanto sou grata por tudo.

Sumário

Lista de Abreviaturas

CLP	*Constraint Logic Programming* (Programação em Lógica com Restrições).
CLP(FD)	*Constraint Logic Programming Over Finite Domain* (CLP sobre domínios finitos).
LPE	Lista de Pontos de Escolha
OO	Orientação a Objetos
PCLP(FD)	*Parallel Constraint Logic Programming Over Finite Domain* (CLP(FD) Paralelo).
PO	Pesquisa Operacional
TAMAGOSHI	***Ta**sk **Si**mulator **Pro**gram and **OR**-**Sch**eduler **I**nterface*
WAM	*Warren Abstract Machine* (Máquina Abstrata de Warren).

Lista de Figuras

Lista de Tabelas

Resumo

Este trabalho é dedicado ao estudo da exploração de paralelismo OU na programação em lógica com restrições em ambientes distribuídos. A programação em lógica, cuja linguagem mais significativa é Prolog, tem como premissa a utilização da lógica de predicados como linguagem computacional. A programação em lógica com restrições (*CLP*) é uma extensão da programação em lógica, onde busca-se a eficiência e a possibilidade de executar novas classes de problemas. Variáveis em CLP podem pertencer a domínios específicos como, por exemplo, reais ou booleanos. O principal conceito introduzido é a restrição. Restrição é uma equação que representa uma certa informação sobre uma variável e a sua relação com outras variáveis. O uso de restrições foi proposto para diminuir o espaço de busca na execução dos programas.

Apesar de mais eficientes que a programação em lógica clássica, para algumas aplicações reais o desempenho das linguagens CLP ainda é insatisfatório. Por isso, é necessário buscar alternativas novas como a execução em paralelo. A exploração de paralelismo implícito em programas em lógica já demonstrou resultados promissores. Vários modelos foram propostos e implementados utilizando as duas principais fontes de paralelismo – E e OU – de forma isolada ou combinada.

O objetivo principal desse trabalho é apresentar o modelo pclp(FD) de exploração de paralelismo OU multi-seqüencial para um ambiente com memória distribuída. O modelo pclp(FD) caracteriza-se pela existência de vários trabalhadores, cada um deles possuindo uma máquina abstrata completa. O escalonamento de tarefas é realizado por uma política dinâmica e distribuída. Uma tarefa em pclp(FD) equivale a um ponto de escolha e a um contexto de execução. O contexto de execução é formado por porções da pilha do exportador.

Para que o importador tenha acesso ao contexto de execução utiliza-se a cópia incremental, que é uma das várias técnicas possíveis. Cada trabalhador possui a sua própria cópia privada das pilhas de execução. A cópia caracteriza-se pelo envio das pilhas de execução do exportador para uma área privada do importador. A cópia incremental é uma técnica mais otimizada que verifica a existência de partes comuns entre os trabalhadores, copiando apenas as partes novas. O algoritmo de cópia incremental proposto no modelo é feito sem nenhuma centralização de informação do estado das pilhas.

O projeto e implementação de um protótipo para esse modelo, utilizando a linguagem clp(FD), que implementa CLP sobre domínios finitos, permitirá uma análise das vantagens e desvantagens do modelo proposto. Os resultados obtidos com a análise servirão de base para trabalhos futuros, visando aprimorar a implementação e o modelo.

Palavras-Chaves: Programação em Lógica com Restrições, Processamento Distribuído, Paralelismo OU, Escalonamento de Tarefas.

TITLE: "OR Parallelism Exploitation in a Constraint Logic Language"

Abstract

This work is dedicated to the study of the exploration of OR parallelism in Constraint Logic Programming for distributed environment. Logic Programming, which the most meaningful language is Prolog, has as premise the use of the logic of predicates as computational language. Constraint Logic Programming or CLP is an extension of the logic programming, where efficiency and the possibility to execute new kinds of problems are searched. A variable in CLP can belong to specific domains as, for example, Real or Boolean. The main concept introduced is the constraint. Constraint is an equation that represents a certain information over a variable and its relation with others variables. The use of constraints was proposed to decrease search space in the program execution.

Although it is more efficient than classic logic programming, for some real applications, the performance of CLP languages still is unsatisfactory. So, it is necessary to search alternatives as parallel execution. The exploration of implicit parallelism in programs in logic has already demonstrated promising results. Several models have been proposed and implemented using the two main sources of parallelism - AND and OR – in an isolated or combined form.

The main objective of this work is to present the pclp(FD) model of exploration of multi-sequential OR parallelism for a distributed memory environment. The pclp(FD) model is characterized for the existence of some workers, each one of them possessing a complete abstract machine. Task scheduling is executed by one dynamic and distributed policy. A task in pclp(FD) is equivalent to a choice point and an execution context. Execution context is formed by portions of the stack of the exporter.

So that importer has access to the execution context, it uses incremental copy, which is one of the several possible techniques. The copy is characterized for sending execution stacks of the exporter to a private area of the importer, that is, each worker possesses its private copy of the execution stacks. The incremental copy is a more optimized technique that verifies the existence of common parts between workers, copying only the new ones. The incremental copy algorithm proposed in the model executes without centralized information of the state of the stacks.

A prototype project and implementation for this model, using the language clp(FD), that implements CLP over finite domains, will allow an analysis of advantages and disadvantages of the considered model. The results gotten with the analysis will serve of base for future works, aiming to improve the implementation and the model.

Keywords: Constraint Logic Programming, Distributed Processing, OR Parallelism, Task Scheduling.

1 Introdução

O tema deste trabalho é o estudo sobre a paralelização de programas em lógica com restrições. A partir de um estudo inicial das técnicas para execução paralela de programas em lógica (com restrições), concebeu-se um modelo de exploração implícita de paralelismo OU multi-seqüencial, direcionado para máquinas de memória distribuída.

1.1 Motivação

A programação em lógica tem como premissa a utilização da lógica de predicados como linguagem computacional. A sua simplicidade e a sua característica declarativa (ao contrário da característica imperativa das linguagens tradicionais) estimulam a sua utilização em aplicações de alta complexidade e em prototipação [COD 95a]. Isso se deve ao fato de um programa ser simplesmente um conjunto de fórmulas lógicas no qual há somente uma estrutura de dados: os termos de primeira ordem.

A linguagem de programação em lógica mais difundida é Prolog. As primeiras implementações são da década de 70 e possuem baixa eficiência, tanto em tempo de execução quanto na utilização de memória. Apesar dos inúmeros avanços, até hoje as linguagens em lógica são rotuladas de ineficientes.

A maioria das implementações atuais utiliza como base a máquina abstrata de Warren (WAM) [AIT 91]. Esse modelo de compilação e execução permitiu a construção de sistemas para a resolução de aplicações simbólicas com desempenho equivalente ao das linguagens imperativas. No entanto, existe uma série de aplicações cuja característica combinatorial causa um aumento exponencial na árvore de busca e, conseqüentemente, um aumento no consumo de memória e no tempo de execução, dificultando a utilização desta linguagem.

A programação em lógica com restrições (*Constraint Logic Programming* ou *CLP*) [JAF 94][FRU 93][COD 95a][COH 90] é uma extensão da programação em lógica. Seus objetivos são aumentar a eficiência e permitir a execução de novas classes de problemas, principalmente com características combinatoriais.

CLP estende o domínio clássico (termos de primeira ordem) pela possibilidade de utilizar domínios de cálculo particulares, tais como booleanos, domínios finitos, reais e inteiros. Sobre variáveis pertencentes a um domínio particular é possível definir restrições (*constraints*). Restrições são equações que representam uma certa informação sobre uma variável e a sua relação com outras variáveis.

A execução de linguagens de programação em lógica com restrições utiliza mecanismos de controle que descartam caminhos inválidos, isto é, que caminhos não podem levar ao resultado. Deste modo, minimiza-se a árvore de busca e conseqüentemente o processamento desnecessário inerente e o retrocesso (*backtracking*).

CLP atualmente é uma das áreas de pesquisa mais promissoras da programação em lógica. Com desempenho comparável ao de linguagens imperativas para várias

aplicações, possui inúmeras aplicações tanto a nível acadêmico quanto industrial. Várias linguagens foram criadas e, apesar de mais eficientes que a programação em lógica clássica, para algumas aplicações reais o seu desempenho ainda é insatisfatório. Por isso, é necessário melhorar o desempenho, utilizando novas técnicas de execução seqüencial e/ou paralela.

Tanto os programas em lógica, quanto os programas em lógica com restrições, são inerentemente paralelos. Essa característica permite a exploração de paralelismo implícito. A principal vantagem do paralelismo implícito é a transparência para o usuário, que não precisa se preocupar com a forma de execução paralela do seu programa. Existem inúmeras implementações paralelas para programação em lógica e várias para CLP. No entanto, ainda há muitos pontos a serem pesquisados, uma vez que a maioria das implementações não são completas ou comerciais.

1.2 Contexto Local

Esse trabalho foi desenvolvido no âmbito do projeto OPERA - Prolog Paralelo, do Instituto de Informática da UFRGS. O objetivo geral deste projeto é o estudo das diversas formas de exploração de paralelismo na programação em lógica.

Diversos trabalhos foram desenvolvidos tais como:

- modelagem e implementação de um sistema de exploração de paralelismo OU multi-seqüencial em uma máquina Transputer [GEY 92]. Esse sistema é bastante eficiente, porém dependente da arquitetura, o que dificulta a sua portabilidade. Uma revisão deste modelo gerou uma nova implementação denominada PLoSys [MOR 97];
- modelagem e prototipação de um sistema de exploração de paralelismo E independente em uma rede de estações [YAM 94] [WER 94]. Esse modelo, assim como os modelos de paralelismo OU citados, utilizam políticas de escalonamento centralizada;
- definição do modelo de análise de granulosidade para paralelismo E e OU denominado GRANLOG [BAR 95] e a implementação de um protótipo para informações E [VAR 95]. Estas informações podem ser utilizadas como auxílio a decisões de escalonamento de tarefas;
- sistema de interpretação abstrata ParTy que obtém informações de tipos para programas Prolog [CAL 96]. Estas informações auxiliam a análise de granulosidade.

Também encontra-se em andamento o estudo de uma política de escalonamento distribuída de tarefas para um modelo que integre E independente e OU [COS 96].

Todos esses trabalhos buscam técnicas para aprimorar o desempenho da programação em lógica. Deste modo, o estudo da programação em lógica com restrições em paralelo vai ao encontro dos objetivos do projeto, uma vez que o uso de restrições pode ser considerado uma nova técnica de otimização da programação em lógica, ao mesmo tempo que concorre para uma melhoria na exploração implícita do paralelismo neste tipo de linguagem.

1.3 Proposta

O objetivo principal deste trabalho é a apresentação de um modelo de exploração de paralelismo em CLP concebido especificamente para detectar implicitamente o paralelismo OU. A partir deste modelo, projetou-se e implementou-se um protótipo com o objetivo de avaliar o modelo.

Deste modo, destaca-se como principais contribuições deste trabalho:

- a definição de um modelo de exploração de paralelismo genérico e modular. Este modelo, pode ser aplicado a qualquer linguagem que utilize, na sua implementação, uma máquina abstrata nos moldes da WAM. Para desenvolver um modelo viável em termos de implementação, vários pontos tiveram que ser definidos, como por exemplo, a interface entre trabalhador e escalonador;
- a proposta de um algoritmo de escalonamento distribuído. A utilização de algoritmos distribuídos é importante quando deseja-se permitir a escalabilidade do sistema. O algoritmo proposto tem como principal vantagem a simplicidade do seu protocolo;
- a definição de um mecanismo de cópia incremental que pode ser feita de forma totalmente distribuída;
- a flexibilidade do protótipo implementado. Minimizou-se a interferência na máquina abstrata e grande parte da gerência de paralelismo foi implementada em Java. Estes dois aspectos facilitam o porte para outros sistemas de programação em lógica;
- a criação de novas frentes de pesquisa dentro do projeto, uma vez que ainda não tinha sido realizado nenhum trabalho com linguagens CLP no escopo do Projeto OPERA.

O restante do texto encontra-se organizado do seguinte modo. No Capítulo 2 introduz-se os principais conceitos de programação em lógica e CLP encontrados na literatura, que são relevantes ao desenvolvimento deste trabalho. No Capítulo 3 apresenta-se vários aspectos importantes da exploração de paralelismo nestes dois paradigmas. No Capítulo 4 apresenta-se um modelo para exploração de paralelismo OU em CLP denominado pclp(FD). No Capítulo 5, o protótipo deste modelo é analisado. Finalmente, no Capítulo 6 apresenta-se as conclusões e os possíveis trabalhos futuros.

2 Programação em Lógica com Restrições

O objetivo deste capítulo é apresentar as principais características de CLP e da Programação em Lógica que são relevantes a este trabalho. São discutidas as principais diferenças entre CLP e a programação em lógica tradicional. Também são apresentados alguns conceitos e terminologia utilizados neste trabalho.

2.1 Programação em Lógica e Prolog

Um programa de computador é formado por um algoritmo e um conjunto de dados. Um algoritmo pode ser descrito pela seguinte equação [KOW 79b]:

ALGORITMO = LÓGICA + CONTROLE.

A lógica é a definição do problema a ser resolvido. O controle coordena o fluxo de execução. Nas linguagens imperativas o controle e a lógica são codificados juntos.

Na programação em lógica é necessário apenas expressar de forma declarativa o problema, enquanto o controle da execução fica transparente.

A proposta de utilização da lógica como linguagem computacional originou o paradigma de programação em lógica [KOW 79a]. Suas principais vantagens são o seu alto poder de expressão e a ausência de informações explícitas para controle da execução.

A execução de programas em lógica é feita por um mecanismo interno à linguagem. Ao resolver um predicado lógico, podem existir várias alternativas (cláusulas) e o sistema de resolução precisa decidir qual caminho seguir. Como o sistema desconhece qual ou quais das várias alternativas podem levar a uma solução satisfatória, ele deve percorrer todos os caminhos ou tentar inferir qual o correto. Esta característica é conhecida como *don't-know nondeterminism* e é, normalmente, a origem do problema típico de explosão combinatorial dessas linguagens [SAR 93].

A linguagem de programação em lógica mais conhecida é o Prolog [STE 94], cujas primeiras implementações são da década de 70. Prolog não pode ser classificada como uma linguagem em lógica pura. Um motivo é a existência de predicados pré-definidos (*built-ins*) que possuem efeitos colaterais, utilizados principalmente para a manipulação de entrada/saída e para cálculos matemáticos. Outra razão é que o programador pode alterar a ordem de execução normal do programa. Interfere-se no mecanismo de execução através do uso de predicados extra-lógicos, principalmente do *cut*.

2.1.1 Conceitos Principais

Faz-se necessário apresentar a terminologia e conceitos que são utilizados no decorrer deste texto. Estes conceitos são relativos à linguagem Prolog, mas vários também são válidos para outras linguagens de programação em lógica, inclusive as linguagens CLP.

As estruturas de dados manipuladas são os **termos** de primeira ordem. Um termo pode ser tanto uma **variável**, iniciada com letra maiúscula ou *underscore* (_), uma **constante**, iniciada com letra minúscula, ou uma **estrutura,** com o formato *f*(*t1*, ..., *tn*), onde *f* é um identificador (*functor*) e *ti* é um termo.

Na programação em lógica, cada **cláusula** expressa que um número (que pode ser zero) de condições implica em um número (que pode ser zero) de conclusões alternativas. As cláusulas que contêm no máximo uma conclusão são chamadas cláusulas de Horn e são a base da linguagem Prolog.

Um programa Prolog é formado por um conjunto de assertivas, que são denominadas cláusulas de Horn, ou simplesmente cláusulas, formadas por termos. As cláusulas podem ser tanto um **fato** quanto uma **regra**. Um fato é simples e incondicional, enquanto uma regra é condicional. Cada regra é formada por uma cabeça e um corpo separados pelo símbolo ":-" . O corpo de uma cláusula, isto é, o lado direito do símbolo ":-", possui literais denominados **objetivos** (*goals*). Esses objetivos podem ser tanto predicados pré-definidos quanto predicados definidos pelo usuário. Uma **consulta** (*query*) é um tipo especial de regra que não possui cabeça (é sempre verdadeira) e, por isso, deve ser sempre avaliada.

Pode-se utilizar uma interpretação procedural dos programas Prolog. Neste caso, um conjunto de **cláusulas** (regras e/ou fatos) definem um **predicado**. Cada predicado é identificado por um nome *f* (**functor**) e pelo número de argumentos *n* (**aridade**), sendo normalmente referenciado como predicado *f*/*n*. As cláusulas que formam um predicado representam logicamente uma disjunção ou ($\vee$), isto é, basta que uma delas seja verdadeira para que o predicado também o seja. Para que uma cláusula seja verdadeira, é necessário que todos os seus objetivos também o sejam, o que representa uma conjunção lógica e ($\wedge$).

Através dessa interpretação procedural, pode-se considerar cada objetivo do corpo de uma cláusula como uma chamada de procedimento. Em um dado passo da execução de uma cláusula, um conjunto de objetivos que ainda devem ser executados, isto é, a continuação da computação, é chamada de **resolvente** corrente. A execução de um programa começa com a colocação da consulta como resolvente inicial e prossegue pela transformação da resolvente corrente em uma nova resolvente, através da execução de operações específicas, até a obtenção do resultado final.

Durante a execução, uma das operações básicas é a **unificação** [STE 94]. Um unificador (*unifier*) de dois termos é a substituição que torna dois termos idênticos. Se dois termos possuem um unificador, diz-se que eles unificam (*unify*). Um algoritmo de unificação obtém o unificador mais geral de dois termos, se ele existir, ou indica falha caso contrário. O unificador mais geral é aquele que limita o mínimo possível o escopo de valores das variáveis, deixando a maior liberdade possível às instanciações posteriores.

Outra operação fundamental é o **retrocesso** (*backtracking*), que permite descartar parte do processamento e avaliar diferentes alternativas, e será ilustrada na próxima seção.

2.1.2 Exemplo de Execução de Programa Prolog

Em Prolog, a execução de um programa cria uma árvore de busca que permite a execução dos diferentes caminhos alternativos que podem levar ao resultado final. Por

exemplo, considere-se o programa para o cálculo de fatorial (**figura 2.2**), obtido quase diretamente da definição lógica de fatorial (figura 2.1):

$$n! \begin{cases} n=0, \ 1 \\ n \geq 1, \ n*(n-1)! \end{cases}$$

FIGURA 2.1 - Definição matemática do cálculo de fatorial.

```
fatorial(0,1).
fatorial(N,R) :-
     N>= 1,
     N1 is N-1,
     fatorial(N1,F),
     R = N*F.
```

FIGURA 2.2 - Cálculo de fatorial codificado em Prolog.

O predicado fatorial/2 possui duas cláusulas que representam duas alternativas possíveis na execução de uma chamada a este predicado. Para calcular o fatorial de 3, utiliza-se a consulta:

```
:- fatorial(3, X).
```

Cujo resultado será

```
X = 6.
```

A execução desta consulta resulta na construção da árvore de busca ilustrada na figura 2.3. O resultado é obtido através da consulta aos ramos desta árvore, que representam os diferentes caminhos alternativos que podem conduzir a resultados válidos. Essa árvore de busca é percorrida em uma ordem pré-fixada da esquerda para direita e em profundidade (*depth-first mechanism*).

Cada predicado com mais de uma cláusula gera um **ponto de escolha** (*choice point*) a partir do qual existem alternativas (ramos). Cada ramo da árvore é avaliado. Caso não seja possível prosseguir por um caminho, isto é, caso uma alternativa não seja uma opção válida, a execução deve retornar ao último ponto de escolha com alternativas pendentes e o próximo ramo deve ser tentado. Esse retorno ao último ponto de escolha é efetuado por um mecanismo denominado **retrocesso** (*backtracking*), cujo objetivo é descartar parte do processamento e encontrar a próxima alternativa ainda não avaliada.

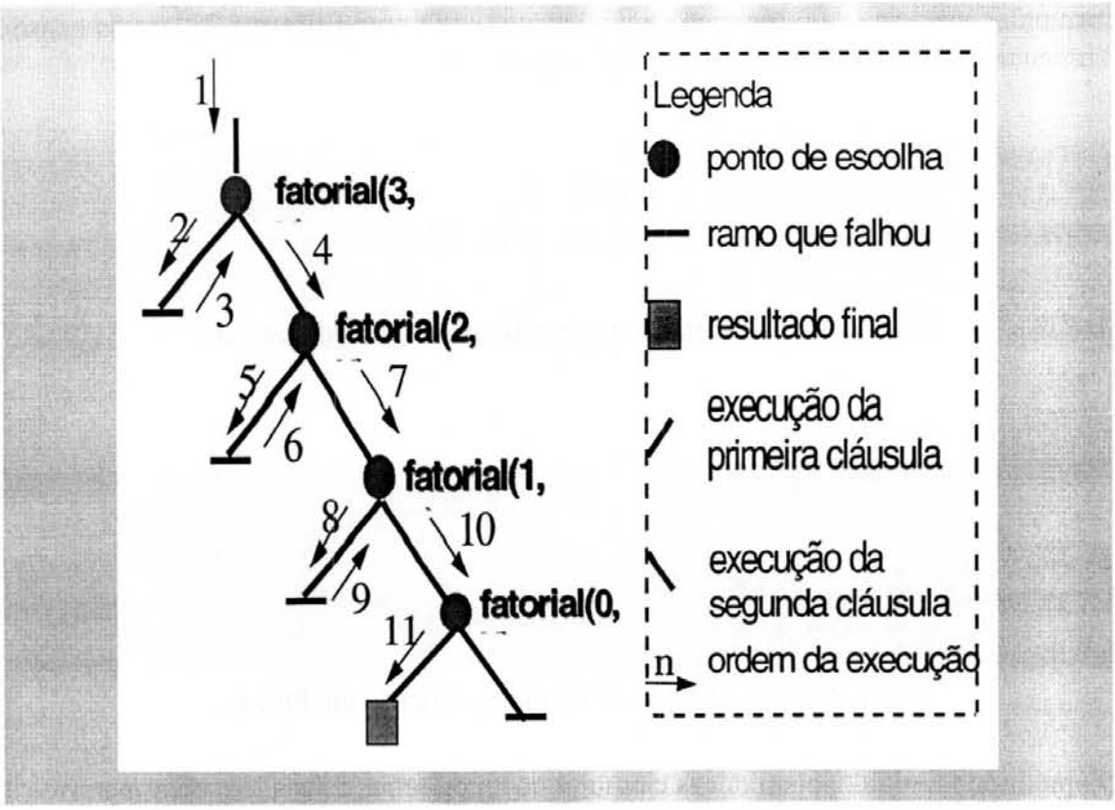

FIGURA 2.3 - Árvore de busca para a consulta *:- fatorial(3,X).*

Para algumas classes de problemas, a árvore de busca dos programas Prolog se torna muito grande, acarretando grande uso de memória para manter as estruturas de dados que controlam a execução. Entretanto, uma árvore é formada em grande parte por ramos que não levam ao resultado, mas que somente são detectadas no momento que falham e causam retrocesso. Ainda que existam mecanismos para diminuir o espaço de busca, como o predicado *cut*, muitos problemas não podem ser resolvidos em tempo e espaço viável. Normalmente, esses problemas são do tipo combinatoriais como, por exemplo, alocações de recursos finitos e composição de horários.

2.2 Programação em Lógica com Restrições

Os termos restrições (*constraints*) e propagação de restrições (*constraint propagation*) têm sido amplamente utilizados em Inteligência Artificial para designar diferentes tipos de técnicas, utilizadas normalmente para tratar problemas combinatoriais com o objetivo de restringir o espaço de busca, minimizando o gasto de memória e de tempo de processamento. Várias linguagens foram criadas com essas características, como THINGLAB (orientada a objetos) e CHARME (imperativa). Deste modo, é natural que tenha sido proposta uma programação em lógica com restrições (*Constraint Logic Programming - CLP*) [JAF 94] [FRU 93] [COH 90] [COD 95a] [VAR 97a] em que vários princípios das linguagens em lógica e das linguagens com restrições foram empregados de modo a obter um melhor desempenho.

A programação em lógica com restrições é uma extensão da programação em lógica onde introduz-se:

1. a possibilidade de uma variável ser definida em função de um domínio matemático;

2. a noção de **restrições** (*constraints*) nos domínios das variáveis. Uma restrição é uma fórmula lógica atômica que representa uma informação sobre seus argumentos representando uma relação flexível entre as variáveis;
3. a utilização de uma operação mais geral chamada **satisfação de restrição** (*constraint satisfaction*) para a manipulação da instanciação de variáveis, ao invés do mecanismo de unificação utilizado em Prolog.

Para ilustrar o poder de expressão destas linguagens, serão apresentados alguns exemplos na seção a seguir. Na seção 2.2.3 são apresentadas as abordagens para resolução de restrições, nas quais as linguagens CLP podem ser classificadas. Na seção 2.2.4, um domínio específico é analisado.

2.2.1 Comparação entre Prolog e CLP

Normalmente, a sintaxe das linguagens CLP é similar a de Prolog. A **tabela 2.1**, apresentada em [VAR 97a], indica as principais diferenças entre a linguagem Prolog e as linguagem de programação em lógica com restrições de modo geral. Note-se que as linguagens CLP, assim como Prolog, também permitem a interferência no controle da execução do algoritmo. Porém, o predicado *cut* pode ser substituído por restrições.

TABELA 2.1 - Principais diferenças entre Prolog e CLP

Prolog	**Linguagens CLP**
• unificação	• satisfação de restrição
• operações aritméticas possuem o formato *A is B op C*. *op* representa uma operação aritmética e B e C devem ser variáveis totalmente instanciadas.	• operações aritméticas são tratadas como restrições não havendo limitações quanto à instanciação das variáveis; equações recebem tratamento algébrico.
• uso intensivo de retrocesso (*backtracking*) para obtenção da solução final.	• retrocesso é minimizado pelo mecanismo de poda antecipada (*a priori pruning*)
• uma variável é instanciada com um único valor, que não pode ser alterado.	• uma variável pertence a um domínio e seu valor pode ser obtido por reduções sucessivas no intervalo de valores possíveis. O resultado final pode ser um valor, um intervalo de valores ou uma restrição envolvendo outras variáveis.

Um aspecto importante que diferencia estes paradigmas é a árvore de execução criada. Isso se reflete nas implementações baseadas na WAM, na criação dos pontos de escolha. A técnica de poda antecipada reduz o número de pontos de escolha (nodos OU) em CLP, mas existem aplicações cujo número de pontos de escolha ainda é muito grande.

Para ilustrar essa característica, executou-se alguns programas nas linguagens clp(FD) e Prolog. Para Prolog, utilizou-se a implementação wamcc [COD 95b]. Tanto clp(FD) quanto wamcc permitem a exibição de uma série de estatísticas de execução, entre as quais o número de pontos de escolha criados (resultado de uma operação *try*), atualizados (resultado de uma operação *retry*) e removidos (resultado de uma operação *trust*). Utilizou-se, para os testes, alguns exemplos fornecidos juntamente com as ferramentas, cujo código encontra-se em anexo. A tabela 2.2 apresenta alguns desses resultados.

O clássico problema de colocação de *n* rainhas em um tabuleiro de xadrez *n* x *n* [STE 94] possui várias propostas de implementação. Ele pode ser resolvido com ou sem o uso de restrições. Utilizou-se como exemplo a colocação de 16 rainhas (`queens 16`). Sem utilização de restrições, nota-se que o número de pontos de escolha criados, atualizados e removidos é equivalente em ambas as linguagens. Já a execução com uso de restrições causa uma redução de aproximadamente 60.000 pontos de escolha criados (98,18%), mas aumenta o número de pontos de escolha com mais de duas alternativas. Um ponto de escolha com várias alternativas significa, potencialmente, um ponto de escolha com maior granulosidade.

Outro exemplo é a descoberta de valores numéricos para letras de forma a satisfazer a equação SEND + MORE = MONEY [COD 95a]. O comportamento para o programa sem restrições novamente não é muito distinto entre as duas linguagens. Salienta-se no entanto que, tanto no programa `queens` quanto no `send`, o tempo de execução na linguagem clp(FD), com o uso de restrições, foi menor. Esse exemplo envolve uma série de testes aritméticos, podendo ser expresso de forma mais natural com restrições. Neste caso, foi drástica a redução no número de pontos de escolha manipulados e no tempo de execução (a execução foi aproximadamente cem vezes mais rápida). Neste exemplo, a introdução de restrições melhorou significativamente o desempenho da aplicação.

Como último exemplo, cita-se `digit 8`, que não consta na tabela por ter sido modelado com restrições e, por isso, a sua execução foi unicamente em clp(FD). Ele apresenta a manipulação, isto é, atualização, de um número expressivo de pontos de escolha, mesmo sendo uma aplicação didática. Foram criados 26 pontos de escolha, atualizados 32.484.374 e removidos 19.

TABELA 2.2 - Pontos de Escolha em wamcc e clp(FD).

	queens 16			send		
	pontos de escolha			pontos de escolha		
	criados	atualizados	removidos	criados	atualizados	removidos
wamcc	63.002	0	62.982	12.071	18.561	7.567
clp(FD) (sem restrições)	63.010	0	62.985	12.079	18.561	7.570
clp(FD) (com restrições)	1.165	5.864	1.147	25	1	17

2.2.2 Exemplos de Aplicações em CLP

Como primeiro exemplo de aplicação de restrições em algoritmos recursivos, considere o clássico cálculo da série de Fibonacci, uma função estatística usada em muitas aplicações. O n-ésimo número de Fibonacci para n maior que 1 é definido como a soma dos números para n-1 e n-2. Os dois primeiros números (0 e 1) são definidos como 1. A **FIGURA 2.4** ilustra a seqüência dos oito primeiros números da série.

1 1 2 3 5 8 13 21 ...

FIGURA 2.4 - Números da série de Fibonacci.

A série de Fibonacci pode ser então definida, conforme a figura 2.5:

$$fibonacci(n)\begin{cases} n=0 & : \quad 1 \\ n=1 & : \quad 1 \\ n>1 & : \quad fibonacci(n-1)+fibonacci(n-2) \end{cases}$$

FIGURA 2.5 - Definição da série de Fibonacci.

Dessa definição, a transposição para Prolog é apenas uma questão sintática, conforme observa-se na figura 2.6

```
fibonacci(0,1).
fibonacci(1,1).
fibonacci(N,R) :-
     N1 is N-1,
     fibonacci(N1, R1),
     N2 is N-2,
     fibonacci(N2, R2),
     R is R1+R2.
```

FIGURA 2.6 - A série de Fibonacci codificada em Prolog.

O predicado *is* é utilizado para efetuar cálculos aritméticos. Ele exige que no seu lado esquerdo exista uma variável livre (aberta) e que no lado direito todas as variáveis já estejam instanciadas (fechadas). Desta forma, o uso do predicado *is* torna o predicado *fibonacci* não inversível: a consulta `fibonacci(5, X)` produz X=8 como resultado, mas a consulta fibonacci(X, 8) falha, pois a variável N não estará instanciada

Uma versão com restrições, codificada na linguagem CLP(R), é apresentada em [COH 90] e faz parte do conjunto de exemplos que são distribuídos com a linguagem. Esse exemplo ilustra a capacidade de inversão de CLP, além da possibilidade de um código mais sucinto:

```
fibonacci(0,1).
fibonacci(1,1).
fibonacci(N,R1+R2) :-
     N > 1,
     fibonacci(N-1, R1),
     fibonacci(N-2, R2).
```

FIGURA 2.7 - A série de Fibonacci codificada em CLP(R).

CLP aceita equações como argumentos de predicados. Essas equações são chamadas de restrições (*constraints*) e são utilizadas também dentro do programa para controlar o valor de variáveis. Essas equações delimitam o escopo de variáveis dentro de um determinado domínio. O resultado final de uma variável pode ser uma equação ou um intervalo e não apenas um valor único como em Prolog.

O exemplo da figura 2.7 apresenta quatro restrições: R1+R2, N-1 e N-2 que são utilizadas como argumentos e N > 1 que é utilizada como controle. Note que em Prolog também é possível incluir a comparação N > 1, mas a forma de tratamento é bastante diferente. Em Prolog, essa condição é testada e tratada como um objetivo normal e, caso N não estiver instanciado, toda a cláusula irá falhar. Já em CLP, caso a variável estiver livre, essa condição irá delimitar o escopo da variável, que deve pertencer ao domínio dos inteiros. N passará a permitir valores de 2 até $+\infty$.

É importante notar que com restrições o predicado *fibonacci* é inversível. Conforme descrito acima, a restrição N > 1 não exige que a variável esteja instanciada, pois não é mais utilizado o predicado *is*. As operações aritméticas necessárias neste exemplo são efetuadas na passagem de argumentos. Se a variável estiver instanciada, o argumento é avaliado e a subtração realizada. Se ela estiver livre, a avaliação da restrição é postergada. Desta forma, tanto a consulta *fibonacci*(5, X) quanto *fibonacci*(X, 8) podem ser respondidas com sucesso.

A figura 2.8 apresenta algumas consultas possíveis para o programa *fibonacci* no sistema CLP(R), ilustrando a flexibilidade obtida, uma vez que restrições também podem ser usadas nas consultas.

```
11 kayser@itaimbe% clpr EXAMPLES/fib
CLP(R) Version 1.2
(c) Copyright International Business Machines Corporation
1989 (1991, 1992) All Rights Reserved
>>> Sample goal: go/0
1 ?- fib(5,X).
X = 8
*** Retry?
2 ?- fib(8,X).
X = 34
*** Retry?
3 ?- 5 <= N, N <= 8, fib(N,R).
R = 8
N = 5
*** Retry? y
R = 13
N = 6
*** Retry? y
R = 21
N = 7
*** Retry? y
R = 34
N = 8
*** Retry? y
*** No
4 ?- halt.
```

FIGURA 2.8 - Consultas ao programa Fibonacci em CLP(R).

O segundo exemplo a ser apresentado é o escalonamento de horários em uma escola. Cabe salientar que o exemplo da série de Fibonacci ilustra o poder de manipulação algébrica das linguagens em lógica com restrições. A outra característica importante de CLP, que é a poda antecipada *(a priori pruning)* da árvore de busca, é melhor ilustrada nesse segundo exemplo.

Timetabling ou escalonamento de horários escolares pode ser descrito como uma seqüência de encontros entre professores e estudantes em um período pré-fixado de tempo (tipicamente uma semana), satisfazendo um conjunto de restrições de vários tipos. Esse é um tipo de problema que freqüentemente contém uma variedade de restrições complexas e que necessitam de estratégias de busca especiais [HEM 96].

Esse problema tradicionalmente tem sido resolvido com técnicas de pesquisa operacional (PO). Atualmente, existem vários estudos que tentam resolver este problema utilizando técnicas de inteligência artificial, tais como algoritmos genéticos e redes neurais.

O uso de CLP para resolver essa classe de problemas é adequado devido à sua característica de modelagem declarativa e à sua execução que proporciona automaticamente uma propagação de alterações em variáveis e um mecanismo eficiente para resoluções combinatoriais. Assim, além da obtenção de um desempenho similar ao de implementações com técnicas consolidadas de PO, a sua característica declarativa facilita a rápida modelagem do problema.

Dentre os vários estudos utilizando CLP para resolver escalonamento de horários cita-se [HEM 96] [MON 96]. Ambos buscam a solução utilizando restrições. Com o uso de restrições, a poda de ramos que não levam a um resultado satisfatório torna a execução mais eficiente.

2.2.3 Técnicas de Resolução de Restrições

Prolog, e a programação em lógica em geral, apresentam duas limitações [FRU 93]. Em primeiro lugar, os objetos manipulados são estruturas não interpretadas e, por isso, a igualdade ocorre apenas entre aqueles objetos que são sintaticamente idênticos. O segundo problema está relacionado à sua execução que, resultando em um procedimento de geração e teste, apresenta problemas de desempenho especialmente para classes de problemas combinatoriais. CLP trata estas duas limitações pela inclusão de restrições e de mecanismos de resolução de restrições. Este tratamento é feito pelas três linhas de resolução de restrições que serão apresentadas: esquema CLP(X), técnica de consistência e técnica de propagação generalizada.

No final da década de 80 começaram a surgir as primeiras implementações de linguagens como CLP(R) [JAF 92], Prolog III [COL 90] e CHIP [DIN 90] [HEN 89]. Estas linguagens objetivam preservar as vantagens da programação em lógica enquanto removem limitações. Esta linha é denominada resolução de restrições ou **esquema CLP(X)** [JAF 94] [FRU 93]. Assim, existe toda uma família de linguagens que partilham a mesma semântica, mas possuem instanciações diferentes de domínios:

- CLP(R): reais;
- clp(Q, R) [HOL 95]: reais e racionais;
- Prolog III: booleanos, números racionais e listas;
- CHIP: domínios finitos, booleanos e números racionais.

O ponto chave deste esquema é permitir que os objetos manipulados sejam projetados como pertencentes a um domínio computacional, como por exemplo, inteiros, reais, conjuntos ou expressões booleanas. Neste esquema, as restrições são relações sobre os domínios, e a computação pode ser feita diretamente sobre estes domínios. A unificação é substituída pela manipulação de restrições, que passa ser a operação central, cujo objetivo é decidir quando uma equação tem solução e, caso exista, retornar a solução mais geral.

A segunda linha é denominada **técnicas de consistência** (*consistency techniques*) [FRU 93] [HEN 89]. As técnicas de consistência foram inicialmente introduzidas em programas de reconhecimento de figuras, que precisavam rotular linhas de forma consistente, e em problemas de satisfação de restrição (*constraint satisfaction problem* ou *CSP*) de modo geral. O número de possíveis rótulos pode ser muito grande enquanto apenas poucos são consistentes. As técnicas de consistência excluem rótulos inconsistentes nos estágios iniciais do processamento e, deste modo, diminuem a busca por rótulos consistentes. Tradicionalmente, estas técnicas trabalham sobre domínios finitos, mas existem estudos mais recentes sobre uso de intervalos finitos.

As linguagens CLP desta abordagem trabalham por propagação de informações sobre variáveis via restrições. A propagação continua até que reduções a novos domínios não possam mais ser extraídas das restrições. Para remover valores inconsistentes entre duas

variáveis, devido a uma restrição, pode-se utilizar uma série de algoritmos de consistência, mas a técnica mais utilizada é a *arc consistency* [FRU 93].

Nesta linha de resolução de restrições, os domínios inconsistentes são podados por propagação, mas a manipulação de restrições não garante a detecção de inconsistências globais. Ou seja, não é garantido que inconsistências existentes nas restrições armazenadas sejam detectadas até que as variáveis obtenham seus valores finais. Por outro lado, as técnicas de consistência provêm um modo eficiente de extrair novas informações das variáveis nas restrições armazenadas.

Estas linguagens possuem uma maior transparência na descrição dos algoritmos utilizados além de possuírem uma boa eficiência. São exemplos, as linguagens clp(FD) [COD 96] e cc(FD) [HEN 93]. Alguns autores indicam a linguagem CHIP como pioneira nesta linha [FRU 93] [HEN 89], pois foi a primeira a introduzir domínios finitos, enquanto outros a classificam como pertencente ao esquema CLP(X) [HEN 92] [JAF 94].

A terceira técnica de resolução de restrições procura remover a exigência de domínios finitos das variáveis. O primeiro passo foi permitir trabalhar com intervalos de valores de números reais. Entretanto, é possível realizar propagação sem exigir domínios ou intervalos associados às variáveis. Esta técnica de consistência é chamada de **propagação generalizada** (*generalized propagation*) [PRO 93] e tenta integrar o esquema CLP(X) e as técnicas de satisfação de restrições.

A idéia principal na propagação generalizada é utilizar predicados como restrições, mesmo que não existam domínios ou intervalos associados aos seus argumentos. No esquema CLP(X) a resposta de um objetivo é um conjunto consistente de restrições sobre as variáveis do problema. Na propagação, informações são extraídas de modo a obter a generalização mais específica (*most specific generalization*) de todas as respostas para o objetivo.

Assim como a propagação das técnicas de consistência, a propagação generalizada é uma forma de computação orientada a restrições. A medida que mais informações sobre as variáveis do problema se tornam disponíveis, através do armazenamento de novas restrições, informações adicionais são extraídas. Todas as informações extraídas são adicionadas ao conjunto de restrições, o que irá permitir que novas propagações ocorram. A propagação é repetida até que não seja mais possível extrair novas informações.

A linguagem Propria [PRO 93] representa esse tipo de abordagem. Esta linguagem tem sua execução baseada nos princípios de concorrência. Existem agentes de propagação que verificam se as suas variáveis foram modificadas para decidir se devem ou não acionar a propagação. Essa mesma noção de concorrência também existe em cc(FD).

É importante salientar que, nos três casos, é essencial que o resolvedor de restrições seja incremental. Por incremental considera-se que quando for adicionada uma nova restrição C para um conjunto de restrições S já resolvido, o resolvedor de restrições não deve começar a solucionar o novo conjunto $S \cup \{C\}$ do início.

Esta pesquisa estará centrada nas linguagens que utilizam técnicas de consistência, principalmente pelo fato dessas implementações utilizarem extensões da WAM.

2.2.4 CLP sobre Domínios Finitos

Como já foi abordado na seção anterior, vários domínios matemáticos já foram implementados em linguagens CLP. Por exemplo, CLP(R) [JAF 92] implementa o domínio dos números reais, enquanto clp(Q, R) [HOL 95] implementa além dos reais o domínio dos racionais. CHIP [DIN 90] trata domínios booleanos e racionais, e foi a primeira linguagem CLP a implementar domínios finitos.

Domínio finito, tal como foi introduzido pela linguagem CHIP, é um conjunto finito (não vazio) de números naturais, isto é, um intervalo [COD 96]. Note-se que um intervalo pode ser formado por um ou mais sub-intervalos.

Os domínios finitos são muito utilizados para resolver numerosas aplicações industriais como, por exemplo, problemas combinatoriais, ordenação, otimização, simulação de circuitos, diagnóstico, ajuda à decisão, ou mesmo, problemas booleanos.

As linguagens cc(FD) [HEN 93] [SAR 93] e clp(FD) [COD 96] [DIA 94a] [DIA 94b] implementam domínios finitos através de uma extensão da WAM. Internamente, a execução manipula uma única restrição *X in r* (onde *X* é uma variável e *r* é um intervalo) que permite expressar um domínio finito. As restrições complexas são traduzidas na compilação em um conjunto de restrições X in r.

Essa restrição X in r é a essência do mecanismo de propagação de restrições. Quando uma restrição utilizar um índice sobre uma variável Y ela se torna *store sensitive* e deve ser verificada cada vez que o domínio de Y for modificado. O *store* é um conjunto finito de restrições manipulado pelo programa, que deve possuir duas características: (a) estar em forma normal: se e somente se contiver exatamente uma restrição X in r para cada variável FD; (b) ser consistente: se e somente se ele não contiver nenhum domínio vazio.

Vários autores justificam a eficiência que essas linguagens possuem através da analogia com processadores RISC: tem-se um conjunto pequeno mas bastante eficiente de instruções.

2.3 Conclusão

A programação em lógica é uma ferramenta muito importante para o desenvolvimento de aplicações simbólicas. No entanto, linguagens como Prolog não conseguem tratar em intervalo de tempo viável aplicações com características combinatoriais.

CLP é uma extensão da programação em lógica em que variáveis podem pertencer a domínios. Linguagens deste paradigma conseguem lidar de forma mais eficiente com problemas combinatoriais devido ao seu mecanismo de poda antecipada.

CLP possui um grande potencial de aplicações possíveis e de estudos para o aprimoramento de seu desempenho.

A execução paralela é uma alternativa importante para aumentar o desempenho tanto de linguagens em lógica quanto de linguagens CLP. Diversos aspectos dessa alternativa são abordados no próximo capítulo.

3 Exploração de Paralelismo em CLP

Neste capítulo analisa-se algumas abordagens utilizadas na exploração de paralelismo na programação em lógica e, especialmente, em CLP. O objetivo é apresentar as formas de exploração de paralelismo implícito e as diferentes técnicas propostas para o controle da execução paralela.

3.1 Motivação

Apesar de CLP ser mais eficiente que a programação em lógica tradicional, é necessário buscar caminhos que melhorem ainda mais o seu desempenho e permitam a resolução de um maior número de problemas reais. A execução de programas em paralelo é uma das alternativas para obter eficiência. Intuitivamente, ao dividir um programa ao meio e executar cada uma dessas partes em um processador diferente, espera-se obter o resultado na metade do tempo (*speedup*). Porém em geral, isso não ocorre na prática devido aos custos adicionais (*overhead*) envolvidos no controle do paralelismo. Outro problema se refere à dificuldade de programação paralela, isto é, a maioria das linguagens obriga o programador a explicitar a criação e a interação de processos, o que não é uma tarefa trivial.

Deste modo, a exploração de paralelismo em CLP deve ter os seguintes objetivos:

- minimizar o *overhead:* buscar a máxima eficiência e somente executar uma tarefa em paralelo quando o seu custo de paralelização for inferior ao custo da sua execução seqüencial.
- explorar o paralelismo implícito: liberar o programador do controle da execução, permitindo reaproveitamento de código e facilitando a manutenção.

3.2 Fontes de Paralelismo na Programação em Lógica

Os programas em lógica com restrições, assim como os programas em lógica, são potencialmente paralelizáveis. As duas principais fontes de paralelismo implícito [KER 92] são:

- **Paralelismo OU**: consiste na execução paralela das cláusulas que compõem um predicado. As várias resolventes que poderiam ser executadas sucessivamente por retrocesso na execução seqüencial, sendo executadas em paralelo, recebem o nome de paralelismo OU pois as várias cláusulas estão conectadas por um operador lógico OU;
- **Paralelismo E**: consiste na execução paralela dos diferentes objetivos que compõem uma cláusula e, portanto, seu nome se deve ao fato dos objetivos estarem relacionados pelo operador lógico E. Esse tipo de paralelismo pode ser explorado de duas formas: (a) E independente: apenas objetivos que não compartilham variáveis podem ser executados em paralelo; (b) E dependente: objetivos que compartilham variáveis podem ser executados em paralelo, gerando uma relação produtor-consumidor.

As primeiras implementações paralelas de CLP, assim como as primeiras implementações paralelas de Prolog, se preocupam com a exploração de paralelismo OU. Alguns dos motivos que justificam essa escolha são:

- Simplicidade: não há necessidade de incluir nenhuma anotação no programa;
- Similaridade: o modelo de execução se mantém próximo ao da versão seqüencial, facilitando o reaproveitamento de técnicas seqüenciais eficientes, o que também facilita a manutenção da semântica original;
- Granulosidade: como normalmente a granulosidade (*granularity*) é alta, isso facilita uma exploração eficiente de paralelismo.
- Aplicações: apesar de não estar presente em todos os programas, como o paralelismo E, existem várias aplicações que se beneficiam desta técnica, como por exemplo, consulta a grandes bases de dados.

As implementações mais eficientes de paralelismo OU utilizam um modelo denominado multi-seqüencial que se caracteriza por ser composto de trabalhadores que possuem máquinas de inferência completas. Isso permite que cada trabalhador resolva uma parte da árvore de busca. Esse modelo também permite a incorporação de técnicas eficientes de execução seqüencial.

Em princípio a implementação de paralelismo OU seria simples, uma vez que os vários ramos da árvore de busca são independentes dos outros e, portanto, exigiriam pouca comunicação entre os trabalhadores [KER 92] [ROC 97]. Entretanto, na prática, a implementação de paralelismo OU é difícil devido ao compartilhamento de pontos de escolha [GUP 93]. Isto é, dado dois nodos em diferentes ramos da árvore, eles podem possuir um ancestral comum. Assim, o problema essencial é gerenciar (nos diferentes trabalhadores) os múltiplos ambientes, isto é, as pilhas, de modo que valores de variáveis aplicáveis somente a um ramo não sejam visíveis nos outros ramos. Tal problema é típico de ambientes de memória compartilhada, existindo várias propostas para essa gerência. Em ambientes de memória distribuída, algumas destas soluções não são adequadas devido ao custo de acesso a memórias não locais.

Para a implementação do paralelismo OU multi-seqüencial, existem diversas técnicas que visam disponibilizar as informações das pilhas de controle de execução da máquina abstrata. Essas técnicas de disponibilização de contexto são acionadas, principalmente, durante a exportação de tarefas. Note que uma **tarefa** é formada por alternativas a serem avaliadas, isto é, um ponto de escolha e um contexto de execução. Essas técnicas, são discutidas nas próximas seções, e podem ser classificadas em três tipos [KER 92] [GUP 93]: compartilhamento, cópia e recomputação.

3.2.1 Compartilhamento de Pilhas

No compartilhamento de pilhas [LUS 90], um trabalhador que possua tarefas compartilha as suas estruturas de dados (globais) com um outro trabalhador que precisa obter trabalho. Nessa técnica, parte das pilhas são compartilhadas e outra parte é privada.

A parte compartilhada é a porção da árvore de busca que é comum, isto é todo o contexto anterior ao da tarefa exportada. A parte privada, isto é, a parte não comum, é utilizada no armazenamento das diversas ligações condicionais às variáveis que foram criadas na parte compartilhada.

A **figura 3.1** ilustra a exportação de tarefa entre dois trabalhadores. Neste caso, é exportada apenas a alternativa no ponto de escolha compartilhado pela memória global.

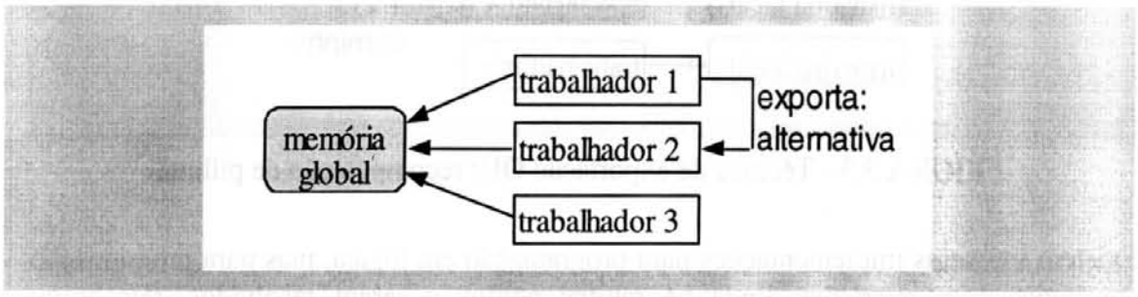

FIGURA 3.1 - Técnica de exportação OU: compartilhamento de pilhas

3.2.2 Cópia de Pilhas

Para a utilização de cópia de pilhas [ALI 90] [PRE 93] [KON 94] [MOR 97] [ROC 97], cada trabalhador possui pilhas privadas completas. Quando uma tarefa é exportada, todas as estruturas que fazem parte do contexto de execução são copiadas para o trabalhador importador. Deve haver um controle no exportador para indicar que uma alternativa, ou várias, foram exportadas. Alguns modelos [ALI 90] optam por tornar um ponto de escolha público, isto é, compartilhado, quando ocorre exportação de uma tarefa deste ponto de escolha.

A figura 3.2 mostra que a exportação agora implica o envio da alternativa e das pilhas de execução. As pilhas são armazenadas na memória local de cada trabalhador.

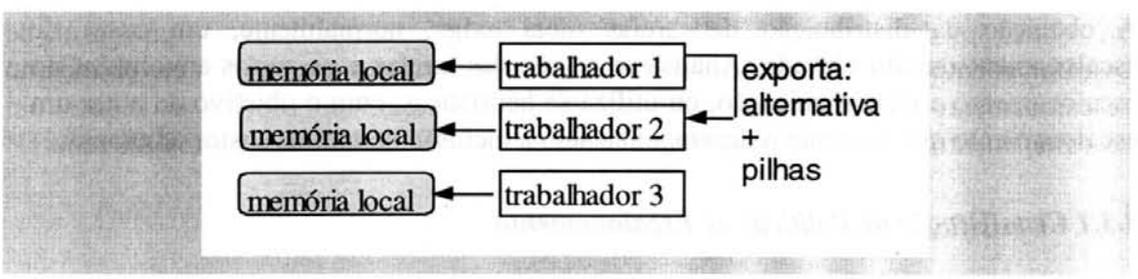

FIGURA 3.2 - Técnica de exportação OU: cópia de pilhas

3.2.3 Recomputação de Pilhas

Na recomputação de pilhas [CLO 87] [MUD 94], cada trabalhador possui pilhas privadas completas. Quando uma tarefa é exportada, o importador recebe uma seqüência de alternativas a serem tomadas na árvore de busca. O importador reexecuta o programa seguindo essa seqüência que permite a construção do contexto de execução.

A figura 3.3 permite que se perceba a similaridade com a técnica de cópia de pilhas. Neste caso, as pilhas também são armazenadas na memória local de cada trabalhador, mas a exportação consiste no envio da alternativa e do caminho a ser reexecutado.

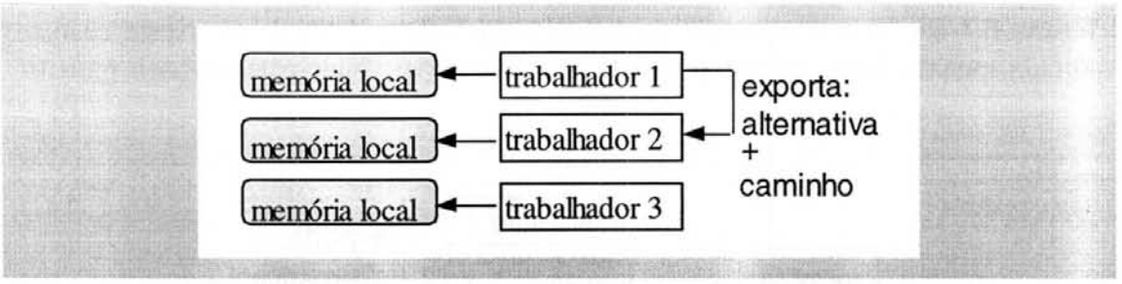

FIGURA 3.3 - Técnica de exportação OU: recomputação de pilhas

Existem inúmeras implementações para programação em lógica, mas para programação em lógica com restrições ainda há muitos pontos a serem resolvidos, tais como escalonamento de tarefas e exploração em ambiente distribuído. Na seção 3.4, serão apresentados alguns trabalhos de exploração de paralelismo.

Um ponto crucial na exploração de paralelismo de modo geral é o escalonamento de tarefas, tal como será apresentado na próxima seção.

3.3 Políticas de Escalonamento de Tarefas

O objetivo do escalonamento de tarefas (*task scheduling*) [ELR 94] é determinar a atribuição das tarefas para elementos de processamento e a ordem em que cada tarefa será executada, de modo que algumas medidas de desempenho sejam otimizadas.

O escalonamento é um problema NP-completo, tanto na sua forma geral quanto em casos restritos. Por isso, normalmente, utilizam-se heurísticas, que reduzem a complexidade do problema, para melhorar o desempenho do algoritmo de escalonamento.

A obtenção da distribuição de tarefas ideal exige, normalmente, um custo de escalonamento muito elevado. Analisa-se o grão das tarefas e os custos envolvidos no escalonamento e na comunicação, ou utiliza-se heurísticas, com o objetivo de obter um escalonamento que aumente o desempenho sem a inclusão de muitos custos adicionais.

3.3.1 Classificação de Políticas de Escalonamento

Existem várias classificações para os algoritmos de escalonamento. Por exemplo, [CAS 88] introduz uma taxonomia para classificar as diferentes políticas de escalonamento. Apresentam-se aqui apenas duas classificações entre as mais utilizadas.

As políticas de escalonamento podem ser classificadas quanto ao número de escalonadores em centralizadas, distribuídas e hierárquicas. Na centralizada, existe um elemento central encarregado das decisões de escalonamento. No distribuído tem-se vários escalonadores que trocam informações a fim de decidir a melhor forma de escalonar as tarefas. As políticas centralizadas possuem um bom desempenho, mas não são escaláveis, enquanto as distribuídas precisam trocar um número maior de mensagens, mas permitem a gerência de um número maior de nodos. As abordagens hierárquicas tentam aproveitar as vantagens de ambas as técnicas, através da criação de grupos (*clusters*) de nodos subordinados a um escalonador que está conectado a outros escalonadores. Existe um escalonamento centralizado dentro dos grupos, buscando

manter a localidade de dados, e um escalonamento distribuído entre os diversos escalonadores.

Os algoritmos de escalonamento também podem ser classificados em relação às informações utilizadas para o escalonamento como estáticos, dinâmicos ou adaptativos. Um algoritmo estático utiliza informações disponíveis sobre o programa antes da sua execução. O dinâmico usa informações sobre o estado do sistema, como por exemplo, a carga em cada nodo ou a fila de espera da CPU em um determinado processador. Os dinâmicos obtêm, potencialmente, resultados mais precisos que os estáticos, mas há um custo adicional na coleta, armazenamento e análise das informações do estado do sistema. O adaptativo caracteriza-se pela utilização de informações sobre o estado do sistema. Ele tem também a capacidade de perceber quando o sistema como um todo está sobrecarregado, situação na qual ele para de coletar informações para evitar um maior custo de execução. Outra classificação possível é o semi-dinâmico, onde informações são obtidas em tempo de compilação, mas a decisão de escalonamento é tomada durante a execução com a coleta de dados adicionais.

3.3.2 Componentes de um Algoritmo de Escalonamento

Um algoritmo para distribuição de carga apresenta quatro componentes [SIN 94]:

1. **Política de Transferência:** determina **quando** um nodo está em um estado adequado para participar de uma transferência de tarefa. A maioria das políticas de transferência é baseada em faixas de valores das unidades de carga (*thresholds*) que indicam se um nodo pode enviar ou receber carga. Uma alternativa é iniciar a transferência quando a política de informação (item 4) detectar que o sistema não está balanceado;
2. **Política de Seleção:** determina **qual tarefa** deve ser transferida. No contexto de sistemas operacionais, a abordagem mais simples é selecionar a tarefa mais recente gerada pelo sistema, porque tal tarefa possui a transferência mais barata por não ser preemptiva. De fato, o mais importante é considerar o custo adicional para envio desta tarefa. O objetivo é a redução no tempo de resposta desta tarefa;
3. **Política de Localização:** determina para **qual nodo** a tarefa a ser transferida deve ser enviada. Um método bastante utilizado é a consulta aos nodos (*polling*) para verificar se eles podem compartilhar tarefas. Essa comunicação pode ser feita de forma aleatória, baseada em informações coletadas pelo sistema, ou pela proximidade dos nodos;
4. **Política de Informação:** é responsável por disparar a coleta de informações sobre o sistema. As políticas de informação podem ser classificadas em três tipos: **(a) por demanda**: um nodo coleta informações quando se torna importador ou exportador; **(b) periódica**: nodos trocam informações periodicamente; **(c) por mudança de estado**: nodos disseminam informações quando seu estado sofre mudança de um certo nível.

A política de localização permite a classificação dos algoritmos de escalonamento como [DAN 97] [KRU 94]: iniciadas pelo importador (*receiver initiated*), iniciadas pelo exportador (*sender initiated*) ou simetricamente iniciadas (*symmetrically initiated*). O desempenho relativo dessas classes de algoritmos tem se mostrado dependente da carga (*workload*) do sistema. A política iniciada pelo exportador possui melhor desempenho que o iniciado pelo importador em sistemas com cargas baixas a moderadas. Isso ocorre

porque a probabilidade de encontrar um nodo inativo (ocioso) é alto. Em sistemas com cargas altas, os iniciados pelo importador são melhores porque é muito mais fácil encontrar um nodo sobrecarregado.

3.4 Trabalhos Relacionados

O tema principal deste trabalho é a exploração de paralelismo OU multi-seqüencial na programação em lógica com restrições, portanto, apenas as pesquisas sobre essa fonte de paralelismo serão analisadas.

Em Prolog, vários sistemas para exploração de paralelismo têm sido propostos e implementados. Para paralelismo OU, os mais conhecidos são Aurora [LUS 90] implementado sobre multiprocessadores com memória compartilhada utilizando a técnica de compartilhamento de pilhas, e MUSE [ALI 90] implementado com memória local e global utilizando a técnica de cópia de pilhas. Ambos obtiveram ganhos de desempenho significativos. O DelPhi [CLO 87] é outro sistema paralelo para Prolog cuja principal contribuição foi a proposta da técnica de recomputação de pilhas.

Vários outros trabalhos também foram propostos entre o final da década de 80 e início da década de 90, sendo ainda um assunto bastante pesquisado. Por exemplo, o sistema YapOr [ROC 97], baseado no MUSE, foi apresentado à comunidade científica no final de 1997.

No contexto de máquinas puramente distribuídas existem poucos trabalhos, como por exemplo [BEN 93]. Outro exemplo significativo é o PLoSys [MOR 97] que é uma evolução do modelo OPERA [GEY 92]. O PLoSys caracteriza-se pela exploração implícita de paralelismo OU multi-seqüencial em um ambiente de memória distribuída, utilizando uma política de escalonamento centralizada.

Em relação a CLP, existem diversos trabalhos recentes sobre exploração do paralelismo OU. A maior parte utiliza derivações da linguagem CHIP. [RÉT 96] busca formas de execução distribuída das linguagens concorrentes com restrições, derivadas do *framework cc* proposto por [SAR 93]. Linguagens concorrentes com restrições permitem a descrição de uma série de processos que interagem entre si e cuja comunicação e sincronização é feita pela inclusão e teste de restrições. [YAN 95] implementou uma linguagem com restrições paralela utilizando a linguagem concorrente KL1. [TON 95] executa a linguagem Firebird (extensão de CHIP) em computadores massivamente paralelos, preocupado-se com escalabilidade.

Exemplos diretamente relacionados com os objetivos do trabalho em questão, isto é, que utilizam linguagens em lógica com restrições em ambientes distribuídos, são PARCS [KON 94], ElipSys [PRE 93] e o trabalho desenvolvido por Mudambi [MUD 94]. Todos paralelizam linguagens CLP sobre domínios finitos.

PARCS objetiva execução em ambiente de memória distribuída e com processadores massivamente paralelos, sem a preocupação com manutenção da semântica de Prolog. Esse sistema utiliza análise estática para gerar informações úteis para a exploração de paralelismo. Para a exportação de uma tarefa é utilizada a cópia de pilhas, sem nenhum tipo de compartilhamento, que pode ser concorrente à computação.

O sistema ElipSys implementa o paralelismo em uma linguagem de programação em lógica com restrições. De fato, é possível explorar paralelismo e/ou restrições em um programa. É possível explorar tanto paralelismo OU quanto E, mas fica a cargo do

programador quando deve ser feita a paralelização de ambas as fontes de paralelismo. A técnica de exportação de contexto adotada é a cópia de pilhas.

Mudambi explora apenas paralelismo OU, mas de forma automática. Além disso, o alvo do protótipo é uma rede heterogênea de computadores. Para a exportação do contexto, utiliza a técnica de recomputação de pilhas.

ElipSys e Mudambi utilizam algoritmos de escalonamento centralizados. PARCS efetua balanceamento de carga através de dois métodos. O primeiro consiste em um processador central que envia periodicamente mensagens coletando informações de estado de todos os processadores. A cada uma destas mensagens, o escalonador informa o estado global do sistema na última coleta, o que permite que os trabalhadores decidam como regular a carga. O segundo algoritmo caracteriza-se pela iniciativa dos trabalhadores sem tarefas (em estado ocioso) de enviarem uma mensagem pedindo tarefas para um trabalhador escolhido aleatoriamente.

A **tabela 3.1** apresenta a comparação entre os trabalhos de exploração de paralelismo mais significativos.

TABELA 3.1 - Comparação entre sistemas em lógica paralelos.

Sistemas / Critérios	Programação em Lógica			Programação em Lógica com Restrições		
	MUSE	PLoSys	Aurora	ElipSys	Mudambi	PARCS
Paralelismo	OU	OU	OU	OU/E	OU	OU
Técnica de Exportação OU						
cópia	√	√		√		√
compartilhamento			√			
recomputação					√	
Ambiente de Execução						
memória compartilhada	√		√			
memória distribuída		√		√	√	√
Escalonamento						
estático						
dinâmico	√	√	√	√	√	√
centralizado		√		√	√	
distribuído com uso de estruturas centralizadas	√		√			
distribuído						√

Apesar dos vários trabalhos existentes, ainda não há um consenso de quais as técnicas mais adequadas para exploração do paralelismo na programação em lógica com restrições.

3.5 Conclusão

A execução paralela é uma alternativa importante para aumentar o desempenho de linguagens CLP. Esta alternativa mostra-se viável devido a dois aspectos principais.

Em primeiro lugar, a execução deste tipo de linguagem é bastante facilitada pela sua característica determinística e pela existência de várias fontes implícitas de paralelismo. Deste modo, a exploração automática de paralelismo é possível, o que é bastante cômodo para o usuário.

Outro ponto refere-se a popularização de arquiteturas paralelas, especialmente redes de computadores. Atualmente, é possível utilizar redes locais como máquinas paralelas pois a comunicação é eficiente e normalmente ocorre a subutilização de máquinas (processadores).

No próximo capítulo, um modelo de exploração implícita de paralelismo OU é apresentado. Como é possível explorar paralelismo em CLP de forma análoga à realizada em Prolog, muitas das decisões tomadas na concepção deste modelo levaram em conta trabalhos realizados em Prolog.

As implementações paralelas na Programação em Lógica e em CLP, em sua maioria, trabalham sobre memória compartilhada. Um dos objetivos do modelo é alcançar a execução eficiente sobre máquinas de memória distribuída.

4 Modelo de Exploração de Paralelismo em CLP

Neste capítulo é apresentado o modelo pclp(FD) [VAR 97b] [VAR 97c] de exploração de paralelismo OU. Antes da análise do modelo propriamente dito, são apresentadas algumas decisões que influenciaram a sua concepção.

4.1 Considerações Iniciais

Nas próximas seções apresentam-se algumas decisões utilizadas como premissas na concepção do modelo pclp(FD). Estes pontos justificam algumas opções de modelagem que serão analisadas posteriormente. Note-se que a questão do ambiente computacional foi decidida pelo contexto de projeto no qual este trabalho está inserido.

4.1.1 Modelo OU Multi-seqüencial

Optou-se por modelar o paralelismo OU, uma vez que vários programas CLP são combinatoriais e não determinísticos o que implica criação de vários pontos de escolha.

O modelo OU apresentado é multi-seqüencial pois existem vários trabalhadores, todos eles com capacidade de resolver de forma seqüencial e autônoma programas Prolog. Esse tipo de abordagem se mostrou a mais adequada para Prolog e tem sido adotada nas implementações de linguagens CLP.

A técnica de poda antecipada reduz o número de pontos de escolha (nodos OU), mas existem aplicações cujo número de pontos de escolha, tal como analisado na seção 2.2.1, justifica tal abordagem.

Por exemplo, o programa digit 8 executado em clp(FD), apresenta a atualização de um número expressivo de pontos de escolha. Tal aplicação possui um pequeno número de pontos de escolha com várias alternativas, o que é uma característica adequada para a exploração de paralelismo OU.

4.1.2 Ambiente de Execução

Memória distribuída é uma característica existente em redes de computadores que constituem-se hoje em uma das plataformas paralelas mais importantes. Por isso, considerou-se que o ambiente computacional, no qual o modelo será executado, não dispõe de memória comum entre os processadores. Isso implica a ausência de variáveis globais para controle de paralelismo.

Também considerou-se que os processadores devem se encontrar organizados fisicamente de tal modo que seja possível a comunicação entre quaisquer dois processadores e, conseqüentemente, entre quaisquer dois trabalhadores.

4.1.3 Linguagem CLP

Outro ponto importante refere-se à linguagem a ser paralelizada. É mais vantajoso o uso de linguagens CLP pois, normalmente, são mais eficientes que Prolog. Como o objetivo deste trabalho é a proposta de um modelo de exploração de paralelismo na programação em lógica, não existe a necessidade de propor uma nova linguagem nem de restringi-lo a uma linguagem específica.

Para que esse modelo possa ser implementado, a linguagem escolhida pode ser qualquer linguagem em lógica que possua a sua implementação baseada na WAM, ou em uma máquina abstrata baseada em pilhas similares à da WAM. Por isso, linguagens sobre domínios finitos são apropriadas pois podem ser implementadas como extensões da WAM.

4.2 Modelo Lógico

O modelo pclp(FD) pressupõe a existência de *n* nodos trabalhadores totalmente interconectados (como ilustrado na figura 4.2). Cada um possui um identificador único.

Todos os trabalhadores possuem capacidade de processamento suficiente para resolver totalmente um programa de forma seqüencial. Um trabalhador, denominado principal (*main*), possui duas tarefas adicionais: (a) iniciar o processamento e (b) detectar o término do programa.

Cada nodo trabalhador é formado por (figura 4.1):

1. uma **máquina abstrata**, que é uma extensão da WAM. Cada trabalhador possui suas próprias pilhas de execução, totalmente privadas, o que lhe permite acesso irrestrito a essas estruturas;
2. um **escalonador**, que é responsável pela exportação e importação de tarefas;
3. uma **interface** entre o escalonador e a máquina abstrata. A interface com a WAM fornece, basicamente, funcionalidades para obter informações sobre o estado do trabalhador e para permitir a importação e a exportação de tarefas, como será abordado posteriormente.

A aplicação que é solucionada pelos trabalhadores, é iniciada pelo trabalhador principal e é composta por várias tarefas que são criadas dinamicamente. A medida que essas tarefas são criadas, elas precisam ser distribuídas entre os trabalhadores para que o processamento possa ocorrer de forma paralela. Esse escalonamento de tarefas é realizado de forma dinâmica, porque as decisões ocorrem durante a execução, e de forma distribuída, pois existem vários escalonadores, um em cada trabalhador.

FIGURA 4.1 - Nodo trabalhador a nível lógico.

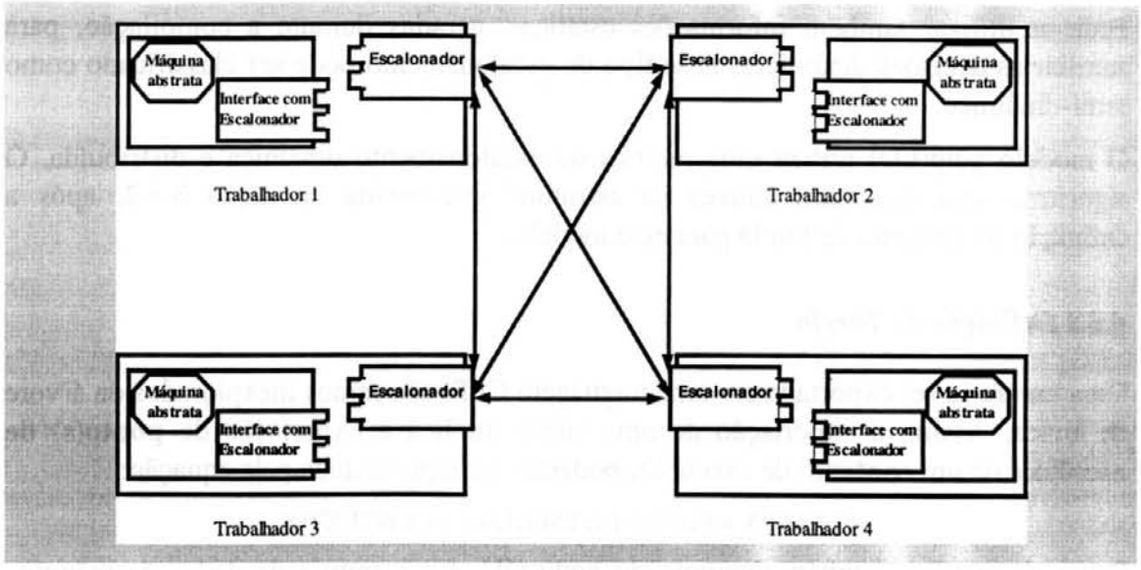

FIGURA 4.2 - Interconexão de quatro nodos trabalhadores.

A definição de uma interface entre o escalonador e a máquina abstrata fornece uma maior flexibilidade ao modelo, devido à separação das atividades que devem ser realizadas da forma como elas são implementadas. Isso permite a combinação de diferentes políticas de escalonamento com o uso de diferentes máquinas abstratas. O sistema Aurora [LUS 90] é um exemplo onde uma mesma máquina abstrata paralela foi utilizada com diferentes políticas de escalonamento.

Neste modelo, o escalonamento é distribuído, com cada nodo trabalhador possuindo um escalonador. Para a implementação de uma política centralizada, o escalonador poderia ser substituído por qualquer componente encarregado da troca de informações com o escalonador central, que mantivesse a interface escalonador/máquina proposta nesse modelo. Da mesma forma, a máquina abstrata poderia ser tanto uma implementação de WAM para Prolog, quanto para qualquer linguagem de programação em lógica.

Nas próximas seções são apresentadas a política de escalonamento, o algoritmo de cópia incremental e o algoritmo de terminação. Essas funcionalidades precisam ser inseridas nos nodos trabalhadores. Por isso, o modelo lógico é mapeado para um conjunto de módulos que se relacionam por interfaces específicas.

4.3 Política de Escalonamento de Tarefas

Para que ocorra o processamento de forma paralela, é necessário que as tarefas sejam escalonadas entre os trabalhadores. Na programação em lógica, é conveniente que o escalonamento seja dinâmico, devido a algumas de suas características de execução [DUT 95]:

- o grau de paralelismo varia durante a execução, exigindo decisões dinâmicas;
- programas em lógica possuem padrões computacionais muito irregulares, não sendo adequados para o particionamento estático;
- um mesmo programa gera árvores de execução diferentes dependendo dos conjuntos de consultas e de dados de entrada.

Pode-se utilizar também informações estáticas, geradas durante a compilação, para auxiliar as decisões dinâmicas. Esse tipo de escalonamento pode ser classificado como semi-dinâmico.

O modelo pclp(FD) utiliza uma política de escalonamento dinâmica e distribuída. O algoritmo será analisado através da estrutura apresentada na seção 3.3.2, após a definição do conceito de tarefa para esse modelo.

4.3.1 Definição de Tarefa

Uma **tarefa** (a ser exportada em uma linguagem CLP) são ramos inexplorados da árvore de busca. Assim, a exportação de uma tarefa implica a exportação de **ponto(s) de escolha** e de um **contexto** de execução, podendo ser representada pela equação:

TAREFA = PONTO DE ESCOLHA + CONTEXTO

Ponto de escolha, também denominado nodo OU, é a estrutura de dados da máquina abstrata que contém **alternativas** não exploradas. O contexto de execução é o conteúdo das **pilhas** e registradores de execução da WAM no qual o ponto de escolha está inserido.

4.3.2 Política de Transferência

Em pclp(FD), um nodo está adequado para participar de uma transferência de tarefa dependendo de seu estado. Os estados de um nodo são definidos por intervalos de valores (*thresholds*), definidos por experimentação, do número de pontos de escolha criados e ainda não explorados.

Conforme o número de pontos de escolha a serem resolvidos em um determinado momento, um trabalhador pode assumir um de três estados:

- **ocioso** (*idle*): quando o trabalhador não possui tarefas a serem executadas. Este é o estado inicial de todos os trabalhadores com exceção do trabalhador principal. Um trabalhador permanece neste estado até que importe uma tarefa de outro trabalhador;
- **ocupado** (*busy*): quando o trabalhador possui um nível baixo de tarefas a serem executadas, isto é, não existe trabalho em nível que justifique a exportação. Este é o estado inicial do trabalhador principal, uma vez que ele inicia a execução do programa;
- **sobrecarregado** (*overloaded*): quando o trabalhador possui tarefas excedentes que podem ser exportadas.

Deste modo, se k for uma constante, definida por experimentação, que indica o número máximo de pontos de escolha que um trabalhador consegue resolver sem exportar; e *npe* o número de pontos de escolha com alternativas ainda não exploradas em um trabalhador, então os estados podem ser definidos como:

$$\begin{cases} npe = 0 & : & ocioso \\ 0 < npe \leq k & : & ocupado \\ npe > k & : & sobrecarregado \end{cases}$$

Note-se que a utilização do número de pontos de escolha para a definição do estado de um trabalhador é a mais utilizada [ALI 90][LUS 90][GEY 92][MOR 97] mas não é obrigatória. Um sistema que obtenha informações de granulosidade durante o processo de compilação poderia utilizar estas medidas para a decisão de estado durante a execução.

A transição de estados sofrida por um trabalhador pclp(FD) é ilustrada na figura 4.3. Esta figura mostra que existe a diferença entre os trabalhadores normais e o trabalhador principal em relação aos seus estados iniciais, mas as transições posteriores são idênticas.

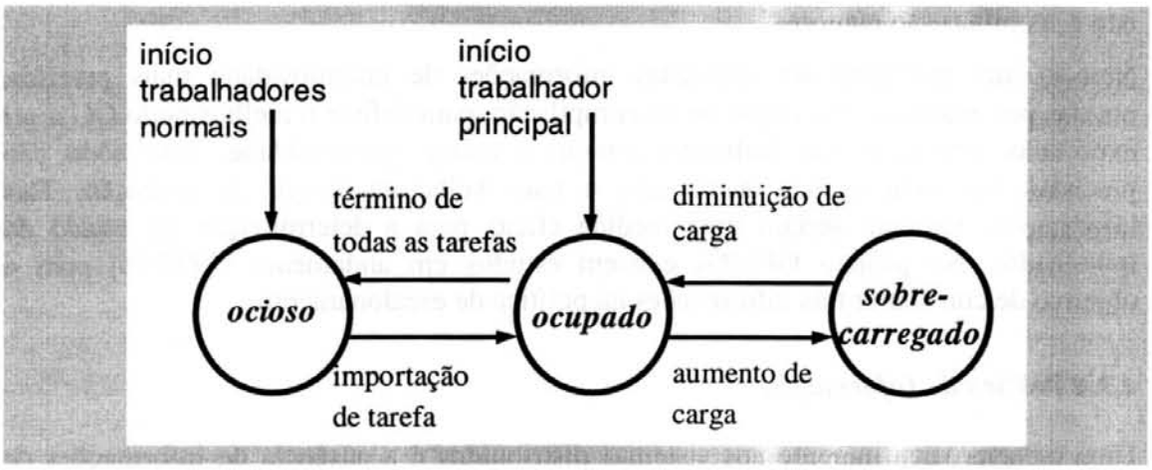

FIGURA 4.3 - Transição de estados em um nodo trabalhador.

Note-se que um trabalhador permanece no estado sobrecarregado até a diminuição da carga. A diminuição da carga ocorre tanto pelo processamento normal (seqüencial), que consome pontos de escolha, quanto pela exportação de tarefas. Já um trabalhador ocioso permanece neste estado até que, através de uma importação, receba uma nova tarefa.

Deste modo, os trabalhadores candidatos a receberem tarefas (candidatos a importadores) são os que estão em estado ocioso. Os trabalhadores candidatos a compartilharem tarefas (candidatos a exportadores) são os que estão em estado sobrecarregado.

4.3.3 Política de Seleção

A escolha de qual ponto de escolha exportar é muito importante na gerência de vários aspectos de um modelo de exploração de paralelismo OU. As duas abordagens mais utilizadas são:

- exportação do ponto de escolha mais antigo (*top-most dispatching*): utilizado nos escalonadores Argonne, Manchester e Wavefront do sistema Aurora

[LUS 90], com o objetivo de minimizar a área compartilhada, uma vez que o controle de contexto é pela técnica de compartilhamento de pilhas. Esses escalonadores são mais adequados para problemas com alta granulosidade. O modelo PLoSys [MOR 97] também utiliza essa abordagem com o objetivo de minimizar as pilhas copiadas.

- exportação do ponto de escolha mais recente (*bottom-most dispatching*): utilizado pelo MUSE [ALI 90] com o objetivo de minimizar o trabalho especulativo (trabalho especulativo, na execução paralela, é aquele que pode ser descartado devido a um comando *cut* em uma cláusula anterior). O escalonador Bristol do sistema Aurora é similar ao escalonador do MUSE e também utiliza a abordagem *bottom-most*. Em Aurora, os escalonadores Bristol e Manchester apresentam desempenhos similares.

Em pclp(FD), a tarefa escolhida para ser exportada é sempre o ponto de escolha mais antigo. A escolha do ponto de escolha mais antigo baseia-se na heurística de que as tarefas de mais alta granulosidade encontram-se próximas da raiz da árvore de busca [KER 92] e no fato de que, quanto mais velho, menor será o contexto a ser exportado, isto é, as pilhas são menores.

Note-se que poderiam ser utilizadas informações de granulosidade mais precisas, obtidas por análise na execução ou na compilação, para definir o melhor nodo OU a ser exportado, isto é, o que realmente possuir a maior granulosidade. Este nodo não precisaria ser nem o mais novo nem o mais velho da árvore de execução. Tais informações também seriam uma medida eficaz para a determinação do estado do trabalhador. No projeto OPERA existem estudos em andamento [COS 96] com o objetivo de considerar tais informações na política de escalonamento.

4.3.4 Política de Informação

Uma característica inerente aos sistemas distribuídos é a ausência de informações de estado global da aplicação. A obtenção de informações precisas exige sincronização absoluta o que é inviável nas aplicações práticas.

Para que o escalonamento possa ser feito de forma distribuída, os diversos escalonadores precisam coletar informações sobre o estado dos outros escalonadores. No modelo pclp(FD) existem dois pontos onde a informação de estado dos trabalhadores é propagada:

1. propagação por mudança de estado: sempre que um trabalhador se tornar ocioso ele avisa a todos os trabalhadores que está disponível para compartilhar trabalho, isto é, envia uma mensagem global (*broadcast*);
2. propagação por demanda: quando uma tarefa é exportada, juntamente com a tarefa é enviada uma lista com os trabalhadores que indicaram estar em estado ocioso.

Deste modo, cada trabalhador possui uma lista, relativamente atualizada, dos nodos que se encontram ociosos. Com o uso desta lista, a política de localização pode comunicar-se com um número menor de escalonadores, diminuindo o número de mensagens para exportação.

4.3.5 Política de Localização

Um trabalhador que esteja em estado sobrecarregado pode exportar parte de seu trabalho para qualquer trabalhador em estado ocioso. Assim, é preciso determinar qual o trabalhador mais adequado para receber a tarefa.

O escalonamento proposto é classificado como *sender initiated,* pois o exportador procura localizar um trabalhador apto a importar, isto é, um trabalhador que esteja em estado ocioso.

A procura do nodo importador passa pelas seguintes fases:

1. o trabalhador exportador informa cada um dos trabalhadores, que esteja cadastrado na sua **lista de ociosos**, que existe uma tarefa a ser exportada. Envia junto uma lista com os pontos de escolha dos tipos importados e exportados, isto é, que podem ser comuns, conforme será visto na seção 4.4.3;
2. um trabalhador que for notificado da existência de tarefas, estando ocioso, confirma o seu interesse na importação indicando o último ponto de escolha comum, ou a inexistência de um ponto de escolha comum. Como a lista pode estar desatualizada, caso um trabalhador não ocioso receber essa mensagem, apenas a ignora;
3. o trabalhador exportador coleta as confirmações até ter obtido um determinado número de respostas ou até um tempo máximo (*timeout*). O exportador avalia o trabalhador cujo custo de envio da tarefa seja menor e o elege como importador da sua tarefa. O custo de envio considera a quantidade de dados a ser transferida - vide seção 4.4 sobre cópia incremental - e o tempo que o trabalhador demorou a responder - privilegiando a localidade e menor custo de comunicação. O escolhido recebe a tarefa e a lista atualizada dos trabalhadores ociosos. Os demais trabalhadores são notificados da impossibilidade de receberem a tarefa e ficam livres para efetuar outras transações de exportação.

4.4 Cópia Incremental

Na seção 4.4.1 apresenta-se a técnica de exportação de contexto escolhida para esse modelo: a cópia incremental de pilhas. A cópia incremental é definida na seção 4.4.2. Na seção 4.4.3 apresenta-se o algoritmo proposto, cuja gerência é efetuada de forma descentralizada, devido a existência de uma política de escalonamento distribuída.

4.4.1 Escolha da Técnica de Exportação de Contexto

Conforme apresentado na seção 3.2, existem três tipos de técnicas para exportação do contexto de uma tarefa no paralelismo OU multi-seqüencial: compartilhamento, cópia e recomputação.

Como o ambiente considerado é distribuído, o compartilhamento de pilhas somente seria viável com a existência de uma camada de *software* e/ou *hardware* que implementasse memória compartilhada distribuída (*distributed shared memory*) permitindo a simulação de memória comum entre diferentes processadores.

As pilhas em CLP são maiores do que em Prolog [MUD 94], aumentando o custo para copiá-las, representando uma vantagem para a recomputação. Entretanto, as operações

de resolução de restrições em CLP são relativamente caras, o que eleva o custo de recomputação.

Optou-se pela cópia de pilhas que evita o desperdício de computação útil e possui uma implementação mais simples. Também considerou-se nesta decisão o fato de que na transmissão de mensagens em uma rede, a latência é mais cara que a banda, isto é, diminuir o tamanho de uma mensagem traz ganho de desempenho menos significativo do que a diminuição no número total de mensagens. Considerando-se que nas técnicas de cópia e de recomputação o número de mensagens é aproximadamente o mesmo, o tempo total de latência também será aproximadamente o mesmo. Assim, o ganho da recomputação provocado pela diminuição da banda não é tão significativo.

Porém, deve-se considerar que, além do fato das pilhas serem maiores por manipularem restrições, em problemas reais a árvore de busca pode ser muito profunda e, portanto, as pilhas a serem copiadas se tornam ainda maiores. A utilização de um mecanismo de **cópia incremental** [ALI 90] [GEY 92], como o proposto por esse modelo, constitui-se de uma importante otimização no custo da cópia de pilhas.

4.4.2 Definição de Cópia Incremental

A cópia incremental baseia-se no princípio de que trabalhadores que já realizaram alguma tarefa podem ter em suas pilhas várias informações comuns a um trabalhador que esteja exportando trabalho.

A figura 4.4 representa a relação entre a execução de uma árvore de busca e a manipulação das pilhas da máquina abstrata. Ela ilustra a existência de partes comuns das pilhas WAM entre trabalhadores que passaram por processos de importação e exportação de tarefas.

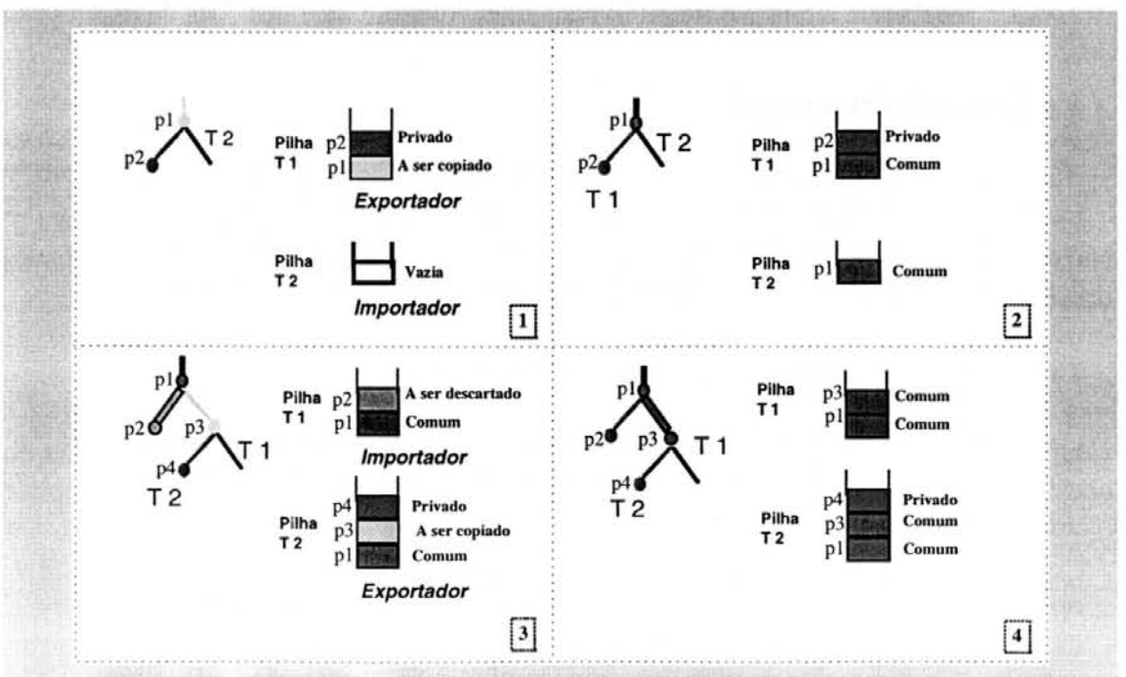

FIGURA 4.4 - Cópia Incremental entre dois trabalhadores.

Na etapa inicial (1), o trabalhador T1 processa um ramo da árvore de busca enquanto o trabalhador T2 encontra-se ocioso e com as suas pilhas vazias. T1 exporta um ponto de escolha *p1* para T2. Após a exportação (2), os trabalhadores T2 e T1 possuem em comum o ponto de escolha exportado (*p1*) e todos os dados anteriores a ele.

No momento (3), o trabalhador T1 deve importar uma tarefa de T2. Neste caso, as porções comuns não precisam ser recopiadas. Apenas as porções de pilha geradas entre o ponto de escolha comum *p1* e o ponto *p3* a ser exportado devem ser copiadas. Assim, em (4) o importador T1 descarta (através de retrocesso) o nodo *p2* que era o mais recente e todos os valores produzidos por este nodo, até o nodo comum *p1*. O ponto *p3*, enviado pelo exportador é, então, acrescentado à pilha de T1.

A **figura 4.5** apresenta um exemplo de árvore de busca. Um trabalhador T2 que terminou a execução de uma tarefa, isto é, está em estado ocioso, deve permanecer com o conteúdo de suas pilhas de execução intacto. Caso um outro trabalhador T1 tenha tarefas para exportar e T1 e T2 possuam um ponto de escolha (nodo OU) comum, o trabalhador exportador T1 somente precisará enviar o contexto de execução no intervalo entre o nodo comum e o nodo a ser exportado.

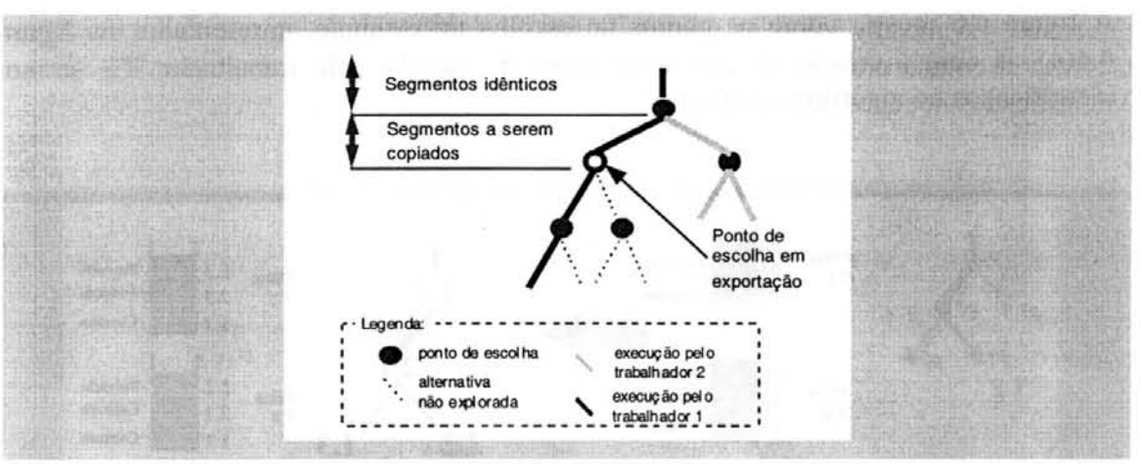

FIGURA 4.5 - Árvore de busca com nodos comuns entre trabalhadores.

Conforme descrito no Anexo 1, a pilha Local da máquina abstrata armazena os pontos de escolha, enquanto na pilha Trail são armazenadas as ligações condicionais. Os pontos de escolha indicam as alternativas pendentes a serem testadas. As ligações condicionais são indicadores para as variáveis globais que foram criadas antes do ponto de escolha e instanciadas ou alteradas em uma alternativa do ponto de escolha. Quando uma alternativa falha, é possível restaurar o valor da variável alterada. Em Prolog seqüencial, esse registro na Trail é apenas do endereço da variável, uma vez que as variáveis somente podem ser instanciadas uma única vez e, portanto, restaurar o valor anterior significa torná-la livre. Em CLP, a Trail guarda também o valor anterior, já que uma variável pode sofrer múltiplas instanciações.

O principal problema enfrentado para a implementação do conceito de cópia incremental é a detecção dos pontos de escolha comuns e, deste modo, a delimitação de quais porções da pilha devem ser transferidas.

Os sistemas que implementam a cópia incremental, descritos na literatura, podem considerar a existência de informações globais, seja por uma área de memória

compartilhada ou por um escalonador centralizado. Um escalonador centralizado, por ser responsável pelo controle dos pontos de escolha exportados, possui a informação completa sobre os compartilhamentos de pontos de escolha.

A utilização de um escalonamento distribuído e de um ambiente com memória distribuída, não permite a disponibilização de informação global sobre o compartilhamento de pontos de escolha. O algoritmo de cópia incremental do modelo pclp(FD) trata esse problema.

4.4.3 Algoritmo de Cópia Incremental

O ponto chave do algoritmo de cópia incremental é a identificação única dos pontos de escolha com o objetivo de permitir a detecção de pontos comuns. Deste modo, cada ponto de escolha passa a possuir um identificador.

O **identificador de um ponto de escolha** é um número composto pelo identificador único do trabalhador criador do ponto de escolha e um número inteiro. Esse segundo número indica a ordem de criação do ponto de escolha no trabalhador, não possuindo nenhuma relação com a profundidade da árvore de busca.

A figura 4.6 mostra como os pontos de escolha do exemplo, apresentados na figura 4.4(4), já com a criação de um novo ponto de escolha pelo trabalhador T1, seriam identificados no algoritmo proposto.

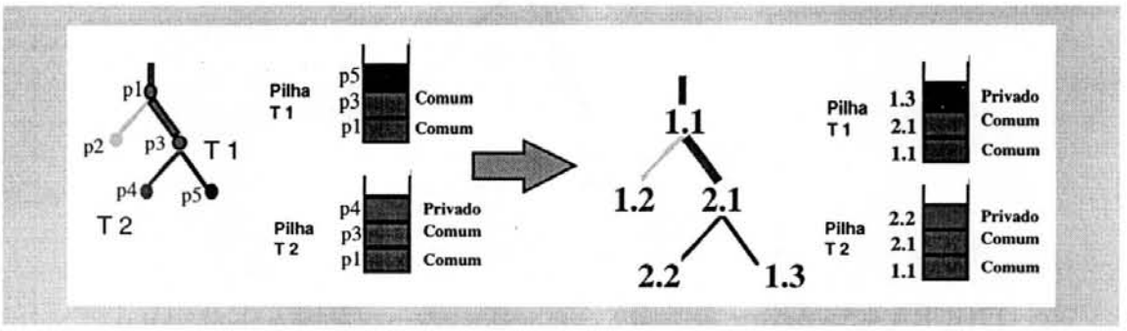

FIGURA 4.6 - Numeração dos pontos de escolha.

Um ponto de escolha ao ser exportado mantém o seu identificador original, como pode ser observado na figura 4.6.

Para permitir o controle da cópia, existe uma estrutura auxiliar que armazena o tipo de cada ponto de escolha criado: a Lista de Pontos de Escolha (LPE). Um ponto de escolha pode ser:

- **local**: foi criado pelo trabalhador e nunca foi exportado, possuindo ainda alternativas pendentes;
- **importado**: é um ponto de escolha que foi criado por outro trabalhador e foi importado. Ainda existem nele alternativas pendentes.
- **exportado**: as alternativas deste ponto de escolha foram exportadas para outro trabalhador e, portanto, não existem alternativas pendentes;

Note-se que um nodo local é removido quando for removida a sua última alternativa.

Para a detecção dos pontos de escolha comuns ser possível, existem algumas condições que devem ser observadas na exportação de tarefas e no processamento normal:

Condição 1: sempre é exportado o ponto de escolha mais velho da árvore de busca, que é marcado no exportador como exportado e no importador como importado;

Condição 2: ao efetuar retrocesso em um ponto de escolha do tipo exportado, ou seja, que tenha sido exportado logicamente mas não fisicamente, não deve-se remover esse ponto de escolha pois não existem mais alternativas pendentes, nem neste nodo nem acima dele[1];

Condição 3: ao remover a última alternativa pendente de um ponto de escolha do tipo importado, esse ponto de escolha somente pode ser realmente removido se houver pontos de escolha anteriores a ele com alternativas pendentes[2].

A condição 1 garante a inexistência de alternativas pendentes anteriores a um ponto de escolha exportado, o que permite a validade das condições 2 e 3. As condições 2 e 3 evitam que o conteúdo das pilhas de execução seja descartado ao término da execução das tarefas.

A cópia incremental é efetuada através das seguintes etapas:

Etapa 1: **detecção de partes comuns** - para que dois trabalhadores possuam porções comuns das pilhas de execução, é necessário que ambos já tenham passado por processos de importação/exportação. Para a determinação de quais porções das pilhas de execução são comuns, compara-se os identificadores dos pontos de escolha importados e exportados existentes na LPE. Assim, quando houver dois trabalhadores que possuam um ou mais pontos de escolha com o mesmo identificador, isso indica que as porções de pilhas anteriores ao ponto de escolha comum mais recente são idênticas em ambos os trabalhadores. Para facilitar a detecção, utiliza-se a estrutura auxiliar LPE, de cada trabalhador, que armazena os tipos dos pontos de escolha existentes nas suas pilhas.

Etapa 2: **cópia das porções de pilhas** - uma vez que seja detectada a existência de pontos de escolha comuns, pode-se efetuar a cópia parcial das pilhas. A cópia em paralelo com a execução normal pode causar o envio de dados inconsistentes para o importador. Para evitar conflitos é necessário introduzir alguns pontos de sincronização. Em primeiro lugar, é necessário evitar que o ponto de escolha que está sendo transferido seja destruído ou atualizado enquanto apenas parte das pilhas foram copiadas. Para isso, deve-se evitar as operações de remoção (*trust*) e atualização (*retry*) durante a cópia da pilha. Também é preciso evitar que variáveis globais sejam atualizadas pelo ramo atual da árvore de busca sem que seja possível retornar ao valor anterior no importador. Para tal, obriga-se que um valor condicional de uma variável seja

---

[1] Isso significa que o trabalhador vai para o estado ocioso pois não há mais trabalho local, não sendo possível realizar retrocesso localmente.

[2] Uma vez que sempre exporta-se os pontos de escolha mais antigos, acima de um ponto de escolha importado pode-se ter apenas pontos de escolha exportados. Somente existirá outros pontos de escolha importados e, portanto, com alternativas pendentes, caso for permitida a exportação de mais de um ponto de escolha simultaneamente.

primeiro colocado na Trail e depois efetivamente feita a alteração na pilha Global, e que a cópia das pilhas seja primeiro da Global e depois da Trail. Assim, caso a cópia ocorra no meio de uma operação de atualização de variável, sempre será possível desfazer os valores de variáveis atribuídos nas partes privadas da execução.

Etapa 3: **ajuste das estruturas no importador** - As pilhas Global e Local são copiadas parcialmente, mas a pilha Trail é transferida em sua totalidade. O primeiro procedimento que o importador deve efetuar após a cópia das pilhas é o retrocesso, com o uso da pilha Trail. O objetivo é desfazer os valores recebidos por variáveis nos ramos da árvore de busca posteriores ao ponto de escolha importado, reconstituindo o contexto de execução.

4.5 Algoritmo de Terminação

Um algoritmo distribuído pode ser considerado **globalmente terminado** quando todos os processadores estiverem em seu estado final de computação e não houverem mensagens em trânsito em nenhum dos canais de comunicação [BAV 93]. Para pclp(FD), a detecção da terminação global é vital para que seja finalizada a máquina abstrata, assim como todos os demais processos envolvidos na execução paralela.

Um escalonador centralizado detecta o final da computação quando todos os trabalhadores reportam a ausência de trabalho, isto é, todos estão requisitando tarefas. No modelo pclp(FD), a utilização de um escalonador distribuído implica uma detecção não trivial do término da execução.

Existem vários algoritmos propostos para detecção do estado final em sistemas distribuídos, como os descritos em livros didáticos como [LYN 96], [BAV 93] e [SIN 94], que poderiam ser utilizados neste trabalho.

O modelo pclp(FD) utiliza uma adaptação do algoritmo de terminação de Hwang, cujo original é descrito em [SIN 94]. Este algoritmo é adequado por não pressupor qualquer topologia específica de interconexão entre os trabalhadores. Além disso, tanto o modelo quanto o algoritmo de terminação pressupõem que a computação inicia em um único trabalhador.

O algoritmo original possui um agente central (*controlling*), que controla a terminação, e uma série de agentes que podem estar em estado ativo ou inativo. O algoritmo tem como idéia básica a atribuição de pesos às tarefas que são exportadas. Todos os agentes devem agir de acordo com quatro regras previstas, de tal forma que, quando o agente controlador recuperar o seu peso original, a computação está globalmente terminada.

Para a utilização deste algoritmo no modelo, foram efetuadas algumas alterações.

O escalonador do trabalhador principal é encarregado de determinar a terminação do processamento (assumindo o papel de agente controlador de terminação). Isso é importante porque o peso do principal não pode ser zerado, caso contrário a terminação nunca será alcançada.

Os demais escalonadores são equivalentes aos agentes normais do algoritmo. No entanto, cada escalonador está associado a um trabalhador que pode assumir três estados - ocioso, ocupado e sobrecarregado - conforme abordado na seção 4.3.2. Esses três estados podem ser mapeados para o algoritmo original do seguinte modo: (1) o estado

ocioso equivale ao inativo; (2) o estado ocupado não possui correspondente; (3) o estado sobrecarregado corresponde ao ativo. O estado ocupado não precisa ser considerado pois escalonadores nesse estado não trocam informações.

As quatro regras sofreram pequenas adaptações, que não alteraram a validade do algoritmo. Essas alterações são basicamente com relação a mudança de nomenclatura dos estados e dos agentes envolvidos. Também evitou-se que o escalonador principal zerasse seu peso, eliminando o envio de uma mensagem para ele mesmo indicando o seu peso anterior.

O algoritmo pode ser informalmente enunciado do seguinte modo:

> O algoritmo de detecção de terminação inicia com a computação normal iniciada pelo trabalhador Principal. O **peso inicial** do trabalhador Principal é 1 e dos trabalhadores normais é 0. Sempre que um trabalhador estiver no estado sobrecarregado, o escalonador pode exportar uma tarefa. Uma tarefa exportada deve conter, além do contexto de execução, um **peso** (*weight*) associado. O valor deste peso é a metade do valor existente no exportador. Sempre que um trabalhador se tornar ocioso, seu peso é zerado (pois não possui mais tarefas em execução) e o seu peso anterior é informado ao escalonador principal. O escalonador principal soma ao seu peso atual os valores informados pelos escalonadores ociosos. Quando o escalonador principal restaurar o seu peso inicial, a terminação global é detectada.

Assim, para que o algoritmo possa ser executado, é necessário que inicialmente o escalonador principal receba peso 1 e os demais recebam peso 0. O processamento prossegue normalmente, com todos os trabalhadores obedecendo as quatro regras enunciadas a seguir, até que a aplicação seja detectada como globalmente terminada.

regra 1: um escalonador, com peso P, para enviar uma tarefa deve:

```
1. obter pesos P1 e P2 que atendam as seguintes condições:
     P1 + P2 == P
     P1>0
     P2>0
2. atualizar o seu peso:
     P ← P1
3. enviar a tarefa com peso P2 para o importador.
```

regra 2: um escalonador, com peso P, ao receber uma tarefa com peso Px deve:

```
1. atualizar o seu peso:
     P ← P + Px
```

regra 3: quando seu trabalhador ao se tornar ocioso,

um escalonador normal deve:

```
1. enviar mensagem de controle com seu peso P para o escalonador
principal.
2. zerar seu peso
     P ← 0
```

o escalonador principal verifica:

```
1. se P=1
     a computação terminou
```

regra 4: o escalonador principal, com peso P, ao receber uma mensagem de controle com peso Px deve:

```
1. atualizar o seu peso:
     P ← P + Px
2. se P=1 e o seu trabalhador está ocioso
     a computação terminou
```

Para facilitar a compreensão do algoritmo, a **figura 4.7** representa uma simulação simplificada com quatro trabalhadores. A troca de pesos entre os trabalhadores, devido a troca de mensagem com tarefas e a mensagem *broadcast* indicando estado ocioso, ocorre até que a terminação seja detectada.

A primeira situação retrata o início do processamento, onde apenas o trabalhador principal executa tarefas e seu escalonador possui peso 1. No momento 2, o trabalhador principal se torna sobrecarregado. O escalonador principal atualiza o seu peso, e exporta uma tarefa para o escalonador 2 com um peso 0,5, seguindo a *regra 1*. O escalonador 2, seguindo a *regra 2*, passa a possuir o peso 0,5.

Estando ainda sobrecarregado, o escalonador 1 utiliza a *regra 1* para exportar ao trabalhador 3 uma tarefa com peso 0,25 e para atualizar o seu peso (3). O escalonador 3 atualiza seu peso para 0,25 (*regra 2*).

No quarto quadro existem dois casos ocorrendo em paralelo. Quando o escalonador 2 se torna ocioso, ele envia uma mensagem global (*broadcast*) informando que está ocioso. Seguindo a *regra 3*, ele precisa zerar o seu peso, que era de 0,5, e informar o seu peso antigo para o trabalhador principal. Aproveitando a mensagem já existente no protocolo de comunicação, o trabalhador 2 informa que está ocioso e que seu peso era 0,5. Apenas ao trabalhador principal essa informação é relevante. Deste modo, seguindo a *regra 4*, o trabalhador principal adiciona 0,5 ao seu peso, ficando com um valor 0,75. Como esse valor é diferente de 1, utilizando ainda a *regra 4*, o trabalhador principal constata que a computação ainda não terminou. Já o trabalhador 3 que está sobrecarregado, envia para o trabalhador 4 uma tarefa com peso 0,125.

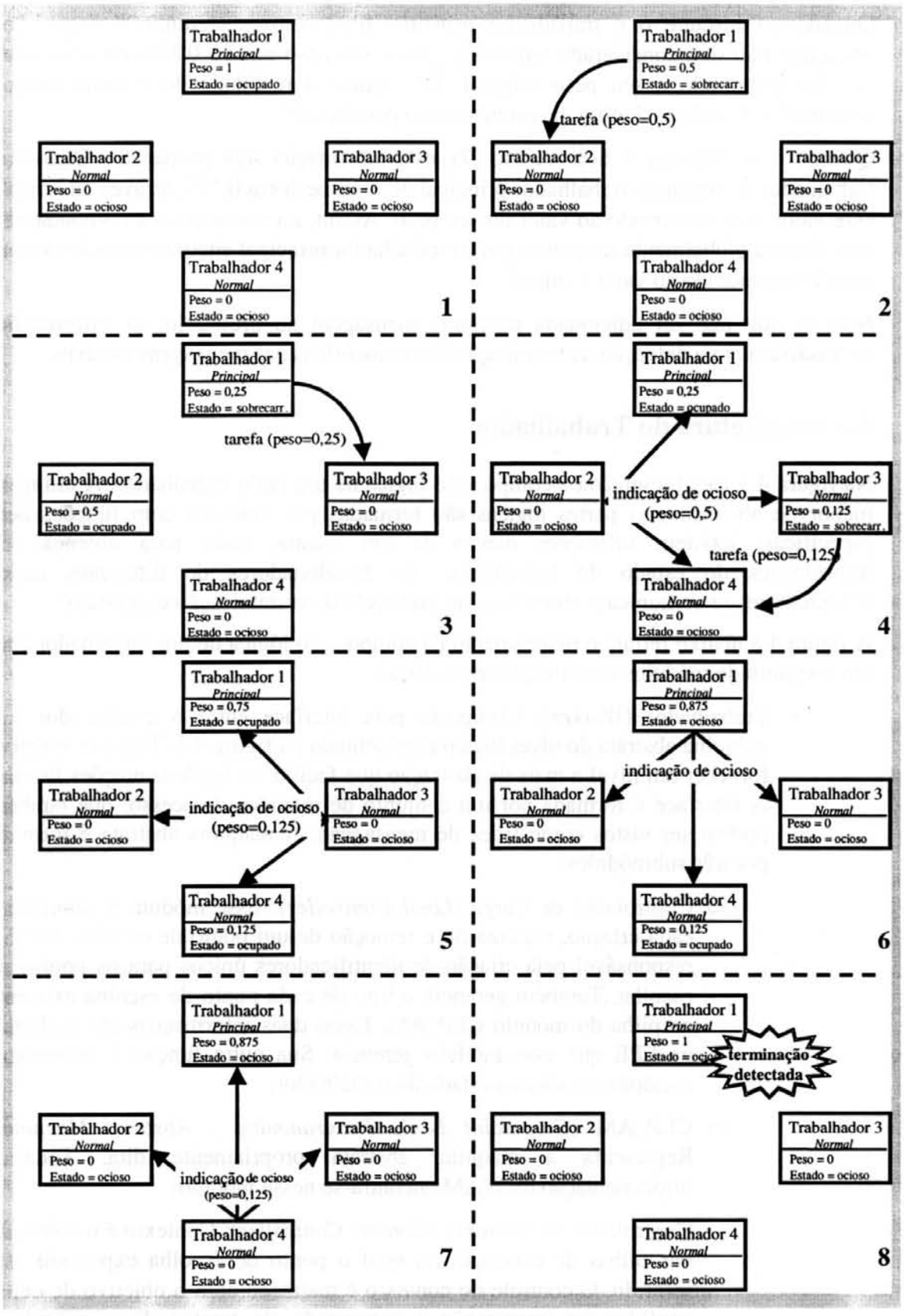

FIGURA 4.7 - Exemplo de execução com utilização do algoritmo de terminação.

Quando o trabalhador 3 fica ocioso, ele procede da mesma forma como o trabalhador 2 agiu: envia uma mensagem global. O peso associado neste caso é 0,125 e o trabalhador principal passa a possuir peso 0,875.

Quando no instante 6 o trabalhador principal ficar ocioso, seguindo a *regra 3*, o programa não será considerado terminado pois o seu peso atual é 0,875, ou seja, ainda não foi restaurado a seu peso original. Isso reflete a realidade do sistema onde o trabalhador 4 ainda está ativo, possuindo como peso 0,125.

Quando o trabalhador 4 ficar ocioso (7), esta informação será propagada. Quando o trabalhador 4 indicar ao trabalhador principal que seu peso era 0,125, através da *regra 4* esse valor será adicionado ao valor do seu peso. Assim, no momento 8 a computação é considerada globalmente terminada pois o trabalhador principal encontra-se ocioso e seu peso foi restaurado ao valor 1 inicial.

Note-se que não foi adicionada nenhuma mensagem ao protocolo: as informações necessárias para a detecção da terminação foram incluídas nas mensagens normais.

4.6 Arquitetura do Trabalhador

Na figura 4.1 foi apresentada a composição lógica de um nodo trabalhador. A um nível menor de abstração, as partes lógicas são formadas por módulos com funções bem específicas. Existem interações dentro de um mesmo nodo para obtenção de informações do estado do trabalhador. Os Escalonadores de diferentes nodos trabalhadores se comunicam através de um protocolo de comunicação específico.

A figura 4.8 abaixo retrata o mapeamento da composição lógica de um trabalhador para um conjunto de módulos com funções específicas:

- Trabalhador (*Worker*): Composto pela interface entre o escalonador e a máquina abstrata do nível lógico representado na figura 4.1. Tem por objetivo fornecer um nível a mais de abstração que facilite as implementações futuras. A interface é formada por um conjunto de métodos de acesso, que também podem ser vistos como troca de mensagens. A máquina abstrata é formada por três submódulos:
  - ⇒ Controlador de Carga (*Load Controller*): Esse módulo é acionado a cada criação, atualização e remoção de um ponto de escolha. Ele é o responsável pela criação de identificadores únicos para os pontos de escolha. Também gerencia o tipo de cada ponto de escolha existente na pilha do módulo CLP_AM. Essas duas informações são incluídas na LPE que esse módulo gerencia. Sua outra função é informar o escalonador sobre o estado do trabalhador;
  - ⇒ CLP_AM (*Constraint Logic Programming - Abstract Machine*): Representa a máquina abstrata propriamente dita. Toda a implementação da WAM encontra-se nesse módulo;
  - ⇒ Controlador de Contexto (*Context Controller*): Contexto é o conteúdo das pilhas de execução no qual o ponto de escolha exportado está inserido. O controle do contexto é realizado com o objetivo de evitar conflitos na importação e exportação de tarefas. Esse módulo é responsável por copiar as pilhas que serão enviadas para um trabalhador importador e por copiar corretamente na máquina abstrata as pilhas recebidas durante uma importação.

- Escalonador (*Scheduler*): Esse é o módulo responsável pelo escalonamento de tarefas, seguindo um protocolo de comunicação com outros escalonadores conforme apresenta-se na seção 4.6.5.

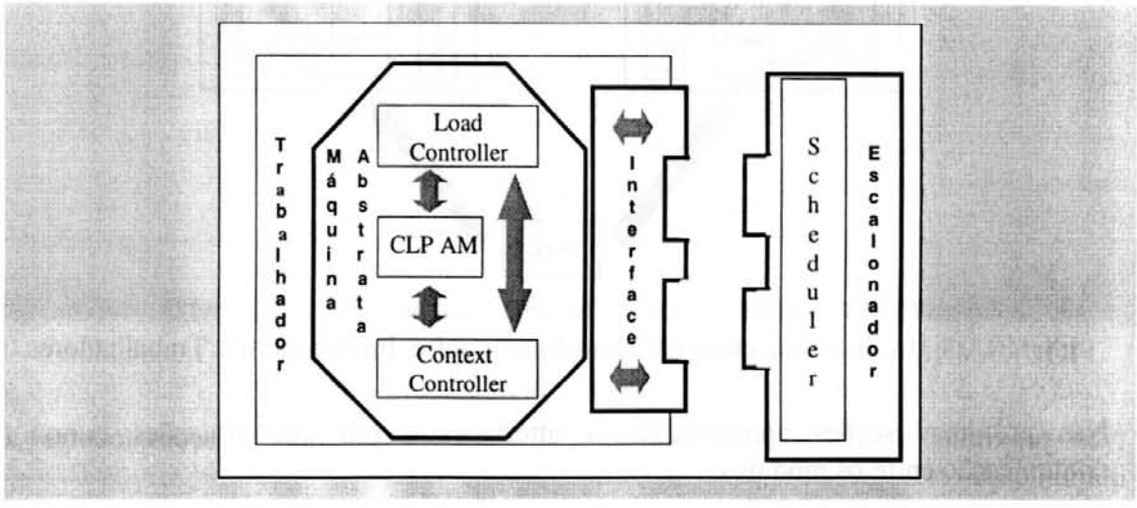

FIGURA 4.8 - Módulos que compõem um nodo trabalhador.

Há momentos específicos em que informações são trocadas entre os vários módulos que compõem o trabalhador. A forma como ocorrem estas trocas é definida por interfaces de comunicação (figura 4.9), que existem entre os todos os módulos que se comunicam diretamente.

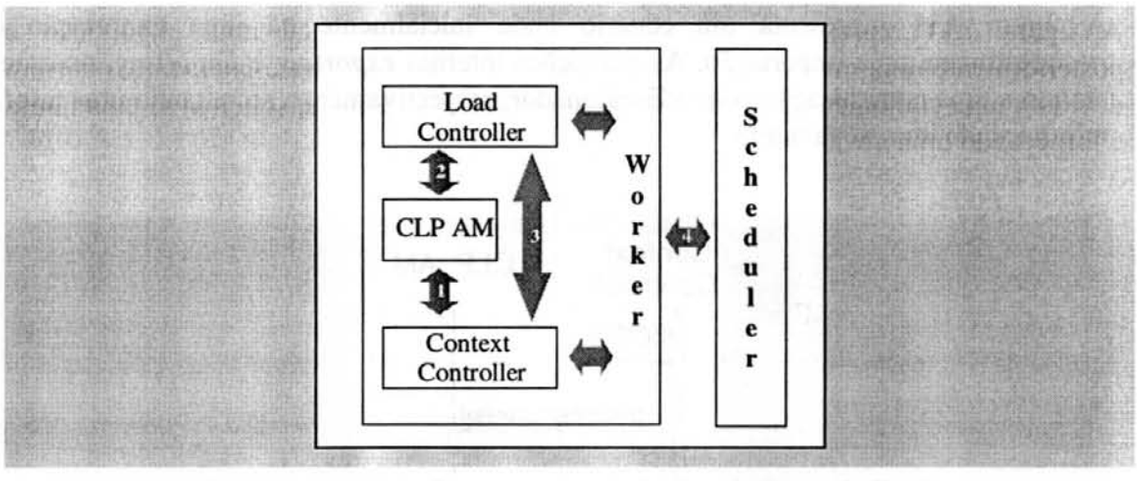

FIGURA 4.9 - Interfaces entre os módulos de um trabalhador.

Para iniciar o processamento, é necessário que um inicializador da arquitetura (*Starter*) crie os trabalhadores e inicialize os seus dados de entrada de forma adequada. Existe um protocolo específico para essa inicialização.

Existe ainda a interface de comunicação entre os escalonadores de trabalhadores distintos. Ambas as interfaces estão esquematizadas na **figura 4.10**.

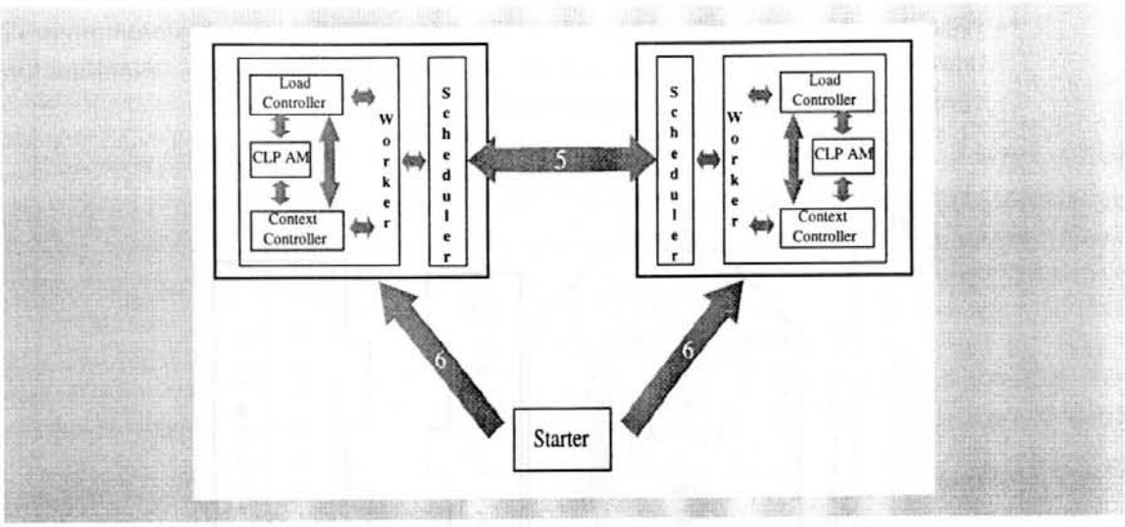

FIGURA 4.10 - Interface entre Escalonadores e entre Inicializador e Trabalhadores.

Nas próximas seções apresenta-se as interfaces e em que situações ocorre a comunicação entre os módulos.

4.6.1 Interface entre Controlador de Contexto e CLP_AM

O Controlador de Contexto (*Context Controller*) possui acesso às pilhas do CLP_AM que constituem o contexto de execução de uma tarefa. Quando ocorre uma importação, esse módulo é o encarregado de enviar o contexto para o CLP_AM. De forma análoga, durante a exportação, esse módulo é encarregado de obter o contexto para enviar para o escalonador.

A figura 4.11 representa um cenário onde inicialmente há uma exportação e posteriormente uma importação. As transições internas *exporting* e *importing*, ocorrem devido a uma comunicação com o Escalonador, respectivamente, solicitando uma tarefa e fornecendo uma nova tarefa.

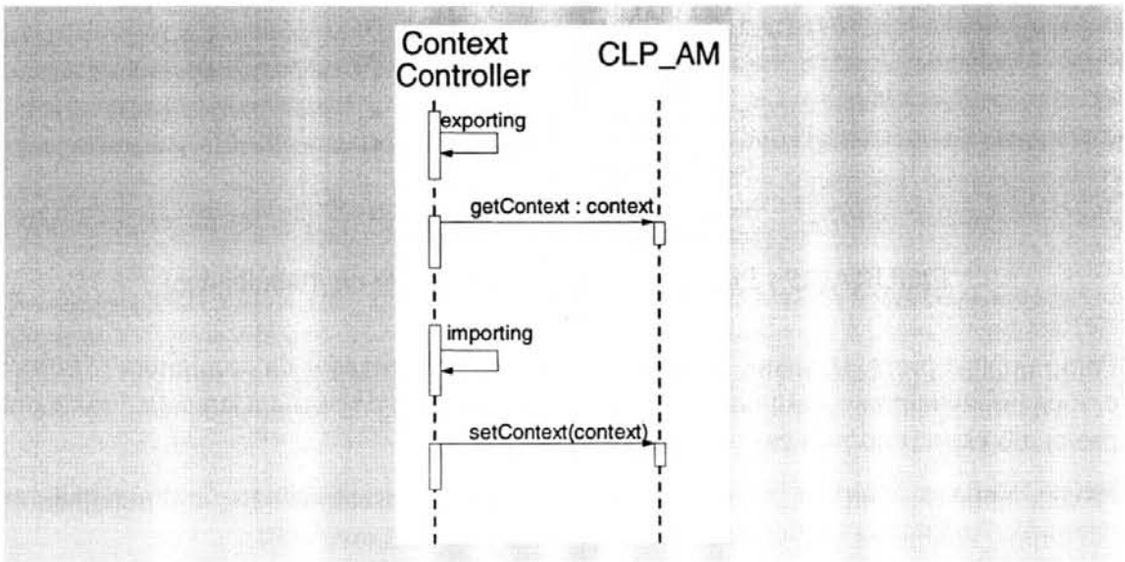

FIGURA 4.11 - Interface Controlador de Contexto e CLP_AM: importação e exportação.

4.6.2 Interface entre CLP_AM e Controlador de Carga

O Controlador de Carga (*Load Controller*) é um módulo praticamente passivo, que mantém uma estrutura com informações sobre todos os pontos de escolha existentes em CLP_AM. Como visto na seção 4.4.3 sobre cópia incremental, existem três tipos de pontos de escolha armazenados na LPE de um trabalhador X: local (existente apenas no trabalhador X), importado (foi gerado em outro trabalhador e importado por X) e exportado (criado no trabalhador X e exportado para outro trabalhador).

Para que a LPE, que é a estrutura de controle dos pontos de escolha, possa ser mantida, o módulo Controlador de Carga é informado cada vez que a carga do sistema é alterada, isto é, cada vez que uma operação que envolva pontos de escolha seja acionada.

Quando uma operação de adição de ponto de escolha for executada em CLP_AM, o Controlador de Carga é acionado para tomar conhecimento do novo ponto de escolha local (**figura 4.12**).

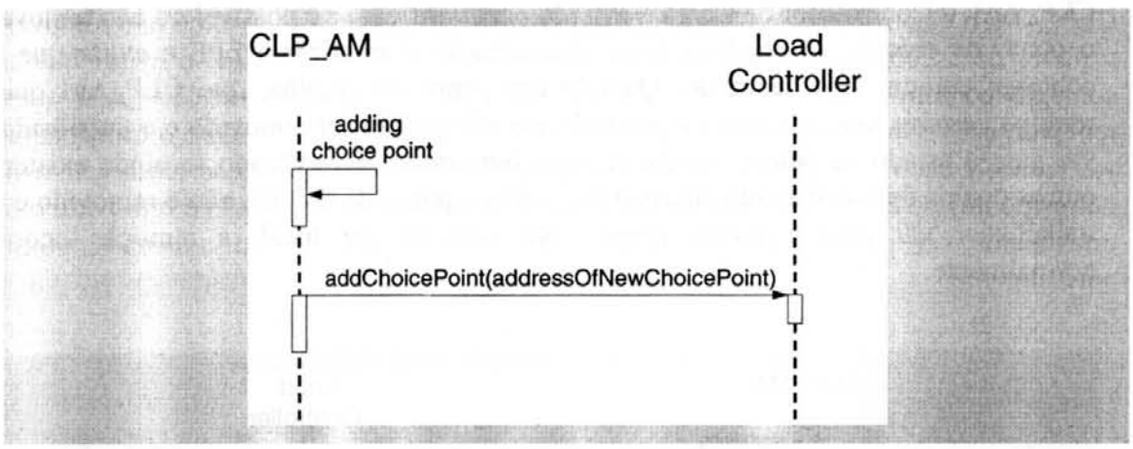

FIGURA 4.12 - Interface Controlador de Carga e CLP_AM: inclusão.

Quando é atualizado um ponto de escolha, isto é, uma alternativa é consumida, CLP_AM consulta o Controlador de Carga (figura 4.13) para verificar se esse ponto de escolha ainda encontra-se disponível, uma vez que o ponto de escolha pode ter sido exportado. De fato, caso não houver mais alternativas disponíveis devido a uma exportação, CLP_AM não poderá continuar a execução e o trabalhador será considerado ocioso.

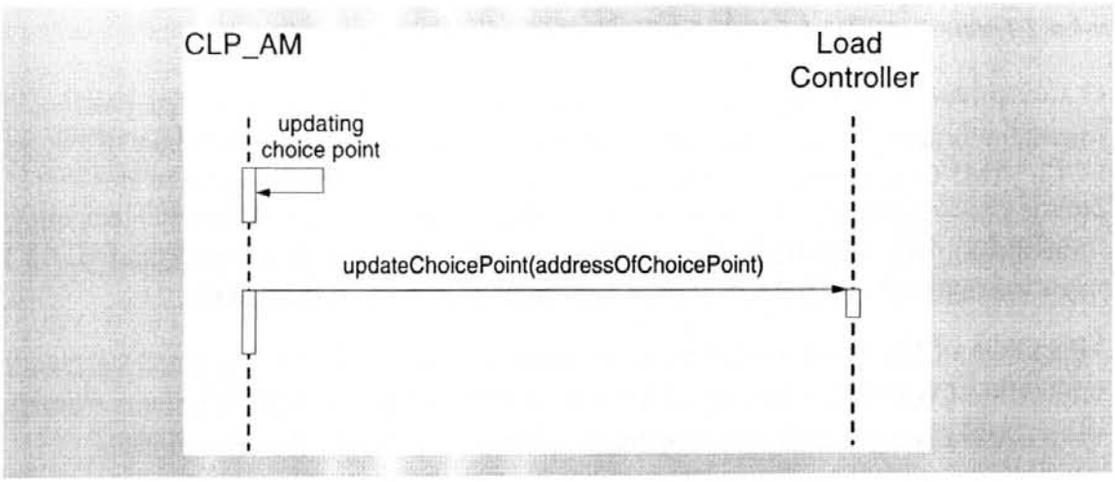

FIGURA 4.13 - Interface Controlador de Carga e CLP_AM: atualização.

Uma operação de remoção de ponto de escolha equivale ao consumo da sua última alternativa. Quando a operação de remoção de um ponto de escolha é acionada em CLP_AM, o Controlador de Carga é avisado para verificar se é possível ou não remover o ponto de escolha. O objetivo desta comunicação é atualizar a LPE e evitar que o contexto comum seja destruído. Quando um ponto de escolha, que CLP_AM quer remover, estiver marcado como *exportado*, ele não poderá ser removido e o trabalhador vai para o estado de ocioso. Se for do tipo *importado*, é verificado se ainda existem outros pontos de escolha com alternativas, senão o ponto de escolha não é removido e o trabalhador vai para o estado ocioso. No caso de ser local, a remoção ocorre normalmente.

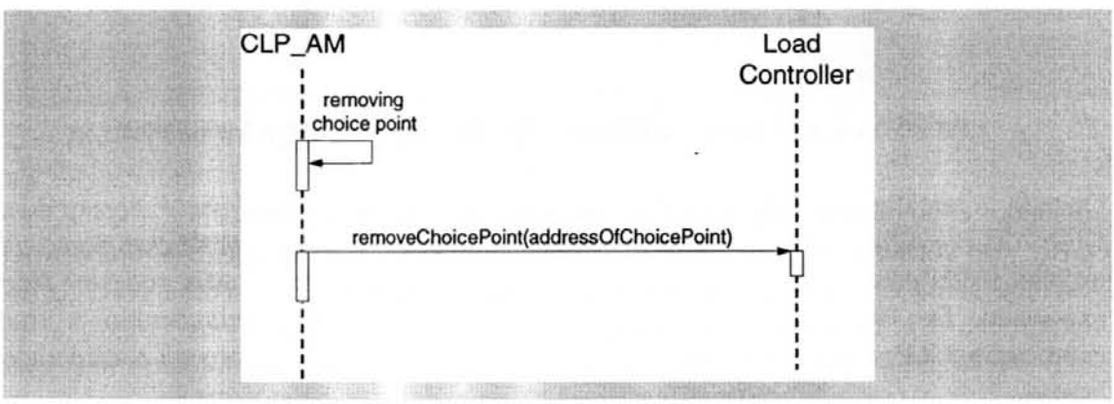

FIGURA 4.14 - Interface Controlador de Carga e CLP_AM: remoção.

4.6.3 Interface entre Controlador de Contexto e Controlador de Carga

A comunicação entre os módulos Controlador de Contexto (*Context Controller*) e Controlador de Cargas (*Load Controller*) somente ocorre durante os processos de importação (figura 4.15) e exportação de tarefas (figura 4.16).

Essa interação é necessária pois o Controlador de Contexto precisa obter as informações sobre os pontos de escolha que o Controlador de Carga possui, isto é, precisa acessar a LPE que contém a relação entre o endereço físico, na máquina abstrata, de cada ponto de escolha e o seu identificador único e o seu tipo.

Na importação, o Controlador de Contexto precisa repassar ao Controlador de Carga a LPE. Quando o Controlador de Contexto receber a nova LPE, também serão atualizadas as informações do número de pontos de escolha e do estado do trabalhador que se tornará ocupado.

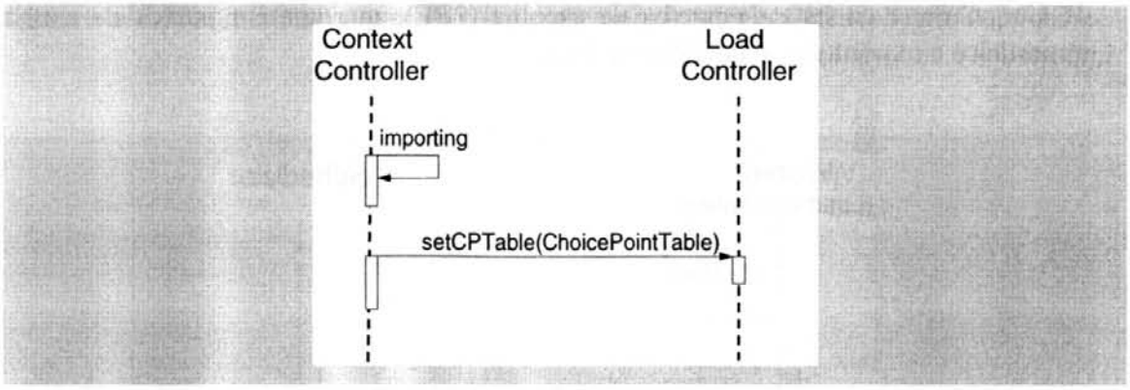

FIGURA 4.15 - Interface Controladores de Contexto e de Carga: importação.

Observa-se que na exportação, além de obter informações da LPE, o ponto de escolha que é exportado precisa ser marcado como exportado para evitar que a máquina abstrata local tente executar esse ponto de escolha posteriormente.

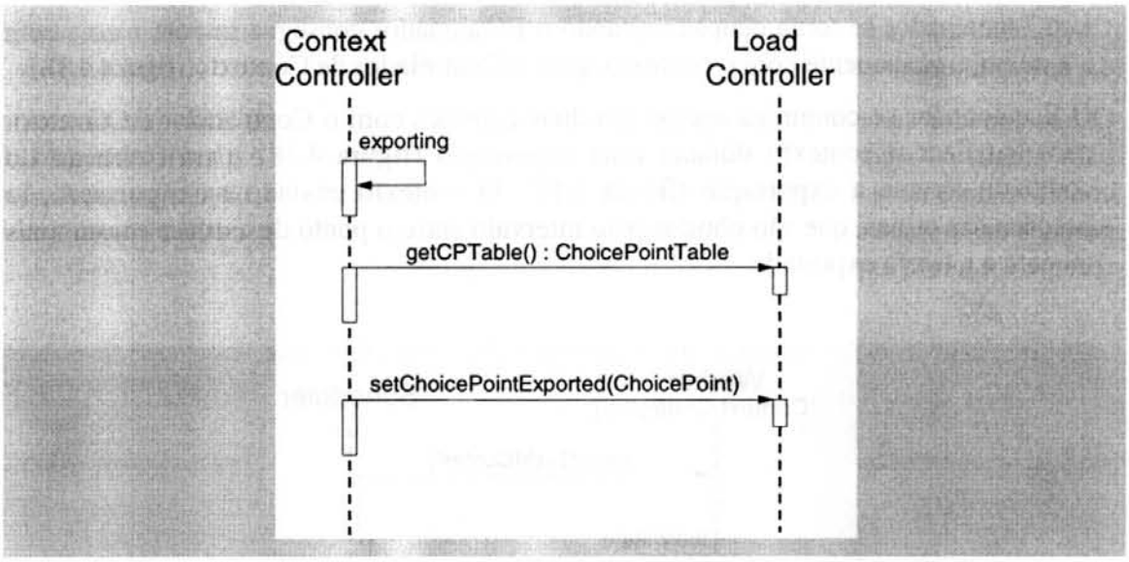

FIGURA 4.16 - Interface Controladores de Contexto e de Carga: exportação.

4.6.4 Interface entre Trabalhador e Escalonador

A interação entre Trabalhador e Escalonador permite que tarefas sejam importadas ou exportadas. O Escalonador nunca interage diretamente com a máquina abstrata (CLP_AM). Ele obtém informações da máquina abstrata através do módulos Controlador de Carga e de Contexto.

O Controlador de Carga tem uma tarefa parecida com a de um espião existente em sistemas como o OPERA/E [YAM 94]. Apesar de não ficar observando, mas sim ser

avisado, ele obtém informações de carga e informa o Escalonador das mudanças de estado do trabalhador.

Quando for removida a última alternativa pendente, o Controlador de Carga percebe que o Trabalhador encontra-se ocioso. Quando o Trabalhador vai para o estado de ocioso (figura 4.17), imediatamente o Escalonador é informado dessa situação pelo Controlador de Carga. Parte da lista de pontos de escolha (LPE), que contém pontos de escolha importados e exportados, é também enviada.

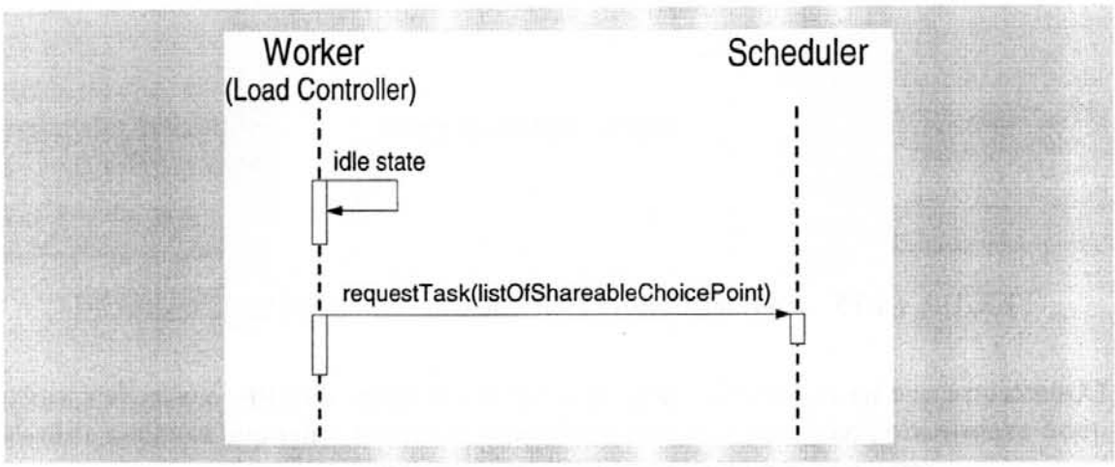

FIGURA 4.17 - Interface Trabalhador e Escalonador: trabalhador ocioso.

Um Trabalhador se torna ocupado quando o Escalonador consegue importar uma tarefa e a entrega, juntamente com o contexto, para o Controlador de Contexto (figura 4.18).

O Escalonador se comunica apenas em duas ocasiões com o Controlador de Contexto: para atualizar o contexto durante uma importação (figura 4.18) e para obtenção do contexto durante a exportação (figura 4.19). O contexto enviado na exportação, são porções das pilhas, que são obtidos pelo intervalo entre o ponto de escolha comum mais recente e a tarefa exportada.

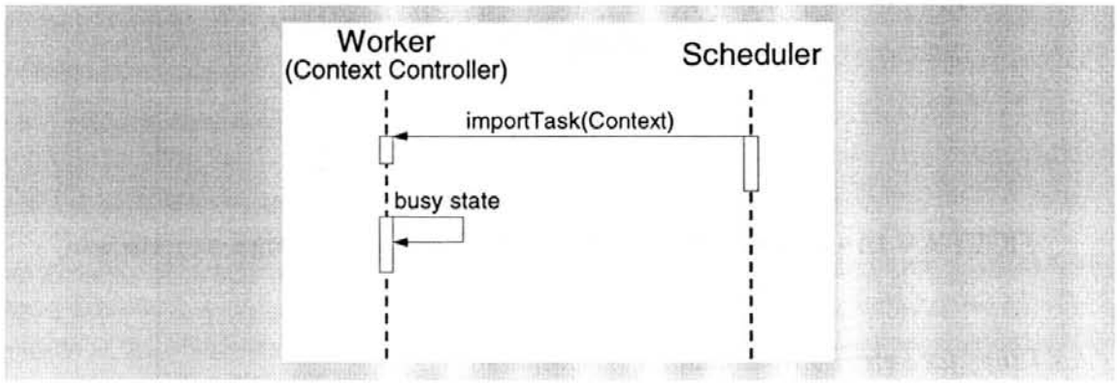

FIGURA 4.18 - Interface Trabalhador e Escalonador: trabalhador ocupado.

Ao atingir um número de pontos de escolha maior que o limite de ocupado, o Controlador de Carga determina que o Trabalhador encontra-se sobrecarregado. Ao ficar sobrecarregado, o Trabalhador (Controlador de Carga) informa ao Escalonador essa situação e envia os pontos de escolha da LPE que possuem os tipos importado e

exportado, formando uma lista de pontos de escolha compartilháveis. A figura 4.19 ilustra uma transação bem sucedida de exportação de tarefas.

Inicialmente o Trabalhador, ao ficar sobrecarregado, informa que há tarefas disponíveis e envia a lista para facilitar a escolha do Escalonador. De fato o Controlador de Carga é o encarregado de enviar essas informações. Quando o Escalonador escolhe um importador adequado, o Trabalhador, isto é, o Controlador de Contexto, recebe a indicação de qual é o último ponto de escolha comum entre ele e o importador. A tarefa é então passada ao Escalonador que se encarrega de enviar ao importador escolhido.

No entanto, a qualquer momento o Trabalhador poderia ter avisado que não estava mais sobrecarregado, cancelando a tentativa de exportação através de uma mensagem `Cancel()`. Essa mensagem de cancelamento somente pode ocorrer antes do início da cópia das pilhas, pois, a partir desse momento, CLP_AM não consegue remover nenhum alternativa não podendo também ir para um estado que justifique o cancelamento. Quando um Escalonador receber a mensagem `Cancel()`, a transação de exportação é abortada, e todos os trabalhadores ociosos que responderam recebem uma mensagem cancelando a transação.

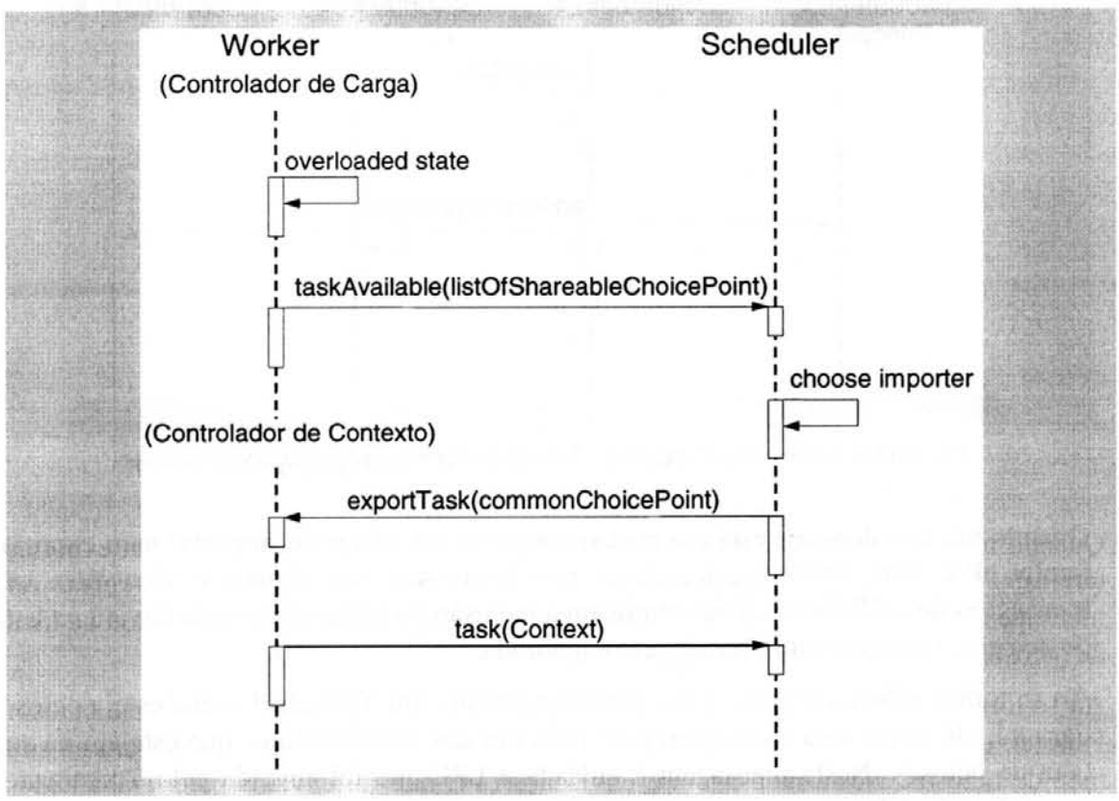

FIGURA 4.19 - Interface Trabalhador e Escalonador: trabalhador sobrecarregado.

Um Trabalhador sempre se torna ocupado com o conhecimento do Escalonador. Por isso, o Controlador de Carga não avisa ao Escalonador quando sofre uma transição para esse estado.

4.6.5 Interface entre Escalonadores

Os Escalonadores existentes nos diferentes trabalhadores precisam se comunicar para efetuar o escalonamento das tarefas. Para ilustrar o protocolo de comunicação, sempre serão mostrado cenários com quatro trabalhadores.

Conforme foi apresentado na seção 4.3, os escalonadores possuem uma lista com os nodos que estão ociosos. Para que essa lista se mantenha atualizada, sempre que um Escalonador se torna ocioso, ele envia uma mensagem para todos os demais Escalonadores (figura 4.20). Essa mensagem contém o peso que esse Escalonador possuía, informação que é utilizada pelo algoritmo de terminação. O Escalonador principal adiciona esse valor ao seu peso atual. Os Escalonadores ocupados ou sobrecarregados que recebem essa mensagem, adicionam o Escalonador a sua lista de ociosos já que podem vir a enviar mensagens para ele tentando exportar uma tarefa. Os Escalonadores ociosos não mantém a lista de ociosos, ignorando as mensagens *addIdle*.

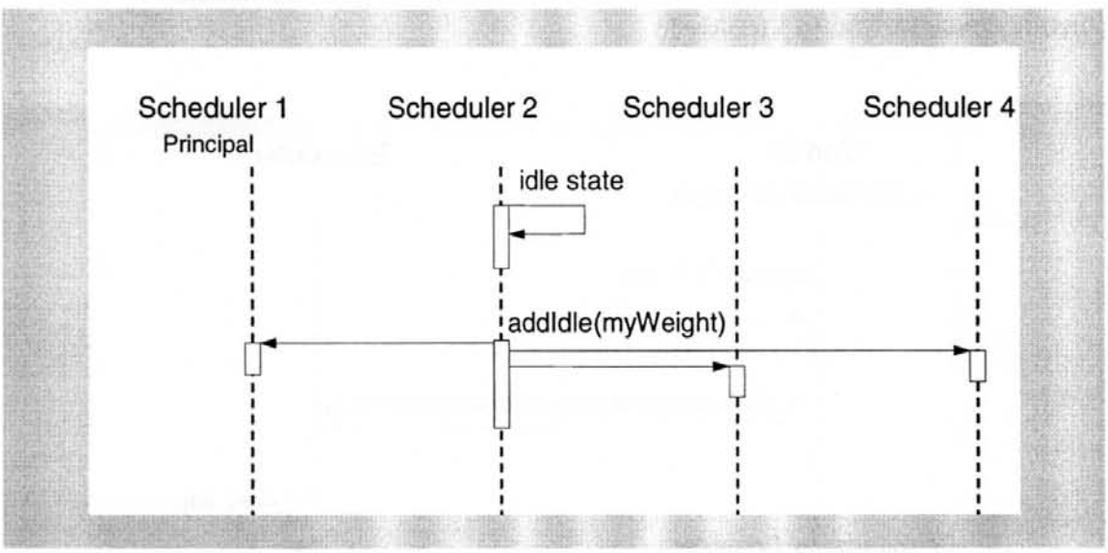

FIGURA 4.20 - Interface entre Escalonadores: propagação de ocioso.

Quando um Escalonador está em estado ocupado, ele não pode importar nem exportar tarefas. Por isso, esses Escalonadores não interessam aos demais e vice-versa. As mensagens de *addIdle* recebidas implicam a inclusão do Escalonador que enviou na lista de ociosos. Qualquer outra mensagem é ignorada.

Ao se tornar sobrecarregado, o Escalonador procura um Trabalhador que esteja ocioso. Para tal, ele envia uma mensagem para cada um dos Escalonadores que estejam na sua lista de ociosos. Nesta mensagem é enviada a LPE que foi enviada pelo Trabalhador para o Escalonador.

A lista pode estar desatualizada, conforme já foi discutido, e por isso trabalhadores em diferentes estados podem receber uma mensagem *taskAvailable*. Cada Escalonador que estiver realmente ocioso descobre qual o último ponto de escolha comum e envia como resposta para o Escalonador. Escalonadores ocupados ou sobrecarregados ignoram a mensagem.

O Escalonador exportador aguarda as mensagens de resposta até que uma das seguintes opções seja atendida:

- recebimento de um número máximo de mensagens, ou
- término do tempo máximo de espera (*timeout*).

Ambos os valores são parâmetros passados ao escalonador. O primeiro tem por objetivo evitar a espera demasiada por respostas permitindo um escalonamento mais ágil. O segundo parâmetro evita que um Escalonador espere indefinidamente por uma resposta, uma vez que pode ocorrer de vários trabalhadores cadastrados como ociosos estarem ocupados no momento da transação de exportação.

Uma vez de posse das respostas, o Escalonador escolhe o importador seguindo dois critérios. Em primeiro lugar o **número de pontos de escolha comuns**, pois quanto mais pontos em comum, menor o contexto a ser exportado. O segundo, como critério de desempate, a **ordem de resposta**, para manter a localidade e por considera-se que quem responde mais rápido também pode receber e executar a tarefa mais rapidamente.

O Escalonador escolhido recebe a tarefa enquanto os demais recebem uma mensagem cancelando a transação. Essa mensagem é enviada para evitar que esses trabalhadores esperem indefinidamente por uma tarefa que nunca será atendida.

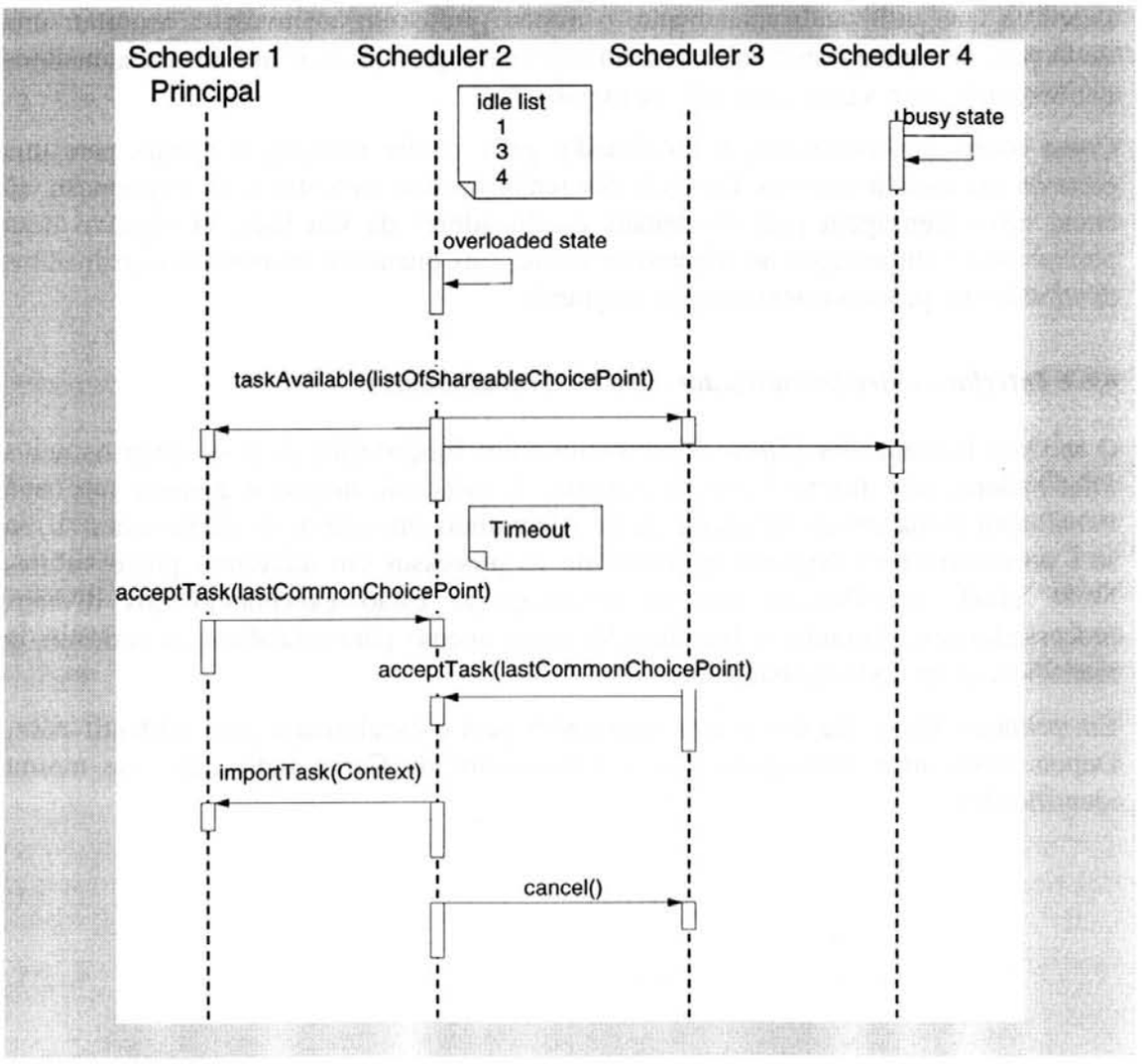

FIGURA 4.21 - Interface entre Escalonadores: exportação.

Caso após o tempo máximo (*timeout*) ter expirado, nenhum Escalonador responder, a transação de exportação falha e o Escalonador aguarda para iniciar nova transação até que: (1) um novo Escalonador informe estar ocioso; ou (2) alguma mensagem em atraso seja recebida.

Uma mensagem pode chegar em atraso por dois motivos: caso houver algum retardo no canal de comunicação para o envio da mensagem, ou caso o Escalonador estiver envolvido em uma outra transação de exportação no momento em que o Escalonador enviou o pedido. Um Escalonador ocioso quando recebe uma mensagem informando a disponibilidade de tarefa, responde imediatamente. Qualquer outra mensagem deste tipo que chegar enquanto a transação de exportação estiver em andamento será armazenada para ser tratada caso seja enviada uma mensagem cancelando a transação.

Para evitar problemas de sincronização, cada transação de exportação recebe um número único que é incluído na mensagem. Essa identificação serve principalmente para tratar mensagens atrasadas. Cada nova transação que é iniciada por um trabalhador, recebe como identificador um número inteiro que é maior em uma unidade que o último identificador utilizado.

Note-se que embora sempre tenha-se tratado da exportação de uma única tarefa, nada impediria que, utilizando exatamente o mesmo protocolo, ao invés de exportar uma tarefa para um único trabalhador, fossem exportadas para mais de um dos Escalonadores que respondessem a uma transação de exportação.

Como forma de otimização, o escalonador pode enviar mensagens apenas para uma parte da sua lista de ociosos. Caso ele não tenha sucesso na tentativa de exportação, ele envia nova mensagem para os demais escalonadores da sua lista. O objetivo deste parâmetro é a diminuição no número de mensagens quando o número de trabalhadores envolvidos no processamento for muito grande.

4.6.6 Interface entre Inicializador e Nodos Trabalhadores

O módulo Inicializador (*Starter*) tem como única função inicializar os diversos nodos trabalhadores que fazem parte do sistema. A principal função é garantir que cada trabalhador tenha um identificador único. Além disso, em termos de implementação, ele será necessário para disparar remotamente os processos em diferentes processadores. Neste nível, considera-se que os processos já estão executando nos diversos processadores e, portanto, o Inicializador serve apenas para inicializar as variáveis de identificação do nodo Trabalhador.

Em primeiro lugar, ele envia uma mensagem para o escalonador com o identificador. Depois envia uma mensagem para o Controlador de Carga indicando esse mesmo identificador.

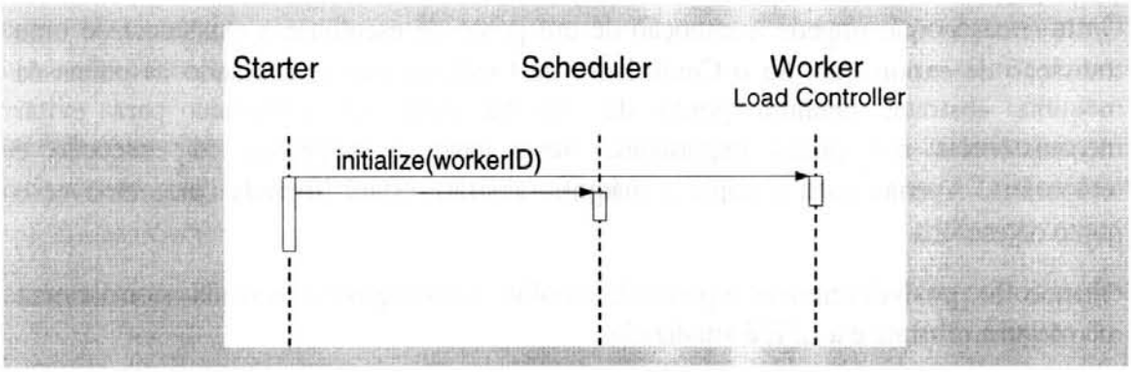

FIGURA 4.22 - Interface Inicializador e Trabalhador.

4.7 Alterações na Máquina Abstrata

Conforme analisado nas seções anteriores, as alterações na máquina abstrata foram minimizadas ao máximo possível. Nenhuma estrutura de dados precisa ser alterada, removida ou inserida na máquina abstrata. Apenas as instruções da classe escolha (*choice*), apresentadas no anexo 1, precisam ser alteradas para incluir chamadas ao módulo Controlador de Carga. A tabela 4.1 resume essas alterações.

Quando for criado um ponto de escolha através das instruções *try* ou *try_me_else*, elas devem acionar o módulo Controlador de Carga. Primeiro a máquina abstrata executa a instrução normalmente e depois ela aciona o Controlador de Carga informando o endereço do novo ponto de escolha criado localmente.

Para atualizar um ponto de escolha, isto é, remover uma alternativa, utiliza-se as instruções *retry* ou *retry_me_else* da máquina abstrata. Antes de consumir a alternativa, a máquina abstrata precisa consultar se o ponto de escolha ainda está disponível, isto é, se ele for do tipo exportado, todas as alternativas que estavam pendentes foram enviadas para um outro trabalhador e por isso não é possível prosseguir a execução e o trabalhador vai para o estado ocioso.

Caso a máquina abstrata estiver tentando fazer uma atualização durante a cópia das pilhas para uma exportação, ela será suspensa até o final da cópia evitando inconsistência no importador. Apenas ao final da cópia o Controlador de Carga permite que a máquina abstrata atualize um ponto de escolha.

Para remover um ponto de escolha através das instruções *trust* ou *trust_me*, a máquina abstrata deve primeiro consultar o Controlador de Carga para saber se pode ou não remover o ponto de escolha. Um ponto de escolha não pode ser removido em três situações.

Em primeiro lugar se um ponto de escolha foi exportado, a alternativa pendente no ponto de escolha está sendo executada em outro trabalhador e não existem alternativas pendentes acima deste ponto de escolha. Neste caso, o processamento neste trabalhador é suspenso e é considerado ocioso. Logo, o Controlador de Carga informa o Escalonador.

Em segundo lugar se o ponto de escolha for do tipo importado, ele foi criado por um outro trabalhador e portanto pode ser comum a um ou mais trabalhadores. Se não houverem mais pontos de escolha com alternativas pendentes, esse ponto de escolha não é removido para permitir a cópia incremental.

Outra situação que impede a remoção de um ponto de escolha é a existência de uma transação de exportação. Se o Controlador de Contexto estiver copiando as pilhas da máquina abstrata, nenhum ponto de escolha deve ser removido para evitar inconsistências nas pilhas exportadas. Neste caso, a suspensão da execução é temporária. Apenas após a cópia a máquina abstrata estará liberada para remover o ponto de escolha.

Quando for possível remover o ponto de escolha, a instrução é executada normalmente na máquina abstrata e a LPE é atualizada.

TABELA 4.1 - Instruções alteradas na máquina abstrata.

Instruções Alteradas	Novo Código com Interface para Controlador de Carga
try e *try_me_else*	`<código original>` **`addChoicePoint`**`(addressOfNewChoicePoint)`
Retry e *retry_me_else*	**`updateChoicePoint`**`(addressOfChoicePoint)` `<código original>`
trust e *trust_me*	**`removeChoicePoint`**`(addressOfChoicePoint)` `<código original>`

4.8 O Modelo pclp(FD) e Trabalhos Relacionados

Na seção 3.4, apresentou-se alguns sistemas paralelos para Prolog e CLP. Para a construção do modelo pclp(FD) observou-se as características e resultados obtidos por esses sistemas.

Notou-se que existem vários trabalhos recentes com memória distribuída, comprovando a importância deste tipo de ambiente. Seguindo a tendência dos sistemas para memória distribuída mais atuais, utiliza-se a cópia como técnica para exportação de contexto.

Outro ponto importante, é a existência de poucos sistemas que, como o pclp(FD) utilizem uma política de escalonamento distribuída. Um dos motivos é a eficiência da gerência centralizada para um número pequeno de processos, e a sua facilidade de implementação, principalmente em ambientes de memória compartilhada.

A tabela 4.2 apresenta uma extensão da tabela 3.1, com a inclusão do modelo pclp(FD).

TABELA 4.2 - Comparação entre pclp e outros sistemas em lógica paralelos.

Sistemas / Critérios	Programação em Lógica			Programação em Lógica com Restrições			
	MUSE	PLoSys	Aurora	ElipSys	Mudambi	PARCS	**pclp**
Paralelismo	OU	OU	OU	OU/E	OU	OU	**OU**
Técnica de Exportação OU							
cópia	√	√		√		√	√
compartilhamento			√				
recomputação					√		
Ambiente de Execução							
memória compartilhada	√		√				
memória distribuída		√		√	√	√	√
Escalonamento							
estático							
dinâmico	√	√	√	√	√	√	√
centralizado	√	√	√	√	√		
distribuído						√	√

4.9 Conclusão

Nesse capítulo definiu-se um modelo de exploração de paralelismo OU em CLP. Como principais características, cita-se a utilização de uma política de escalonamento distribuída e de um algoritmo de cópia incremental de contexto.

A política de escalonamento proposta pode ser otimizada mantendo-se as políticas de seleção e de transferência fixas. A execução destas duas políticas está diretamente ligada a interface escalonador/máquina abstrata. As políticas de informação e de localização são as responsáveis pela introdução da maior parte do custo de escalonamento, pois a troca de mensagens entre os escalonadores ocorre devido a essas duas políticas. Ambas podem ser alteradas sem interferir no funcionamento do restante do modelo.

O grande mérito do algoritmo de cópia incremental é propor uma forma de controle descentralizado dos nodos comuns.

Além disso, um algoritmo de terminação consolidado na literatura foi adaptado para o modelo facilitando a implementação.

O modelo prevê um mínimo de interferência na máquina abstrata, sendo poucas as alterações a serem realizadas diretamente na máquina abstrata. Isso tem como vantagem a facilidade para a substituição da máquina abstrata.

Outra característica do modelo é a possibilidade de execução da máquina abstrata de forma concorrente à cópia das pilhas. Durante a transação de exportação, isto é, enquanto é escolhido o importador mais adequado, a máquina abstrata pode continuar executando. No entanto, enquanto estiver sendo copiada as pilhas que formam o contexto de execução, qualquer tentativa de atualizar ou remover um ponto de escolha causará a interrupção temporária da máquina abstrata, servindo como uma sincronização.

No próximo capítulo será descrito um protótipo para o modelo pclp(FD).

5 Protótipo pclp(FD)

Neste capítulo apresenta-se o protótipo pclp(FD) que foi projetado com o objetivo de avaliar os principais conceitos do modelo proposto. Por se tratar de um protótipo, não foram implementadas todas as funcionalidades previstas no modelo. Além disso, otimizações de desempenho são limitadas devido a dois objetivos principais: portabilidade e facilidade para alterações.

Na seção 5.1 são apresentadas as duas escolhas de implementação mais importantes: a máquina abstrata e a linguagem de implementação. Na seção seguinte, é apresentado o projeto do protótipo.

5.1 Opções de Implementação

Existem duas questões vitais para a implementação de protótipos como o em questão. A primeira é a escolha da máquina abstrata a ser utilizada como base do nodo trabalhador. A segunda questão é a escolha da linguagem a ser utilizada na gerência do paralelismo.

As implementações atuais da linguagem Prolog utilizam a técnica de compilação para a WAM, isto é, um programa Prolog é traduzido para um programa equivalente para a WAM, que é executado por um emulador. Este emulador é implementado da forma mais eficiente possível com o objetivo de obter um bom tempo de execução com pouco gasto de memória. Essas otimizações podem ser tanto no funcionamento da máquina, isto é, extensões, quanto em utilização de particularidades de uma máquina específica através da escrita em código nativo (*assembly*). Normalmente, essas otimizações implicam uma complexidade no código que dificulta a manutenção e a inclusão de novas características.

Na seção 5.1.1 apresenta-se a opção pela máquina abstrata de clp(FD). Na seção 5.1.2 apresenta-se uma série de vantagens que levaram à escolha da linguagem Java.

5.1.1 Máquina Abstrata CLP: clp(FD)

Vários sistemas em lógica paralelos utilizam o SICSTus, que é um sistema comercial, como base do trabalhador. O SICSTus é eficiente por possuir uma série de otimizações, mas essas otimizações tornam difícil a tarefa de alterar, tanto o compilador quanto o emulador. Outros optaram por máquinas mais simples como o sistema PLoSys [MOR 97] que utiliza a wamcc como máquina abstrata e o YapOr [ROC 97] que utiliza a Yap.

Optou-se pela utilização da máquina abstrata da linguagem clp(FD) [COD 96] no protótipo pclp(FD) pelos seguintes motivos:

- o código fonte, bem documentado e estruturado, encontra-se disponível, facilitando a sua alteração;
- estende o compilador e o emulador *wamcc* [COD 95b], que é uma implementação eficiente da WAM, o que contribuirá para a eficiência do protótipo pclp;

- a linguagem implementa domínios finitos, permitindo a modelagem e a solução de problemas de satisfação de restrições discretas, possuindo várias aplicações práticas.

A linguagem clp(FD) é fortemente influenciada pelas linguagens CHIP e cc(FD). Como já foi dito, o sistema clp(FD) é uma extensão de wamcc para implementação de domínios finitos, sendo que a arquitetura e a estrutura de dados básicas da WAM não foram alteradas. O compilador wamcc traduz um programa WAM para um programa em linguagem C, com o objetivo de criar uma implementação simplificada, eficiente e portável da WAM de modo a facilitar a extensão para outros trabalhos.

5.1.2 Linguagem de Programação: Java

Grande parte dos sistemas paralelos atuais optam pela implementação com o uso da linguagem C associada a uma biblioteca de comunicação. Uma biblioteca de comunicação é composta por um conjunto de funções que permitem o envio e o recebimento de mensagens entre processos. As duas bibliotecas mais utilizadas são MPI (*Message Passing Interface*), que foi adotada como padrão pela ISO, e PVM (*Parallel Virtual Machine*), que é um padrão de fato.

Porém o uso dessas bibliotecas, torna a programação mais suscetível a erros. Isso se deve principalmente ao fato da implementação ser realizada em um baixo nível de abstração, manipulando tipos primitivos como *byte, float* e *integer.*

No protótipo pclp(FD), optou-se pela implementação da gerência de paralelismo na linguagem Java com o uso de acesso remoto a métodos (*Remote Method Invocation* ou RMI)

Java é uma linguagem moderna que realmente implementa os conceitos de OO. Outras linguagens comerciais como C++ e Object Pascal (disponível na ferramenta Borland Delphi) possuem apenas extensões para a utilização destes conceitos.

Orientação a objetos (OO) é um paradigma que permite um alto nível de abstração. Ele facilita a construção de protótipos por permitir o desenvolvimento através de refinamentos sucessivos. A manutenção de sistemas OO também é mais simples.

Java apresenta uma série de vantagens, como por exemplo: simplicidade sintática, facilidades para o uso de serviços de redes de computadores, portabilidade e mecanismo de *threads* inerente à linguagem [SUN 97]. Também é possível integrar sistemas já desenvolvidos em outras linguagens. Por exemplo, Java pode acessar funções C, da mesma forma que programas escritos C podem fazer uso de métodos escritos em Java. Isso é possível através de uma interface para sistemas nativos (*Java Native Interface* ou JNI), que faz parte da linguagem.

A comunicação por troca de mensagens pode ser feita via *sockets*, através da manipulação de um conjunto pré-definido de classes. Outra forma de interação entre objetos em processadores distintos é a invocação remota de métodos (*Remote Method Invocation* ou RMI), que é análoga à chamada remota de procedimentos (RPC) em linguagens imperativas.

A desvantagem desta linguagem é a baixa eficiência das implementações atuais. Essa ineficiência está associada em grande parte ao fato de Java ser uma linguagem interpretada. Essa ineficiência está sendo contornada pela implementação de

compiladores JIT (*Just-In-Time compilers*) [SUN 98], e por interpretadores bastante eficientes [ARM 98].

Java é uma linguagem dinâmica, que utiliza um código intermediário interpretado por uma máquina virtual. Existe a possibilidade de compilar esse código intermediário em código nativo, durante a execução. Por isso, o JIT é realmente útil por compilar, de forma integrada à máquina virtual, cada método antes da sua chamada. Existem vários artigos disponíveis na internet explicando o processo de compilação JIT, como por exemplo [MCM 96]. O artigo [MAN 98] apresenta uma comparação de tempos de execução de programas escritos em C++ e em Java com JIT, obtendo um desempenho similar nas duas linguagens.

5.2 Conceitos Básicos sobre Orientação a Objetos

Os dois principais conceitos de OO [FOW 87] relevantes para esse texto são:

- Classe: conjunto de objetos que compartilham o mesmo comportamento e a mesma estrutura;
- Objeto: entidade com estado, comportamento e identidade. A estrutura e comportamento de objetos similares são definidos na sua classe comum, isto é, um objeto é uma instância de uma classe. Um objeto representa um indivíduo, um item identificável, uma unidade ou uma entidade, real ou abstrata, com um papel bem definido no domínio do problema.

Foram explorados dois tipos de relacionamento:

- Especialização: mecanismo pelo qual uma classe, denominada subclasse ou classe descendente, herda métodos e atributos de uma classe denominada superclasse ou classe ancestral. A subclasse pode sobreescrever métodos ou criar novos;
- Associação: mecanismo pelo qual um objeto pode guardar referência para outro objeto, permitindo assim acesso aos métodos deste.

5.3 Projeto

Conforme o modelo apresentado no capítulo anterior, cada nodo trabalhador é dividido em módulos que possuem interfaces de comunicação bem definidas. O protótipo pclp(FD) realiza a implementação de cada um desses módulos e de todas as interfaces, assim como do protocolo de comunicação entre os escalonadores.

Nesta seção será apresentado o projeto do protótipo pclp(FD) utilizando técnicas de orientação a objetos. Cada módulo do trabalhador foi mapeado para uma ou mais classes.

A **figura 5.1** apresenta as classes que compõem um nodo trabalhador, quais sejam:

- *ChoicePoint:* representa a relação entre o endereço físico de um ponto de escolha, seu tipo e o seu identificador único;
- *ChoicePointTable:* implementa a lista de pontos de escolha (LPE), que guarda informações sobre os pontos de escolha, referenciando um conjunto de objetos *ChoicePoint*;

- *LoadController:* recebe informações sobre variação de carga diretamente da máquina abstrata. Na implementação, ele não é um objeto, mas um módulo C integrado com a máquina abstrata. *ChoicePoint, ChoicePointTable* e *LoadController* implementam o módulo Controlador de Carga;
- *clpFD:* executa as instruções WAM. Corresponde ao módulo CLP_AM (*CLP Abstract Machine*). Totalmente implementado em C, possui alterações para se comunicar com os módulos Controladores de Carga e de Contexto;
- *ContextController*: exclui e inclui tarefas na máquina abstrata, pois possui livre acesso às suas estruturas de dados, implementando o módulo Controlador de Contexto;
- *Scheduler*: coordena a importação e a exportação de tarefas, implementando o protocolo de comunicação entre os escalonadores;
- *SchedulerController*: é uma subclasse de *Scheduler* que tem a capacidade de detectar a terminação global do programa;
- *ImportTransaction:* envia e recebe mensagens relacionadas com a importação de tarefas. Objetos dessa classe são acionados por escalonadores em estado ocioso;
- *ExportTransaction:* envia e recebe mensagens relacionadas com a exportação de tarefas. Objetos dessa classe são acionados por escalonadores em estado sobrecarregado;
- *TimeoutController*: avisa quando *Scheduler* deve parar de esperar por respostas em uma transação de exportação. *Scheduler, SchedulerController, ImportTransaction, ExportTransaction* e *TimeoutController* formam o módulo Escalonador.

Um nodo trabalhador pode ser do tipo normal ou principal. O que diferencia um trabalhador de outro é o Escalonador. O Escalonador do trabalhador principal além de disparar o início do processamento, possui funcionalidades que detectam o término do programa.

O Escalonador dos trabalhadores normais é formado por objetos das classes *Scheduler, ImportTransaction, ExportTransaction* e *TimeoutController*. O Escalonador do trabalhador principal é formado por objetos das classes *SchedulerController,* subclasse de *Scheduler, ImportTransaction, ExportTransaction* e *TimeoutController.*

A classe *Scheduler* possui métodos para a implementação do protocolo de comunicação do algoritmo de terminação. Esses métodos envolvem o envio e recebimento de pesos associados à tarefas e a propagação do peso de uma tarefa finalizada.

A classe *SchedulerController* sobreescreve os métodos da classe *Scheduler*, adicionando a capacidade de detecção da terminação.

Observa-se que o *Scheduler* se relaciona somente com o *ChoicePointTable*, que faz parte do módulo Controlador de Carga, e com o *ContextController* que implementa o módulo Controlador de Contexto, obedecendo as interfaces apresentadas no modelo.

Na **figura 5.1** também existem outras três classes que integram o protótipo pclp(FD):

- *Starter*: efetua a instanciação remota de processos trabalhadores e controla as suas inicializações;

- *InitCLP*: inicializa os objetos que fazem parte da máquina abstrata;
- *SchedulerFactory*: cria escalonadores e inicializa as suas listas de ociosos.

Starter, InitCLP e *SchedulerFactory* compõem o módulo Inicializador.

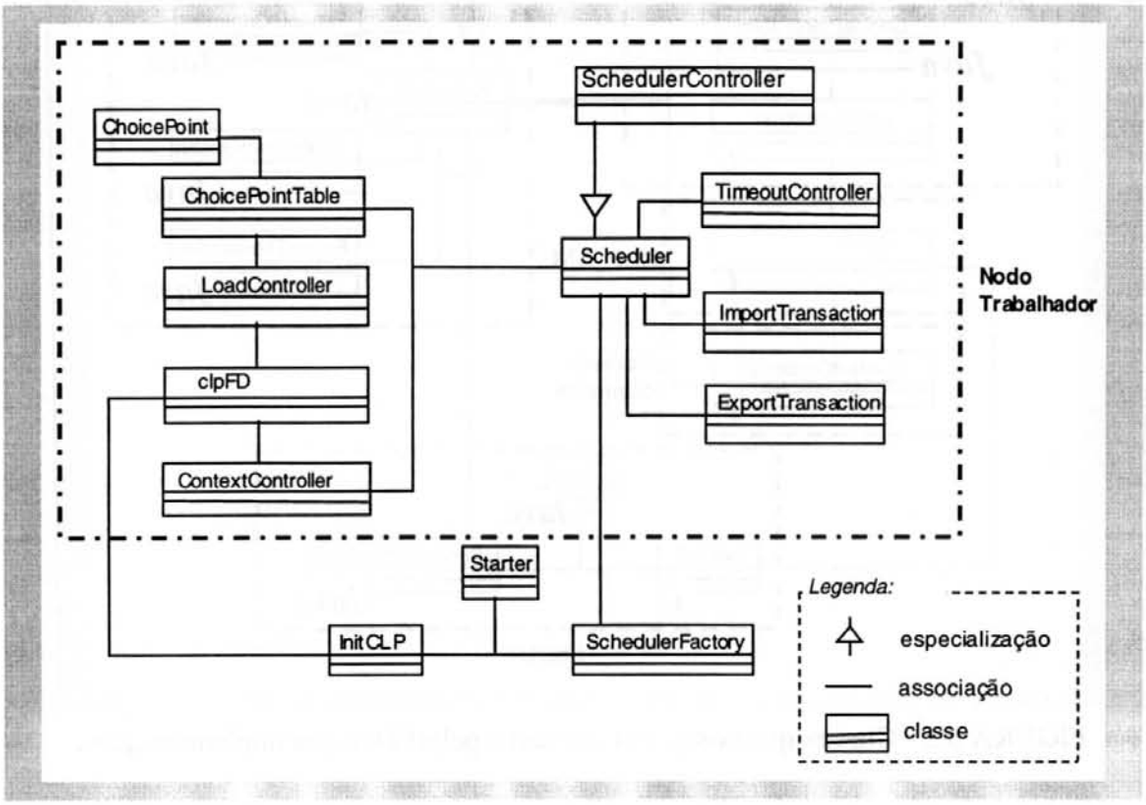

FIGURA 5.1 - Classes que compõem o sistema pclp(FD).

Como a máquina abstrata encontra-se implementada em C, existem partes do sistema que estão representadas como objetos, mas que, de fato, são implementadas como um conjunto de rotinas C. A figura 5.2 apresenta o mapeamento dos objetos para os módulos do modelo e indica a sua linguagem de implementação.

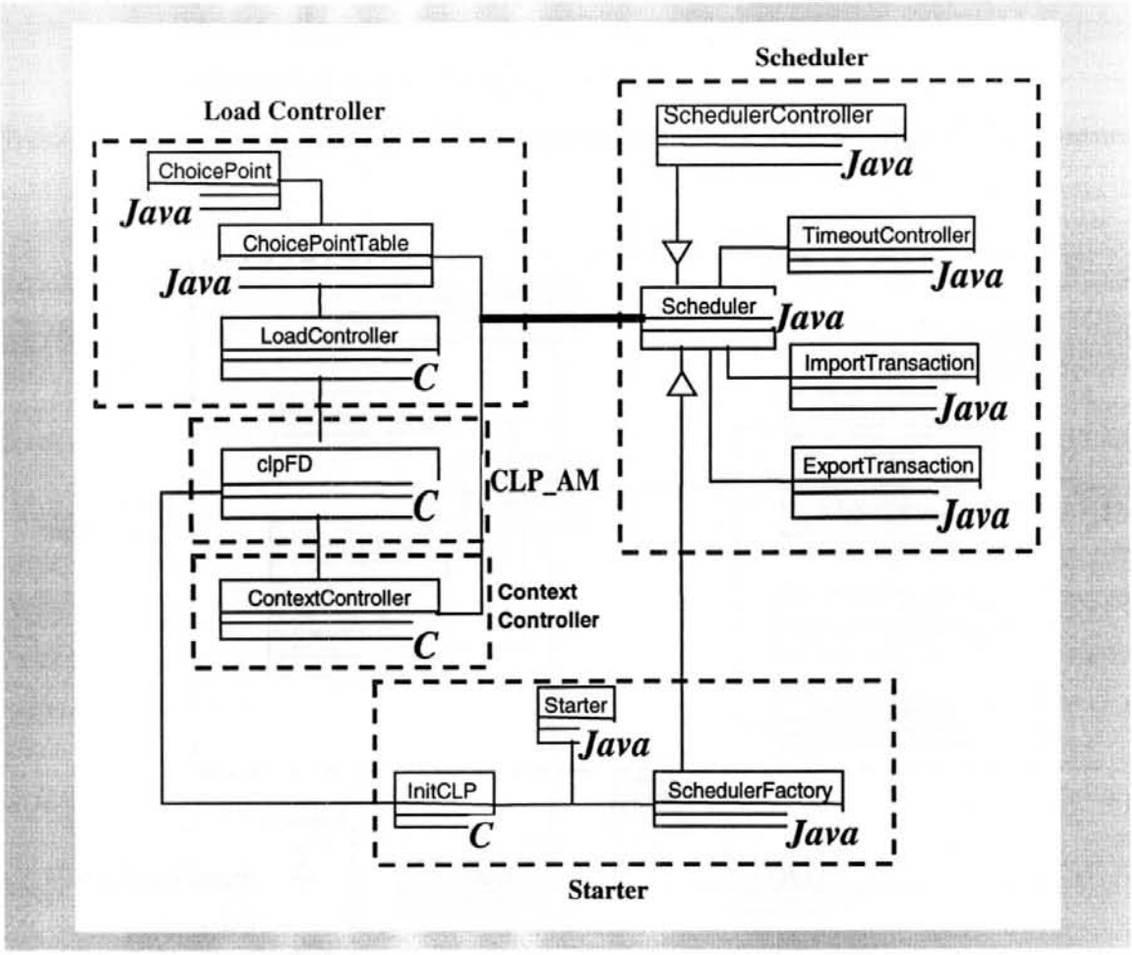

FIGURA 5.2 - Classes que compõem o sistema pclp(FD) e sua implementação.

A identificação única dos trabalhadores, exigida pelo modelo, é implementada como um número inteiro que corresponde à ordem de criação do trabalhador. O trabalhador principal sempre será o primeiro a ser instanciado e por isso recebe o identificador 1. O segundo trabalhador recebe o número 2 e assim sucessivamente. A atribuição dos identificadores únicos é responsabilidade do módulo Inicializador que é formado pelas classes *Starter*, *InitCLP* e *SchedulerFactory*.

5.4 Detalhes do Protótipo

5.4.1 Processo de Compilação

O sistema pclp(FD) efetua um processo de compilação dividido em três etapas.

Na primeira etapa um programa codificado em Prolog é traduzido para um conjunto de instruções WAM equivalentes, através do compilador disponível no sistema clp(FD). Essas instruções são agrupadas em um arquivo juntamente com as instruções de inicialização da máquina abstrata. Cada instrução da WAM corresponde a uma função ou macro disponibilizada pela biblioteca clp(FD). Deste modo, para cada aplicação Prolog, é criado um programa C.

FIGURA 5.3 – O processo de compilação: primeira etapa.

Na segunda etapa, o programa gerado é submetido a um pré-compilador que efetua alterações na função principal (*main*). O programa C, gerado originalmente pelo compilador clp(FD), possui as rotinas de inicialização, execução e finalização da WAM agrupadas na função *main*. O programa pré-compilado disponibiliza funções para a inicialização, execução e finalização da WAM, respectivamente:

- `InitializeWAM()`, chamada pelo Inicializador;
- `ExecuteWAM()`, chamada pelo Inicializador no trabalhador principal e pelo Controlador de Contexto após uma importação;
- `FinalizeWAM()`, chamada pelo Escalonador após a detecção da terminação global.

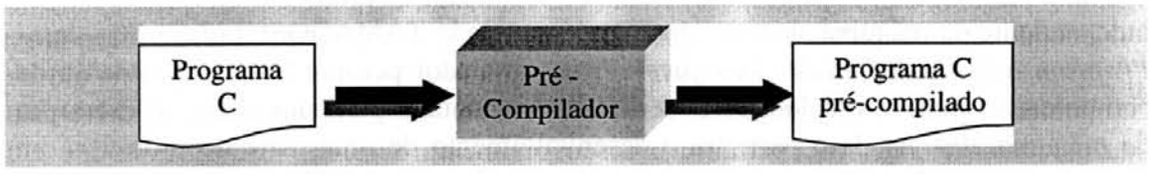

FIGURA 5.4 – O processo de compilação: segunda etapa.

O programa pré-compilado é submetido ao compilador C juntamente com a biblioteca clp(FD), gerando o executável. A biblioteca clp(FD) é formada por um conjunto de instruções que implementa a WAM.

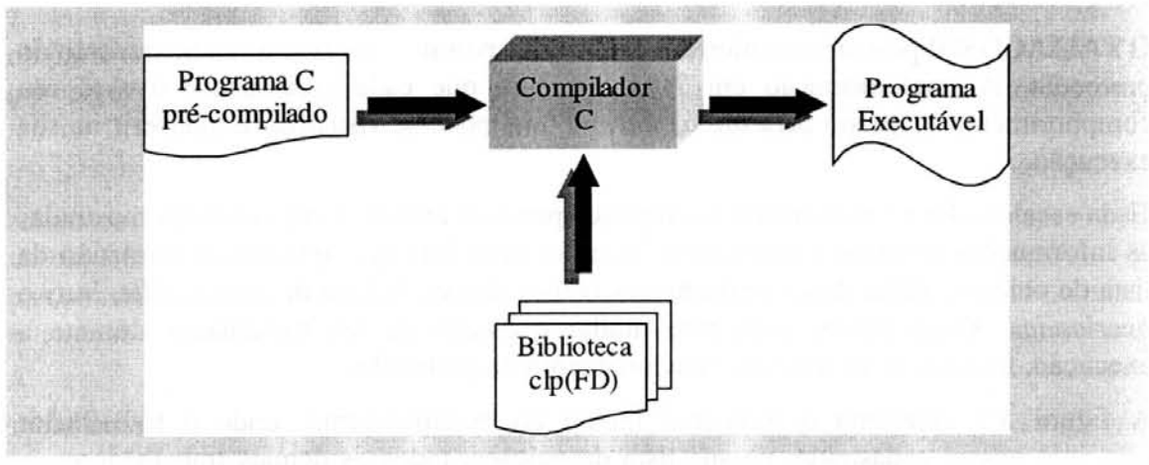

FIGURA 5.5 – O processo de compilação: terceira etapa.

5.4.2 Inicializador

Para que o protótipo possa executar em paralelo, é necessário que antes da execução propriamente dita os processos sejam instanciados remotamente.

Devido à comunicação entre Java e C existente no protótipo pclp(FD), utilizando a interface para métodos nativos (JNI), o Inicializador precisa iniciar como um processo Java. Isso se deve à limitação da implementação atual de JNI, que exige que o programa Java inicie o programa C para que a comunicação seja possível.

Outra característica da inicialização está intimamente relacionada à implementação de clp(FD). O Inicializador deve acionar, em cada trabalhador, a rotina de inicialização da máquina abstrata, que foi alterada pelo pré-compilador. Essa rotina tem por objetivo principal alocar memória para as pilhas de execução.

Outra característica importante do Inicializador é garantir a sincronização dos trabalhadores. A execução só é iniciada após todos os trabalhadores inicializarem as suas máquinas abstratas.

5.4.3 Escalonamento

Em uma fase inicial do trabalho foi criado um simulador de protocolo de escalonamento independente da máquina abstrata clp(FD) denominado TAMAGOSHI (***Task Simulator Program and OR-Scheduler Interface***). Esse simulador permite testar o protocolo de comunicação entre os escalonadores sem levar em conta as particularidades de execução da máquina abstrata. Isso permitiu o desenvolvimento de uma parte do protótipo em paralelo com as alterações necessárias da máquina abstrata clp(FD).

A máquina abstrata foi substituída por um simulador elementar, que apenas gera cargas aleatórias, sem basear-se em um padrão de execução de aplicações reais. Cada vez que um trabalhador importa uma tarefa, é criado um número aleatório de tarefas. Cada tarefa gerada possui um tempo de execução, gerado aleatoriamente, que causa não uma execução, mas um atraso na execução, correspondente ao tempo que levaria para resolver a tarefa.

O TAMAGOSHI possui uma interface gráfica que permite observar o funcionamento do protocolo. A implementação em Java, permitiu que cada escalonador tivesse seu comportamento mapeado para um *applet*, no qual pode-se visualizar e interferir na sua execução.

Cada escalonador é representado na interface por uma área de texto, onde são mostradas as informações relativas a sua execução, e por uma lista que apresenta o conteúdo da lista de ociosos. Além disso, existem três botões abaixo da lista de ocioso: *idle*, *busy* e *overloaded*. Esses botões permitem mudar o estado de um trabalhador durante a execução, facilitando os testes de funcionamento do protocolo.

A figura 5.6 apresenta o momento inicial do escalonamento, onde o trabalhador principal possui cadastrado, na sua lista de ociosos, todos os demais trabalhadores, e encontra-se em estado ocupado processando os primeiros pontos de escolha.

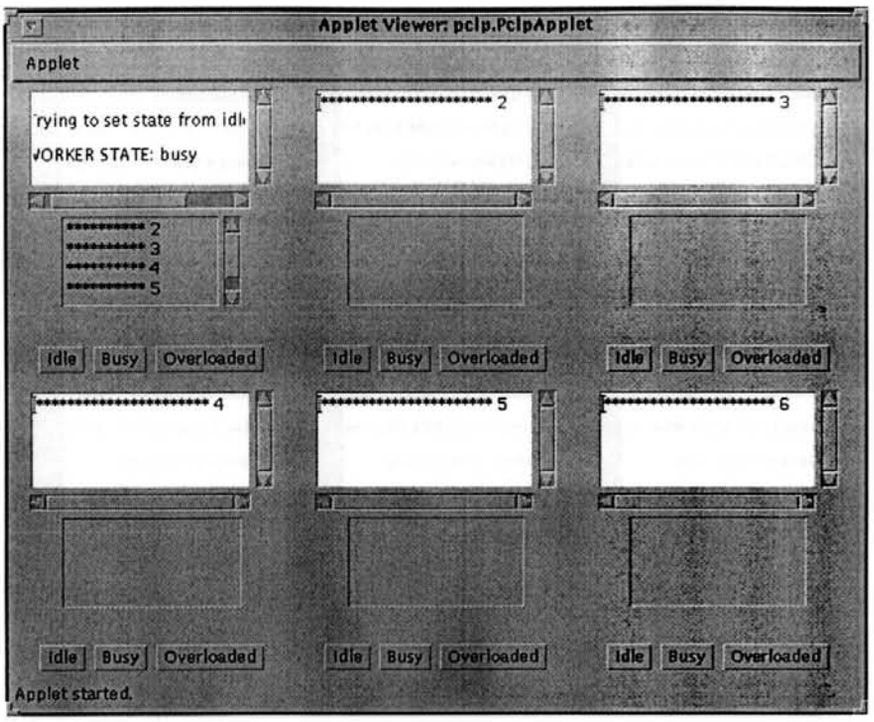

FIGURA 5.6 – Simulador de Protocolo de Execução: situação inicial.

Através da observação do comportamento do TAMAGOSHI, notou-se que o escalonamento consegue manter os trabalhadores ocupados a maior parte do tempo.

Notou-se também que a lista de ociosos tende a se tornar desatualizada no decorrer do processamento. É comum um trabalhador achar que um ou mais escalonadores ocupados estão ociosos, como pode-se observar na figura 5.7. Nesse instante de uma simulação, o trabalhador 1 considera que o trabalhador 5 está em estado ocioso quando na verdade ele se encontra em estado ocupado.

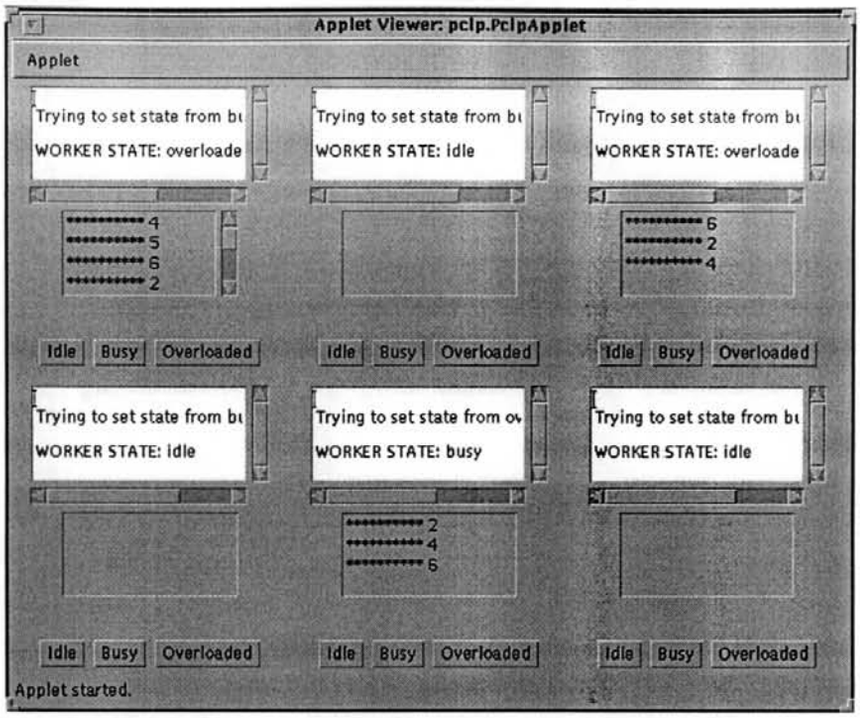

FIGURA 5.7 – Simulador de Protocolo de Execução: durante uma simulação.

Porém, em nenhuma das simulações, ocorreu a existência de trabalhadores sobrecarregados desconhecendo a existência de ociosos. Isso ocorre pois, sempre que um trabalhador vai para o estado ocioso, ele avisa a todos os demais trabalhadores da sua disponibilidade para importar tarefas.

5.4.4 Arquitetura de Processos

Cada nodo trabalhador é formado por um Escalonador e uma máquina abstrata, composta pelo Controlador de Carga, CLP_AM e Controlador de Contexto. O Escalonador é totalmente implementado em Java, por isso roda como um processo independente da máquina abstrata. CLP_AM e os Controladores de Carga e de Processo rodam como *threads* que se comunicam segundo as interfaces definidas no modelo (seção 4.6). É importante lembrar que uma parte do Controlador de Carga é implementado em Java, pertencendo ao mesmo processo Escalonador.

Um dos momentos críticos do processamento é a cópia das pilhas para a exportação de tarefas. O processo de exportação ocorre em paralelo com a execução normal do programa Prolog, por isso é preciso utilizar mecanismos de sincronização para evitar o envio de dados inconsistentes. Estudou-se duas alternativas para sincronizar a cópia com a execução.

Na primeira alternativa, no momento que o Controlador de Carga fosse informado do estado sobrecarregado do trabalhador, o Controlador de Contexto faria uma cópia local das pilhas. O Controlador de Carga deveria monitorar e detectar a tentativa de remoção do ponto de escolha que foi copiado para a exportação. Caso isso ocorresse, ele deveria cancelar a tentativa de exportação. A vantagem desse método é não utilizar semáforos para o controle do momento em que a cópia estiver sendo realmente feita, simplificando a implementação. Por outro lado, há localmente uma duplicação das pilhas, implicando

consumo de tempo e de memória, aguardando uma exportação que talvez não venha a ocorrer.

Na alternativa escolhida, a cópia das pilhas só ocorre após a determinação de um nodo trabalhador importador. Se a CLP_AM tentar remover ou atualizar um ponto de escolha durante a cópia, ele ficará bloqueado até o término da mesma. Durante a tentativa de encontrar um trabalhador importador, o escalonamento e a execução normal ocorrem de maneira concorrente. Para efetuar esse bloqueio, um semáforo deve ser obtido para a realização das operações de remoção e atualização de pontos de escolha no Controlador de Carga. Para efetuar a cópia das pilhas, esse mesmo semáforo também deve ser obtido, garantindo a consistência da cópia.

O módulo Inicializador é um processo a parte. Ele tem como objetivo, além de inicializar as estruturas de cada nodo trabalhador, garantir a sincronização inicial, isto é, a execução do programa só começa após a inicialização de todos os nodos trabalhadores.

5.4.5 Comunicação entre os Escalonadores

Em Java existem duas formas de comunicação entre objetos distribuídos: sockets e RMI. Sockets permite a comunicação via troca de mensagens, utilizando classes disponibilizadas na própria linguagem. RMI permite o acesso a métodos de objetos remotos de forma quase transparente.

[ORF 97] apresenta uma série de comparações entre RMI e Sockets. RMI fornece um nível maior de abstração, mas possui um desempenho inferior às implementações com sockets. Testes realizados na rede de estações Sun do II/UFRGS comprovaram o alto custo de envio de comunicação de RMI.

No protótipo pclp(FD) os escalonadores se comunicam via RMI, pois essa é a forma mais intuitiva e alto nível, facilitando tanto a implementação quanto a manutenção.

De qualquer modo, é possível mapear a implementação de RMI para sockets com o objetivo de melhorar o desempenho.

5.5 Conclusão

Tanto no modelo quanto na implementação, procurou-se minimizar ao estritamente necessário as alterações no sistema clp(FD). O principal objetivo desta estratégia é facilitar o porte do sistema paralelo para outros sistemas em lógica, assim como permitir que seja incorporado a novas versões do clp(FD) com um mínimo de esforço de programação.

6 Conclusão

Neste trabalho foi apresentada a proposta de um modelo de exploração de paralelismo OU em CLP para um ambiente de memória distribuída. CLP é uma área relativamente nova na comunidade científica e pouco explorada no Instituto de Informática da UFRGS. Esse trabalho permitiu a criação de novas frentes de pesquisa dentro do projeto OPERA, sem por isso perder a integração com os trabalhos desenvolvidos anteriormente.

Nos dois capítulos iniciais deste trabalho, apresentou-se alguns conceitos fundamentais da Programação em Lógica e de CLP, seguido de aspectos de paralelização de linguagens destes paradigmas.

A exploração de paralelismo implícito em CLP é bastante facilitada devido ao paralelismo inerente as linguagens que implementam o paradigma CLP. Uma das fontes é o OU, que se caracteriza pela execução paralela de diferentes cláusulas de um mesmo predicado. Esse tipo de paralelismo está presente, principalmente, em programas não determinísticos.

Existem vários trabalhos nesta área, tal como foi apresentado no Capítulo 3. Os dois trabalhos mais importantes em Prolog são o MUSE [ALI 90] e o Aurora [LUS 90], ambos implementados em ambientes de memória compartilhada. O principal aspecto que os distingue é a forma de exportação do contexto de execução em que uma tarefa está inserida, isto é, o conteúdo das pilhas de execução da máquina abstrata. Existem basicamente três formas de exportar o contexto:

- compartilhamento de pilhas: uma parte das pilhas é compartilhada enquanto outra parte é privada (usada para armazenar as diversas ligações condicionais às variáveis das partes compartilhadas);
- cópia de pilhas: o trabalhador importador copia as porções de pilha com o contexto;
- recomputação de pilhas: o trabalhador importador recebe um caminho pré-determinado da árvore de busca, o que permite a construção do contexto.

MUSE utiliza cópia enquanto Aurora utiliza compartilhamento.

Em CLP também existem vários trabalhos, sendo vários deles inspirados nos princípios do MUSE. Quase todos os sistemas paralelizam linguagens sobre domínios finitos. Existem dois principais motivos para essa escolha. Em primeiro lugar, esse domínio pode ser utilizado para resolver uma série de aplicações reais. Além disso, a exploração de paralelismo OU nessas linguagens é bastante similar à Prolog.

O modelo pclp(FD) de exploração de paralelismo, apresentado neste trabalho, foi proposto com o objetivo inicial de explorar paralelismo OU especificamente em CLP. No entanto, esse modelo mostra-se adequado tanto para linguagens CLP baseadas na WAM, quanto para Prolog.

Uma das decisões anteriores à concepção do modelo, foi o uso de ambientes com memória distribuída, pois esse é o ambiente no qual o projeto OPERA desenvolve os seus trabalhos. Essa decisão também deve-se ao fato das redes de computadores, que formam atualmente a principal arquitetura paralela na maior parte das instituições, possuírem memória distribuída. A principal vantagem das redes de computadores é o seu baixo custo em comparação com máquinas paralelas convencionais, além de ser de uso genérico.

Memória distribuída implica a ausência de variáveis globais para controle de paralelismo, por isso, várias soluções existentes em memória compartilhada não podem ser aplicadas. O uso de um sistema com memória compartilhada distribuída (*Distributed Shared Memory* ou DSM) [PRJ 98] resolveria essa limitação pois, mesmo utilizando um ambiente fisicamente distribuído, o programa pode considerar a existência de memória compartilhada. A utilização de DSM em sistemas paralelos em lógica exige um estudo mais aprofundado, fugindo ao escopo deste trabalho.

Uma das contribuições mais importantes deste modelo é o detalhamento do mecanismo de cópia incremental. Embora alguns autores proponham essa técnica no modelo [GEY 92] [BEN 93] ou no modelo e na implementação [ALI 90], não se tem conhecimento de trabalhos detalhando o algoritmo de cópia incremental de forma independente do escalonamento.

Os sistemas que implementam a cópia incremental, em ambiente de memória compartilhada ou distribuída, são baseados na existência de informações globais, como por exemplo em um escalonador centralizado. Um escalonador centralizado, por ser responsável pelo controle dos pontos de escolha exportados, possui a informação completa sobre os compartilhamentos de pontos de escolha. Os trabalhadores importadores podem ser escolhidos com base no menor número de pontos de escolha a serem copiados, utilizando as informações centralizadas no escalonador. Entretanto, em um escalonamento distribuído, essa informação global não está disponível. Também não existem variáveis compartilhadas que possam guardar informações sobre os pontos de escolha comum em um ambiente com memória distribuída.

No algoritmo proposto, a cópia incremental pode ser realizada independente do tipo de escalonamento adotado. Cada trabalhador possui a informação de quais pontos de escolha podem ser comuns a outros trabalhadores. Deste modo, a decisão de escolha do importador pode levar em conta os pontos de escolha comuns, através da comparação das informações locais dos trabalhadores.

Um ponto importante é a possibilidade de exportar mais de um ponto de escolha. Seria possível exportar mais de um ponto de escolha para aumentar a granulosidade das tarefas. É importante ressaltar que neste caso as considerações feitas para o escalonamento e cópia incremental continuam válidas. Mas no caso específico da cópia incremental, o ponto de escolha comum seria o ponto de escolha importado mais antigo, pois é o único que pode-se garantir que esteja explorando o mesmo caminho.

Nesse modelo também está proposta uma política de escalonamento distribuída. Uma vantagem deste algoritmo é a simplicidade do seu protocolo de comunicação. Outra vantagem, é a inexistência da exigência de uma topologia específica de interconexão entre os trabalhadores, como por exemplo um anel ou árvore. Isso é adequado para a implementação em uma rede de computadores conectados por barramento.

Outra contribuição importante é a definição de uma arquitetura de processos onde a interferência na máquina abstrata é mínima. Um modelo implementado com essa arquitetura possui uma independência da máquina abstrata que facilita a portabilidade para diferentes máquinas abstratas, isto é, linguagens de programação em lógica.

Para a implementação do protótipo, optou-se pela utilização da linguagem clp(FD). Essa linguagem, além de atender a premissa de ter a máquina abstrata baseada na WAM, implementa domínios finitos que permitem a construção de uma série de aplicações reais. Outra característica importante dessa linguagem é a sua eficiência ao mesmo tempo em que não possui muitas otimizações na máquina abstrata, pois tais otimizações dificultariam alterações.

Como trabalhos futuros destaca-se a necessidade de depuração e avaliação do protótipo do modelo. Além disso, existem várias pesquisas que poderiam ser feitas a partir do modelo pclp(FD).

A política de escalonamento distribuída proposta poderia ser avaliada em relação a outras propostas a fim de verificar as suas vantagens e desvantagens. Uma forma de avalia-lo em um sistema real, seria a implementação deste escalonador no sistema PLoSys [MOR 97], que possui um escalonador centralizado. A comparação de vários parâmetros entre esses dois escalonadores seria um bom subsídio para futuras implementações de sistemas paralelos.

Outro ponto que poderia ser avaliado é o impacto do aumento das pilhas da máquina abstrata CLP em relação à WAM original. A opção por cópia poderia ser avaliada de forma experimental através de duas alternativas:

- Implementação do modelo em um ambiente Prolog, como, por exemplo, a wamcc. Isso permitiria comparar as diferenças na execução de programas Prolog e CLP, e que características melhoram o desempenho. Além disso, o impacto no aumento das pilhas poderia ser avaliado pela comparação nos tamanhos das mensagens enviadas;
- Implementação, na mesma linguagem CLP, de um modelo utilizando recomputação para permitir a comparação de desempenho a nível de gasto de memória, de troca de mensagens, de tamanhos de mensagens e de tempo de execução. Com isso poderia-se avaliar qual a melhor técnica para ambientes de memória distribuída.

Outra pesquisa importante é a análise de granulosidade em programas CLP. Utilizando uma extensão do modelo GRANLOG [BAR 95], poderiam ser obtidas informações de granulosidade que poderiam ser utilizadas no auxílio a decisões de escalonamento.

Anexo 1 Máquina Abstrata de Warren

A Máquina Abstrata de Warren (WAM) é atualmente o padrão de fato para a implementação de Prolog. Por isso, a maioria das implementações paralelas de Prolog tem como base a paralelização de versões da WAM.

O objetivo deste anexo é apresentar os principais aspectos da WAM, que são necessários para a compreensão de vários aspectos do modelo de paralelismo proposto neste trabalho.

Histórico

As primeiras implementações de Prolog possuíam um desempenho não satisfatório com relação às linguagens imperativas. No entanto, esse quadro mudou com a introdução de técnicas de compilação eficientes, propostas por David H. D. Warren em 1983. Essas técnicas levaram a uma estratégia padrão de implementação que utiliza a máquina abstrata de Warren, conhecida como WAM (*Warren Abstract Machine*) [AIT 91].

A WAM é uma estratégia de implementação na qual um programa Prolog é compilado em uma linguagem para uma máquina abstrata baseada em pilhas. Normalmente, a implementação da WAM possui dois componentes: um compilador e um emulador. O compilador é responsável por transformar os programas Prolog em instruções WAM, enquanto o emulador é responsável pela execução das instruções geradas.

Estrutura Básica

A WAM é normalmente definida através de uma organização de memória, estruturas de dados, registradores e instruções.

A estrutura básica da WAM contém um conjunto de três pilhas:

- LOCAL (*Local Stack*): é usada para o controle da chamada de predicados e da indexação de cláusulas. A Local contém dois tipos de elementos:
  - ⇒ Ambiente (*environment*): corresponde a uma estrutura de ativação de procedimento, que contém as variáveis locais e um ponteiro para a continuação da resolvente;
  - ⇒ Ponto de Escolha ou Nodo OU (*choice point*): contém o estado da máquina quando da chamada de um procedimento não determinista, isto é, com mais de uma cláusula a ser avaliada.
- GLOBAL (*Heap*): esta pilha armazena os termos estruturados (estruturas e listas) criados dinamicamente e as variáveis globais.
- TRAIL: contém ponteiros para as variáveis, cujas ligações precisam ser desfeitas quando ocorre retrocesso, para que tais variáveis voltem ao estado de não instanciadas. Com isso é possível desfazer os efeitos da execução de uma cláusula a ser descartada.

Ainda existem duas áreas importantes. A PUSH DOWN LIST, que é usada para operações de unificação e a ÁREA DE CÓDIGO (*Code Area*), que armazena o código compilado. Todas essas pilhas e áreas encontram-se representadas na figura a.1.

Existe ainda um conjunto de registradores, utilizados principalmente na manipulação das pilhas.

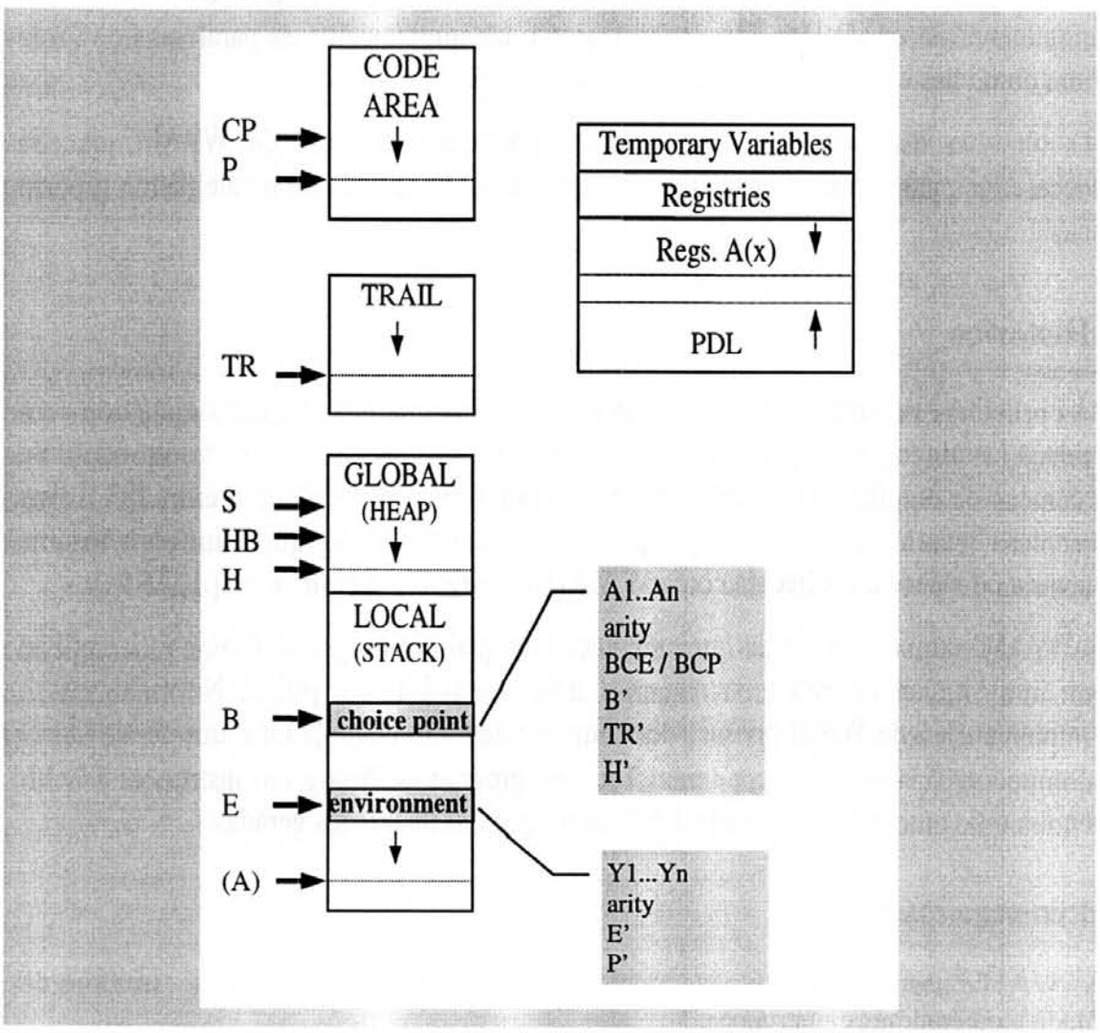

FIGURA A.1 - Pilhas e registradores da WAM

Conjunto de Instruções

As instruções da WAM podem ser divididas em seis grupos [YAM 94][WER 94]:

- *put*: manipula os argumentos do corpo da cláusula;
- *get*: manipula os argumentos da cabeça da cláusula;
- *unify*: corresponde aos argumentos de listas e estruturas;
- *control*: controla as chamadas e retornos de objetivos (procedimentos);
- *choice*: manipula os pontos de escolha;
- *indexing*: utilizado na indexação de cláusulas.

Esse conjunto de instruções encontra-se detalhado em [AIT 91].

As instruções do tipo *choice* serão analisadas pois a exploração de paralelismo OU está diretamente relacionada a elas.

Um ponto de escolha (*choice point*), também chamado de nodo OU, marca a existência de outras alternativas na árvore de busca a serem testadas, isto é, cláusulas adicionais de um predicado. Existem basicamente três operações que podem ser feitas em ponto de escolha:

- a criação de um ponto de escolha: *try* e *try_me_else*
- a atualização do ponto de escolha: *retry* e *retry_me_else*
- a sua remoção: *trust* e *trust_me*.

Note-se que por atualização considera-se o consumo de uma alternativa pendente. Um ponto de escolha é removido quando for consumida a última alternativa.

Anexo 2 Programas utilizados como teste

Queens 8

Código sem Restrições

```
:- main.

q:-
                write('N ?'), read_integer(N),
                statistics(runtime,_),
                queens(N, Qs), statistics(runtime,[_,Y]),
                write(Qs), nl,
                write('time : '), write(Y), nl,
                halt_or_else(0,true).

queens(N,Qs):-
                range(1,N,Ns),
                queens(Ns,[],Qs).

queens([],Qs,Qs).

queens(UnplacedQs,SafeQs,Qs):-
                sel(UnplacedQs,UnplacedQs1,Q),
                not_attack(SafeQs,Q),
                queens(UnplacedQs1,[Q|SafeQs],Qs).

not_attack(Xs,X):-
                not_attack(Xs,X,1).

not_attack([],_,_).

not_attack([Y|Ys],X,N):-
                X1 is Y+N, X =\= X1,
                X2 is Y-N, X =\= X2,
                N1 is N+1,
                not_attack(Ys,X,N1).

sel([X|Xs],Xs,X).
sel([Y|Ys],[Y|Zs],X):-
                sel(Ys,Zs,X).

range(N,N,[N]).
range(M,N,[M|Ns]):-
                M < N,
                M1 is M+1,
                range(M1,N,Ns).

:- q.
```

Código com Restrições

```
:- main.

q:-
                get_labeling(Lab), write('N ?'), read_integer(N),
                statistics(runtime,_),
                queens(N,L,Lab), statistics(runtime,[_,Y]),
                write(L), nl,
                write('time : '), write(Y), nl,
```

```
                halt_or_else(0,true).

queens(N,L,Lab):-
                fd_vector_max(N),
                length(L,N),
                domain(L,1,N),
                safe(L),
                lab(Lab,L).

safe([]).
safe([X|L]):-
                noattack(L,X,1),
                safe(L).

noattack([],_,_).

/*
% faster than the original PVH's version

noattack([Y|L],X,I):-
                I1 is I+1,
                noattack(L,X,I1),
                diff(X,Y,I).
*/

noattack([Y|L],X,I):-
                diff(X,Y,I),
                I1 is I+1,
                noattack(L,X,I1).

/*
diff(X,Y,I):-
                X in -({val(Y)}:{val(Y)-I}:{val(Y)+I}),
                Y in -({val(X)}:{val(X)-I}:{val(X)+I}).
*/

diff(X,Y,I):-
                X in -{val(Y)} & -{val(Y)-I} & -{val(Y)+I},
                Y in -{val(X)} & -{val(X)-I} & -{val(X)+I}.

/*
diff(X,Y,I):-
                X in 'C_diff'(val(Y),I),
                Y in 'C_diff'(val(X),I).
*/

/*
diff(X,Y,I):-
                X in (0..val(Y)-1   : val(Y)+1  ..infinity) &
                     (0..val(Y)-I-1 : val(Y)-I+1..infinity) &
                     (0..val(Y)+I-1 : val(Y)+I+1..infinity),

                Y in (0..val(X)-1   : val(X)+1  ..infinity) &
                     (0..val(X)-I-1 : val(X)-I+1..infinity) &
                     (0..val(X)+I-1 : val(X)+I+1..infinity).
*/

/*
diff(X,Y,I):-
                diff1(X,Y),
                diff2(X,Y,I),
                diff2(Y,X,I).

diff1(X,Y):-
                X in 0..val(Y)-1 : val(Y)+1..infinity,
                Y in 0..val(X)-1 : val(X)+1..infinity.

diff2(X,Y,I):-
                X in 0..val(Y)-I-1 : val(Y)-I+1..infinity,
                Y in 0..val(X)+I-1 : val(X)+I+1..infinity.
*/
```

```
/*
diff(X,Y,I):-
                X in (0..max(Y)-1   : min(Y)+1  ..infinity) &
                     (0..max(Y)-I-1 : min(Y)-I+1..infinity) &
                     (0..max(Y)+I-1 : min(Y)+I+1..infinity),
                Y in (0..max(X)-1   : min(X)+1  ..infinity) &
                     (0..max(X)-I-1 : min(X)-I+1..infinity) &
                     (0..max(X)+I-1 : min(X)+I+1..infinity).
*/

/*
diff(X,Y,I):-
                'x<>y'(X,Y),
                'x<>y+c'(X,Y,I),
                'x<>y+c'(Y,X,I).

'x<>y+c'(X,Y,C):-
                X in -{val(Y)+C},
                Y in -{val(X)-C}.
*/

lab(normal,L):-
                labeling(L).
lab(ff,L):-
                labelingff(L).

get_labeling(Lab):-
                argc(C),
                get_labeling1(C,Lab).

get_labeling1(1,normal).
get_labeling1(2,Lab):-
                argv(1,Lab).

:- q.
```

Send+More = Money

Código sem Restrições

```
% Cryptoaddition:
% Find the unique answer to:
%  SEND
% +MORE
% -----
% MONEY
% where each letter is a distinct digit.

:- main.

q :-
                statistics(runtime,_), send,
                statistics(runtime,[_,Y]),
                write('time : '),write(Y), nl,
                halt_or_else(0,true).

send :-
                digit(D), digit(E), D=\=E,
                sumdigit(0, D, E, Y, C1),
                digit(N), N=\=Y, N=\=E, N=\=D,
                digit(R), R=\=N, R=\=Y, R=\=E, R=\=D,
                sumdigit(C1,N, R, E, C2),
                digit(O), O=\=R, O=\=N, O=\=Y, O=\=E, O=\=D,
                sumdigit(C2,E, O, N, C3),
                leftdigit(S), S=\=O, S=\=R, S=\=N, S=\=Y, S=\=E,
```

```
                S=\=D,
            leftdigit(M), M=\=S, M=\=O, M=\=R, M=\=N, M=\=Y,
                M=\=E, M=\=D,
            sumdigit(C3,S, M, O, M),
            write(' '),write(S),write(E),write(N),write(D),nl,
            write('+'),write(M),write(O),write(R),write(E),nl,
            write('-----'),nl,
            write(M),write(O),write(N),write(E),write(Y),nl,nl,
            fail.
send.

sumdigit(C, A, B, S, D) :-
            X is (C+A+B),
            (X<10
            -> S=X,         D=0
            ;  S is X-10,   D=1
            ).

digit(0).
digit(1).
digit(2).
digit(3).
digit(4).
digit(5).
digit(6).
digit(7).
digit(8).
digit(9).

leftdigit(1).
leftdigit(2).
leftdigit(3).
leftdigit(4).
leftdigit(5).
leftdigit(6).
leftdigit(7).
leftdigit(8).
leftdigit(9).

:- q.
```

Código com Restrições

```
/*-------------------------------------------------------------------------*/
/* Benchmark (Finite Domain)          INRIA Rocquencourt - ChLoE Project */
/*                                                                         */
/* Name           : send.pl                                                */
/* Title          : crypt-arithmetic                                       */
/* Original Source: P. Van Hentenryck's book                               */
/* Adapted by     : Daniel Diaz - INRIA France                             */
/* Date           : September 1992                                         */
/*                                                                         */
/* Solve the operation:                                                    */
/*                                                                         */
/*      S E N D                                                            */
/*  +   M O R E                                                            */
/*  -----------                                                            */
/*  = M O N E Y                                                            */
/*                                                                         */
/* (resolution by line)                                                    */
/*                                                                         */
/* Solution:                                                               */
/*  [S,E,N,D,M,O,R,Y]                                                      */
/*  [9,5,6,7,1,0,8,2]                                                      */
/*-------------------------------------------------------------------------*/

:- main.

q:-
            get_labeling(Lab), statistics(runtime,_),
            send(LD,Lab), statistics(runtime,[_,Y]),
            write(LD), nl,
```

```
                write('time : '), write(Y), nl,
                halt_or_else(0,true).

send(LD,Lab):-
                LD=[S,E,N,D,M,O,R,Y],
                alldifferent(LD),
                domain(LD,0,9),
                S in 1..9,
                M in 1..9,

                   1000*S+100*E+10*N+D + 1000*M+100*O+10*R+E
                #= 10000*M+1000*O+100*N+10*E+Y,

                lab(Lab,LD).

lab(normal,L):-
                labeling(L).
lab(ff,L):-
                labelingff(L).

get_labeling(Lab):-
                argc(C),
                get_labeling1(C,Lab).

get_labeling1(1,normal).
get_labeling1(2,Lab):-
                argv(1,Lab).

:- q.
```

Digit 8

Código com Restrições

```
/*------------------------------------------------------------------------*/
/* Benchmark (Finite Domain)          INRIA Rocquencourt - ChLoE Project */
/*                                                                        */
/* Name           : digit8.pl                                             */
/* Title          : particular 8 digit number                             */
/* Original Source: Daniel Diaz - INRIA France                            */
/* Adapted by     :                                                       */
/* Date           : October 1993                                          */
/*                                                                        */
/* Find the 8 digit number N such that:                                   */
/*                                                                        */
/*   - N is a square                                                      */
/*   - if we put a 1 in front of the decimal notation of N then it still  */
/*     is a square                                                        */
/*                                                                        */
/* Solution:                                                              */
/*  [N,X,M,Y]                                                             */
/*  [23765625,4875,123765625,11125]                                       */
/*  [56250000,7500,156250000,12500]                                       */
/*------------------------------------------------------------------------*/

:- main.

q:-
                get_labeling(Lab),
                statistics(runtime,_),
                (digit8(L,Lab),
                write(L), nl,
                fail
                ;
                write('No more solutions'), nl),
                statistics(runtime,[_,Y]),
                write('time : '), write(Y), nl,
                halt_or_else(0,true).

digit8(L,Lab):-
                L=[N,X,M,Y],
```

```
                N in 10000000..99999999,
                'xx=y'(X,N),
                100000000+N #= M,
                'xx=y'(Y,M),
                lab(Lab,L).

lab(normal,L):-
        labeling(L).
lab(ff,L):-
        labelingff(L).

get_labeling(Lab):-
        argc(C),
        get_labeling1(C,Lab).

get_labeling1(1,normal).
get_labeling1(2,Lab):-
        argv(1,Lab).

:- q.
```

Bibliografia

[AIT 91] AIT-KACI, Hassan. **The WAM:** A (Real) Tutorial. Paris: Digital Equipment Corporation, Research Laboratory, 1991. 114p.

[ALI 90] ALI, Khayri A. M.; KARLSSON, Roland. The MUSE Approach to Or-Parallel Prolog. **International Journal of Parallel Programming**, Belgium, v.19, n.2, p.129-162, April 1990.

[ARM 98] ARMSTRONG, Erick. **HotSpot: A new breed of virtual machine**: Sun's Next-generation Dynamic Compiler Generates Bytecodes that Scream. 1998. Disponível em html em http://www.javaworld.com/javaworld/jw-03-1998/jw-03-hotspot.html.

[BAR 95] BARBOSA, Jorge Luís Victória. GRANLOG: Um Modelo para Análise Automática de Granulosidade na Programação em Lógica. In: SIMPÓSIO BRASILEIRO DE ARQUITETURAS DE COMPUTADORES E PROCESSAMENTO DE ALTO DESEMPENHO, 1995, Canela-RS. **Anais...** Porto Alegre: II/UFRGS, 1995. p.61-75.

[BAV 93] BARBOSA, Valmir. **Massively Parallel Models of Computation:** Distributed Parallel Processing in Artificial Intelligence and Optimization. Ellis Horwood: Great Britain, 1993. 253p.

[BEN 93] BENJUMEA, V.; TROYA, J. M. An OR Parallel Prolog Model for Distributed Memory Systems. In: INTERNATIONAL SYMPOSIUM ON PROGRAMMING LANGUAGE IMPLEMENTATION AND LOGIC PROGRAMMING, 5., August 1993 Tallinn, Estonia. **Proceedings...** Germany: Springer Verlag, 1993. p.291-301.

[CAL 96] CASTRO, Luís Fernando Pias de. Um Modelo de Analisador Estático Baseado na Interpretação Abstrata Direcionado à Paralelização de Programas em Lógica. In: SEMANA ACADÊMICA DA COMPUTAÇÃO, 1., 1996, Porto Alegre. **Anais...** Porto Alegre: CPGCC/UFRGS, 1996. p.291-294.

[CAS 88] CASAVANT, Thomas L.; KUHL, Jon G. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. **IEEE Transactions on Software Engineering**, New York, v.14, n.2, p. 141-154, February 1988.

[CLO 87] CLOCKSIN, Willian F. Principles of the DelPhi Parallel Inference Machine. **The Computer Journal,** London, v.30, n.5, p.386-392, October 1987.

[COD 95a] CODOGNET, Philippe. Programmation Logic avec Constraints: une Introduction. **RAIRO Technique et Science Informatiques**, [S.l.], v.14, n.6, 1995.

[COD 95b] CODOGNET, Philippe; DIAZ, Daniel. wamcc: Compiling Prolog to C. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 12., Tokyo, Japan. **Proceedings...** New York: MIT Press, 1995. p.317-331.

[COD 96] CODOGNET, Philippe; DIAZ, Daniel. Compiling Constraints in clp(FD). **Journal of Logic Programming**, New York, v.27, n.1, 1996. p.185-226.

[COH 90] COHEN, Jacques. Constraint Logic Programming Languages. **Communications of the ACM**, New York, v.33, n.7, p.52-68, July 1990.

[COL 90] COLMERAUER, Alain. An Introduction to Prolog III. **Communications of the ACM**, New York, v.33, n.7, p.69-90, July 1990.

[COS 96] COSTA, Cristiano André da. Uma Proposta de Escalonamento Distribuído para Exploração do Paralelismo na Programação em Lógica. In: SEMANA ACADÊMICA DA COMPUTAÇÃO, 1., 1996, Porto Alegre. **Anais...** Porto Alegre: CPGCC/UFRGS, 1996. p.287-290.

[DAN 97] DANDAMUDI, Sivarama. **The Effect of Scheduling Discipline on Dynamic Load Sharing in Heterogeneous Distributed Systems**. [S.l.: s.n.], 1997. (Technical Report TR-96-26)

[DIA 94a] DIAZ, Daniel; CODOGNET, Philippe. A Minimal Extension of the WAM for clp(FD). In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 10., 1993, Budapest, Hungary, **Proceedings...** New York: MIT Press, 1994. p. 774-790.

[DIA 94b] DIAZ, Daniel. **clp(FD) 2.21 - User's Manual**. July 1994. 8p. Disponível em ftp em ftp://ftp.inria.fr/INRIA/Projects/ChLoE/LOGIC_PROGRAMMING/clp_fd/

[DIA 95] DIAZ, Daniel. **Étude de la Compilation des Langages Logiques de Programmation par Contraintes sur les Domaines Finis**: le Système clp(FD). Orléans: L'Université D'Orléans, 1995. 270p. Thése.

[DIN 90] DINCBAS, Mehmet; SIMONIS, Helmut; HENTENRYCK, Pascal van. Solving Large Combinatorial Problems in Logic Programming. **Journal of Logic Programming**, New York, v.8, n.1-2, p. 74-94, 1990.

[DUT 95] DUTRA, Inês de Castro. **Distributing And- and Or-Work in the Andorra-I Parallel Logic Programming System**. Bristol: University of Bristol, February 1995. 264p. Thesis.

[ELR 94] EL-REWINI, Hesham; LEWIS, Theodore G.; ALI, Hesham H. **Task Scheduling in Parallel and Distributed Systems**. New Jersey: Prentice Hall, 1994. 290p.

[FOW 87] FOWLER, Martin; SCOTT, Kendall. **UML Distilled**: Applying the Standard Object Modeling Language. Reading : Addison-Wesley, 1997. 179p.

[FRU 93] FRÜHWITH, Thom et al. **Constraint Logic Programming:** An Informal Introduction. Müchen, Germany: European Computer-Industry Research Center, February 1993. 25p. Disponível em ftp em ftp.ecrs.de:/pub/ECRC_tech_reports. (Technical Report ECRC-93-5).

[GEY 92] GEYER, Cláudio Fernando Resin et al. Otimizações Importantes para Paralelismo Ou em Prolog sobre Máquinas com Memória Distribuída. In: CONFERÊNCIA LATINO-AMERICANA DE INFORMÁTICA, 18., 1992, Las Palmas de Gran Canária. **Anais...** Las Palmas de Gran Canária: CLEI, 1992. p.540-547.

[GUP 93] GUPTA, Gopal; ALI, Khayri A.M.; CARLSSON, Mats; HERMENEGILDO, Manuel V. **Parallel Execution of Prolog Programs**: A Survey. 1993. Disponível em ftp em http://www.cs.nmsu.edu:80/lldap/pub_para/survey.html (Preliminary version)

[HEM 96] HENZ, Martin; WÜRTZ, Jörg. Using Oz for College Time Tabling. **Journal of Applied Artificial Intelligence**, [S.l.], v.10, n.5, October 1996.

[HEN 89] HENTENRYCK, Pascal van. **Constraint Satisfaction in Logic Programming**. London: MIT Press, 1989. 224p.

[HEN 92] HENTENRYCK, Pascal van; SIMONIS, Helmut; DINCBAS, Mehmet. Constraint Satisfaction using Constraint Logic Programming. **Artificial Intelligence**, Amsterdam, v.58, n.1-3, p.113-159, December 1992

[HEN 93] HENTENRYCK, Pascal van; SARASWAT, Vijay A.; DEVILLE, Yves. **Design, Implementation, and Evaluation of the Constraint Language cc(FD).** [S.l.:s.n], 1993. 22p. (Technical report CS-93-02).

[HOL 95] HOLZBAUR, Christian. **OFAI clp(Q, $\mathcal{R}$) Manual**: Edition 1.3.3. Vienna: Austrian Research Institute for Artificial Intelligence, 1995. 24p. (Technical report TR-95-09).

[JAF 92] JAFFAR, Joxan et all. The CLP(R) Language and System. **ACM Transactions on Programming Languages and Systems**, New York, v.14, n.3, p.339-395, July 1992.

[JAF 94] JAFFAR, Joxan; MAHER, Michael J. Constraint Logic Programming: A Survey. **The Journal of Logic Programming**, New York, v.19/20, p.503-581, May/July 1994.

[KAN 94] KANNAT, S.E. et al. A Platform to Study Load Balancing Functions for Parallel Logic Systems. In: INTERNATIONAL WORKSHOP ON PARALLEL PROCESSING, 1., 1994, Bangalore, India, **Proceedings...** [S.l.:s.n], 1994.

[KER 92] KERGOMMEAUX, Jacques Chassin; CODOGNET, Philippe. **Parallel Logic Systems**. Grenoble, France: Institut IMAG, 1992. Technical Report.

[KON 94] KONNO, Kazuriro et al. PARCS: An MPP-Oriented CLP Language. In: PASCO, 1994, [S.l.]. **Proceedings...** [S.l]: World Scientific Publishing Company, 1994.

[KOW 79a] KOWALSKI, Robert. **Logic for Problem Solving**. New York: Elsevier Science Publishing, 1979. 287p.

[KOW 79b] KOLWALSKI, Robert A. Algorithm = Logic + Control. **Communications of the ACM**, New York, v. 22, n.7, p.424-436, July 1979.

[KRU 94] KRUEGER, Phillip; SHIVARATRI, Niranjan G. Adaptative Location Policies for Global Scheduling. **IEEE Transactions on Software Engineering**, New York, v. 20, n. 6, p.432-444, June 1994.

[LUS 90] LUSK, Ewing; WARREN, David H. D. et al. The Aurora OR-Prolog System. **New Generation Computing**, Berlin, v.7, n.2/3, p.243-271, 1990.

[LYN 96] LYNCH, Nancy A. **Distributed Algorithms.** San Francisco : Morgan Kaufmann, 1996. 872p.

[MAN 98] MANGIONE, Carmine. **Performance tests show Java as fast as C++**: Java Has Endured Criticism for Its Laggard Performance (Relative to C++) Since Its Birth, But the Performance Gap is Closing. February 1998. Disponível em html em http://www.javaworld.com/javaworld/jw-02-1998/jw-02-jperf.html

[MCM 96] McMANIS, Chuck. **Just In Time Compilation**: A Look at the Technology that Turbo Charges Java Applications. 1996. Disponível em html em http://www.javacats.com/us/articles/chuckmcmanis091696.html.

[MON 96] MONFROGLIO, Angelo. Timetabling through Constrained Heuristic Search and Genetic Algorithms. **Software - Practice and Experience**, London, v. 26, n.3, p.251-279, March 1996.

[MOR 97] MOREL, Érick; BRIAT, Jacques; KERGOMMEAUX, Jacques Chassin de. Cuts and Side-effects in Distributed Memory OR-Parallel Prolog. **Parallel Computing**, Netherlands, v. 22, p.1883-1896, February 1997.

[MUD 94] MUDAMBI, Shyam; SCHIMPF, Joachim. Parallel CLP on Heterogeneous Networks. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 11., 1994, Santa Marherita Ligure, Italy. **Proceedings...** New York: MIT Press, 1994. p.124-141.

[OAK 97] OAKS, Scott; WONG, Henry. **Java Threads**. Cambridge, USA: O'Reilly & Associates, 1997. 252p.

[ORF 97] ORFALI, Robert; HARKEY, Dan. **Client/Server Programming with Java and CORBA**. New York: John Wiley & Sons, 1997. 655 p.

[PRE 93] PRESTWICH, Steven. **ElipSys Programming Tutorial.** [S.l]: ECRC, April 1993. 30p. Technical Report.

[PRJ 98] PROTIC, Jelica; TOMASEVIC, Milo; MILUTINOVIC, Veljko. An Overview of Distributed Shared Memory. In: PROTIC, Jelica; TOMASEVIC, Milo; MILUTINOVIC, Veljko (Eds.). **Distributed Shared Memory**: Concepts and Systems. USA: IEEE Computer Society Press, 1998. p.12-41.

[PRO 93] PROVOST, Thierry Le; WALLACE, Mark. Generalised Constraint Propagation Over the CLP Scheme. **Journal of Logic Programming**, New York, v.16, July 1993. Também como: relatório técnico ECRC-92-1, ECRC, 1992.

[RÉT 96] RÉTY, Jean-Huges. **A Distributed Concurrent Constraint Programming Language**. [S.l: s.n.], March 1996.

[ROC 97] ROCHA, Ricardo; SILVA, Fernando; COSTA, Vítor S. **YapOr: an OR-Parallel Prolog System based on Environment Copying**. Porto: Universidade do Porto, December 1997. 15p. (Technical Report DCC-97-14). Submitted to VECPAR'98.

[SAR 93] SARASWAT, Vijay A. **Concurrent Constraint Programming**. Cambridge: MIT, 1993. 486p. ACM doctoral dissertation awards.

[SIN 94] SINGHAL, Mukesh; SHIVARATRI, Niranjan G. **Advanced Concepts in Operating Systems**: Distributed, Database, and Multiprocessor Operating Systems. New York: MIT Press, 1994. 522p.

[STE 94] STERLING, Leon; SHAPIRO, Ehud Y. **The Art of Prolog**. London: MIT Press, 1994. 509p.

[SUN 97] SUN. **The Java^{tm} Language**: an overview. 1997. Disponível em html em http://java.sun.com/docs/overviews/java/java-overview-1.html.

[SUN 98] SUN. **Java JIT Compiler Overview**. 1998. Disponível em html em http://www.sun.com/solaris/jit/

[TON 95] TONG, Bo-Ming; LEUNG, Ho-Fung. Concurrent Constraint Logic Programming on Massively Parallel SIMD Computers. In: INTERNATIONAL SYMPOSIUM IN LOGIC PROGRAMMING, 1993, Vancouver, British Columbia, Canada. **Proceedings...** New York: MIT Press, 1993. p.388-402.

[VAR 95] VARGAS, Patrícia Kayser. **Implementação de um Analisador de Granulosidade para Prolog**. Porto Alegre: CIC/UFRGS, 1995. 71p. Projeto de diplomação.

[VAR 97a] VARGAS, Patrícia Kayser. **Um Estudo sobre as Linguagens em Lógica com Restrições**: trabalho individual. Porto Alegre: CPGCC/UFRGS, 1997. 40p. (TI-621).

[VAR 97b] VARGAS, Patrícia Kayser. Exploração de Paralelismo OU em uma Linguagem em Lógica com Restrições. In: SEMANA ACADÊMICA DO CPGCC, 2., 1997, Porto Alegre, RS. **Anais...** Porto Alegre: CPGCC/UFRGS, 1997. p.31-34.

[VAR 97c] VARGAS, Patrícia Kayser; GEYER, Cláudio Fernando Resin. Introduzindo o Paralelismo OU na Programação em Lógica com Restrições. In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES - PROCESSAMENTO DE ALTO DESEMPENHO, 9., 1997, Campos do Jordão, SP. **Anais...** São Paulo: Escola Politécnica da USP, 1997. p.381-396.

[WER 94] WERNER, Otília. **Uma Máquina Abstrata Estendida para o Paralelismo E na Programação em Lógica.** Porto Alegre: CPGCC/UFRGS, 1994. 170p. Dissertação de mestrado.

[YAM 94] YAMIN, Adenauer Corrêa. **Um Ambiente para Exploração de Paralelismo na Programação em Lógica**. Porto Alegre: CPGCC/UFRGS, 1994. 204p. Dissertação de mestrado.

[YAN 95] YANG, Rong. Implementing a Finite Domain Constraint Solving System. In: ICLP95 - WORKSHOP ON PARALLEL LOGIC PROGRAMMING, Kanagawa. **Proceedings...** [S.l: s.n.], June 1995.

Informática  UFRGS

CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Exploração de Paralelismo OUem uma Linguagem em Lógica com Restrições

por

Patrícia Kayser Vargas

Dissertação apresentada aos Senhores:

Profa. Dra. Ana Maria de Alencar Price

Prof. Dr. Philippe Olivier Alexandre Navaux

Prof. Dr. Celso Maciel da Costa (PUC/RS)

Vista e permitida a impressão.
Porto Alegre, 14/08/98.

Prof. Dr. Cláudio Fernando Resin Geyer,
Orientador.

Profa. Carla Maria Dal Sasso Freitas
Coordenadora do Curso de Pós-Graduação
em Ciência da Computação
Instituto de Informática - UFRGS