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

Uma Proposta de Escalonamento Distribuído para Exploração de Paralelismo na Programação em Lógica

por

CRISTIANO ANDRÉ DA COSTA

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

Prof. Cláudio Resin Geyer
Orientador

Porto Alegre, abril de 1998.

CIP - CATAlogação na publicação

Costa, Cristiano André da

Uma proposta de escalonamento distribuído para exploração de paralelismo na programação em lógica / por Cristiano André da Costa.- Porto Alegre: CPGCC da UFRGS, 1998.

104 f.:il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Sul, Curso de Pós-graduação em Ciência da Computação, Porto Alegre, BR-RS, 1997. Orientador: Geyer, Cláudio F.R..

1.Processamento Paralelo. 2.Programação em Lógica. 3.Paralelismo E. 4.Paralelismo OU. 5.Escalonamento Distribuído. 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. Roberto Tom Price
Coordenador do CPGCC: Prof. Flávio Rech Wagner
Bibliotecária-Chefe do Instituto de Informática: Zita Prates de Oliveira

"Nada é tão complexo ou tão simples quanto parece. Com vontade e determinação tudo pode ser alcançado."

- Cristiano Costa

Agradecimentos

Gostaria de agradecer inicialmente ao meu orientador professor Cláudio Geyer pela participação constante, acompanhamento e estímulo nesta fase da minha vida.

Ao amigo Adenauer Yamin que sempre me apoiou nos trabalhos realizados. Em especial pelo incentivo ao mestrado e pelas inúmeras e incontáveis ajudas.

Ao amigo Jorge Barbosa pelas discussões sobre granulosidade, pela amizade e por todo apoio durante o mestrado.

Ao colega do projeto OPERA que já conhecia antes do mestrado, Luís Fernando Castro, pelo convívio durante o mestrado. Por ter me "aturado sobre o mesmo teto" durante dois anos de nossas vidas.

Às colegas do projeto OPERA que conheci durante o mestrado, Denise Bandeira e Patrícia Kayser pelo grande apoio, almoços e outros bons momentos que compartilhamos durante o mestrado. Hoje vocês são grandes amigas.

Aos bolsistas do projeto Charles Trein e Leonardo Cervo sem os quais o protótipo nunca teria sido implementado. Agradeço pela inestimável colaboração.

Aos colegas de mestrado, em especial a Mauro Baioneta e Rafael Santos pela amizade e receptividade.

Aos professores e funcionários do CPGCC, em especial ao Prof. Philippe Navaux por todo convívio e à funcionária Eliane Iranco pelo seu carinho.

À minha família que me apoiou na decisão de ir para o mestrado e sempre me ajudou durante sua realização.

À minha namorada Adriana Roehe pelas cobranças e estímulos para que concluísse a dissertação.

A todas as outras pessoas que me ajudaram de forma direta ou indireta para a realização do mestrado.

Ao CNPq pelo apoio financeiro, sem o qual este trabalho não aconteceria.

Sumário

Lista de Abreviaturas **7**
Lista de Figuras **8**
Lista de Tabelas **10**
Resumo **11**
Abstract **12**
1 Introdução **13**
1.1 Tema **13**
1.2 Motivação **13**
1.3 Contribuição do Autor **14**
1.4 Estrutura da Dissertação **14**
2 Programação em Lógica e Prolog **15**
2.1 Programação em Lógica **15**
2.1.1 Fatos 15
2.1.2 Consultas 15
2.1.3 Regras 16
2.2 Prolog **17**
2.2.1 O Modelo de Execução 17
2.2.2 Operadores de Corte 17
2.2.3 Efeitos Colaterais 18
2.3 A Warren Abstract Machine (WAM) **18**
2.4 Conclusões **22**
3 Programação em Lógica e Paralelismo **23**
3.1 Paralelismo OU **23**
3.2 Paralelismo E **24**
3.2.1 Paralelismo E Independente 25
3.2.2 Paralelismo E Dependente 25
3.3 Paralelismo E/OU **26**
3.4 Principais Sistemas **26**
3.4.1 Andorra-I 26
3.4.2 ACE 27
3.4.3 ROPM 28
3.4.4 IDIOM 29
3.5 Comparações entre os Principais Sistemas **30**
3.5.1 Semântica de Prolog seqüencial 30
3.5.2 Linguagem Suportada 31
3.5.3 Arquiteturas Suportadas 31
3.5.4 Baseados em Processos versus Baseados em Enlaces (*Threads*) 31
3.5.5 Escalonamento 31
3.5.6 Suporte à Análise em Tempo de Compilação 32
3.5.7 Overhead do Paralelismo 32
3.5.8 Desempenho Geral dos Sistemas 32
3.5.9 Quadro Comparativo 32

3.6 Conclusões 33

4 OPERA e GRANLOG 34

4.1 OPERA OU 34

4.1.1 PloSys 34

4.2 OPERA E 36

4.3 GRANLOG 37

4.4 Conclusões 39

5 Escalonamento 40

5.1 Conceitos Básicos 40

5.2 Escalonamento na Programação em Lógica 43

5.3 Escalonamento Distribuído 44

5.4 Conclusões 47

6 Escalonador Hierárquico: DSLP 48

6.1 Princípios Básicos 48

6.2 Visão Geral 49

6.3 Descrição do Modelo DSLP 50

6.3.1 Estruturas de Dados 54

6.3.2 Funcionamento do Escalonador 57

6.3.3 Trabalhador OU 74

6.3.4 Trabalhador E 76

6.3.5 Espião (*spy)* 77

6.4 Extensões ao DSLP 78

6.5 Conclusões 79

7 Protótipo do Escalonador DSLP 80

7.1 Descrição do Protótipo 80

7.1.1 Ferramentas Utilizadas 80

7.1.2 Organização Básica 81

7.2 Simulador de Programas Prolog 82

7.3 Resultados Obtidos 87

7.3.1 Quantidade de Trabalhadores e Tempo de Inatividade do Espião 88

7.3.2 Uso das Informações de Granulosidade 90

7.3.3 Quantidade de Escalonadores e Trabalhadores 91

7.4 Conclusões 92

8 Conclusões 93

Anexo 1 *System Resource Table* 95

Anexo 2 *Application Characteristic Table* 96

Anexo 3 Programas Utilizados nos Testes 97

Bibliografia 99

Lista de Abreviaturas

ACE	And/Or-parallel Copying-based Execution of Logic Programs
ACRPL	Acumulado da Complexidade das Resolventes Pendentes Locais
ACT	Application Characteristic Table
CC	Complexidade da Cláusula
CCpri	Complexidade da Cláusula Primeira
CGE	Conditional Graph Expression
CM	Complexidade da Meta
CPE	Complexidade do Ponto de Escolha
CPGCC	Curso de Pós-graduação em Ciência da Computação
CRPL	Complexidade da Resolvente Pendente Local
CTPE	Complexidade Total dos Pontos de Escolha
DAP	Dependent And-Parallel phase
DSLP	Distributed Scheduler for Logic Programming
EGL	Executing Goals List
GRANLOG	Granularity Analyzer for Logic Programming
IAP	Independent And-Parallel phase
IDIOM	Integrated Dependent- Independent and Or-parallel Model
IP	Internet Protocol
LR	List of Results
MIPS	Milhões de Instruções por segundo
ORP	Or-Parallel phase
PDL	Push-down List
PGL	Parallel Goals List
PLOSYS	Parallel Logic System
ROPM	REDUCE-OR Process Model
SGL	Sequential Goals List
SRT	System Resource Table
UCP	Unidade Central de Processamento
UFRGS	Universidade Federal do Rio Grande do Sul
WAM	Warren Abstract Machine

Lista de Figuras

FIGURA 2.1 - Uma Regra e seus Componentes 16
FIGURA 2.2 - Organização de Memória e Registradores da WAM 19
FIGURA 2.3 - Lista de Instruções da WAM 21
FIGURA 3.1 - Árvore de Busca e Paralelismo OU 23
FIGURA 3.2 - Árvore de Busca e Paralelismo E 24
FIGURA 4.1 - Arquitetura de Processos do PloSys 35
FIGURA 4.2 - Arquitetura de Processos do OPERA E 36
FIGURA 4.3 - Organização do GRANLOG 38
FIGURA 4.4 - Programa Granulado Gerado pelo GRANLOG 38
FIGURA 5.1 - Taxonomia de Escalonamento de Casavant 40
FIGURA 5.2 - Escalonamento Centralizado 43
FIGURA 5.3 - Escalonamento Totalmente Distribuído 45
FIGURA 5.4 - Escalonamento Hierárquico 46
FIGURA 6.1 - Etapas de Execução de um Programa Prolog 50
FIGURA 6.2 - Escalonador Hierárquico 51
FIGURA 6.3 - Anel Lógico Formado Entre Escalonadores 51
FIGURA 6.4 - Componentes do DSLP 53
FIGURA 6.5 - Informações de Carga do Escalonador 58
FIGURA 6.6 - Ciclo de Mensagens do Escalonador 59
FIGURA 6.7 - Fase OU de Escalonamento 61
FIGURA 6.8 - Algoritmo de Escalonamento para os Nodos OU 62
FIGURA 6.9 - Algoritmo de Recebimento de Trabalho OU pelos Vizinhos 63
FIGURA 6.10 - Exemplo de Configuração de Nodos OU 64
FIGURA 6.11 - Exportação de Trabalho com Vizinhos em Estado *idle* 65
FIGURA 6.12 - Exportação de Trabalho com Apenas um Vizinho em Estado *idle* 65
FIGURA 6.13 - Exportação de Trabalho com Vizinhos em Estado *quiet* 66
FIGURA 6.14 - Todos os Vizinhos Consumindo Trabalho 66
FIGURA 6.15 - Nodo Fica Disponível Novamente 67
FIGURA 6.16 - Estados dos Vizinhos 67
FIGURA 6.17 - Fase E de Escalonamento 68
FIGURA 6.18 - Algoritmo de Escalonamento para os Nodos E 70
FIGURA 6.19 - Algoritmo de Recebimento de Trabalho E pelos Vizinhos 72
FIGURA 6.20 - Exemplo de Configuração de Nodos E 72
FIGURA 6.21 - Exportação de Trabalho Entre Nodos E Vinculados 73
FIGURA 6.22 - Informações Armazenadas na *Stack* 76

FIGURA 6.23 - Uso da Granulosidade no Trabalhador E..77
FIGURA 7.1 - Arquitetura do Protótipo DSLP 81
FIGURA 7.2 - Código do Fibo de 5 para o Simulador..82
FIGURA 7.3 - Algoritmo do Simulador no Protótipo DSLP 86
FIGURA 7.4 - Comparação no Tempo de Execução das Séries de *Fibonacci*......................89
FIGURA 7.5 - Influência das Informações de Granulosidade...90
FIGURA 7.6 - Diversas Configurações de Trabalhadores e Escalonadores.........................92

Lista de Tabelas

TABELA 3.1 - Quadro Comparativo dos Sistemas que Integram o Paralelismo E/OU 33
TABELA 5.1 - Comparação Entre as Políticas de Escalonamento 47
TABELA 6.1 - Cálculo de Índices de Importação...74
TABELA 7.1 - Tempos de Execução com Espião Inativo por 6s...88
TABELA 7.2 - Tempos de Execução com Espião Inativo por 3s...89
TABELA 7.3 - Tempos de Execução com Espião Inativo por 1s...89
TABELA 7.4 -Tempos de Execução sem e com o Uso das Informações de Granulosidade 90
TABELA 7.5 - Tempos de Execução com Diversas Configurações Diferentes.....................91

Resumo

Este trabalho apresenta um modelo de escalonamento hierárquico para exploração do paralelismo E Independente e do paralelismo OU na programação em lógica. O modelo utiliza informações de granulosidade geradas pelo GRANLOG (*Granularity Analyzer for Logic Programming*) para o auxílio ao escalonamento.

Um estudo detalhado de ambientes de programação em lógica explorando o paralelismo é apresentado. A partir deste, é feita uma comparação destacando as principais características de cada um. O escalonamento em linhas gerais também é descrito e uma enfâse maior é dada ao escalonamento dinâmico. As principais vantagens e desvantagens de cada escalonador são mostradas.

O modelo proposto recebe o nome de DSLP – *Distributed Scheduler for Logic Programming* e realiza o escalonamento em duas fases. Inicialmente é executada a Fase OU, na qual todo paralelismo OU é explorado. Em seguida, é iniciada a Fase E onde ocorre a exploração do paralelismo E Independente. A estratégia de escalonamento proposta, utiliza informações de complexidade do GRANLOG para determinar o trabalho a ser exportado, bem como o nível de sobrecarga dos nodos.

Para validação do trabalho, um protótipo utilizando o ambiente *Parallel Virtual Machine* foi implementado. O protótipo é um simulador de programas Prolog e implementa a fase E de escalonamento.

Palavras-chave: processamento paralelo, programação em lógica, paralelismo E, paralelismo OU, escalonamento hierárquico.

TITLE: "A DISTRIBUTED SCHEDULER PROPOSAL FOR EXPLORATION OF PARELLELISM IN LOGIC PROGRAMMING"

Abstract

This work presents a hierarchical scheduling model for exploration of the Independent AND parallelism and OR parallelism in logic programming. The model uses granularity information generated by GRANLOG (Granularity Analyzer for Logic Programming) to aid the scheduler.

A detailed study of parallel logic programming environments is presented. Starting from this, it is made a comparison highlighting the main characteristics of each one. Scheduling in general is also described and the dynamic scheduling is pointed out. The main advantages and disadvantages of each scheduler are shown.

The proposed model receives the name of DSLP – Distributed Scheduler for Logic Programming and it accomplishes the scheduling in two phases. Initially the OR Phase is executed and the whole OR parallelism is explored. Soon after, it is initiate the AND Phase with the exploration of the Independent AND parallelism. The scheduling strategy proposed uses complexity information generated by GRANLOG to determinate the task to be exported, as well as the nodes overloaded level.

For work validation, a prototype using the Parallel Virtual Machine was implemented. The prototype is a Prolog simulator and it implements the scheduling AND phase.

Keywords: parallel processing, Logic programming, OR parallelism, AND parallelism, hierarchical scheduling.

1 Introdução

1.1 Tema

O tema central desta dissertação é o estudo do escalonamento distribuído na programação em lógica. A partir do estudo de várias políticas de escalonamento, foi definida uma política distribuída e hierárquica. O trabalho utiliza informações de granulosidade no auxílio ao escalonamento de programas Prolog.

No trabalho é proposto um modelo de escalonamento que integra as duas principais formas de exploração de paralelismo na programação em lógica. O modelo é denominado DSLP - *Distributed Scheduler for Logic Programming (*Escalonador Distribuído para a Programação em Lógica). Para a validação do modelo de escalonamento, foi implementado um protótipo.

1.2 Motivação

O processamento paralelo vem sendo utilizado como uma alternativa para o aumento de desempenho nos sistemas de computação uma vez que os processadores individualmente estão atingindo seus limites físicos, impostos pela tecnologia dos componentes eletrônicos utilizados em sua implementação. A exploração do paralelismo é também estimulada pelo contínuo crescimento na complexidade das aplicações.

Para suportar o processamento paralelo, é desejado que a linguagem empregada seja capaz de detectar o paralelismo inerente a cada programa, utilizando de forma eficiente os processadores disponíveis na arquitetura. Nesta categoria, existem dois grandes grupos de linguagens, a saber: as que administram o processamento paralelo de forma explícita, através de construções específicas para tal fim, e as que o fazem de maneira implícita, mantendo a mesma sintaxe quando do processamento seqüencial. Um dos objetivos das linguagens que exploram o processamento paralelo implícito é a não re-implementação de todo o *software* utilizado quando da substituição de máquinas seqüenciais por paralelas.

Por sua vez, a programação em lógica permite a computação de programas de forma seqüencial ou paralela sem comprometer sua semântica declarativa. A possibilidade de paralelização implícita é conseqüência de uma característica deste tipo de programação que especifica a lógica do programa sem impor uma ordem de execução.

Deste modo um programa em lógica pode ser decomposto em uma parte lógica que especifica o significado declarativo do programa e uma parte de controle que provê um meio de executar esta especificação ([STE 86]). Dentre as linguagens que implementam a programação em lógica destaca-se Prolog.

Prolog é uma linguagem de programação baseada na programação em lógica com cláusulas de Horn, proposta por Kowalski nos anos 70, e implementada pela primeira vez por Alain Colmerauer na Universidade de Marselha. Atualmente, existem várias implementações de Prolog para multiprocessadores (memória compartilhada) e multicomputadores (memória distribuída) ([COS 96a], [SWE 95] e [YAM 92]).

Dentre as implementações de Prolog paralelo, destaca-se o OPERA ([GEY 91]). O projeto OPERA foi iniciado em Grenoble e encontra-se hoje em desenvolvimento no Instituto de Informática da UFRGS. Para o aumento de desempenho do sistema foram propostas técnicas de análise de granulosidade pelo GRANLOG ([BAR 96a]).

1.3 Contribuição do Autor

A principal contribuição do trabalho é um modelo de escalonador distribuído para a programação em lógica, denominado DSLP (*Distributed Scheduler for Logic Programming*). Este modelo faz uso das principais formas de exploração de paralelismo na programação em lógica, quais sejam: paralelismo OU e paralelismo E.

Como outras contribuições do autor, pode-se citar:

- definição de uma proposta para utilização de informações de granulosidade geradas pelo GRANLOG no auxílio ao escalonamento em processamento paralelo;
- o desenvolvimento de um protótipo de escalonamento para a validação do modelo compatível com o projeto OPERA;
- análise comparativa das principais propostas que integram o paralelismo E/OU na programação em lógica;
- estudo das principais políticas de escalonamento.

1.4 Estrutura da Dissertação

A dissertação está organizada em 8 capítulos, sendo os primeiros 4 capítulos introdutórios para o restante do texto. Inicialmente são apresentados conceitos básicos, sobre cada um dos temas que serão abrangidos no trabalho. Em seguida é descrito o modelo de escalonamento proposto. Finalmente, é apresentado o protótipo desenvolvido.

O capítulo 2 abrange os conceitos relacionados à programação em lógica e Prolog. Na seqüência, o capítulo 3 apresenta as formas de exploração de paralelismo na programação em lógica, bem como os principais sistemas que integram as duas formas principais de exploração de paralelismo.

No capítulo 4 é descrito o projeto OPERA e o GRANLOG. A seguir, no capítulo 5 é feita uma introdução sobre o tema escalonamento e são apresentados os principais conceitos relacionados.

O capítulo 6 apresenta o modelo de escalonamento hierárquico. O protótipo desenvolvido para a validação do modelo é descrito no capítulo 7.

Finalmente no capítulo 8 estão as conclusões do trabalho.

2 Programação em Lógica e Prolog

Neste capítulo são apresentadas noções básicas sobre programação em lógica, Prolog e sobre uma das implementações mais eficientes de Prolog a *Warren Abstract Machine* (WAM). Para estudos mais aprofundados em programação em lógica e Prolog, consultar [DER 96], [OKE 94], [STE 94], [HOG 84] e [CLO 81]. Para saber mais sobre a WAM, consultar [AIT 90] e [WAR 83].

2.1 Programação em Lógica

Programação em lógica é um modelo de linguagem derivada da lógica dos predicados. Um programa neste modelo é formado por um conjunto de axiomas definindo relações entre objetos.

A programação em lógica é um paradigma de linguagem de alto nível que possibilita o desenvolvimento de programas mais concisos e em menos tempo do que na programação Imperativa. Entretanto, a execução dos programas normalmente é menos eficiente neste paradigma.

Existem três construções básicas na programação em lógica, a saber: fatos, consultas e regras. A seguir são apresentadas estas construções, bem como conceitos básicos relacionados.

2.1.1 Fatos

São declarações simples que expressam relacionamentos entre objetos. Um exemplo simples é:

irmao(pedro,carlos).

Este fato estabelece uma relação entre Pedro e Carlos, significando que eles são irmãos. A relação é chamada de predicado. Os nomes pedro e carlos são chamados de átomos. Normalmente nomes de predicados e átomos são iniciados por letras minúsculas.

2.1.2 Consultas

As consultas são meios pelos quais é possível obter informação de um programa em lógica. Exemplo de uma consulta:

$$\leftarrow irmao(pedro,carlos).$$

Esta consulta pode ser interpretada como: "É fato que Pedro e Carlos são irmãos?". A resposta pode ser obtida verificando se a consulta é uma *conseqüência lógica* do programa. Para tal, um fato idêntico à consulta pode ser encontrado e então a resposta será afirmativa. A resposta também será verdadeira se for encontrada uma regra (seção 2.1.3) que satisfaça a consulta.

Consultas podem ser realizadas ainda, através do uso de *variáveis lógicas*. Veja o exemplo:

$$\leftarrow irmao(pedro, X).$$

O significado desta consulta é: "Pedro tem algum irmão?". No exemplo, X é uma *variável lógica* que juntamente com as *constantes* e os *termos compostos* constituem os termos.

Termos compostos são termos na forma $f(t_1, t_2, \ldots, t_n)$, onde f é o *functor* e os t_{is} são os argumentos. Um *functor* é identificado pelo seu nome e por sua *aridade* (quantidade de argumentos).

Variáveis quando ocorrem em programas estão inicialmente livres e para que a instanciação aconteça deve haver uma *substituição*. Uma *substituição* é um par, (por exemplo, $\{X = carlos\}$) que aplicado a um termo com uma variável livre (por exemplo, *irmao(pedro, X).*) resulta em um termo com uma *variável ligada* (a variável é um apontador para o termo).

2.1.3 Regras

Através das regras é possível estabelecer uma relação a partir de relações já existentes. Veja o exemplo:

$$irmao(X, Y) \leftarrow progenitor(Z, X) \wedge progenitor(Z, Y).$$

Existem duas maneiras de interpretar o exemplo dado. Poderia ser interpretado como "Para responder se X é irmão de Y, responda a conjunção das consultas: Z é progenitor de X e Z é progenitor de Y". Esta visão é chamada de *procedural*. Ela expressa consultas complexas em termos de consultas simples.

A segunda forma de interpretação é a visão *declarativa*, que interpreta regras como axiomas lógicos. Para o exemplo dado, a interpretação seria: "Para todo X, Y e Z; X é irmão de Y se Z é progenitor de X e Z é progenitor de Y".

Uma regra divide-se em duas partes: *cabeça* e *corpo*. Além disso cada parte do corpo é chamada de *meta*, conforme mostra a figura 2.1.

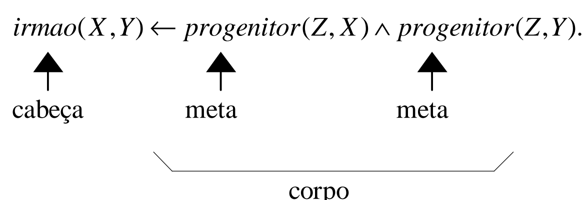

FIGURA 2.1 - Uma Regra e seus Componentes

Para que determinada meta seja avaliada, é necessário que ocorra uma *unificação*. Dois *termos* unificam se:

1. eles são idênticos, ou

2. uma ou mais substituições aplicadas a dois termos os tornam iguais, resultando em uma instanciação mais geral.

Por exemplo, os termos $a(X,45,Y)$ e $b(teste,G,H)$ unificam. Uma instanciação que torna os dois idênticos é:

X é instanciado com $teste$.

G é instanciado com 45.

Y é instanciado com H.

2.2 Prolog

Prolog é uma linguagem de programação desenvolvida a partir da programação em lógica com cláusulas de Horn. A utilização destas cláusulas faz com que Prolog seja visto como uma restrição da programação em lógica. Em Prolog as cláusulas de Horn são representadas como:

$$a(X,Y):-b(Z,X),c(Z,Y).$$

O símbolo "$:-$" representa a implicação ($\leftarrow$) e a vírgula representa a conjunção ($\wedge$). As cláusulas podem possuir vários argumentos e metas. Assim como na programação em lógica as constantes são iniciadas por letras minúsculas e as variáveis por maiúsculas.

2.2.1 O Modelo de Execução

Existem várias implementações de Prolog e consequentemente diversos modelos de execução. Será apresentado nesta seção o modelo de execução de Prolog puro e seqüencial.

A execução de um programa Prolog consiste em percorrer uma árvore de busca onde os nodos (pontos de escolha ou *choice-points*) contém alternativas que podem satisfazer determinada *meta*.

Esta árvore é percorrida primeiro de cima para baixo, então as metas são executadas da esquerda para a direita. Este algoritmo de caminhamento é chamado de *depth-first left-to-right*. Em um determinado passo da execução de um programa Prolog, o conjunto de metas que devem ser executadas são chamadas de *resolvente*.

Quando uma *meta* falha, entra em funcionamento o mecanismo de retrocesso (*backtracking*). Este mecanismo consiste em rever parte do que foi feito, tentando novas alternativas e satisfazendo as metas a partir de uma outra unificação.

2.2.2 Operadores de Corte

Prolog ao percorrer a árvore de busca realiza o retrocesso de forma automática. Algumas vezes o programador quer evitar a pesquisa de determinadas alternativas. Isto pode ser obtido com o uso de *pruning operators* (operadores que "cortam" ramos da árvore de busca).

Dentre esta categoria de operadores é destacado o *cut*. Ele é representado pelo símbolo “!” e é o único operador de corte encontrado em Prolog padrão ([DER 96]). O *cut* é utilizado para desconsiderar alternativas durante o retrocesso. Na execução normal, o *cut* é sempre um objetivo bem sucedido.

A utilização do *cut* ocorre por dois motivos principais:

- O programa irá ficar mais rápido, uma vez que ramos que não contribuem para determinada solução não serão percorridos;
- O programa ocupará menos espaço de memória, uma vez que alguns pontos de retrocesso não necessitarão ser armazenados.

Por outro lado, o uso do *cut* reduz o aspecto declarativo do programa, o que traz um aumento de sincronismo quando da paralelização do processo de busca na árvore de metas ([YAM 92]).

2.2.3 Efeitos Colaterais

Prolog pode ocasionalmente ter efeitos colaterais (*side-effects*) durante a execução de um programa. Metas e predicados que produzem efeitos colaterais incluem predicados como *assert*, *retract, read, write, cut* e outros.

Basicamente existem dois tipos de predicados com *side-effects*, quais sejam:

- os operadores de corte (seção 2.2.2);
- operadores que modificam ou dependem do ambiente, como predicados de entrada e saída, *data-base* e outros.

A semântica operacional de Prolog consiste na execução seqüencial de metas contidas em uma cláusula, em ordem, da esquerda para a direita. Além disso, múltiplas cláusulas de um único procedimento são tentadas uma por vez, de cima para baixo. Quando as cláusulas contém predicados com *side-effects,* estas duas ordens produzem um determinado comportamento. Para a execução paralela de Prolog manter a semântica normal, é necessário manter a ordem dos predicados com efeitos colaterais. Com esta ordem mantida, as metas podem ser executadas em qualquer seqüência.

2.3 A Warren Abstract Machine (WAM)

A WAM é uma técnica de implementação na qual um programa Prolog é compilado em uma linguagem para uma máquina abstrata baseada em pilhas. Normalmente este tipo de implementação possui dois componentes: um compilador e um emulador.

O compilador é responsável por transformar os programas Prolog em instruções WAM, que devem ser executadas pelo emulador. Este emulador faz com que a linguagem Prolog seja mais rápida que Prolog interpretado e mais portável. Além disso, com a utilização da máquina abstrata proposta por Warren, a linguagem Prolog é capaz de obter desempenhos muito próximos ao das linguagens procedurais.

A *Warren Abstract Machine* é normalmente definida através de uma organização de memória, estruturas de dados, registradores e instruções. A figura 2.2 apresenta a organização de memória e os registradores da máquina abstrata.

A memória da WAM está organizada da seguinte forma:

- **área de código**: local onde fica armazenado o programa Prolog compilado;
- ***heap***: na pilha *heap* são armazenados os termos estruturados. Também chamada de *Global*;

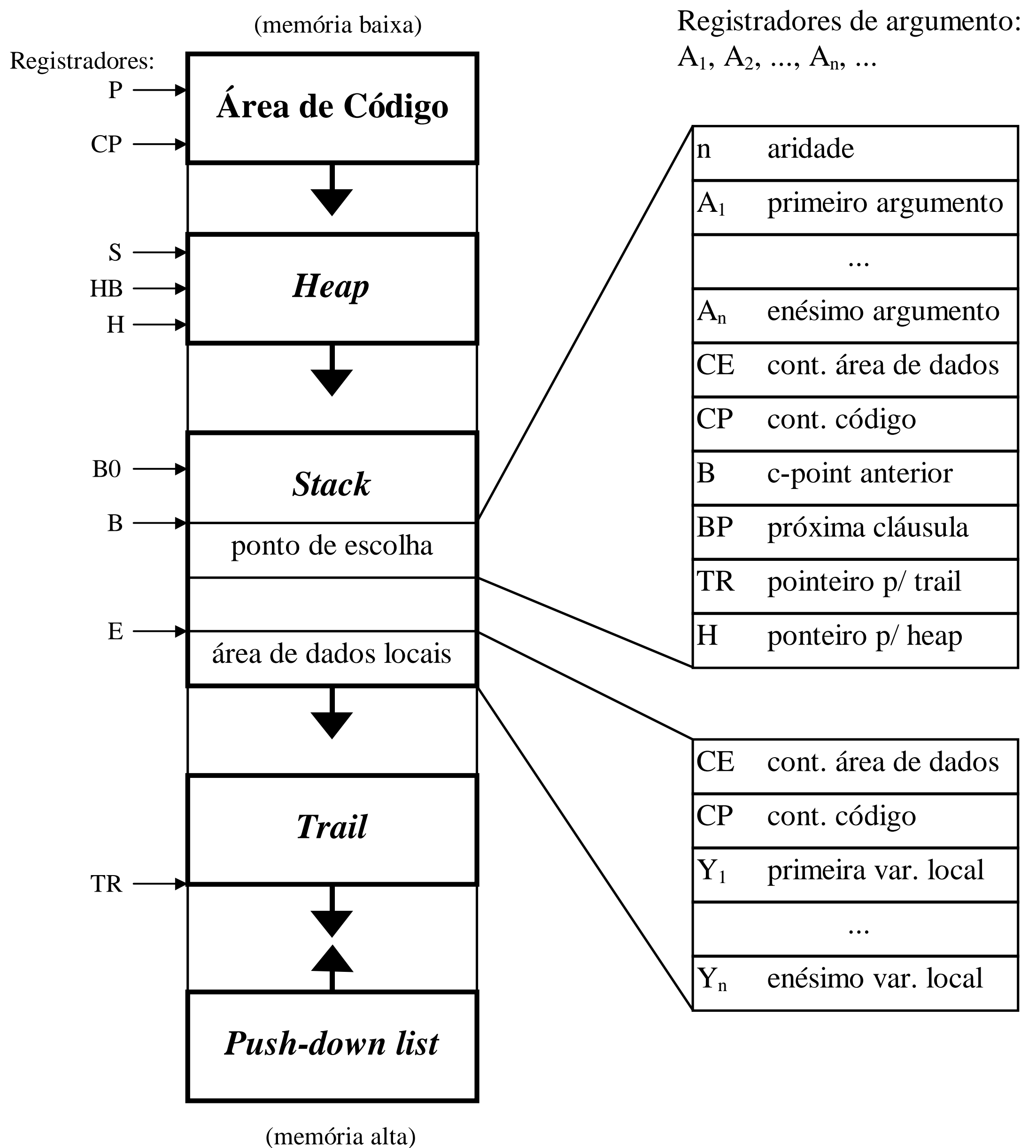

FIGURA 2.2 - Organização de Memória e Registradores da WAM

- ***stack***: a pilha *stack* é usada para controle da chamada de predicados e do retrocesso, sendo responsável também pela alocação de variáveis locais à determinada cláusula. Muitas vezes a *Stack* é chamada de pilha *Local*. Esta pilha armazena basicamente dois tipos de estruturas:

    1. **áreas de dados locais** (*environment*) que contêm as variáveis locais de uma cláusula;

    2. **pontos de escolha** (nodo OU) que contêm informações sobre o estado da WAM, possibilitando o retrocesso.

- ***trail***: é uma pilha que registra as variáveis instanciadas. Isso é necessário para desfazer as ligações durante o retrocesso. A *trail* armazena as variáveis unificadas;

- ***push-down list* (PDL)**: é utilizada pelo algoritmo de unificação. É sempre esvaziada a cada nova unificação.

Existem alguns registradores globais que apontam para áreas de memória da WAM. Dentre estes destacam-se:

- **P**: aponta para o endereço da próxima instrução a ser executada;

- **CP**: contém o endereço da instrução a ser executada após o retorno de uma chamada;

- **H**: aponta para o próximo espaço disponível no *heap*;

- **HB**: aponta para o registrador **H** do ponto de escolha anterior. Este registrador é necessário pois somente as variáveis cujo o endereço é menor do que **HB** devem ser armazenadas na pilha *trail*;

- **S**: aponta para o próximo termo a ser tratado na unificação;

- **B**: aponta para o último ponto de escolha. Em caso de falhas, a computação deve retornar para o estado indicado por **B**;

- **B0**: é utilizado para suportar a implementação do *cut*;

- **E**: indica o endereço da área de dados locais;

- **TR**: aponta para o topo da pilha *trail*;

- $\mathbf{A_1, A_2, ..., A_n}$: armazenam os argumentos de um termo.

Na figura 2.3 são listadas as instruções da WAM. Estas instruções estão agrupadas por grupos funcionais:

- **instruções *put* e *set***: são instruções que normalmente armazenam dados na memória da WAM ou atribuem valores a registradores;

- **instruções de *controle***: são instruções para alocação e desalocação de áreas

na *stack*, chamadas de predicados (*procedures*);

- **instruções de indexação**: determinam qual instrução deve ser executada de acordo com termos (caso sejam constantes, estruturas, etc.);
- **instruções get**: estas instruções normalmente lêem dados da memória da WAM ou de registradores;
- **instruções de unificação**: instruções empregadas na unificação. Podem estar em dois modos: escrita ou leitura. No primeiro os termos são armazenados na *Heap* e no segundo os termos são unificados com aqueles da *Heap*;
- **instruções de escolha**: estas instruções auxiliam na manipulação de pontos de escolha para múltiplas cláusulas;
- **instruções cut**: instruções relacionadas com o operador de corte *cut*.

Instruções *put*

```
put_variable Xn,Ai
put_variable Yn,Ai
put_value Vn,Ai
put_unsafe_value Yn,Ai
put_structure f,Ai
put_list Ai
Put_constant c,Ai
```

Instruções *set*

```
set_variable Vn
set_value Vn
set_local_value Vn
set_constant c
set_void n
```

Instruções de Controle

```
allocate
deallocate
call P,N
execute P
proceed
```

Instruções de Indexação

```
switch_on_term V,C,L,S
switch_on_constant N,T
switch_on_structure N,T
```

Instruções *get*

```
get_variable Vn,Ai
get_value Vn,Ai
get_structure f,Ai
get_list Ai
get_constant c,Ai
```

Instruções de Unificação

```
unify_variable Vn
unify_value Vn
unify_local_value Vn
unify_constant c
unify_void n
```

Instruções de Escolha

```
try_me_else L
retry_me_else L
trust_me
try L
retry L
trust L
```

Instruções *Cut*

```
neck_cut
get_level Yn
cut Yn
```

FIGURA 2.3 - Lista de Instruções da WAM

2.4 Conclusões

A programação em lógica foi apresentada neste capítulo destacando suas principais características. A linguagem Prolog foi introduzida e caracterizado seu modelo de execução. Posteriormente, foi apresentada a *Warren Abstract Machine*, como uma das implementações mais eficientes de Prolog.

3 Programação em Lógica e Paralelismo

Neste capítulo são apresentadas as técnicas para exploração de paralelismo na programação em lógica, bem como os principais sistemas que as implementam. Inicialmente é apresentado o paralelismo OU, em seguida o paralelismo E e finalmente são apresentadas propostas que integram os dois tipos.

Para saber mais sobre as técnicas para exploração de paralelismo consultar [KER 94]. O paralelismo OU é abordado em [YAM 92] e [GEY 91]. O paralelismo E, por sua vez, é descrito em [WER 94a], [YAM 94], [BUE 93] e [HER 91],. Para consultas sobre propostas que integram o paralelismo E/OU ver [COS 96a] e [GUP 95].

3.1 Paralelismo OU

O paralelismo OU consiste na execução concorrente de várias resolventes, ao contrário da execução seqüencial onde estas resolventes seriam computadas sucessivamente por retrocesso (*backtracking*). Examinando a árvore de busca (figura 3.1), é observado que o paralelismo OU consiste na exploração em paralelo de cada ramo da árvore.

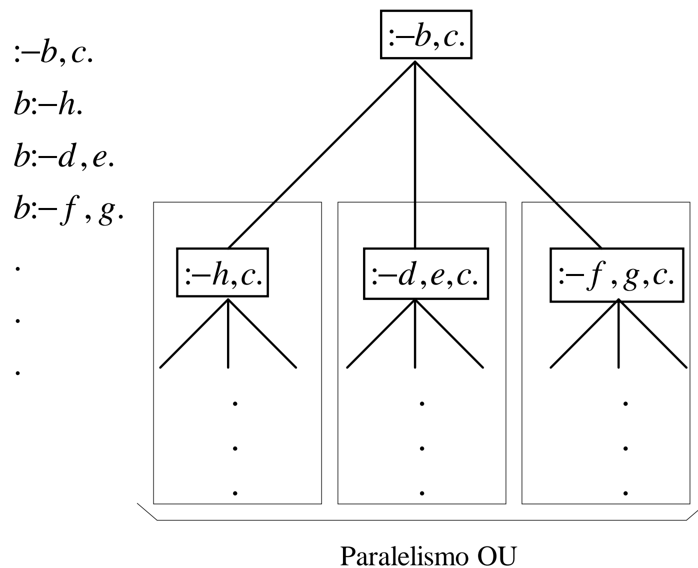

FIGURA 3.1 - Árvore de Busca e Paralelismo OU

No paralelismo OU cada resolvente pode fazer ligações diferentes de variáveis na pilha *trail*. Várias técnicas foram propostas para administrar a execução simultânea destas resolventes:

- **cópia de pilhas**: neste esquema, cada trabalhador mantém uma cópia completa das pilhas no seu espaço de trabalho. O trabalhador livre copia a porção da *Stack* e da *Heap* com o estado da computação anterior ao ponto de escolha (*choice-point*), que contém a alternativa não explorada. Um sistema que emprega esta técnica é o OPERA ([GEY 91]);

- **compartilhamento de pilhas**: partes da *Stack* e do *Heap* são compartilhadas e partes são privadas: os trabalhadores compartilham as partes das pilhas que correspondem às porções da árvore de busca que eles têm em comum. Existe necessidade de uma área distinta para cada ligação em variáveis compartilhadas, sendo uma para cada alternativa. O sistema Aurora é baseado nesta técnica ([WAR 87]);

- **recomputação de pilhas**: neste esquema é possível que trabalhadores explorem independentemente caminhos completos da árvore de busca. As partes das pilhas, que seriam compartilhadas ou copiadas anteriormente, são recomputadas para cada trabalhador. Uma descrição desta técnica pode ser observada em ([CLO 88]).

3.2 Paralelismo E

O paralelismo E consiste na computação simultânea de vários objetivos de uma cláusula. A figura 3.2 mostra que nesta abordagem ocorre a execução paralela de uma ramificação.

Uma vez que as metas são executadas independentemente no paralelismo E, é necessária a compatibilidade das ligações feita por cada ramo. Isto leva a dois modelos de execução diferentes.

No paralelismo E *Independente* somente metas que não compartilham variáveis são executadas em paralelo, enquanto que as outras metas são executadas seqüencialmente (serialização). Já no paralelismo E *Dependente* a execução concorrente de metas que compartilham variáveis é possibilitada através de uma relação produtor/consumidor, isto é, o consumidor aguarda o produtor ter uma ou mais soluções disponíveis (sincronização).

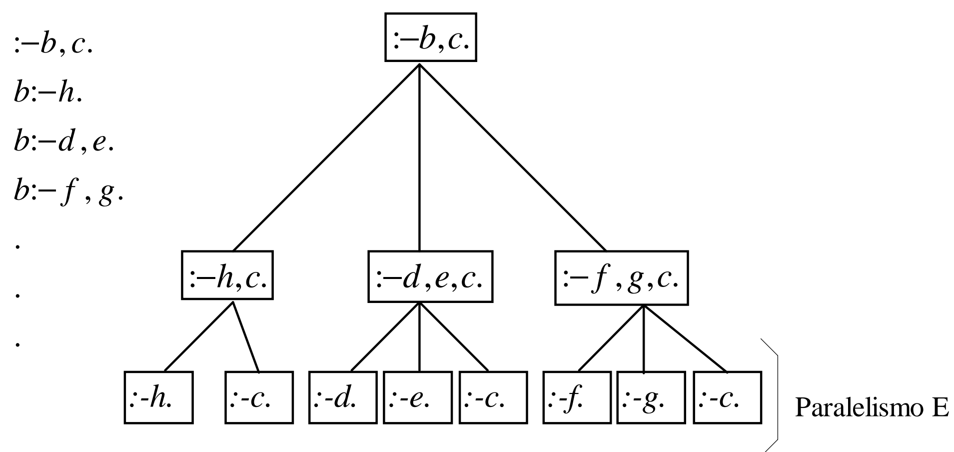

FIGURA 3.2 - Árvore de Busca e Paralelismo E

3.2.1 Paralelismo E Independente

O paralelismo E Independente pode ser de dois tipos: paralelismo E Restrito ou paralelismo E Não-restrito (normalmente chamado de paralelismo E Independente).

O paralelismo E Restrito ocorre quando duas metas não compartilham nenhuma variável, enquanto que o Não-restrito ocorre também em alguns casos que compartilham variáveis, desde que as metas envolvidas não concorram para a ligação destas variáveis.

Normalmente quando variáveis são compartilhadas por metas e estas variáveis estão fechadas (*ground*) no momento da execução das resolventes, o paralelismo E Independente realiza estas metas concorrentemente. Da mesma forma se é constatado que em determinada meta uma variável permanece livre durante sua execução, a paralelização também pode ocorrer.

O maior problema no paralelismo E Independente é a detecção de independência entre metas. Pode-se detectar independência em tempo de execução, em tempo de compilação, ou ainda, parte na compilação e parte na execução.

3.2.1.1 Paralelismo E Restrito

Proposto por DeGroot em 1984, este paralelismo realiza parte da detecção de independência durante a compilação. Programas são compilados em *Conditional Graph Expressions* (CGEs). Estas construções incluem testes simples de independência para serem realizados em tempo de execução.

A CGE tem a seguinte forma geral:

$$(\text{i_cond} \;\; => \;\; goal_1 \;\&\; goal_2 \;\&\; \ldots \;\&\; goal_N$$

$$;goal_1 \;,\;\; goal_2 \;,\;\; \ldots \;,\;\; goal_N \;).$$

Na CGE cada $goal_i$ pode ser um objetivo Prolog normal ou uma outra CGE, enquanto i_cond é uma condição que se satisfeita garante a independência dos objetivos. O significado da CGE é “verifique i_cond; se satisfeita, execute os objetivos em paralelo, caso contrário seqüencialmente”.

3.2.2 Paralelismo E Dependente

O paralelismo E Dependente é normalmente empregado entre metas determinísticas, isto é, metas que unificam com no máximo uma cláusula ([KER 94]). Para exploração deste paralelismo é usado o modelo *goal process* (processo meta).

Um *goal process* é responsável por tentar cada cláusula até encontrar uma que unifique, criando assim processos meta para os objetivos do corpo da cláusula. Devido a este fato, sistemas que utilizam o paralelismo E Dependente tendem a manipular um alto número de processos com pouca granulosidade gerando uma sobrecarga grande na execução.

Para diminuir a sobrecarga na execução, muitas vezes é limitado o número de processos que podem ser criados. Além disso, o custo da troca de processos é

minimizado através do uso de uma política de escalonamento eficiente.

3.3 Paralelismo E/OU

Apesar da exploração de um tipo de paralelismo possibilitar um desempenho satisfatório para a linguagem Prolog, uma vasta quantidade de programas não pode ser beneficiada: é o caso de programas determinísticos utilizando paralelismo OU e programas com grandes buscas usando o paralelismo E ([KER 94]). Segundo Kalé ([KAL 87]), qualquer modelo que pretenda explorar o máximo paralelismo existente nos programas em lógica deverá analisar estas duas fontes.

Pode-se classificar os sistemas que exploram mais de um tipo de paralelismo basicamente em:

- sistemas que exploram o paralelismo E Independente/OU;
- sistemas que exploram o paralelismo E Dependente/OU;
- sistemas que exploram o paralelismo E Dependente, E Independente/OU.

3.4 Principais Sistemas

Nesta seção são apresentadas as principais implementações de Prolog que exploram conjuntamente o paralelismo E/OU. Estas implementações são particularmente interessantes para o trabalho, uma vez que a proposta de escalonamento consiste na exploração conjunta dos dois tipos principais de paralelismo.

3.4.1 Andorra-I

O Andorra-I ([COS 91a] e [COS 91b]) é um sistema que explora o paralelismo E Dependente/OU e estabelece inicialmente a execução de metas determinísticas (metas que unificam com no máximo uma cláusula). O Andorra-I combina duas formas de paralelismo implícito e provê a capacidade das linguagens *flat commited-choice* ([SHA 89]).

Uma *flat language* é caracterizada por utilizar um conjunto fixo de predicados primitivos (principalmente testes de unificação e aritmética). Nestas linguagens uma cláusula pode realizar somente uma computação simples, especificada por uma conjunção de átomos com predicados primitivos, antes de fazer a escolha não-determinística (*commited-choice*).

Os programas no Andorra-I são executados por times de trabalhadores (grupos de processadores). Cada um destes times pode estar executando em uma fase determinística ou não-determinística, a saber:

- **determinística**: todas as metas determinísticas são candidatas para avaliação imediata. Esta fase acaba quando não existem mais metas determinísticas disponíveis ou quando uma meta falha. No primeiro caso, o time passa para a fase não-determinística. No segundo ocorrerá o retrocesso para que um novo nodo OU seja explorado.

- **não-determinística:** se nenhuma meta determinística existe, a meta mais a esquerda (ou uma especificada pelo usuário) é avaliada. Um ponto de escolha é criado, gerando nodo OU. O time atual explora um destes nodos. Caso existam mais times disponíveis, estes times podem explorar os outros nodos gerados.

Em um time de trabalhadores, um processador é o mestre e os outros são escravos, sendo que:

a) cada time trabalha em um nodo OU diferente (na fase não-determinística e no retrocesso só o mestre trabalha);

b) cada trabalhador no mesmo time trabalha no mesmo nodo OU, explorando o paralelismo E Dependente.

A quantidade de times determina o paralelismo OU máximo, enquanto que a quantidade de trabalhadores por time indica o paralelismo E máximo em um time.

O Andorra-I possui dois componentes principais: uma máquina (*engine*) e um pré-processador. A máquina do Andorra executa programas descritos em dois níveis: o código principal representando as definições de cláusulas e o código de determinação, que representa gráficos de decisão para cada procedimento (gerado pelo pré-processador).

A versão seqüencial do Andorra-I é 2.5 vezes mais lenta que o SICStus Prolog (ambiente de programação Prolog disponível comercialmente e amplamente difundido na comunidade acadêmica) ([SWE 95]) e a sobrecarga para o suporte de implementações paralelas é na média de 40%. Na versão paralela do Andorra-I é obtido desempenho melhor com programas que contém paralelismo E e OU, do que com programas que exploram essencialmente um único tipo de paralelismo.

3.4.2 ACE

O *And/Or-parallel Copying-based Execution of Logic Programs* - ACE ([PON 95], [PON 94] e [GUP 93]) é um sistema Prolog que explora o paralelismo OU e E Independente. O sistema suporta a linguagem Prolog de uma maneira elegante pois recomputa metas quando explora o paralelismo E Independente, ao contrário dos outros modelos estudados que compartilham as soluções obtidas paralelamente.

Quando utiliza-se o compartilhamento, é necessário que o conjunto de soluções das metas envolvidas sejam estáticas (quando executadas várias vezes as metas devem gerar as mesmas respostas). Porém isto só é verdadeiro quando as cláusulas utilizam lógica pura. Entretanto se for empregada a recomputação o paralelismo E pode suportar mais facilmente *cuts* e *side-effects*. Além disso, é importante ressaltar que o uso de recomputação não causa diminuição no desempenho do sistema, pois só é utilizada pelo ACE nas mesmas situações empregadas por Prolog.

Para que a implementação da linguagem fosse possível sem alteração da semântica seqüencial de Prolog utilizou-se um modelo abstrato chamado Árvore de Composição (*Composition Tree*), *ou seja,* uma árvore contendo nodos OU e E. Os nodos OU representam as múltiplas cláusulas que unificam com uma meta, enquanto

que os nodos E representam as várias submetas no corpo da cláusula sendo executada em paralelismo E Independente.

No ACE os processadores são organizados em times de forma análoga ao Andorra-I. O paralelismo E Independente é explorado entre processadores de um mesmo time, enquanto que o processamento OU é realizado entre times. Desta forma, um processador pertencente a um time comporta-se como um processador em sistemas puramente E, ao passo que, times de processadores comportam-se como processadores nos sistemas puramente OU.

Até o presente momento só está implementado o componente de paralelismo E Independente do ACE, chamado de &ACE. O &ACE apresenta um *overhead* de aproximadamente 10% sobre o SICStus Prolog. Os tempos de execução obtidos no &ACE são em geral melhores do que os obtidos pelo &- Prolog (sistema que explora o paralelismo E Independente) ([BUE 93]), apesar do fato de que o &ACE contém uma implementação mais completa e permite retrocesso total entre metas paralelas. Além disso, o sistema possui a vantagem de ter sido construído para trabalhar também com o paralelismo OU.

3.4.3 ROPM

O ROPM - REDUCE-OR Process Model ([RAM 92] e [RAM 90]), muitas vezes chamado apenas como REDUCE-OR, é um modelo que combina o paralelismo E Independente e OU para a implementação de Prolog padrão. O sistema é capaz de gerar o paralelismo máximo disponível em programas Prolog. Além disso, todas as soluções de determinado problema são encontradas. O REDUCE-OR é baseado no compartilhamento das soluções obtidas pelo paralelismo E, ao invés da recomputação.

No REDUCE-OR cada regra é representada por um gráfico chamado *Data Join Graph* (DJG). Neste gráfico cada arco (*arc-relation*) denota um literal no corpo de uma cláusula, enquanto que os nodos (*node-relation*) armazenam as soluções parciais até aquele ponto. O objetivo do gráfico é mostrar a dependência dos dados entre literais no corpo de uma cláusula.

O ROPM começa executando a meta de mais alto nível. Após acontecer a unificação, as ligações de variáveis são inseridas no primeiro nodo do DJG e todos os arcos que originam-se neste nodo são disparados.

Em problemas com muitas soluções diferentes o REDUCE-OR cria um processo OU encarregado de encontrar todas as cláusulas que unificam com determinado nome de predicado. Podem ser encontrados dois tipos de cláusulas, a saber: *fatos*, o REDUCE-OR faz a unificação do fato com o predicado e retorna as ligações como resposta; *regras,* o REDUCE-OR tenta inicialmente unificar a meta e a cabeça da cláusula. Se conseguir, um processo é disparado para executar o corpo da cláusula, retornando as ligações ao final.

Nodos em que entram dois ou mais arcos unem as soluções dos arcos para produzir uma solução que é inserida no nodo da junção. A junção de duas soluções obtidas pelo paralelismo E envolve essencialmente a união das duas soluções (compartilhamento), uma vez que os dois nodos E são independentes. Porém para que isto ocorra de forma eficiente alguns algoritmos foram propostos, dentre os quais

destacam-se os algoritmos propostos em [RAM 92].

Quando uma solução está disponível para o último nodo ela é unificada (*back-unified)* com as variáveis da consulta que antecede o nodo, fornecendo assim as ligações para estas variáveis.

O desempenho obtido pelo ROPM quando comparado com o SBProlog ou o Quintus Prolog é considerado satisfatório ([RAM 89]), porém nos testes não foi utilizado o *cut,* uma vez que o REDUCE-OR não o suporta (o *cut* representa em média uma melhora de desempenho de 25% nos programas seqüenciais, em relação aos programas que não o utilizam).

3.4.4 IDIOM

O Integrated Dependent- Independent- and Or-parallel Model, ou simplesmente IDIOM ([GUP 91]), é a primeira estratégia de implementação que explora as três formas de paralelismo. Além de suportar a linguagem Prolog tradicional, o IDIOM suporta as linguagens *flat commited choice*, ao estilo do Andorra-I.

Assim como no sistema Andorra foi utilizado o princípio de execução imediata de metas determinísticas e a execução de metas não determinísticas por paralelismo OU. Além disso, foram empregadas as técnicas de execução paralela de metas independentes e o compartilhamento de soluções.

A combinação de soluções no IDIOM é feita através de um produto cartesiano (do inglês, *cross-product*). Neste esquema, supondo-se duas metas a serem executadas, p e q, não é necessário que para cada nova solução de p, q seja recomputado. O IDIOM explora o paralelismo OU quando uma meta unifica com mais de uma cláusula, e as resolventes resultantes são executadas em paralelo.

O sistema IDIOM trabalha com CGEs para expressar o paralelismo E Independente. Estas CGEs podem ser obtidas em tempo de compilação através de um pré-processador. A execução do IDIOM é constituída de três fases principais:

- fase de paralelismo E Dependente ou *Dependent And-Parallel phase (*DAP);
- fase de paralelismo E Independente ou *Independent And-Parallel phase* (IAP);
- fase de paralelismo OU ou *Or-Parallel phase* (ORP).

Inicialmente o IDIOM executa na fase DAP todas as metas determinísticas em paralelo (incluindo aquelas dentro das CGEs). Quando não existirem mais metas determinísticas a serem executadas, a meta mais a esquerda é examinada para determinar se é uma meta com CGE. Em caso afirmativo o IDIOM entra na fase IAP, caso contrário entra na fase ORP.

Na fase ORP, as várias alternativas de um predicado são executadas em paralelismo OU. Após ocorrer a unificação da cabeça da cláusula em uma destas alternativas, o IDIOM retorna para a fase DAP.

Na fase IAP, por sua vez, ocorre a avaliação da CGE. Se verdadeira, as metas são executadas em paralelismo E Independente e será iniciada então a fase ORP, processando a meta mais a esquerda em paralelismo OU. Se a avaliação da CGE resultar falsa, a fase ORP é iniciada imediatamente para que a meta mais a esquerda seja executada em paralelismo OU.

Quando a execução de uma meta nas fases ORP ou DAP tiver sucesso e se a meta for componente de uma CGE, é feito o cruzamento das soluções incrementalmente. A continuação da execução retorna para a fase DAP.

No IDIOM os vários componentes de uma CGE são avaliados em paralelo. Cada componente pode ter mais de uma solução, encontrada por paralelismo OU. Quando um processador produz uma solução para um problema, inicialmente ele verifica se ao menos uma solução foi encontrada pelos outros componentes. Se não, ele começa a trabalhar em um componente não tentado. Se não existem componentes não tentados e nenhuma solução foi encontrada, o processador simplesmente faz o retrocesso e tenta achar outra solução. Porém, se ao menos uma solução foi encontrada para todos os componentes, é feito o *cross-product* de todas as soluções.

A execução do paralelismo E Dependente é similar ao Andorra-I. Fora da fase DAP, o processo escravo pode mudar seu modo e ajudar a realizar trabalho E Independente e OU.

Considerando a bibliografia atualmente disponível não está operacional uma implementação do IDIOM até o presente momento. Sendo assim, o estudo foi feito com base na proposta teórica.

3.5 Comparações entre os Principais Sistemas

Nesta seção são apresentadas as comparações entre os modelos estudados. As comparações são na sua maioria qualitativas e estão dividas de acordo com características consideradas importantes para um modelo de exploração de paralelismo E/OU na programação em lógica.

3.5.1 Semântica de Prolog seqüencial

Uma característica que pode ser avaliada quanto aos sistemas de Prolog paralelos é o fato de seguir a semântica de Prolog. Muitos sistemas estudados, mesmo suportando a linguagem Prolog padrão, não executam programas da mesma forma. Dentre os sistemas estudados apenas o ACE mantém a semântica de Prolog. O Andorra-I e o IDIOM mudam esta semântica pois executam metas determinísticas primeiro.

Outra questão importante relativa à semântica de Prolog ocorre em sistemas que exploram o paralelismo E Independente. Sistemas como o ACE recomputam as metas à direita, de forma análoga à linguagem Prolog padrão. O REDUCE-OR e o IDIOM, por sua vez, utilizam uma semântica diferente de Prolog pois ao invés de recomputar as metas de determinada cláusula, eles compartilham as soluções.

Em geral para duas metas independentes G_1 e G_2 com m e n soluções respectivamente, o custo de computação pode diminuir de $m * n$ para $m+n$ quando compartilham-se as soluções através do produto cartesiano. Entretanto, para metas com

pequenas granulosidades, o ganho do compartilhamento não compensa o custo do produto cartesiano. Além disso, a recomputação é melhor para metas independentes que contém *side-effects* e predicados extra-lógicos. O uso de compartilhamento ou recomputação não altera os resultados do programa nem a semântica a nível de usuário.

3.5.2 Linguagem Suportada

Outro fator que deve ser observado é o fato de suportar as características extra-lógicas (*pruning operators*, *side-effects*, etc.). Dentre os sistemas estudados o REDUCE-OR é o único a não suportar estas características.

Além disso, o Andorra-I e o IDIOM têm a opção de suportar também as características das *flat commited choice languages*.

3.5.3 Arquiteturas Suportadas

Realizada uma análise de todos os sistemas estudados, é constatado que todos eles foram projetados para multiprocessadores (excluindo-se o IDIOM que não foi implementado). O Andorra-I e o ACE suportam arquiteturas multiprocessadoras, como o *Sequent Symmetry*. Já o REDUCE-OR além de suportar estas arquiteturas, suporta também multicomputadores (arquiteturas com memória distribuída), como o Intel iPCS/2.

Alguns dos sistemas estudados ([COS 96a]) podem ser executados em estações de trabalho, porém seqüencialmente.

3.5.4 Baseados em Processos versus Baseados em Enlaces (*Threads*)

Os sistemas para exploração da programação em lógica paralela podem ser implementados baseados em processos e baseados em enlaces ou processos leves (*threads*).

Em sistemas baseados em processos, como o REDUCE-OR, um processo é criado para cada meta encontrada durante a execução. Estes processos comunicam-se para a troca de ligações e informações de controle. Por outro lado, em sistemas baseados em enlaces múltiplas *threads* são criadas e executadas em paralelo. O ACE, o ANDORRA-I e o IDIOM são exemplos de sistemas que utilizam enlaces.

Os sistemas baseados em *threads* são mais adequados quando existe mais de um processo por processador, ou ainda, quando os processos de um mesmo processador necessitam compartilhar informações.

3.5.5 Escalonamento

O escalonador é um dos componentes cruciais para um bom desempenho de sistemas Prolog paralelos. Por exemplo, em sistemas que suportam *cut*, o escalonador deve optar por realizar primeiro trabalhos que não estão no escopo de nenhum *cut*, uma vez que este *pruning operator* pode fazer com que o trabalho realizado possa se tornar inútil.

Afim de explorar o paralelismo E e OU simultaneamente, Andorra-I e ACE

realizam escalonamento em duas fases. Com isso, aproveitam os escalonadores já existentes para sistemas que exploram um tipo de paralelismo. Estes sistemas dividem os processadores em times e devem ser capazes de decidir quando processos devem migrar de um time para outro ou quando deve criar novos times. Uma vantagem importante do Andorra-I é que ele pode usar um dentre os vários escalonadores disponíveis para o Aurora ([YAM 92]).

O escalonamento por ser tema de especial importância para o trabalho é tratado no capítulo 5. Neste capítulo, a seção 5.2 trata do escalonamento na programação em lógica.

3.5.6 Suporte à Análise em Tempo de Compilação

A análise em tempo de compilação é outra característica importante para a eficiência de um sistema paralelo para a programação em lógica. Ferramentas para este tipo de análise, baseadas em Interpretação Abstrata foram empregadas em todos os sistemas estudados que foram implementados.

O ACE faz análise de *sharing* e *freeness* para geração automática de CGEs em tempo de compilação. Estas características foram herdadas do sistema &-Prolog. O Andorra-I utiliza análise em tempo de compilação para detecção de determinância de metas. O ROPM, por sua vez, faz análise de dependência durante a compilação gerando os *Data Join Graphs*.

3.5.7 Overhead do Paralelismo

Dentre os sistemas aqui apresentados, o &ACE e o REDUCE-OR apresentam *overheads* pequenos se comparados a ambientes Prolog seqüencial amplamente difundidos, como o SICStus Prolog e o Quintus Prolog.

O Andorra-I apresenta uma sobrecarga de 40% sobre o SICStus, enquanto que no &-ACE a sobrecarga é de apenas 10%. O REDUCE-OR quando comparado ao Quintus Prolog apresenta uma sobrecarga considerada satisfatória pelos seus criadores.

3.5.8 Desempenho Geral dos Sistemas

O desempenho geral dos sistemas estudados foi considerado satisfatório. Muitas vezes foram obtidos *speedups* próximos ao limite teórico máximo, conforme pode ser observado em [PON 94], [YAN 93] e [RAM89].

Neste trabalho não foi contemplada uma comparação de desempenho entre os sistemas relacionados, uma vez que os testes foram obtidos de trabalhos publicados e não apresentavam os programas fontes. Configurações de arquiteturas e número de processadores também foram diferentes. Além disso, não foi possível realizar os testes no Instituto de Informática da UFRGS, uma vez que durante a realização do trabalho nenhuma arquitetura multiprocessadora suportada pelos sistemas estava disponível. Por sua vez, os esforços foram concentrados na análise qualitativa dos sistemas.

3.5.9 Quadro Comparativo

Neste item é feita uma comparação resumindo as principais características dos

sistemas estudados para a exploração do paralelismo E/OU na programação em lógica. A tabela 3.1 apresenta o respectivo quadro comparativo.

TABELA 3.1 - Quadro Comparativo dos Sistemas que Integram o Paralelismo E/OU

Características:	Andorra-I	ACE	REDUCE-OR	IDIOM
Paralelismo OU	●	●	●	●
Paralelismo E Dependente	●	○	○	●
Paralelismo E Independente	○	●	●	●
Mantém semântica de Prolog seqüencial	○	●	○	○
Recomputação de metas	N/A	●	○	○
Compartilhamento de soluções	N/A		●	●
Suporte a predicados extra-lógicos	●	●	○	N/A
Suporte à linguagens *flat commited choice*	●	○	○	●
Suporte à arquitetura com memória compartilhada	●	●	●	N/A
Suporte à arquitetura com memória distribuída	○	○	●	N/A
Sistemas baseados em processos	○	○	●	○
Sistemas baseados em enlaces	●	●	○	●
Escalonamento em duas fases	●	●	○	○
Suporte à análise em tempo de compilação	●	●	●	N/A

● → implementa; ○ → não implementa; N/A → Não aplicável

3.6 Conclusões

Este capítulo mostrou as principais formas de exploração do paralelismo na programação em lógica, quais sejam: paralelismo OU e paralelismo E. Particularmente, foram enfatizados sistemas que combinam os dois tipos de paralelismo. Após a descrição dos sistemas foram apresentadas comparações entre os sistemas estudados.

4 OPERA e GRANLOG

Este capítulo apresenta o projeto OPERA, no qual o trabalho encontra-se inserido. Posteriormente, é descrito também o GRANLOG.

Para saber mais sobre o OPERA consultar [WER 94a], [WER 94b], [YAM 94] e [GEY 91]. Maiores informações sobre o paralelismo OU no ambiente OPERA podem ser encontradas no projeto PloSys que é descrito em [MOR 96]. Para estudos aprofundados sobre o GRANLOG consultar [BAR 96a], [BAR 96b], [BAR 96c], [BAR 95a] e [KAY 95].

4.1 OPERA OU

O ambiente OPERA foi projetado para exploração implícita do paralelismo na programação em lógica. O sistema foi concebido inicialmente para explorar o paralelismo OU, em uma arquitetura multiprocessada com memória distribuída baseada em Transputers.

O modelo é multiseqüencial e executa o Prolog padrão, incluindo *cut* e alguns predicados com *side-effects* (seção 2.2.3) , como os predicados de entrada e saída. Cada trabalhador tem uma TWAM (*Transputer Warren Abstract Machine*) e usa uma cópia local do programa Prolog. O paralelismo OU é explorado a partir da cópia de pilhas (ver seção 3.1).

Posteriormente, o projeto OPERA OU foi estendido permitindo seu uso em outras arquiteturas com memória distribuída. Além disso, permite a utilização de alguns outros predicados com *side-effects* (como predicados *data-base*). Este modelo estendido recebeu o nome de PloSys (*Parallel Logic System*).

4.1.1 PloSys

O PloSys é um sistema que explora o paralelismo OU na execução de Prolog em arquiteturas com memória distribuída. O ambiente foi implementado de forma a permitir o uso de *cut* e *side-effects*, e com a preocupação de manter o número de mensagens entre processadores reduzido.

Atualmente, o PloSys utiliza uma política de escalonamento centralizada, na qual um processador é responsável por essa tarefa (ver capítulo 5). Estudos vem sendo feitos para a utilização de outras políticas.

Um programa Prolog é executado em paralelo por vários trabalhadores, os quais contêm: uma máquina de inferência e um enlace exportador (figura 4.1). A máquina de inferência é uma máquina abstrata seqüencial baseada na WAM (seção 2.3). A importação é feita pela própria máquina de inferência a partir do momento que este recebe a mensagem de autorização do escalonador.

No PloSys trabalhadores ociosos procuram trabalhadores que possuam uma carga maior que determinado valor preestabelecido (*threshold).* Isso é feito através do escalonador, que mantém um estado global do sistema. Periodicamente, os trabalhadores enviam o seu estado ao escalonador. Uma exportação de trabalho pode

falhar se o trabalhador importador não estiver mais sobrecarregado.

Um trabalhador ocioso importa trabalho de um ativo copiando o estado do trabalhador exportador até o ponto de escolha mais antigo (mais acima na árvore de busca) do programa Prolog em execução. Após a cópia, o trabalhador faz o desligamento das ligações inválidas (feitas pelo exportador depois do *choice-point*) usando a pilha *trail*. E então, simula uma falha para executar a alternativa mais recente não tentada, dentre as alternativas copiadas.

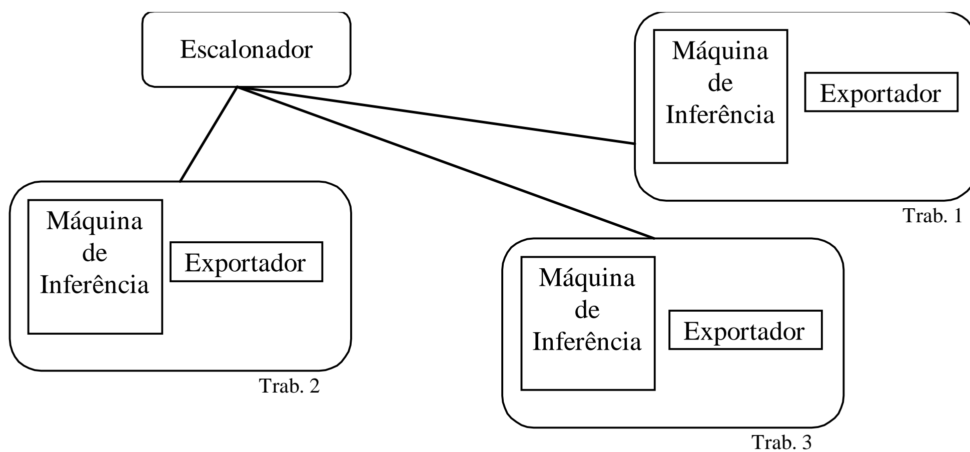

FIGURA 4.1 - Arquitetura de Processos do PloSys

O PloSys trata os predicados com *side-effects* mantendo a ordem de execução seqüencial dos programas Prolog. Uma lista é empregada para guardar a posição relativa dos trabalhadores na árvore de execução. Quando um trabalhador exporta trabalho, é armazenado na lista o trabalhador importador. Trabalhadores disponíveis são removidos da lista, até receberem novas transferências.

O gerenciamento dos *pruning-operators* é feito em duas fases. Inicialmente, é identificada na compilação a natureza especulativa de alguns pontos de escolha. Na execução, é controlada a exportação de trabalho especulativo (predicados que contém operadores de corte). Este controle é feito através de informação passada aos trabalhadores importadores a respeito dos predicados enviados (se possuem ou não predicados com *cut*). Quando o retrocesso ocorre, é verificada a pilha de *choice-points* e mensagens de invalidação são enviadas aos trabalhadores executando predicados no escopo de um *prunig-operator*.

Existe um protótipo do PloSys que foi implementado utilizando a WAMCC ([DIA 94]) e o ambiente de programação paralela Athapascan ([BRI 96]). A WAMCC é um compilador que traduz programas Prolog para instruções da máquina abstrata de Warren e então para a linguagem C. Ela foi utilizada pois sua máquina de inferência não é muito complexa, e pode ser modificada. Além disso, a WAMCC possui um bom desempenho (1.5 vezes mais rápida do que programas emulados no SICStus) e apresenta uma portabilidade razoável ([MOR 96]).

4.2 OPERA E

O modelo OPERA E é baseado no paralelismo E Restrito (seção 3.2.1.1), executando em paralelo os literais de cláusulas que não apresentam variáveis comuns. Alguns literais que compartilham variáveis poderão ser executados em paralelo se e somente se no momento da execução, as variáveis comuns estiverem fechadas (instanciadas com valores definidos). Os literais que contiverem estruturas e listas como variáveis são executados seqüencialmente, em função do custo para verificação de dependências ([YAM 94]).

O sistema foi concebido de forma a ser passível de integração com o OPERA OU, pois opera em arquiteturas multiprocessadas com memória distribuída, explora o paralelismo implícito em Prolog e utiliza uma máquina abstrata baseada na WAM.

O OPERA E é composto por um processo mestre e vários trabalhadores. O Mestre é responsável pela gerência da arquitetura, controlando o ambiente de execução, criando trabalhadores e escalonando serviços. Os trabalhadores são compostos por dois processos: o processo *Solver* e o *Spy*. O *Solver* é a máquina abstrata responsável pela computação do programa (WAM - E), enquanto que o *Spy* está encarregado de manter o Mestre informado sobre o estado dos trabalhadores. A figura 4.2 apresenta a arquitetura de processos de OPERA E.

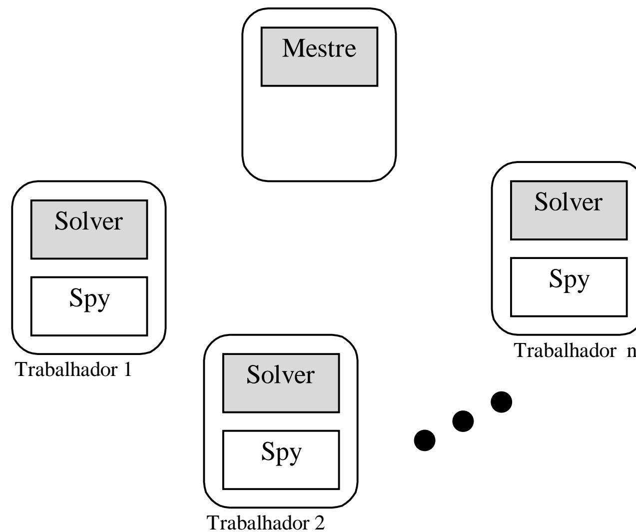

FIGURA 4.2 - Arquitetura de Processos do OPERA E

No OPERA E, assim como no PloSys, o programa a ser executado está disponível na memória local de cada processador onde ocorrerá a execução paralela. O processamento começa com uma consulta do usuário sendo atendida por um trabalhador escolhido como principal.

A avaliação da carga de trabalho paralelizável é feita dinamicamente. Cada trabalhador mantém duas pilhas: uma paralela e outra seqüencial. A primeira contém expressões que devem ser executadas em paralelo, enquanto a segunda contém expressões que serão executadas seqüencialmente. Se a pilha seqüencial esvaziar, o

trabalhador retirará expressões da pilha paralela para execução. Se a pilha paralela também estiver vazia, o trabalhador irá informar ao escalonador que está disponível.

O OPERA E é baseado na heurística de associar carga ao número de literais na pilha paralela. A partir da carga, os trabalhadores podem estar em um dos estados abaixo:

- **idle:** não existe trabalho para ser executado. O trabalhador está aguardando uma autorização do mestre para importar trabalho;

- **quiet:** o trabalhador está ativo, mas não dispõe de trabalho em volume suficiente, que justifique a exportação;

- **overloaded:** o trabalhador está ativo e o volume de trabalho justifica uma exportação.

Um protótipo do OPERA E foi implementado em uma rede de estações de trabalho SUN. O protótipo foi desenvolvido utilizando a linguagem C e o ambiente PVM - *Parallel Virtual Machine* ([GEI 94]). O protótipo encontra-se atualmente em fase de testes.

4.3 GRANLOG

O GRANLOG (*Granularity Analyzer for Logic Programming*) é um modelo para análise automática de granulosidade na programação em lógica. O modelo realiza uma análise estática, gerando informações que podem ser utilizadas dinamicamente para tarefas como o escalonamento.

A análise de granulosidade determina o tamanho dos grãos, isto é, a complexidade dos módulos que deverão ser executados seqüencialmente em um único processador. Devido a isso, devem ser realizadas considerações sobre dependências, complexidade dos grãos e custos envolvidos na paralelização. Essa análise torna-se especialmente importante para o projeto, uma vez que um dos objetivos é a busca pelo paralelismo implícito na programação em lógica.

O GRANLOG é composto basicamente por três módulos, a saber: Analisador Global, Analisador de Grãos e Analisador de Complexidade. A figura 4.3 mostra a organização básica do modelo.

O primeiro módulo analisa o programa em lógica e determina os modos (direcionalidade do fluxo de dados vinculado a um argumento), tipos (especificação que limita os valores que determinado argumento pode assumir), medidas (tamanho dos argumentos) e dependências (dependências de dados existentes entre os literais de cada uma das cláusulas de um programa em lógica).

Com as informações de modos, tipos, medidas e dependências o Analisador de Grãos determina os grãos em potencial do programa e informações relacionadas a estes. Três tipos de grãos podem ser gerados: grão meta (utilizado para exploração do paralelismo E), grão metas (utilizado para exploração do paralelismo E, contendo mais de uma meta) e grão cláusula (utilizado para exploração do paralelismo OU).

O último módulo, o Analisador de Complexidade, a partir das informações dos grãos determina a complexidade OU e a complexidade E de um programa em lógica. A complexidade OU consiste na determinação da complexidade da cláusula que cria o ramo da árvore de busca e a complexidade da resolvente pendente no momento da criação do ramo. Por sua vez a complexidade E determina os recursos computacionais que serão consumidos durante a execução de uma meta.

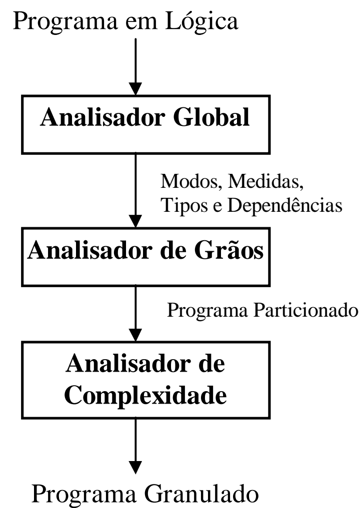

FIGURA 4.3 - Organização do GRANLOG

A unidade básica de medida de complexidade é o número de resoluções executadas numa chamada de procedimento. Este número indica a quantidade de chamadas realizadas durante a execução de Prolog. Como o modelo é baseado no estudo do pior caso é determinado por tanto, a quantidade máxima de resoluções. O modelo GRANLOG utiliza esta unidade de medida, pois foi constatado que durante a execução de um procedimento em um determinado ambiente, o tempo de execução de uma resolução mantém-se praticamente constante ([BAR 96a]).

A figura 4.4 apresenta um programa em lógica granulado que foi gerado pelo GRANLOG. O programa apresentado é para cálculo da série de *fibonacci*.

```
:- granularity(e1,1.45*exp(1.62,$1)+0.55*exp(-0.62,$1)-1).
:- out_size(r1,[0,0]).
:- out_size(r2,[0,1]).
:- out_size(r3,[$1,0.45*exp(1.62,$1)-0.45*exp(-0,62,$1)]).

:- grain_clause(fibo/2,g1,[i,o],[void,int],1,1,r1).
:- grain_clause(fibo/2,g2,[i,o],[void,int],1,1,r2).
:- grain_clause(fibo/2,g3,[i,o],[int],11,e1,r3).
:- grain_goal(fibo/2,g3_4,[i,o],[int],[int],5, 5, e1,r3).
:- grain_goal(fibo/2,g3_5,[i,o],[int],[int],0, 5, e1,r3).

fibo(0,0).
fibo(1,1).
fibo(M,N) :-      M>1, M1 is M-1, M2 is M-2,
                  fibo(M1,N1) & fibo(M2,N2), N is N1 + N2.
```

FIGURA 4.4 - Programa Granulado Gerado pelo GRANLOG

As quatro últimas linhas da figura apresentam o programa Prolog para o cálculo da série de *fibonacci*. As linhas restantes mostram as anotações de granulosidade. A anotação *granularity* contém a expressão de complexidade que será utilizada para determinar a granulosidade dos grãos meta. A declaração *out-size*, por sua vez, registra a relação empregada para determinar o tamanho das saídas em função das entradas. Por fim, *grain-clause* e *grain-goal* armazenam os modos, tipos e medidas das cláusulas e das metas respectivamente.

Além das expressões de complexidade, o GRANLOG infere estaticamente as complexidades simplificadas das metas e das cláusulas (representadas em negrito nas anotações *grain-clause* e *grain-goal*). Estas anotações não consideram o número de chamadas recursivas, uma vez que somente durante a execução do programa em lógica esse número é conhecido. Apesar de mais imprecisas que as expressões de complexidade, as complexidades simplificadas são interessantes pois sua utilização não introduz *overhead* durante a execução do programa.

O protótipo implementado do GRANLOG está atualmente disponível na Rede de Computadores SUN do Instituto de Informática da UFRGS. O programa dedica-se a determinação de granulosidade para o paralelismo E independente.

Como analisador de complexidade E do GRANLOG é utilizado o sistema CASLOG ([DEB 93]). Este sistema analisa programas em lógica e gera expressões de complexidade e relações de tamanho entre entradas e saídas de procedimentos para a exploração do paralelismo E. O GRANLOG realiza as adaptações necessárias para a compatibilização das informações geradas pelo CASLOG.

4.4 Conclusões

Este capítulo introduziu o projeto no qual o presente trabalho está inserido. Inicialmente foi apresentado o OPERA OU, em particular o PloSys, e a seguir o OPERA E. Finalmente, o GRANLOG foi descrito. Foram apontadas as principais características de cada um dos sistemas, bem como a sua situação atual.

5 Escalonamento

Este capítulo apresenta um estudo sobre escalonamento, traçando conceitos básicos e justificando o modelo de escalonador proposto no presente trabalho. Para saber mais sobre escalonamento de modo geral consultar [ELR 94], [KUN 91], [CAS 88], [ZHO 88] e [EAG 86]. O escalonamento na programação em lógica é tratado em [SHE 97], [GUP 95], [KER 94] e [CIE 91]. As principais propostas de escalonamento distribuído encontram-se em [DAN 96a], [LOD 96], [EVA 94], [SER 94], [SON 94] e [KUN 91].

5.1 Conceitos Básicos

O escalonamento consiste em atribuir processos a processadores e determinar em que ordem estes processos serão executados. Ele é de extrema importância para sistemas paralelos e distribuídos, podendo ser considerado um dos problemas mais desafiantes nesta área ([ELR 94]).

Segundo Casavant ([CAS 88]), o escalonamento pode ser visto como um recurso para gerenciar recursos. Este gerenciador é basicamente um mecanismo ou uma política usada para controlar o acesso e o uso de vários recursos por seus vários consumidores. Duas propriedades que devem ser consideradas no escalonamento: a satisfação do consumidor em relação a como o escalonador manipula os recursos em questão (desempenho); e a satisfação do consumidor em relação à dificuldade ou custo de acesso ao gerenciamento de recursos em si (eficiência).

Existem vários tipos de escalonamentos e algumas taxonomias foram propostas ([ELR 94], [MUL 93] e [CAS 88]). Nenhuma destas taxonomias abrange todas as características que um escalonador pode possuir, devido a isso existem características que serão descritas a parte. Dentre as taxonomias estudadas, a proposta por Casavant ([CAS 88]) é a mais aceita e completa. A figura 5.1 mostra a taxonomia de escalonamento proposta pelo mesmo.

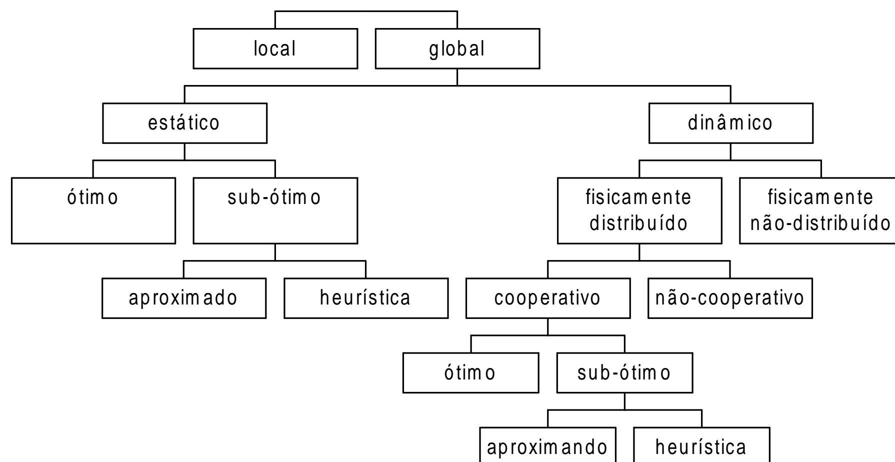

FIGURA 5.1 - Taxonomia de Escalonamento de Casavant

Na taxonomia apresentada o escalonamento local está relacionado com a ordenação de processos em um único processador. Neste trabalho será enfocado o escalonamento global, o qual consiste em decidir num ambiente com vários processadores qual destes irá executar a tarefa.

O escalonamento estático é aquele que é resolvido antes da execução das tarefas de maneira determinística. Quando é possível determinar todas as informações destas tarefas (dependência entre tarefas, custo de comunicação, custo de execução, etc.) o escalonamento pode ser ótimo, em caso contrário sub-ótimo. Nesta última categoria, é possível optar pela busca de uma 'boa' solução ao invés de procurar no espaço de soluções inteiro (aproximado). Os modelos heurísticos representam a categoria de algoritmos estáticos que assumem um conhecimento *a priori* sobre os processos e as informações do sistema.

Muitas vezes as informações a respeito das tarefas e do sistema não podem ser obtidas antes da execução, levando à utilização de um escalonamento dinâmico. Dependendo se o escalonador deve residir fisicamente em um único processador ou deve estar fisicamente distribuído entre processadores, classifica-se os escalonadores respectivamente em distribuídos e não-distribuídos (ou centralizados).

Os escalonadores distribuídos são divididos entre aqueles que cooperam entre si e aqueles nos quais cada processador individual toma decisões independentemente das ações tomadas pelos outros. Os escalonadores cooperativos são classificados de forma análoga aos estáticos.

Existem várias características consideradas importantes que não estão presentes na classificação proposta anteriormente. Abaixo estão as principais encontradas na literatura:

- **adaptável e não-adaptável:** a diferença entre elas é o fato de que os adaptáveis mudam dinamicamente seu mecanismo de controle através do uso de parâmetros do sistema, enquanto que os não-adaptáveis não alteram seu algoritmo;

- **alocação única e realocação dinâmica:** na primeira opção os trabalhos são alocados uma única vez a determinado processador, enquanto que na segunda podem migrar várias vezes antes de serem executados. Claramente, a segunda opção traz uma sobrecarga maior ao sistema;

- **preemptivo e não-preemptivo:** em um sistema preemptivo uma tarefa pode ser interrompida, uma vez que iniciou sua execução, enquanto que no não-preemptivo isto não é possível;

- **sistema de várias aplicações e de única aplicação:** existem sistemas onde o escalonador está trabalhando com uma aplicação específica, e portanto seu objetivo é minimizar o tempo total para realizar esta aplicação. Por outro lado, podem existir casos onde o escalonamento é para diversas aplicações e deve ser minimizado o tempo médio para resolução de cada aplicação envolvida;

- **balanceamento de carga:** esta política de escalonamento usa a filosofia de

que ser justo com o uso dos recursos de *hardware* do sistema é bom para os usuários. A idéia básica é tentar balancear a carga em todos os processadores de maneira que executem a uma mesma taxa. Os nodos agem em conjunto, removendo trabalho de nodos muito carregados para nodos pouco carregados. Esta solução é mais eficiente quando os nodos são homogêneos, pois isso permite que todos os nodos saibam a estrutura do outro;

- **compartilhamento de carga:** esta política de escalonamento tem por objetivo distribuir a carga do sistema de nodos muito carregados para nodos pouco carregados no sistema. A idéia básica é aumentar o desempenho do sistema distribuindo a carga através dos nodos. Não existe a necessidade de que a taxa de execução dos nodos seja semelhante, e sim, de que não existam simultaneamente nodos sobrecarregados e nodos pouco carregados. Esta solução é importante para eficiência tanto em sistemas homogêneos como heterogêneos.

No projeto de um escalonador de tarefas uma das decisões mais importantes é a escolha entre uma política estática ou dinâmica. As políticas estáticas não utilizam informações sobre o estado do sistema para escalonar tarefas. Algoritmos estáticos assumem informações sobre os tempos médios de execução de tarefas e necessidades de intercomunicação entre todas as tarefas ([EAG 86]). Algoritmos dinâmicos, entretanto, decidem a distribuição de carga no sistema através do estado atual do sistema ou mais recente. Como o estado do sistema sofre alterações dinamicamente, políticas dinâmicas costumam possibilitar melhoras significantes de desempenho ([DAN 96a]).

Dois componentes importantes em um escalonamento dinâmico são a política de transferência e a política de localização. A política de transferência determina quando uma tarefa deve ser processada localmente ou remotamente, enquanto que a política de localização determina o nodo para o qual a tarefa, selecionada para possível execução remota, deve ser enviada ([DAN 96a]). Normalmente, políticas de transferência empregam algum tipo de índice de carga (*threshold*) para determinar quando o nodo está muito carregado ou não.

A escolha de um bom índice de carga tem um efeito considerável no desempenho do sistema, e por isso, vários índices de carga foram propostos ([KUN 91] e [ZHO 88]). Índices como tamanho da fila de execução da UCP, tempo médio da fila da UCP, utilização da UCP, quantidade de memória disponível, etc. Apesar de intuitivamente parecer óbvio que o uso de índices sofisticados melhora o desempenho, via de regra não é o que ocorre. Quanto mais cálculos e parâmetros são utilizados, maior é a sobrecarga do sistema, mais difícil é a obtenção e a distribuição do índice ([KUN 91]). Dentre os índices possíveis é um consenso na literatura de que o número de tarefas na fila de espera da UCP é o melhor índice para o escalonamento de tarefas ([DAN 96a], [LOD 96], [KUN 91] e [EAG 86]).

A maior parte dos sistemas dinâmicos são da categoria fisicamente distribuído, cooperativos, sub-ótimos e heurísticos ([EVA 92] e [CAS 88]). Entretanto, o escalonamento dinâmico fisicamente não-distribuído, também denominado centralizado, apresenta importantes vantagens para multicomputadores. A figura 5.2 apresenta o modelo de escalonamento centralizado.

As políticas centralizadas são mais eficientes que as distribuídas para ambientes

com um número razoável de processadores (até cem processadores) em uma rede local (velocidades de comunicação idênticas entre processadores). Políticas centralizadas apresentam problemas de escalabilidade, não tolerância a falhas, perda de paralelismo, localidade da atividade de escalonamento e informações globais imprecisas ([HEI 96]).

A escalabilidade diz respeito à capacidade de um sistema paralelo suportar quantidades diferentes de processadores. Normalmente, políticas centralizadas apresentam problemas de "gargalo" quando são submetidas a vários processadores ([KUC 90]). Escalonadores centralizados não são tolerantes a falhas, uma vez que qualquer falha que ocorra no processador o qual mantém a execução da tarefa escalonadora, causa a interrupção total da aplicação ([DAN 96a]).

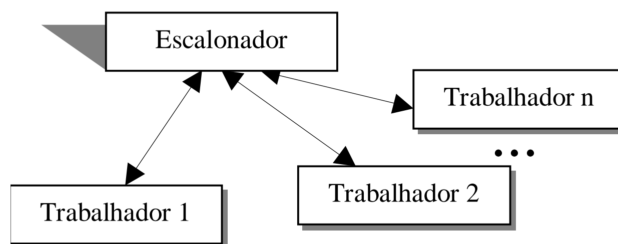

FIGURA 5.2 - Escalonamento Centralizado

Em alguns sistemas com escalonamento centralizado não é possível explorar todo o paralelismo disponível. Isso é devido ao fato de que um único escalonador não consegue detectar e distribuir de forma eficiente todo o paralelismo do problema. O que causa a não detecção do paralelismo é a localidade da atividade de escalonamento (um único processo é responsável).

Como vantagens dos sistemas centralizados são citadas a simplicidade, maior eficiência para um número razoável de processadores, maior eficiência para sistemas com memória compartilhada e menor custo de comunicação ([DAN 96a] e [YAM 94]).

5.2 Escalonamento na Programação em Lógica

Alternativas dinâmicas são utilizadas para o escalonamento na programação em lógica devido às características não-determinísticas das linguagens. Outro fator que influencia nesta escolha é o fato de que alguns sistemas são concebidos para execução em arquiteturas diferentes, com número variável de processadores e até mesmo poderes computacionais diferentes. Evans ([EVA 92]) afirma que o problema de escalonamento dinâmico é muito complexo e necessita que sejam considerados um grande número de parâmetros para obter ganhos significativos de desempenho.

Conforme analisado na seção 3.4, grande parte dos escalonadores utilizados por implementações de Prolog paralelo são centralizados. Isso se deve principalmente a dois fatores: os ambientes são na sua maioria para arquiteturas com memória compartilhada (vide seção 3.5.3); e, o foco principal dos trabalhos nesta área muitas vezes é a implementação de uma linguagem em lógica paralela deixando o escalonamento em segundo plano.

O escalonamento na programação em lógica pode alterar a semântica

operacional de Prolog, isto é, executar metas em ordem diferente daquela seguida pelo Prolog tradicional. Para a programação em lógica, o objetivo do escalonador é terminar a execução paralela o quanto antes, alocando processadores para a melhor tarefa ([DUT 97]). Um algoritmo de escalonamento alcança mais facilmente este objetivo se não causar muita sobrecarga, levar em conta as características do sistema e usar análise de granulosidade.

O escalonamento para o paralelismo E/OU é sempre dinâmico, por causa da natureza imprevisível do paralelismo inerente. Os principais problemas no escalonamento OU são a interação entre o modelo de memória e o escalonador, e as técnicas para ordenação automática dos efeitos colaterais e manipulação do trabalho especulativo ([DUT 97] e [CIE 91]). Na seção 4.1.1 pode ser observado como o Plosys trata os problemas de efeito colateral e trabalho especulativo.

O escalonamento para o paralelismo E tem como principais problemas a dependência entre metas (E Dependente), e a política empregada com as metas (compartilhamento ou recomputação). Na seção 3.4 podem ser vistas as vantagens e desvantagens destas políticas.

Quando é desejada a exploração do paralelismo E/OU o principal problema é decidir quando executar as cláusulas e quando executar as metas em paralelo. Isso leva a duas alternativas: executar o paralelismo OU abaixo do paralelismo E ou executar o paralelismo E abaixo do paralelismo OU.

O modelo atual do OPERA propõe a execução de E abaixo de OU, de forma que o paralelismo OU seja explorado em uma primeira fase (até que esteja esgotado) e após a execução seja concentrada no paralelismo E.

Devido às poucas propostas distribuídas encontradas na programação em lógica, foi feita uma ampla busca dentre os vários artigos existentes sobre o tema em outras áreas da Ciência da Computação como programação de sistemas, processamento paralelo e sistemas operacionais. A seção a seguir apresenta uma resumida análise das principais propostas encontradas.

5.3 Escalonamento Distribuído

Dentre as principais políticas utilizadas nos sistemas fisicamente distribuídos é possível destacar:

- ***bidding:*** nesta política de escalonamento os processadores cooperam para que o envio de uma tarefa a um processador beneficie o sistema como um todo. Através de lances (unidades de carga) é escolhido o processador em qual determinada tarefa irá executar. Uma característica importante desta classe de escalonadores é que todos os nodos geralmente têm autonomia total, uma vez que têm o poder de decidir para onde enviar uma tarefa. Além disso, eles têm autonomia pois não são forçados a aceitar determinado trabalho;

- **probabilísticos:** este esquema é motivada pelo fato de que em muitos problemas o número de permutações entre trabalho disponível e processadores é tão grande que examinar analiticamente o espaço de solução inteiro, demandaria um tempo excessivo de execução. A idéia básica é de

escolher aleatoriamente (de acordo com alguma distribuição) processos para serem escalonados. Usando este método repetidamente, um número grande de estratégias de escalonamento podem ser geradas, e então este conjunto é analisado e a melhor escolhida.

Além das políticas apresentadas, várias outras que não se enquadram em categorias específicas são empregadas. Como política de localização para sistemas distribuídos são utilizadas basicamente duas alternativas, a saber: iniciado pelo emissor (*sender-initiated*) e iniciado pelo receptor (*receiver-initiated*). Várias análises comparando as duas alternativas foram feitas na literatura ([DAN 96b], [TOW 94] e [SHI 92]).

Nas políticas iniciadas pelo emissor, nodos sobrecarregados procuram transferir trabalho para nodos com pouca carga. Por outro lado, em políticas iniciadas pelo receptor nodos com pouca carga procuram nodos sobrecarregados. A política iniciada pelo emissor comporta-se melhor para sistemas com carga baixa ou moderada ([EAG 86]). Isso porque, com estas cargas, a probabilidade de encontrar nodos com carga baixa é maior do que a de encontrar nodos com carga alta. Além disso, a política iniciada pelo receptor apresenta grandes problemas de desempenho se a carga não é homogênea no sistema (apenas alguns nodos geram a carga) ou se existe um grande variação de tempos entre o surgimento de novos trabalhos ([SHI 92]).

Pode-se classificar as políticas de escalonamento distribuído quanto à sua localização física em dois tipos, a saber: totalmente distribuída e hierárquica.

O escalonamento totalmente distribuído é aquele em que todos os processadores envolvidos na computação possuem além de trabalhadores, tarefas responsáveis pelo escalonamento. Esta política de escalonamento apresenta um custo de comunicação alto, uma vez que várias mensagens são necessárias para manter a informação de carga global do sistema ([SUE 92]). Por outro lado, a política totalmente distribuída resolve em parte os problemas de escalabilidade e de tolerância a falhas da proposta centralizada. A figura 5.3 apresenta o modelo de escalonamento totalmente distribuído.

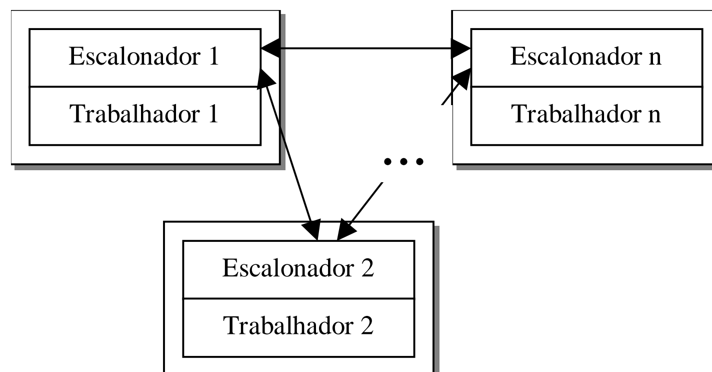

FIGURA 5.3 - Escalonamento Totalmente Distribuído

As políticas de escalonamento totalmente distribuído apresentam algoritmos extremamente complexos, uma vez que vários processos idênticos interagem simultaneamente. É importante registrar que os custos de comunicação as tornam

proibitivas em arquiteturas fracamente acopladas e com muitos processadores. Muitas vezes, para que sejam obtidos desempenhos satisfatórios, é necessário que apenas alguns nodos (aleatoriamente selecionados) sejam consultados quando da exportação ou importação de tarefas ([DAN 96a] e [EAG 86]).

A última categoria de política de escalonamento distribuído que será apresentada, a hierárquica, representa um meio termo entre a centralizada e a totalmente distribuída. Ela resolve os problemas de escalabilidade e tolerância a falhas da proposta centralizada sem apresentar os custos de comunicação excessivos da proposta totalmente distribuída ([DAN 96a] e [ARA 94]). A figura 5.4 apresenta um modelo de escalonamento hierárquico.

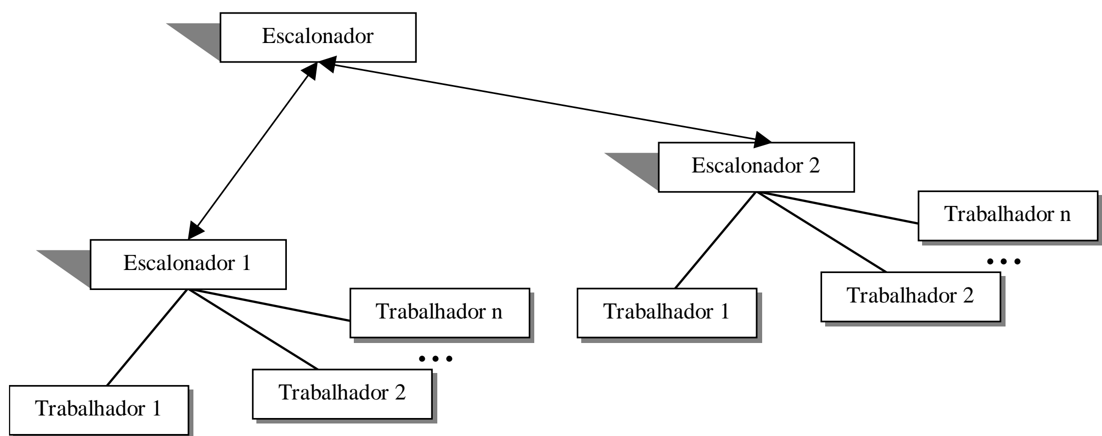

FIGURA 5.4 - Escalonamento Hierárquico

A política hierárquica caracteriza-se por possuir escalonadores em alguns processadores do sistema. Estes processadores podem ser dedicados ou não, isto é, podem possuir processos escalonadores e trabalhadores no mesmo processador. Esse tipo de escalonamento possui um conjunto de nodos responsável por manter o estado do sistema, ao invés de um único nodo, como no centralizado. No escalonamento hierárquico, o sistema é dividido logicamente em grupos (*clusters*) e cada grupo de nodos tem um único processo responsável por manter o estado dos nodos do grupo.

Dentre as políticas estudadas a hierárquica apresenta a melhor relação custo/benefício para arquiteturas com memória distribuída. A tabela 5.1 sintetiza as principais características dos três tipos de escalonamento apresentados.

TABELA 5.1 - Comparação Entre as Políticas de Escalonamento

Características:	Centralizado	Totalmente distribuído	Hieráquico
Complexidade dos algoritmos	☺☺☺	☹☹☹	😐😐😐
Escalabilidade	☹☹☹	😐😐😐	☺☺☺
Precisão das informações de carga dos processadores	☺☺☺	☹☹☹	😐😐😐
Quantidade de mensagens	☺☺☺	☹☹☹	😐😐😐
Tolerância a falhas	☹☹☹	☺☺☺	☺☺☺
Desempenho (menos de 100 processadores)	☺☺☺	😐😐😐	☺☺☺
Desempenho (mais de 100 processadores)	☹☹☹	☹☹☹	☺☺☺

Boa →☺☺☺; Média →😐😐😐; Ruim →☹☹☹

5.4 Conclusões

Este capítulo apresentou uma introdução ao escalonamento, concentrando-se no escalonamento dinâmico e distribuído. Através deste estudo, foram identificados os principais componentes de um escalonador, bem como feita uma comparação entre as principais políticas de escalonamento existentes.

Foram apresentadas características dos escalonadores na programação em lógica bem como dos utilizados em outras áreas da Ciência da Computação. A busca por outras políticas de escalonamento ocorreu uma vez que alternativas de escalonamento fisicamente distribuídas para arquiteturas fracamente acopladas foram até hoje muito pouco empregadas na programação em lógica.

6 Escalonador Hierárquico: DSLP

Este capítulo apresenta o modelo de escalonador proposto para exploração do paralelismo na programação em lógica. O modelo proposto é classificado segundo a taxonomia apresentada como global, dinâmico, fisicamente distribuído, cooperativo, sub-ótimo e heurístico. Quanto à localização física é um escalonador hierárquico, pois nem todos os processadores necessitam ter escalonadores.

Para minimizar os custos envolvidos na tarefa de escalonar a opção foi por uma política que utilize alocação única, que seja não-preemptiva e para uma única aplicação. O escalonador integra o paralelismo E Independente e OU na programação em lógica. Foram preservadas todas as características necessárias para a futura integração com o OPERA e o GRANLOG. Algumas publicações foram geradas como produto do trabalho, ver [CER 97], [TRE 97] e [COS 96b].

6.1 Princípios Básicos

A idéia central do trabalho é definir uma proposta de escalonamento hierárquica na execução paralela da programação em lógica. O escalonador consiste em um modelo dinâmico e distribuído, integrando o paralelismo E Independente e OU na programação em lógica.

O modelo criado tem o nome *Distributed Scheduler for Logic Programming* (DSLP) – Escalonador Distribuído para a Programação em Lógica. Para a tarefa de escalonar o DSLP utiliza dois tipos de informações, quais sejam: informações de granulosidade, obtidas através de anotações feitas pelo GRANLOG, e informações do sistema. Através destas informações o DSLP executará em paralelo a aplicação Prolog determinada pelo usuário. Para tal, o modelo fará uso de máquinas abstratas WAM com suporte a paralelismo E Independente e OU.

Os princípios básicos do modelo DSLP são:

- **realizar escalonamento dinâmico e distribuído**: o modelo proposto trabalha durante a execução dos programas, devido às características não-determinísticas da linguagem Prolog. Além disso ele é fisicamente distribuído, cooperativo, sub-ótimo e heurístico. A proposta consiste em um modelo hierárquico, pois segundo a seção 5.3 este apresenta as melhores características dentre as propostas distribuídas estudadas. No DSLP alguns processadores possuem o escalonador. Estes processadores podem não ser dedicados à tarefa de escalonar;

- **integrar paralelismo E Independente e OU**: o modelo proposto explora as duas principais formas de paralelismo encontradas na programação em lógica. Para tal, o escalonador utiliza máquinas abstratas específicas para o paralelismo E Independente e para o paralelismo OU multiseqüencial. O DSLP explora o paralelismo E abaixo de OU em duas fases distintas;

- **utilizar informações de granulosidade**: para a tarefa de escalonar, são utilizadas informações de granulosidade para o auxílio na tomada de decisões. Através delas, é possível determinar se o custo de exportação de determinada

tarefa é maior que o custo de execução. O ambiente decide dinamicamente se deve ou não fazer a exportação, em função destas informações;

- **utilizar informações do sistema:** para que o escalonamento funcione adequadamente em várias arquiteturas de forma eficiente, são utilizados parâmetros do sistema. É possível citar como parâmetros do sistema: custos de comunicação, tamanho de fila de espera da UCP e poder computacional. Dependendo do sistema em questão, alguns parâmetros podem ser desconsiderados (por exemplo: em um sistema homogêneo o poder computacional não necessita ser utilizado pelo DSLP);

- **compatibilidade com o OPERA:** o modelo DSLP é compatível com o OPERA. Por isso, explora apenas o paralelismo E Independente e OU multiseqüencial, não explorando em uma primeira fase o paralelismo E dependente. Além disso, o modelo explora paralelismo E abaixo de OU, conforme é proposto no OPERA. Outra característica do DSLP que o torna passível de integração com o OPERA é o fato de não necessitar de memória compartilhada entre processadores;

- **compatibilidade com o GRANLOG:** o DSLP é compatível com o GRANLOG. Para tal, o escalonador pressupõe informações de granulosidade no formato sugerido pelo GRANLOG. O modelo tenta aproveitar ao máximo as informações do analisador de granulosidade em questão.

O escalonador foi proposto a partir de vários estudos sobre escalonamento, sintetizados no capítulo 5. O DSLP é um escalonador hierárquico devido a duas características principais: o uso dessa classe de políticas de escalonamento representa uma tendência na área de processamento paralelo e distribuído e, essa classe de escalonamento apresenta a melhor relação custo/benefício dentre as políticas estudadas.

6.2 Visão Geral

Inicialmente um programa em Prolog é processado pelo GRANLOG, adicionando assim informações de granulosidade ao programa fonte. Após, o programa Prolog acrescido destas informações é submetido a um compilador paralelizador. Este compilador gera um código intermediário, formado por instruções para a máquina WAM.

O compilador paralelizador insere instruções específicas para que o ambiente de execução saiba quando pode executar uma tarefa (seja por exploração do paralelismo E Independente ou paralelismo OU) em outro processador. Além disso, ele organiza em tabelas as informações de granulosidade anotadas.

O conjunto de instruções geradas pelo compilador passa por um processo de montagem que o transforma em código decimal (*byte code*). O montador transforma as tabelas de granulosidade em listas que contém referências aos trechos de códigos onde devem ser empregadas.

O código decimal é então submetido a um emulador, o qual executa o programa Prolog. Este emulador é composto por máquinas abstratas e escalonadores. O DSLP da forma como é apresentado neste capítulo é o próprio Emulador. A figura 6.1 mostra as

etapas necessárias para a execução de programa Prolog pelo ambiente OPERA integrado com o DSLP.

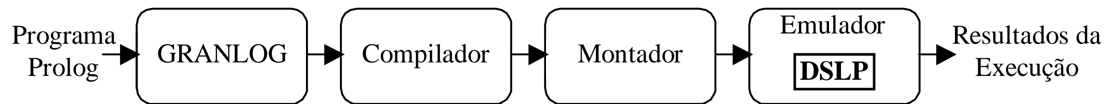

FIGURA 6.1 - Etapas de Execução de um Programa Prolog

O escalonador realiza a execução do programa em duas fases. Primeiramente é explorado o paralelismo OU multiseqüencial. Quando esta exploração estiver esgotada, o escalonador determina que o paralelismo E Independente seja utilizado. Além disso o escalonador é responsável pela distribuição de tarefas aos processadores e pela garantia da ordem de execução.

Para que o DSLP faça o escalonamento é necessário que este tenha conhecimento de algumas informações, a saber: parâmetros da aplicação e parâmetros do sistema.

Os parâmetros da aplicação informam a quantidade de processadores que serão utilizados na execução, quantos processadores devem possuir escalonadores e quais são trabalhadores (estão dedicados exclusivamente para a execução de programas Prolog). Além disso, outros parâmetros como nível de sobrecarga, intervalo de verificação de carga e variação de carga necessária para exportação são determinados.

Os parâmetros do sistema podem ser divididos em dois tipos: aqueles que são estáticos, isto é, não se alteram no decorrer da execução paralela e os dinâmicos. O poder computacional das máquinas, o conjunto de todas as máquinas que podem ser utilizadas e as velocidades de comunicação são alguns parâmetros considerados estáticos.

Os parâmetros dinâmicos do sistema devem ser definidos para que o DSLP determine o estado dos processadores de maneira precisa. É enquadrado nesta categoria o parâmetro tamanho da fila de espera da UCP.

6.3 Descrição do Modelo DSLP

Cada nodo do sistema pode conter um *módulo escalonador*. Os nodos que o contém não são dedicados, são responsáveis pelo escalonamento de tarefas e pela execução das resolventes em paralelo, e por isso são chamados de *nodos OU*. Os outros nodos do sistema (sem o módulo escalonador) ficam ligados aos escalonadores mais próximos e são responsáveis pela execução paralela de metas independentes, chamados de *nodos E Independente*. Nesta arquitetura, é explorado o paralelismo E abaixo de OU como mostra a figura 6.2.

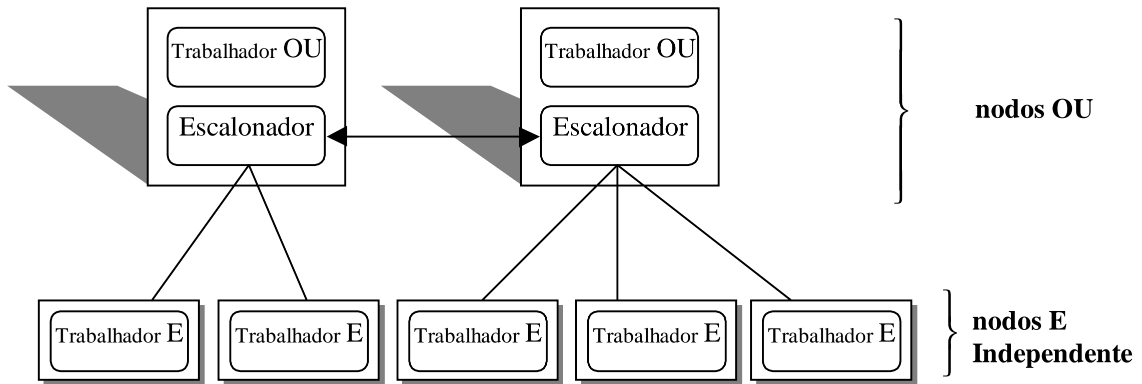

FIGURA 6.2 - Escalonador Hierárquico

Cada escalonador deve ter conhecimento da existência e da carga de seus vizinhos, através dos quais haverá um fluxo de mensagens. Para isso, entre os módulos escalonadores é formado um anel lógico (figura 6.3). O anel é bidirecional, para que cada escalonador possa se comunicar com ambos os vizinhos.

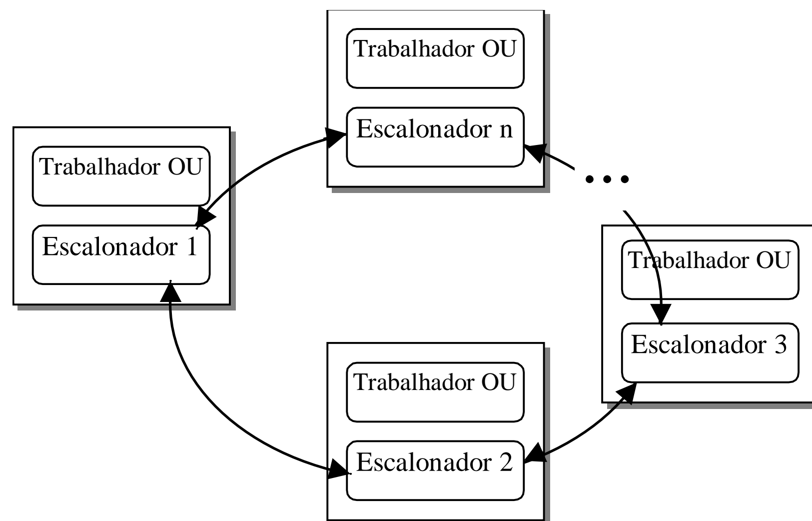

FIGURA 6.3 - Anel Lógico Formado Entre Escalonadores

Outras alternativas poderiam ser empregadas para a comunicação entre os escalonadores ao invés do anel lógico. Propostas de escalonadores algumas vezes utilizam primitivas de difusão (*broadcast*) e difusão seletiva (*multicast*) para distribuir informações entre os escalonadores.

Neste trabalho, foi procurado minimizar o custo envolvido com a comunicação ao máximo, devido a prioridade de que o sistema trabalhe em arquiteturas fracamente acopladas. Por este motivo as primitivas de *broadcast* não foram utilizadas. Primitivas deste tipo podem onerar os custos de comunicação do sistema ([HEI 97]).

Em algumas arquiteturas, o custo de envio de um *broadcast* é o mesmo do que o

de envio de uma mensagem para um único destinatário. Porém, todos os processadores que recebem a mensagem tem custos envolvidos no recebimento e desempacotamento desta. Heiss ([HEI 97]), considera que o custo de recebimento e empacotamento é alto e que algoritmos de escalonamento que utilizam primitivas de *broadcast* distribuem mensagens que não necessitam ser recebidas por todos os processadores do sistema (por exemplo, nodos sobrecarregados não necessitariam receber mensagens de exportação). Por isso a difusão não deve ser utilizada.

Os nodos OU contém os módulos escalonadores e comunicam-se através de um anel lógico. Os nodos OU informam suas mudanças de carga ao processo escalonador, executando na própria máquina através de uma variável compartilhada. Os nodos E possuem um processo espião (*spy*) que informa ao escalonador associado, através de troca de mensagens, mudanças significativas de carga.

Os escalonadores são responsáveis por decidir os nodos exportadores e importadores, a partir das análises de carga. Um nodo exporta suas tarefas diretamente para o importador, isto é, o escalonador envia mensagens aos nodos envolvidos na troca de carga dizendo que determinado trabalho pode ser exportado/importado.

O modelo DSLP permite que o usuário determine a quantidade inicial de escalonadores e de trabalhadores que o sistema deve possuir para a execução de determinada aplicação. Com isso, a arquitetura pode váriar de acordo com a quantidade de paralelismo E/OU existente no programa, bem como de acordo com o multicomputador em questão.

Dependendo da quantidade de paralelismo E/OU inerente ao problema é feita a configuração da arquitetura, podendo até mesmo explorar apenas uma das fontes do paralelismo na programação em lógica, seja ela o paralelismo E ou OU. Para arquiteturas com muitos processadores é possível configurar um número maior de escalonadores, bem como para arquiteturas com poucos processadores pode-se optar por uma política de escalonamento centralizada. A seção 6.4 apresenta considerações para a detecção automática da quantidade inicial de escalonadores em função da aplicação e do sistema, bem como para a reconfiguração dinâmica.

O DSLP mantém os trabalhadores OU localizados nos mesmos processadores que os escalonadores, empregando entre os nodos OU uma política de escalonamento totalmente distribuída. Entre os nodos E é utilizada uma política de escalonamento centralizada (para nodos ligados ao mesmo escalonador) e hierárquica (para nodos ligados a escalonadores diferentes). Quando o paralelismo E e OU é explorado simultaneamente a política de escalonamento é hierárquica.

O *Distributed Scheduler for Logic Programming* tem como sua principal característica a flexibilidade. A proposta permite uma adequação do escalonamento ao problema, uma vez que é possível ao usuário determinar em quais nodos os escalonadores estão presentes. Caso esteja presente em apenas um nodo, o escalonamento passa a ser centralizado e o paralelismo OU não é explorado. Caso esteja presente em todos os nodos, o escalonamento é totalmente distribuído e a exploração do paralelismo E Independente não é feita.

Outra característica importante da proposta é a utilização de informações de granulosidade para o auxílio ao escalonamento. Até o presente momento de acordo com

o conhecimento do autor, não existem propostas de escalonamento para a exploração de paralelismo na programação em lógica que utilizem tais informações. Com a análise de granulosidade é possível melhorar o escalonamento das tarefas, uma vez que as tarefas mais complexas serão executadas antes, por processadores mais ágeis e as menos complexas não serão executadas em paralelo.

O uso de informações do sistema em conjunto com a análise de granulosidade, permite uma melhor adequação em relação à arquitetura utilizada. O DSLP não pressupõe nenhum tipo de arquitetura, nem homogeneidade ou velocidades idênticas de comunicação entre as máquinas. Facilmente o ambiente pode ser adequado a qualquer configuração de máquinas e interconexão disponíveis.

Resumindo a descrição da arquitetura apresentada, é possível dividi-la em dois componentes: os nodos OU e os nodos E. Os nodos OU são compostos por dois processos, a saber: os trabalhadores OU e os escalonadores. Os nodos E também possuem dois processos, quais sejam: os trabalhadores E e os espiões. A figura 6.4 sintetiza os componentes da arquitetura.

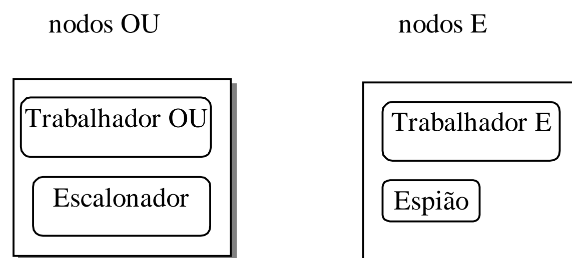

FIGURA 6.4 - Componentes do DSLP

Os escalonadores são associados aos trabalhadores OU e não aos trabalhadores E, pelos seguinte motivos:

- **quantidade de paralelismo:** o paralelismo E é muito mais presente nos programas em lógica do que o paralelismo OU. Com isso, o trabalho executado pelos nodos E é mais constante que o executado pelos nodos OU deixando os nodos OU menos sobrecarregados;

- **complexidade dos grãos:** normalmente os grãos E possuem uma complexidade menor do que os grãos OU, isto é, os trabalhadores OU realizam mais processamento (sem comunicação) do que os trabalhadores E. Com o escalonador associado a cada nodo, a informação geral das cargas E seria mais imprecisa, e portanto, o escalonamento seria pior;

- **localidade:** como o modelo possui um escalonador para um grupo de nodos E, o trabalho gerado fica o mais próximo possível do nodo que o originou. Isso é especialmente importante para o paralelismo E, pois as instanciações das variáveis devem ser retornadas para os nodos os quais originaram o paralelismo.

A proposta do DSLP permite que os nodos OU não tenham escalonadores

associados, porém no presente trabalho os nodos OU sempre os possuem por questões de simplificação. No caso da existência de um único escalonador só é explorado o paralelismo E.

Para a manipulação da arquitetura DSLP é proposta a criação de três programas:

- ***createArch***: realiza a criação da arquitetura, isto é, dispara nos processadores envolvidos na computação os escalonadores, trabalhadores OU, trabalhadores E e espiões. O *createArch* pode realizar esta tarefa de duas formas distintas: a partir de uma configuração pré-determinada pelo usuário ou a partir de detecção automática (sugerida na seção 6.4);

- ***startArch:*** dispara uma aplicação em uma arquitetura já criada. Este módulo faz com que seja executado um programa em lógica no sistema DSLP já previamente montado pelo *createArch*. Ao encerrar a execução da aplicação o *startArch* apresenta os resultados obtidos e mantém a arquitetura pronta para a execução de outras aplicações. Este programa pode ser executado diversas vezes, sem a necessidade da recriação da arquitetura;

- ***changeArch:*** este módulo é utilizado para modificar a arquitetura existente, em função de uma nova configuração especificada pelo usuário ou devido à detecção de paralelismo de outra aplicação. O módulo pode ser acionado também para destruir totalmente a arquitetura montada.

6.3.1 Estruturas de Dados

Para a execução de programas em lógica pelo emulador, é necessária inicialmente a criação da arquitetura através do programa *createArch*. Esta criação é feita a partir da leitura de duas tabelas: a *System Resource Table* (SRT) – Tabela de Recursos do Sistema e a *Application Characteristic Table* (ACT) – Tabela de Características da Aplicação. As tabelas foram propostas inicialmente em ([YAM 94]) porém ampliadas e modificadas para o trabalho atual.

A SRT contém informações sobre os recursos disponíveis no ambiente onde a aplicação será executada. As informações contidas nesta tabela são:

- **nome ou número IP das máquinas:** são listadas todas as máquinas que poderão participar do DSLP;

- **poder computacional das máquinas:** são listadas os poderes computacionais das máquinas descritas no primeiro item. O poder computacional é descrito em um índice baseado na velocidade do processador em MIPS (Milhões de instruções por segundo). Este dado pode ser omitido em sistemas homogêneos, isto é, onde todas as máquinas possuem a mesma capacidade de processamento;

- **velocidade de comunicação entre máquinas:** caso existam velocidades diferentes de comunicação entre as máquinas que poderão ser envolvidas na computação, estas são apresentadas. Este dado pode ser omitido, caso a velocidade seja constante entre todas as máquinas.

A SRT contém as três informações descritas uma após a outra separadas por espaços. Cada linha inicia com o nome ou número IP de uma máquina, seguido do poder computacional e da velocidade de comunicação. Os dois últimos ou o último parâmetro podem ser omitidos. Comentários de uma linha podem ser introduzidos desde que o primeiro caractere da linha seja o cancela (#).

A tabela ACT contém informações sobre as características que serão utilizadas para a execução de um conjunto de aplicações. Ela é organizada de forma que o primeiro caractere de cada linha indica o parâmetro passado, seguido do valor. Comentários de uma linha são suportados de forma análoga aos da SRT. As informações contidas nesta tabela são:

- **nodos OU (*p*rincipal) :** indica as máquinas que conterão os módulos escalonadores e os trabalhadores OU. Para tal, colocar a letra *p* seguida de um espaço e do nome da máquina ou número IP. A primeira máquina relacionada é aquela que iniciará a execução dos programas em lógica. No caso da detecção automática de nodos E/OU é utilizada a palavra reservada *automatic* nesta categoria, e todos os nodos que são envolvidos na computação são listados na categoria a seguir;

- **nodos E (*m*achine):** indica as máquinas que conterão os trabalhadores E e os processos espiões. Caso o item anterior contenha a palavra reservada *automatic*, esta categoria informa as máquinas que irão conter nodos E ou nodos OU, determinados automaticamente;

- **tempo de inatividade do espião (*s*py sleep time):** é o tempo, em milisegundos, de inatividade do processo espião. Separa duas verificações consecutivas de carga do trabalhador;

- **nível de sobrecarga (*o*verload level):** indica o número de resoluções a partir do qual o trabalhador é considerado sobrecarregado, justificando pedido de exportação de seu trabalho para outros nodos. É empregada a mesma unidade utilizada para a determinação de complexidade do GRANLOG;

- **variação de flutuação (*flutuation* **r***ange*):** estabelece a flutuação mínima que deve ocorrer na carga do trabalhador para justificar uma atualização do escalonador com seu novo valor. Como unidade é utilizado também o número de resoluções.

Exemplos de tabelas SRT e ACT para o protótipo implementado podem ser encontrados nos Anexos 1 e 2 respectivamente . Cabe observar que as tabelas foram simplificadas para o protótipo implementado, conforme descrito na seção 7.1.2.

Os critérios para definir bons valores para os parâmetros da ACT é uma tarefa árdua e ainda foco de pesquisas. Atualmente, muitos sistemas de escalonamento determinam valores dessa natureza experimentalmente, uma vez que dependendo da aplicação e do sistema utilizados parâmetros diferentes podem obter desempenhos melhores ou piores de acordo com o caso.

Após a leitura das tabelas o *createArch* cria o ambiente de execução do DSLP, disparando os processos da aplicação nos processadores determinados e distribuindo as

informações necessárias a cada componente. Os escalonadores são informados dos nodos OU vizinhos, dos nodos E vinculados a eles e do número total de escalonadores do sistema. Os nodos E e OU recebem a variação da flutuação e o nível de sobrecarga que serão considerados na aplicação. Além destas informações, os nodos E recebem ainda o tempo de inatividade do espião e a informação de que nodo OU contém o escalonador ao qual ele está vinculado. As estruturas de dados do sistema são inicializadas e o *createArch* termina deixando a arquitetura pronta para a execução de uma aplicação.

A aplicação é disparada a partir do *startArch*. Esse módulo recebe como parâmetro o nome do arquivo em código decimal que deve ser executado e uma consulta é iniciada em um nodo OU. A partir deste momento começa a ser executada a aplicação. A medida que vão surgindo resolventes com potencial para paralelização (determinado através da análise de granulosidade) estas são repassadas para outros nodos OU pelo escalonador.

Cada escalonador possui um estado de *carga OU*, além do estado de carga dos nodos E Independente ligados a ele. Caso seja necessário exportar trabalho OU para outros nodos, mensagens de exportação de trabalho OU são enviadas para os escalonadores vizinhos, até encontrar um escalonador com carga OU baixa.

Quando o paralelismo OU está esgotado, as metas são repassadas para os nodos E Independente. A partir deste momento só é explorado o paralelismo E. Nesta fase, metas paralelizáveis e o número máximo de resoluções necessárias para sua execução (determinadas pela análise de granulosidade) são colocadas em uma lista.

Periodicamente o escalonador é informado da quantidade de resoluções existentes nesta estrutura (*carga E)*. Quando existir uma carga E considerável, o escalonador exporta a meta com o maior número de resoluções, preferencialmente para um nodo E Independente ligado a ele. Caso isso não seja possível, o trabalho é passado para um outro escalonador, contendo nodos E Independente com carga E baixa.

As cargas E e OU são definidas em complexidade (números de resoluções máximas). Além disso, em função do nível de sobrecarga (obtido a partir da tabela ACT) podem assumir três estados possíveis:

- **disponível (*idle*):** neste estado não há trabalho a ser executado. O trabalhador aguarda uma tarefa para importação, a ser determinada pelo escalonador;

- **trabalhando (*quiet*):** neste estado o trabalhador está ativo, mas não dispõe de trabalho em volume suficiente que justifique o envio de tarefas para outro processador;

- **sobrecarregado (*overloaded*):** neste estado, o trabalhador está ativo e com volume de trabalho que justifique a exportação para outros trabalhadores. Durante este estado, o trabalhador pode receber mensagens para exportação determinadas pelo escalonador.

Quando o *startArch* termina a execução da aplicação são apresentados os resultados da execução. O módulo deixa as estruturas de dados consistentes e prontas para a execução de uma nova aplicação. Neste momento pode ser chamado o *startArch*

novamente, quantas vezes for necessário.

Se o usuário quiser alterar ou encerrar a arquitetura deve chamar o módulo *changeArch*. Este módulo, caso receba o parâmatro *end*, termina todos os processos do ambiente DSLP em execução. Em caso contrário o *changeArch* analisa as mudanças na tabela ACT e reorganiza a arquitetura de forma a refletir as novas alterações.

A seguir o funcionamento detalhado de cada um dos processos do DSLP é descrito.

6.3.2 Funcionamento do Escalonador

O modelo proposto realiza um escalonamento baseado no compartilhamento de carga, distribuindo a carga de nodos sobrecarregados (*overloaded) p*ara nodos disponíveis (*idle).* O escalonador utiliza alocação única, é não-preemptivo e não-adaptável. Isso ocorre devido aos motivos apresentados na seção 5.3. Como política de localização, o DSLP é sempre iniciado pelo emissor.

O processo escalonador é o responsável pelo escalonamento de trabalho E e de trabalho OU no sistema DSLP. O escalonamento é realizado em duas fases. Inicialmente o escalonador concentra-se em explorar o paralelismo OU (fase OU). Quando este paralelismo não pode mais ser explorado, o escalonador busca a exploração de paralelismo E. Esta característica é devido à limitação do modelo OPERA de exploração de paralelismo E abaixo de OU, isto é, resolver todo o paralelismo OU e a partir deste momento só explorar o paralelismo E (não retornando para a fase anterior). O escalonador foi proposto de maneira a suportar a exploração do paralelismo E/OU simultaneamente. Para tal, basta empregar no sistema um novo modelo de execução.

O escalonador necessita armazenar dois tipos de carga: carga OU e carga E. O escalonador contém a carga OU do trabalhador situado no próprio processador, além da carga OU dos escalonadores vizinhos. Além disso, o escalonador possui um vetor contendo as cargas E de todos os nodos E vinculados a ele e uma informação de carga E dos vizinhos.

As cargas são dividas em duas informações, a saber: índice (loadI*ndex)* e estado (*loadState*). O índice representa a complexidade de todas as tarefas que estão disponíveis para a execução paralela em determinado nodo. O estado informa se o nodo está disponível, trabalhando ou sobrecarregado. O escalonador utiliza o índice para escolher o nodo que irá exportar uma tarefa quando está no estado *overloaded*. No estado *quiet* e *idle* o índice não é considerado.

As cargas dos vizinhos possuem uma semântica um pouco diferente das outras cargas do sistema. A carga OU de um vizinho é *overloaded* somente quando não existem vizinhos disponíveis em determinada direção no anel lógico. Carga OU *idle* indica que o próprio vizinho (aquele imediatamente conectado) está livre. Por sua vez, carga de vizinho *quiet* informa que o próximo vizinho no anel não está livre, porém outros nesta direção podem estar. A carga E dos vizinhos é tratada de forma análoga, porém basta apenas um nodo E vinculado estar livre para que o estado seja *idle*. A figura 6.5 sintetiza os dados de carga do escalonador.

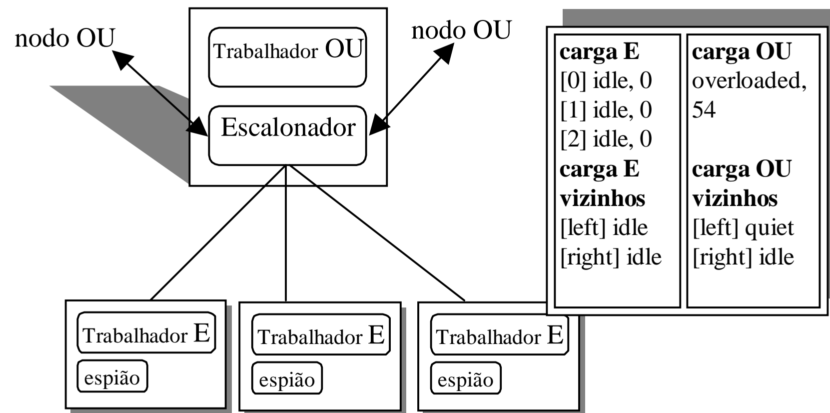

FIGURA 6.5 - Informações de Carga do Escalonador

Além dos dados de carga, as principais informações que o escalonador contém são:

- **passo máximo (*MaxStep*):** a partir do número total de escalonadores é determinado o número máximo de vezes que uma mensagem pode ser enviada no anel bidirecional. Sempre que uma mensagem de um escalonador para outros é gerada, o passo máximo é empacotado. Quando uma mensagem de escalonador é recebida e deve ser enviada a outro vizinho, o passo é decrementado e re-enviado. O passo máximo é obtido dividindo-se o número de escalonadores por dois. Caso o número de escalonadores seja par o passo para a esquerda é uma unidade menor que o da direita;

- **índice de importação das máquinas (*importIndex*):** esta informação permite escolher a máquina que irá receber tarefas de nodos sobrecarregados (nodos E). O índice é obtido a partir do tamanho da fila de espera da UCP. Quando o sistema não é homogêneo, o poder computacional das máquinas tem um peso de 20%. Quando a velocidade de comunicação entre as máquinas não é idêntica, a velocidade de comunicação tem um peso de 20%;

- **poder computacional dos nodos:** esta informação é utilizada para geração do índice de importação. Em sistemas homogêneos esse dado é desconsiderado;

- **velocidade de comunicação entre processadores:** esta informação é utilizada para geração do índice de importação. Em sistemas com velocidade de comunicação idêntica entre as máquinas esse dado é desconsiderado. Caso contrário, tem um peso de 20%;

- **tamanho da fila de espera da UCP dos nodos:** esta informação é utilizada para geração do índice de importação. Mensagens enviadas pelo espião do nodo E contém atualizações da fila de espera, bem como mensagens provenientes dos vizinhos. Quando a carga OU é alterada e o nodo OU fica disponível, o próprio escalonador obtém o tamanho da sua fila de espera da UCP para enviar aos seus vizinhos;

- **nodos E vinculados:** identificadores dos processos nas máquinas que contém os nodos E para que a comunicação possa ser realizada;

- **nodos OU vizinhos:** identificadores dos processos nas máquinas que contém os nodos OU para que a comunicação possa ser realizada.

O escalonador inicialmente espera uma mensagem de inicialização da aplicação (*STARTAPPS*). Esta mensagem é enviada pelo programa *startArch* para o escalonador principal. O escalonador principal envia então para seus vizinhos uma mensagem de inicialização (*STARTNEIGHBOR*). Cada escalonador que recebe esta mensagem, decrementa o passo e caso necessário (passo > 0) re-envia a mensagem para seu vizinho.

A inicialização é necessária para que os escalonadores sejam informados que uma nova aplicação foi disparada. Com isso, cada escalonador entra na fase OU e obtém seus índices de importação iniciais. Os índices de carga são inicializados em zero e o estado em *idle*, com exceção do nodo OU que iniciar a computação, pois este recebe o estado *quiet.* O escalonador principal informa o trabalhador OU localizado no mesmo processador para iniciar a execução do programa em lógica. Caso só exista um único escalonador, a fase E é iniciada imediatamente.

A fase OU é iniciada com uma mensagem *OR_STARTFASE* que é propagada por todo o anel lógico. Esta fase é terminada com a mensagem *OR_ENDFASE.* A terminação é detectada pelo escalonador que iniciou a computação. O escalonador principal dispara então a fase E, com uma mensagem *AND_STARTFASE*.

Caso o escalonador receba a mensagem de finalização da aplicação (*ENDAPPS*), seu vizinho é informado (*ENDNEIGHBOR*). Este por sua vez, envia a mensagem a um outro vizinho e assim sucessivamente, quantas vezes for necessário. O escalonador principal envia os resultados (*RESULT*) para o programa *startArch*, que os apresenta na tela. Todos os escalonadores devem ser avisados do término da aplicação para que não fiquem esperando outras tarefas a serem executadas. A figura 6.6 sintetiza o ciclo de mensagens do escalonador.

O escalonador proposto tem como índice de carga a soma de todas as complexidades de tarefas exportáveis em determinado nodo. Até o presente momento, todos os sistemas estudados tem como índice o número de tarefas exportáveis. Claramente, a complexidade das tarefas é uma informação mais precisa que o número de tarefas, o que torna o escalonamento mais eficaz.

```
recebe STARTAPPS do startarch
envia STARTNEIGHBOR para vizinhos
envia OR_STARTFASE para vizinhos
      FASE OU
      recebe OR_ENDFASE de trabalhador OU
      envia AND_STARTFASE para vizinhos
      FASE E
      recebe ENDAPPS de nodo E vinculado
      envia ENDNEIGHBOR para vizinhos
      envia RESULT para startarch
```

FIGURA 6.6 - Ciclo de Mensagens do Escalonador

As seções a seguir descrevem os algoritmos de escalonamento empregados nas fases OU e E, respectivamente. Nos algoritmos apresentados são empregadas duas funções para envio e recebimento de mensagens. A sintaxe utilizada é:

```
envia (identificador do destino, dados p/ enviar, TAG DA MENSAGEM)
recebe (identificador de origem, dados a receber, TAG DA MENSAGEM)
```

O identificador contém o número do processo destino ou o qual deve originar a mensagem. O TAG é um rótulo necessário para descrever a mensagem. O `envia` apresentado é do tipo não-bloqueante, isto é, as mensagens são enviadas de maneira assíncrona. O `recebe` utilizado é do tipo bloqueante, isto é, só é encerrado quando recebe uma mensagem com o TAG especificado (síncrono). Cabe ressaltar ainda, que o `recebe` pode receber mensagens de qualquer procedência e com qualquer TAG, para tal basta especificar nestes parâmetros variáveis inicializadas com zero.

Além das funções acima, é necessária uma função para verificar se existem mensagens a serem recebidas. Sua sintaxe é:

```
temMensagem ():valor lógico
```

Esta função retorna o valor lógico *verdadeiro* caso exista alguma mensagem para ser recebida, caso contrário retorna *falso*. Além das funções descritas são utilizadas construções padrões encontradas em qualquer linguagem de programação[1], tais como atribuições, estruturas de controle, seleção, repetição, etc.

6.3.2.1 Fase OU

O escalonador fica inoperante até o recebimento da mensagem *OR_STARTFASE*. Com esta mensagem a fase OU é iniciada. A figura 6.7 apresenta o algoritmo da FASE OU.

```
endComp := falso;
    oldState := Orstate;
    enquanto endComp <> verdadeiro faça
    início
      se ORstate <> oldState então {mudança na carga do trabalhador}
      início
        se Orstate = overloaded então
          {exportação de tarefas p/ vizinhos}
        se Orstate = idle então
          {avisa mudança de carga p/ vizinhos (OR_UPDATELOAD)}
        oldState := Orstate;
      fim
      se temMensagem ()= verdadeiro então
      início
        addrSent := 0;
        tag := 0;
        recebe (addrSent, , tag);
        caso tag de
          OR_UPDATELOAD:  {atualização da carga OU de um dos vizinhos}
```

[1] Semelhante a linguagem Pascal, somente traduzido para o português;

```
          OR_HAVEWORK:    {requisição de trabalhador disponível por
                           um dos vizinhos}
          OR_ENDFASE:     {terminada a execução da FASE OU, dispara a
                           fase E }
          AND_STARTFASE:  {indicação de que a execução da FASE E deve
                           ser iniciada}
                           endComp := verdadeiro;
        fim
      fim
    fim
```

FIGURA 6.7 - Fase OU de Escalonamento

O escalonamento de tarefas pode ocorrer sempre que exista uma variação de carga, seja do trabalhador OU presente no mesmo processador ou de um dos seus vizinhos. A variação de carga no trabalhador OU presente no próprio processador é detectada através da verificação de uma variável compartilhada (*ORState*) entre o escalonador e o trabalhador OU, a qual contém o estado do trabalhador OU. A variação na carga dos vizinhos é informada através de trocas de mensagens (*OR_UPDATELOAD*) pelo anel lógico.

Além das mensagens de atualização de carga, na fase OU o escalonador pode receber requisições de exportação de trabalho de outros escalonadores (*OR_HAVEWORK*). Quando um escalonador recebe uma mensagem *OR_ENDFASE*, é a indicação de que a exploração do paralelismo OU está esgotada naquele trabalhador. Esta mensagem é recebida pelo escalonador principal. Este escalonador é responsável pela detecção do término da fase OU. Quando isso ocorre, ele informa seus vizinhos do início da fase E (*AND_STARTFASE*). Essa informação é propagada pelo anel lógico.

As mudanças de carga do trabalhador OU presente no próprio escalonador, detectadas através da variável compartilhada *ORState*, podem gerar trocas de mensagens. Caso seja detectada uma variação para *idle*, os vizinhos que possuem carga *quiet* ou *overloaded* são avisados e propagam a mensagem enquanto houver vizinhos livres até o passo máximo. Caso o escalonador detecte sobrecarga no trabalhador OU presente no mesmo processador, através da variável compartilhada, é executado o algoritmo da figura 6.8. Mudanças de estado para *quiet* não são propagadas pelo anel, evitando tráfego de mensagens adicionais.

```
se HaveNeighbor = verdadeiro então {tem vizinhos}
  se (NeighborState[0] <> overloaded) OU
     (NeighborState[1] <> overloaded) então {pode exportar}
  início
    answers:=1;
    se addrNeighbor[1] = 0 então
      {tem um vizinho apenas}
      se NeighborState[0] = idle então
        envia (addrNeighbor[0], MaxStep e Myaddr, OR_HAVEWORK)
      senão
        answers:=2; {não pode exportar}
    senão
      {caso tem um idle envia p/ ele. Senão envia p/ vizinho
       com maior índice de importação}
      se (NeighborState[0] =idle) E (NeighborState[1] <> idle) então
        envia (addrNeighbor[0], MaxStep e Myaddr, OR_HAVEWORK)
      senão
```

```
        se(NeighborState[1]=idle) E (NeighborState[0]<>idle) então
          envia (addrNeighbor[1],MaxStep e Myaddr, OR_HAVEWORK)
        senão
        início
          answers:=0;
          se NeighborState[0]<>overloaded então
            envia (addrNeighbor[0],MaxStep e Myaddr, OR_HAVEWORK);
          senão
            answers := answers + 1;
          se NeighborState[1]<>overloaded então
            envia (addrNeighbor[1],MaxStep e Myaddr, OR_HAVEWORK);
          senão
            answers:= answers + 1;
        fim
    taken := 0;
    enquanto answers < 2 faça
    início
      {recebe de qualquer vizinho}
      addrTemp := 0;
      tag := 0;
      recebe (addrTemp, ,tag);
      se tag = TAKEN então
        se taken = 0 então
        início
          envia (addrTemp, addrOrWrk ,CONFIRM);
          taken := 1;
          answers := answers + 1;
          se addrTemp = Neighbor[0] então
            NeighborState[0] := quiet
          senão
            NeighborState[1] := quiet;
        fim
        senão {trabalho já exportado. Cancelar o outro.}
        início
          envia(addrTemp, , CANCEL);
          answers := answers + 1;
        fim
      senão
      {tag é NOIDLE - os vizinhos estão sobrecarregados e seus
       vizinhos também, pois a mensagem é propagada. NOIDLE só é
       enviado quando o passo é zero.}
      início
        answers := answers + 1;
        se addrTemp = Neighbor[0] então
          NeighborState[0] := overloaded
        senão
          NeighborState[1] := overloaded;
      fim
    fim
  fim
```

FIGURA 6.8 - Algoritmo de Escalonamento para os Nodos OU

No algoritmo, inicialmente o escalonador verifica a existência de vizinhos. Caso possua vizinhos, e os vizinhos não estejam sobrecarregados, o algoritmo tenta exportar tarefas. Quando o estado de um vizinho é sobrecarregado significa que não existe nenhum nodo livre naquela direção do anel, o estado *quiet* indica que o vizinho não está livre, porém outros nodos naquela direção do anel podem estar. Por fim, o estado *idle*

indica que o vizinho está livre.

Caso exista apenas um vizinho livre, este recebe a tarefa. Caso existam dois, o com maior índice de importação recebe a tarefa. Se não existirem vizinhos livres, a mensagem é propagada através dos vizinhos com estado *quiet* pelo anel. O escalonador após propagar as mensagens de pedido de exportação de trabalho (*OR_HAVEWORK*) fica aguardando alguma resposta. As respostas vem diretamente dos nodos escalonadores com trabalhador OU livre e podem ser: *TAKEN* ou *NOIDLE*. Uma mensagem *NOIDLE* indica que não existem trabalhadores livres naquela direção do anel, e portanto o estado deve ser alterado para *overloaded*. Uma mensagem *TAKEN* é respondida com uma mensagem *CONFIRM* e o escalonamento de tarefas é realizado.

Pode ocorrer do escalonador receber duas mensagens *TAKEN*, quando envia para os dois vizinhos no anel. Neste caso, o primeiro escalonador que responde é selecionado e o segundo recebe uma mensagem *CANCEL*.

```
recebe (addrTemp ,step e addrStart, OR_HAVEWORK);
    se (addrTemp = Neighbor[0]) então
    início
      se (NeighborState[0] = idle) então
        NeighborState[0] := quiet;
    fim
    senão
      se NeighborState[1] = idle então
        NeighborState[1] := quiet;
    se Orstate = idle então
    início
      envia (addrStart, ,TAKEN);
      recebe (addrStart, ,tag);
      se tag = CONFIRM então
        envia (addrOrWrk, addrStart, IMPORT);
    fim
    senão
    início
      step := step - 1; {decrementa o passo}
      se step > 0 então
      início
        se addrTemp = addrNeighbor[0] então
          indexTemp := 1 {descobre índice do outro vizinho}
        senão
          indexTemp := 0;
        se NeighborState[indexTemp]<>overloaded então
          envia(addrNeighbor[indexTemp],step e addrStart,
                OR_HAVEWORK)
        senão
          envia(addrStart, , NOIDLE);
      fim
      senão
        envia(addrStart, , NOIDLE);
    fim
```
FIGURA 6.9 - Algoritmo de Recebimento de Trabalho OU pelos Vizinhos

O envio da mensagem *TAKEN* é a indicação de que um nodo está livre e portanto pode receber trabalho. Neste momento o escalonador aguarda apenas o recebimento de dois tipos de mensagem: *CONFIRM* ou *CANCEL*. Nenhuma outra

mensagem pode ser recebida pelo escalonador, evitando assim mais de um escalonador sobrecarregado requisitar o mesmo nodo disponível simultaneamente. Caso o escalonador não possua trabalhador livre, a mensagem *OR_HAVEWORK* é propagada no anel (até o passo máximo). O algoritmo de recebimento das mensagens pelos vizinhos é apresentado na figura 6.9.

Quando o escalonador recebe uma mensagem *OR_HAVEWORK* inicialmente é verificado o seu estado através de uma variável compartilhada. Caso o estado seja *idle*, o escalonador envia uma mensagem *TAKEN*, indicando a aceitação do trabalho. Em caso contrário, o escalonador continua a propagação da mensagem no anel.

Uma requisição de trabalho não é mais propagada no anel quando o passo é zero ou o estado do próximo vizinho é *overloaded*. Neste caso, uma mensagem *NOIDLE* é enviada diretamente para o escalonador que disparou o processo.

Um otimização que é feita, porém não apresentada nos algoritmos, é o envio da carga de um trabalhador OU sempre em qualquer mensagem. Por exemplo, sempre que um escalonador envia uma mensagem de requisição de trabalho envia junto a carga. Com isso, a informação da carga geral do sistema fica mais precisa.

O escalonamento de tarefas pode ser realizado também quando o escalonador recebe uma mensagem *OR_UPDATELOAD*. Estas mensagens são propagadas somente quando o estado de determinado trabalhador é alterado para *idle*. O algoritmo de escalonamento empregado é o mesmo apresentado na figura 6.8 e descrito anteriormente.

Para esclarecer o funcionamento da fase OU considere a configuração apresentada na figura 6.10. Neste exemplo, existem quatro escalonadores cada um com seu trabalhador OU associado.

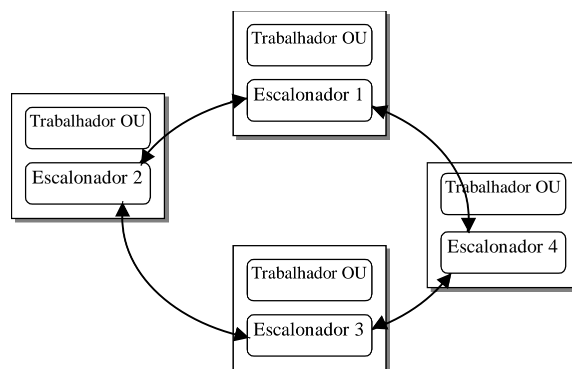

FIGURA 6.10 - Exemplo de Configuração de Nodos OU

Inicialmente todos os nodos possuem o estado *idle*, com exceção do nodo 1 que possui o estado *quiet* por ter iniciado a computação. Com o passar do tempo, novos pontos de escolha surgem no trabalhador 1 e o estado é alterado para *overloaded*. Neste momento, o nodo 1 envia mensagem de pedido de exportação de trabalho para ambos os vizinhos, uma vez que ambos estão disponíveis. O escalonador dois responde antes, e recebe o trabalho. O escalonador 4 recebe uma mensagem de cancelamento, conforme

figura 6.11. Nesta figura são apresentadas as mensagens trocadas pelos escalonadores bem como o estado de cada escalonador e de seus vizinhos. Os estados estão abreviados pela letra inicial, e não são repetidos quando se mantêm inalterados.

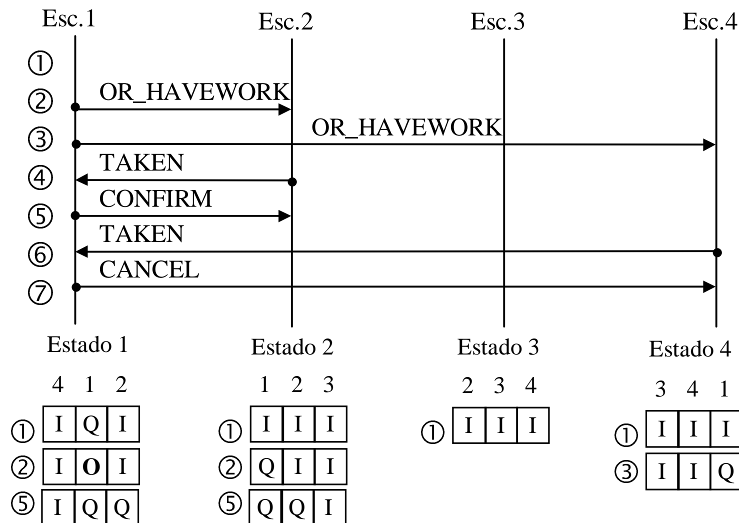

FIGURA 6.11 - Exportação de Trabalho com Vizinhos em Estado *idle*

Caso o trabalhador 1 fique novamente sobrecarregado (figura 6.12) é solicitada a exportação de trabalho apenas para o vizinho 4, pois este é o único em estado *idle*. Novas sobrecargas no nodo 1 fazem com que o pedido de trabalho seja propagado em ambos os sentidos no anel lógico. O estado *quiet* para um vizinho não é a garantia de que todo os nodos em determinada direção do anel estejam consumindo ou sobrecarregados. Essa situação é apresentada na figura 6.13.

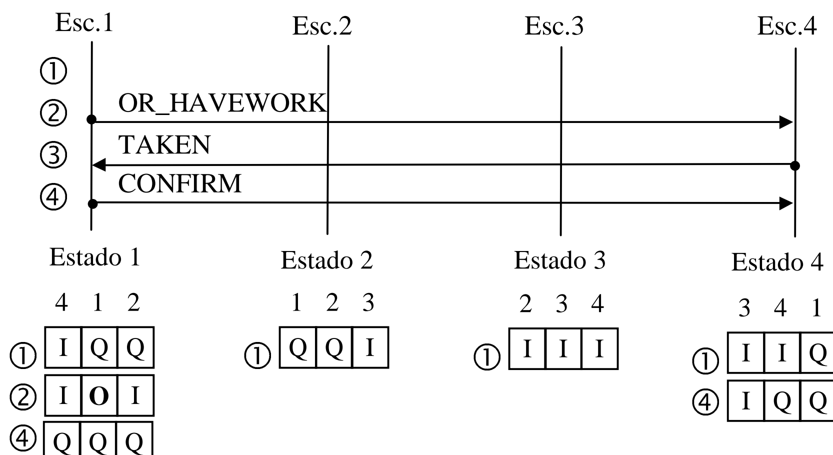

FIGURA 6.12 - Exportação de Trabalho com Apenas um Vizinho em Estado *idle*

Como pode ser observado na figura 6.13, quando a mensagem *OR_HAVEWORK* chega no escalonador 2 este a propaga para o próximo escalonador a direita, uma vez que o anel não foi totalmente percorrido nesta direção. Por sua vez, o escalonador 4 ao receber a mensagem *OR_HAVEWORK* responde com uma mensagem *NOIDLE*, uma vez que não é mais possível percorrer o anel nesta direção e o seu estado não é *idle*. O escalonador 3 responde a requisição de exportação de trabalho do nodo 1. A

comunicação é feita diretamente entre os dois nodos.

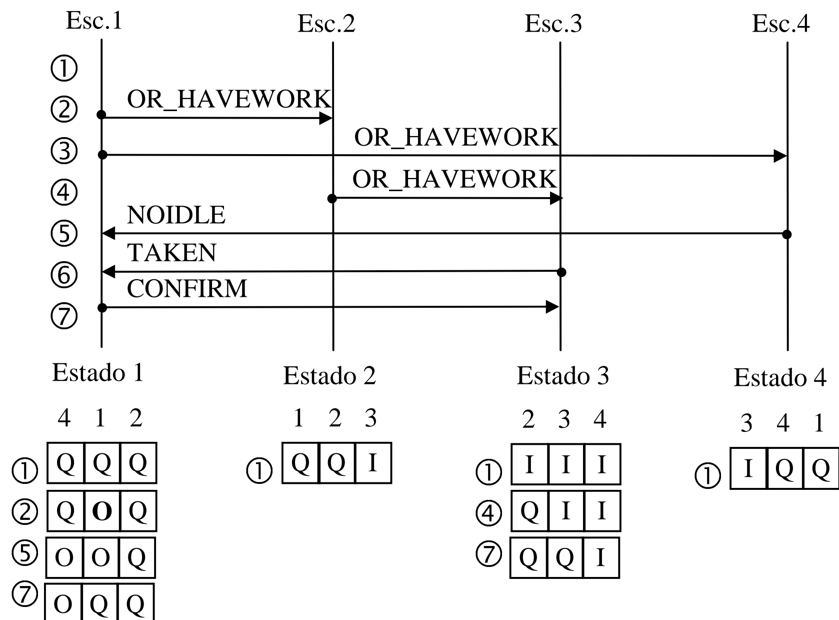

FIGURA 6.13 - Exportação de Trabalho com Vizinhos em Estado *quiet*

Se o escalonador 1 continuar sobrecarregado, é feita uma tentativa de exportação pela direita (uma vez que o estado ainda é *quiet*). Esta tentativa (figura 6.14) resulta em uma mensagem *NOIDLE*. Neste momento, apesar do nodo 1 estar sobrecarregado não é possível a exportação de trabalho pois todos os trabalhadores estão ocupados.

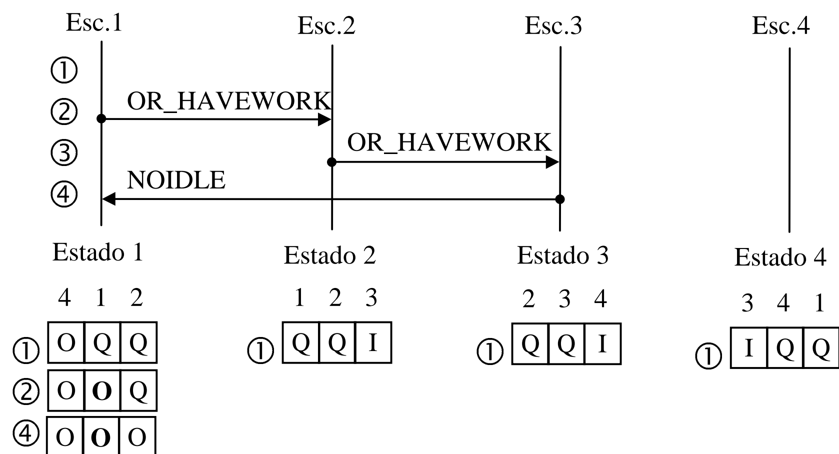

FIGURA 6.14 - Todos os Vizinhos Consumindo Trabalho

Quando um trabalhador fica novamente *idle* seu estado é propagado dentre seus vizinhos não *idle* no DSLP (figura 6.15). Um nodo OU ao receber a mensagem *OR_UPDATELOAD,* caso esteja sobrecarregado, envia imediatamente uma mensagem *OR_HAVEWORK*. Caso contrário, se o próximo vizinho na direção do anel estiver em estado *quiet,* continua a propagação da informação.

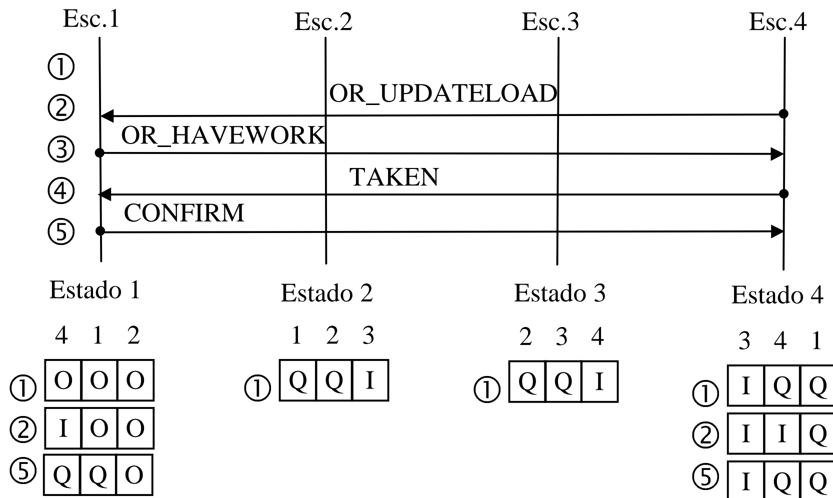

FIGURA 6.15 - Nodo Fica Disponível Novamente

Conforme pôde ser observado o estado do vizinho tem um significado diferente do estado do próprio trabalhador. O estado *overloaded* indica que não existem vizinhos livres em determinada direção do anel lógico. As possíveis mudanças de estados nos vizinhos ocorrem com mensagens *CONFIRM*, *NOIDLE* e *OR_UPDATELOAD*. A figura 6.16 apresenta resumidamente estas mudanças de estado.

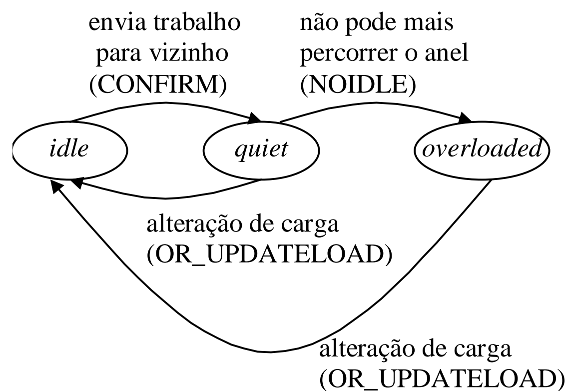

FIGURA 6.16 - Estados dos Vizinhos

6.3.2.2 Fase E

A fase E é iniciada com o envio da mensagem *AND_STARTFASE* do escalonador principal para os outros nodos escalonadores. Os escalonadores obtém os índices de importação iniciais dos nodos E vinculados. Além disso o nodo E com maior índice de importação começa a execução do programa em lógica.

Na fase de execução do paralelismo E (figura 6.17), o escalonador fica sempre esperando mensagens. As mensagens possíveis são: atualização de carga (*AND_UPDATELOAD*), atualização de índice de importação (*CPUQUEUECHANGE*), pedidos de exportação de trabalho para os vizinhos (*AND_HAVEWORK*), mudança no estado da carga E dos vizinhos (*AND_NEIGHBORSTATE*) e término da aplicação (*ENDAPPS* e *ENDNEIGHBOR*).

```
endComp := falso;
enquanto endComp <> verdadeiro faça
início
  addrSent := 0;
  tag := 0;
  recebe (addrSent, , tag);
  caso tag de
    AND_UPDATELOAD:   {atualização da carga E no vetor de carga de
                       nodos E vinculados e ativa escalonamento}
    CPUQUEUECHANGE:   {atualização no índice de importação do
                       nodo E que enviou a mensagem}
    AND_HAVEWORK:     {pedido de nodo E disponível por um dos
                       vizinhos}
    AND_NEIGHBORSTATE:{alteração no carga E do vizinho}
    ENDAPPS:          {terminada a execução da FASE E}
    ENDNEIGHBOR:      {terminada a execução da FASE E}
  fim
fim
```

FIGURA 6.17 - Fase E de Escalonamento

Caso o escalonador receba uma mensagem de atualização de carga de um nodo E vinculado (*AND_UPDATELOAD*) é executado o algoritmo da figura 6.18.

```
indexWkr := índice do nodo E que enviou a mensagem;
recebe (addrSpy[indexWkr], loadIndex [indexWkr], LOADINDEX);
recebe (addrSpy[indexWkr], loadState [indexWkr], LOADSTATE);
recebe (addrSpy[indexWkr], cpuQueueSize, CPUSIZE);
calculateImportIndex (importIndex[indexWkr], cpuQueueSize);
maxLoadIndex := 0;
indexWrk := -1;
{procura por nodo sobrecarregado de maior índice de carga}
para  indexAux := 0 até NumWkr – 1 faça
  se loadState [indexAux] = overloaded então
    se loadIndex [indexAux] > maxLoadIndex então
    início
      maxLoadIndex := LoadIndex [indexAux];
      indexWrk := indexAux;
    fim
maxImportIndex := 0;
indexImp := -1;
{caso tenha nodo sobrecarregado procura um disponível}
se indexWrk <> -1 então
início
{procura por nodo disponível dentre os vinculados com maior
 índice de importação}
  para indexAux := 0 até NumWkr – 1 faça
    se loadState[indexAux] = idle então
      se importIndex [indexAux] > maxImportIndex então
      início
        maxImportIndex := importIndex [indexAux];
        indexImp := indexAux;
      fim
  se indexImp<>-1 então {caso tenha disponível e sobrecarregado}
  início
    envia (addrWrk[indexImp], addrWrk [indexWrk], IMPORT);
    { Atualiza estados dos trabalhadores }
    recebe (addrSpy[indexWkr], loadIndex [indexWkr], LOADINDEX);
    recebe (addrSpy[indexWrk], loadState [indexWkr], LOADSTATE);
```

```
          recebe (addrSpy[indexWkr], cpuQueueSize, CPUSIZE);
          calculateImportIndex (importIndex[indexWkr], cpuQueueSize);
          recebe (addrSpy[indexImp], loadIndex [indexImp], LOADINDEX);
          recebe (addrSpy[indexImp], loadState [indexImp], LOADSTATE);
          recebe (addrSpy[indexImp], cpuQueueSize, CPUSIZE);
          calculateImportIndex (importIndex[indexImp], cpuQueueSize);
        fim
        senão {caso tenha que propagar no anel, não tem nodo
               vinculado livre}
          se HaveNeighbor = verdadeiro então {tem vizinhos}
          início
            se addrNeighbor[1] = 0 então {tem um vizinho apenas}
              envia (addrNeighbor[0], MaxStep e Myaddr, AND_HAVEWORK)
            senão
            {caso tenho um idle manda p/ ele. Senão manda p/ vizinho
             com maior índice de importação}
             se (NeighborState[0]=idle)E(NeighborState[1]<>idle) então
               envia (addrNeighbor[0], MaxStep e Myaddr, AND_HAVEWORK)
             senão
              se(NeighborState[1]=idle)E(NeighborState[0]<>idle) então
                envia (addrNeighbor[1],MaxStep e Myaddr, AND_HAVEWORK)
              senão
              início
                answers:=0;
                se NeighborState[0]<>overloaded então
                  envia(addrNeighbor[0],MaxStep e Myaddr,
                        AND_HAVEWORK);
                senão
                  answers := answers + 1;
                se NeighborState[1]<>overloaded então
                  envia (addrNeighbor[1],MaxStep e Myaddr,
                         AND_HAVEWORK);
                senão
                  answers := answers + 1;
              fim
          fim
          taken := 0;
          enquanto answers < 2 faça
          início
            {recebe de qualquer vizinho, mensagens de outros nodos
             não devem ser recebidas}
            addrTemp := 0;
            tag := 0;
            recebe (addrTemp, addrWrkIdle ,tag);
            se tag = TAKEN então
              se taken = 0 então
              início
                envia (addrTemp, ,CONFIRM);
                envia (addrWrkIdle, addrWrk [indexWrk], IMPORT);
                taken := 1;
                answers := answers + 1;
                se addrTemp = Neighbor[0] então
                  NeighborState[0] := quiet
                senão
                  NeighborState[1] := quiet;
              fim
              senão {trabalho já exportado. Cancelar o outro.}
              início
                envia(addrTemp, , CANCEL);
                answers := answers + 1;
              fim
```

```
      senão
      {tag é NOIDLE - os vizinhos estão sobrecarregados e seus
       vizinhos também, pois a mensagem é propagada. NOIDLE só é
       enviado quando o passo é zero.}
      início
        answers := answers + 1;
        se addrTemp = Neighbor[0] então
          NeighborState[0] := quiet
        senão
          NeighborState[1] := quiet;
      fim
    fim
    se taken = 1 então
    { Atualiza estado do trabalhador que exportou}
    início
      recebe(addrSpy[indexWkr],loadIndex [indexWkr], LOADINDEX);
      recebe(addrSpy[indexWrk],loadState [indexWkr], LOADSTATE);
      recebe(addrSpy[indexWkr], cpuQueueSize, CPUSIZE);
      calculateImportIndex (importIndex[indexWkr],cpuQueueSize);
    fim
fim
{Avisa vizinho se mudou de IDLE p/ QUIET ou vice-versa}
se HaveNeighbor = verdadeiro então {tem vizinhos}
início
  haveIdle := falso;
  para aux := 0 até aux < NumWrk – 1 faça
    se loadState[aux] = idle então
      haveIdle := verdadeiro;
  se (haveIdle = verdadeiro E andState = quiet) então
  início
    andState := idle;
    envia (addrNeighbor[0], andState, AND_NEIGHBORSTATE);
    se addrNeighbor[1] <> 0 então {tem dois vizinhos}
      envia (addrNeighbor[1], andState, AND_NEIGHBORSTATE);
  fim
  se (haveIdle = falso E andState = idle) então
  início
    andState := quiet;
    envia (addrNeighbor[0], andState, AND_NEIGHBORSTATE);
    se addrNeighbor[1] <> 0 então {tem dois vizinhos}
      envia (addrNeighbor[1], andState, AND_NEIGHBORSTATE);
  fim
fim
```

FIGURA 6.18 - Algoritmo de Escalonamento para os Nodos E

Quando o escalonador recebe a mensagem *AND_UPDATELOAD* de um espião esta contém a carga, o índice (complexidade do nodo) e o tamanho da fila da UCP (para cálculo do índice de importação) de um nodo E vinculado. A primeira tarefa do algoritmo é procurar por algum nodo E vinculado que esteja sobrecarregado. Caso não exista nenhum nodo com carga alta o escalonador envia mudanças na sua carga E para os vizinhos, ficando no aguardo de outra mensagem. Com isso é encerrada a execução do algoritmo apresentado.

O escalonador, quando possuir nodos sobrecarregados, escolhe aquele com o maior índice para exportação. É procurado, então, um nodo E disponível. Caso exista algum nodo E *idle* vinculado ao escalonador, o nodo com o maior índice de importação

é selecionado e a migração de tarefas concretiza-se. Porém pode ocorrer de todos os nodos E vinculados estarem consumindo tarefas (em estado *quiet* ou *overloaded*). Neste caso, mensagens de pedido de exportação de trabalho (*AND_HAVEWORK*) são encaminhadas aos escalonadores vizinhos em estado *idle*. Quando ambos os vizinhos estão em estado *quiet*, as mensagens *AND_HAVEWORK* são enviadas pois existe a possibilidade de outros nodos E estarem disponíveis no sistema. Somente quando os vizinhos estão *overloaded*, mensagens de pedido de exportação de trabalho não são enviadas. A requisição de exportação de trabalho aos vizinhos funciona de maneira análoga ao escalonamento OU.

Sempre que uma exportação ou importação ocorre são feitas atualizações no sistema para garantir consistência. Novas distribuições de tarefas podem ser feitas a cada alteração de carga no sistema. As alterações na fila de espera da UCP (*CPUQUEUECHANGE*) são enviadas quando o estado do nodo é *idle* e ocorrem mudanças significativas nesta fila. Em sistemas dedicados apenas à execução de uma aplicação no DSLP este dado é desconsiderado.

Mensagens *AND_NEIGHBORSTATE* são propagadas somente quando o próprio estado é alterado novamente para *idle*, isto é, não existiam nodos E em estado *idle* e agora existe pelo menos um nodo E vinculado disponível. Mudanças para *quiet* não são informadas aos vizinhos para minimizar o número excessivo de mensagens. Mudanças para *overloaded* causam tentativas de exportação de trabalho.

O algoritmo de recebimento de pedido de exportação de trabalho dos vizinhos (*AND_HAVEWORK*) é apresentado na figura 6.19.

```
recebe (addrTemp ,step e addrStart, AND_HAVEWORK);
se (addrTemp = Neighbor[0]) então
início
  se (NeighborState[0] = idle) então
    NeighborState[0] := quiet;
fim
senão
  se NeighborState[1] = idle então
    NeighborState[1] := quiet;
{procura por nodo disponível dentre os vinculados com maior
 índice de importação}
maxImportIndex := 0;
indexImp := -1;
para indexAux := 0 até NumWkr - 1 faça
  se loadState[indexAux] = idle então
    se importIndex [indexAux] > maxImportIndex então
    início
      maxImportIndex := importIndex [indexAux];
      indexImp := indexAux;
    fim
    se indexImp<>-1 então {caso tenha nodo disponível}
    início
      envia (addrStart, addrWrk[indexImp], TAKEN);
      recebe (addrStart, ,tag);
      se tag = CONFIRM então
      início
        { Atualiza estado do trabalhador}
        recebe (addrSpy[indexImp],loadIndex[indexImp], LOADINDEX);
        recebe (addrSpy[indexImp],loadState[indexImp], LOADSTATE);
        recebe (addrSpy[indexImp], cpuQueueSize, CPUSIZE);
```

```
        calculateImportIndex (importIndex[indexImp],cpuQueueSize);
      fim
    fim
    senão {nenhum nodo E vinculado idle}
    início
      step := step - 1; {decrementa o passo}
      se step > 0 então
      início
        se addrTemp = addrNeighbor[0] então
          indexTemp := 1 {descobre índice do outro vizinho}
        senão
          indexTemp := 0;
        se NeighborState[indexTemp]<>overloaded então
          envia(addrNeighbor[indexTemp],step e addrStart,
                AND_HAVEWORK)
        senão
          envia(addrStart, , NOIDLE);
      fim
      senão
        envia(addrStart, , NOIDLE);
    fim
```

FIGURA 6.19 - Algoritmo de Recebimento de Trabalho E pelos Vizinhos

No recebimento de uma mensagem *AND_HAVEWORK* o escalonador inicia a busca por um nodo E com o estado *idle*. Caso o nodo seja encontrado nos nodos E vinculados ou através de seu outro vizinho, uma mensagem *TAKEN* é enviada diretamente ao escalonador que originou o pedido. Em caso contrário, uma mensagem *NOIDLE* é enviada, causando uma atualização no estado do vizinho para *overloaded*. Essa mensagem só é enviada no caso de não ser mais possível propagar mensagens *AND_HAVEWORK* no anel lógico ou se o próximo vizinho estiver no estado *overloaded*.

Para ilustrar a fase E de escalonamento considere a configuração da figura 6.20. Neste exemplo, existem três nodos OU (com escalonadores) e seis nodos E (dois associados a cada escalonador disponível).

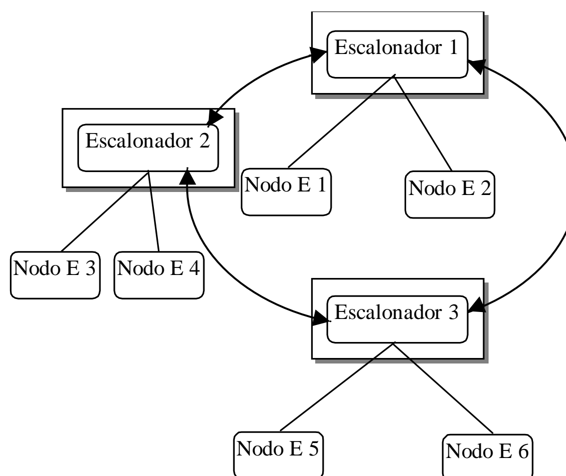

FIGURA 6.20 - Exemplo de Configuração de Nodos E

Inicialmente todos os escalonadores possuem estado *idle* para os nodos E associados e para os escalonadores vizinhos, com exceção do nodo E 1 que possui estado *quiet*. A figura 6.21 mostra a exportação de trabalho entre nodos E vinculados a um mesmo escalonador. São apresentados os estados dos trabalhadores associados e dos escalonadores vizinhos, de forma análoga à apresentada na seção anterior.

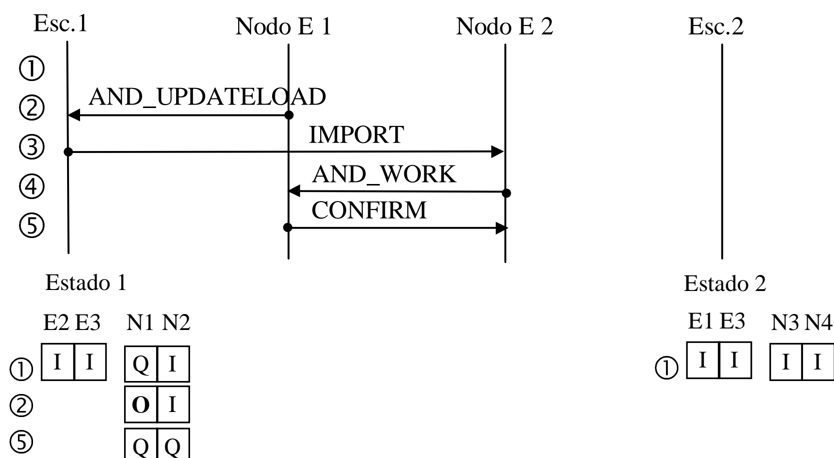

FIGURA 6.21 - Exportação de Trabalho Entre Nodos E Vinculados

Através da mensagem *AND_UPDATELOAD* enviada pelo nodo E 1 é detectada a existência de metas paralelizáveis (estado *overloaded*). Então, como existe outro nodo em estado *idle* vinculado ao mesmo escalonador ele é eleito como candidato imediato para importação. Caso exista mais de um nodo E associado disponível, é selecionado aquele com maior índice de importação. Caso não existam nodos E vinculados livres é realizada a propagação de mensagens *AND_HAVEWORK* no anel, de maneira semelhante à fase OU de escalonamento.

6.3.2.3 Cálculo do Índice de Importação

O índice de importação é utilizado quando o escalonador possui mais de um trabalhador livre no momento da exportação de tarefas. Ele só é aplicado na fase E de escalonamento, pois a fase OU exporta através do anel lógico para vizinhos livres. A partir do valor obtido como índice é determinado qual o processador mais adequado (com uma sobrecarga menor, maior poder computacional, etc.) para executar a tarefa.

O cálculo do índice de importação faz uso do tamanho da fila de espera da UCP, poder computacional das máquinas e velocidade da rede. O tamanho da fila de espera da UCP é obtido dinamicamente, enquanto que o poder computacional e a velocidade da rede são estáticos.

Quando o sistema é homogêneo, isto é, processadores idênticos interligados por uma rede com velocidade de transmissão de dados constante, somente a fila de espera da UCP é utilizada. Caso o sistema seja heterogêneo, é utilizado no cálculo do índice, de acordo com o caso, a velocidade da rede e o poder computacional. A tabela 6.1 sintetiza as heurísticas utilizadas em cada caso para o cálculo do índice de importação,

com seus respectivos pesos. É utilizado um peso maior para o tamanho da fila da UCP pelos motivos expostos no capítulo anterior.

TABELA 6.1 - Cálculo de Índices de Importação

Sistemas heterogêneos	Heurística	Peso
Totalmente	1 / Tamanho da Fila da UCP	60%
	Velocidade da Rede	20%
	Poder Computacional	20 %
Velocidade da rede Idêntica	1 / Tamanho da Fila da UCP	80%
	Poder Computacional	20%
Poder computacional Idêntico	1 / Tamanho da Fila da UCP	80%
	Velocidade da Rede	20%

Sistemas homogêneos	Heurística	Peso
	1 / Tamanho da Fila da UCP	100%

Caso o sistema seja dedicado para a execução do DSLP, isto é, não existirem outras aplicações em execução simultânea, a heurística tamanho da fila de espera da UCP pode ser desconsiderada.

6.3.3 Trabalhador OU

Esta seção apresenta as modificações necessárias no trabalhador do OPERA OU para suportar o modelo DSLP. Basicamente, deve ser adicionada a gerência das informações de granulosidade do GRANLOG. O trabalhador utilizado no OPERA OU está descrito em [MOR 96] e [GEY 91].

O trabalhador OU é o processo responsável pela exploração do paralelismo OU no sistema. Para tal, o trabalhador mantém na *stack* os pontos de escolha que podem ser executados em paralelo. Estes pontos de escolha são armazenados na pilha juntamente com suas respectivas complexidades OU. Em função desta complexidade OU é determinado qual ponto de escolha será exportado.

O trabalhador mantém em um registrador a complexidade total dos pontos de escolha armazenados na *stack*. Esta complexidade total é determinada a partir da soma de todas as complexidades OU dos pontos de escolha. Através desta complexidade total, é armazenado o estado do trabalhador na variável compartilhada *OrState*. A determinação do estado depende do valor *overloaded level* na tabela SRT.

Para calcular a *complexidade do ponto de escolha* (CPE[2]) são necessárias as seguintes informações: *complexidade da meta* (CM) que originou o ponto de escolha, *complexidade da primeira cláusula do procedimento* (CCpri), número de cláusulas no procedimento (N) e o *acumulado das complexidades das resolventes pendentes locais* (ACRPL). As informações CM e Ccpri são disponibilizadas pelo GRANLOG. O

[2] Os nomes das variáveis utilizadas para análise de granulosidade no DSLP foram deixadas em português para compatibilidade com a nomenclatura do modelo GRANLOG;

número de cláusulas no procedimento é facilmente obtido em tempo de execução. Já o ACRPL deve ser calculado dinamicamente.

O GRANLOG determina em cada trecho do programa o valor da *complexidade da resolvente pendente local* (CRPL). Sempre que determinada cláusula é computada, o trabalhador OU utiliza a CRPL para incrementar o ACRPL. Esta informação indica a cada momento a complexidade das resolventes pendentes. A resolvente pendente deve ser calculada em tempo de execução uma vez que depende dos caminhos percorridos na árvore.

A complexidade OU de um ponto de escolha é calculada através da seguinte fórmula:

$$CPE = CM - CCpri + (N-1) * ACRPL$$

A complexidade do ponto de escolha é obtida através da complexidade da meta que originou o ponto de escolha acrescida da resolvente pendente multiplicada pelo número de cláusulas ainda não tentadas. A complexidade da primeira cláusula do procedimento (a cada momento) não é considerada, uma vez que ele é imediatamente executado.

A cada novo ponto de escolha armazenado na *stack*, o registrador que armazena a *complexidade total dos pontos de escolha* (CTPE) é incrementado com a complexidade do mesmo. Juntamente com o ponto de escolha é armazenado na *stack* a CPE e valor do ACRPL naquele momento. Esse último valor é necessário para calcular a complexidade de um ponto de escolha no momento do *backtracking*.

Quando ocorre o retrocesso, a CPE do ponto exportado deve ser decrescida de CCpri + ACRPL, isto é, a complexidade da cláusula a ser executada mais o valor da resolvente pendente no momento da criação desta. A CTPE também é atualizada.

O OPERA utiliza a cópia de pilhas para gerenciar as resolventes pendentes. A cada exportação de trabalho, é copiada uma porção das pilhas para um outro trabalhador. Neste momento, a CTPE é decrementada com a complexidade do ponto de escolha enviado. O ponto de escolha exportado é aquele que possui a maior CPE dentre os pontos armazenados na *stack*.

Quando um trabalhador recebe um ponto de escolha ele determina a execução imediata da primeira cláusula do procedimento. A CPE é decrementada de CCpri + ACRPL, isto é, a complexidade da primeira cláusula do procedimento acrescida do valor do ACRPL armazenado na *stack*.

Os pontos de escolha exportados na *stack* são marcados através de uma variável lógica (*Alive*) como falsos. Para facilitar o gerenciamento de memória eles não são removidos da *stack*. Todos os outros pontos de escolha não exportados possuem o estado de vivo (com o valor de *Alive* igual a verdadeiro) na variável lógica.

Quando determinado ponto de escolha é exportado, o trabalhador importador verifica seus pontos de escolha mortos. Estes pontos de escolha podem ser comuns aos pontos de escolhas do trabalhador exportador. Caso exista pelo menos um ponto de escolha em comum nas pilhas dos trabalhadores exportador e importador é possível efetuar a cópia incremental. Desta forma apenas os pontos de escolha mais recentes, ou

seja, entre o último ponto de escolha em comum e o exportado, precisam ser copiados. Com isso, os dados enviados de um trabalhador para outro podem ser reduzidos drasticamente.

A figura 6.22 apresenta todas as informações armazenadas na *stack*. As informações que foram acrescentadas estão destacadas das demais. Maiores detalhes sobre a *stack* são encontrados na seção 2.3.

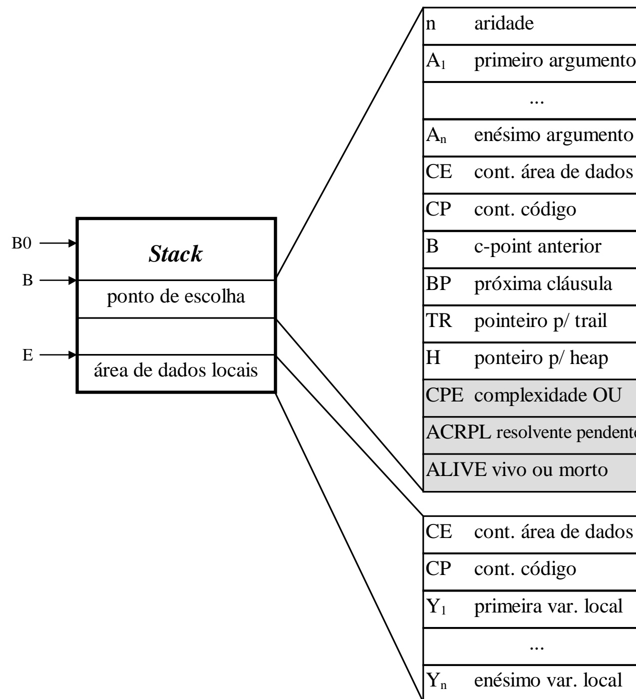

FIGURA 6.22 - Informações Armazenadas na *Stack*

6.3.4 Trabalhador E

De forma análoga a seção anterior, aqui são apresentadas as modificações necessárias no trabalhador E para dar suporte ao modelo DSLP. O trabalhador E utilizado é o OPERA E ([WER 94a] e [YAM 94]).

O trabalhador E contém uma máquina abstrata para a execução de programas em lógica explorando o paralelismo E Independente, semelhante ao trabalhador do OPERA E. Sempre que surgem metas com potencial para paralelização (determinado através de análise de granulosidade) estas são colocadas na lista de objetivos paralelos – *Parallel Goals List* (PGL). Caso contrário, as metas são armazenadas na lista de objetivos seqüenciais[3] – *Sequential Goals List* (SGL).

Quando uma nova meta é encontrada pela máquina abstrata, sua complexidade é avaliada. A meta é colocada na PGL caso sua complexidade seja maior que o custo de exportação, isto é, o tempo necessário para enviar e receber os dados necessários para seu processamento, considerando a velocidade de comunicação, é maior que o tempo de

[3] A WAM gerencia as metas não paralelizáveis sem a necessidade da criação de uma estrutura adicional. Porém por questões de simplificação e necessidade no protótipo ela foi criada.

execução da meta. As metas na PGL são armazenadas em ordem de complexidade, sendo as mais complexas armazenadas no início e as menos complexas no final.

A complexidade de meta utilizada pelos trabalhadores do DSLP é a complexidade simplificada ao invés da avaliação das expressões de complexidades. Essa decisão ocorreu em função do custo adicional envolvido no cálculo das expressões de granulosidade em tempo de execução.

Sempre que o trabalhador E termina de executar uma meta, ele verifica a existência de metas na SGL. Caso a SGL contenha alguma meta, esta será executada. Em caso oposto, o trabalhador tenta executar alguma meta da PGL.

O trabalhador E entra em estado *overloaded* quando a complexidade total da PGL for maior que o *overloaded level* determinado na ACT. Quando o escalonador detecta a sobrecarga e determina a exportação, é enviada a primeira meta da PGL, ou seja, a tarefa de maior granulosidade.

A figura 6.23 apresenta um resumo da utilização das informações de granulosidade pelo trabalhador E. Sempre que a complexidade da meta é maior que o custo de exportação a meta é armazenada na PGL, ordenada por granulosidade. Caso contrário ela é inserida no fim da SGL.

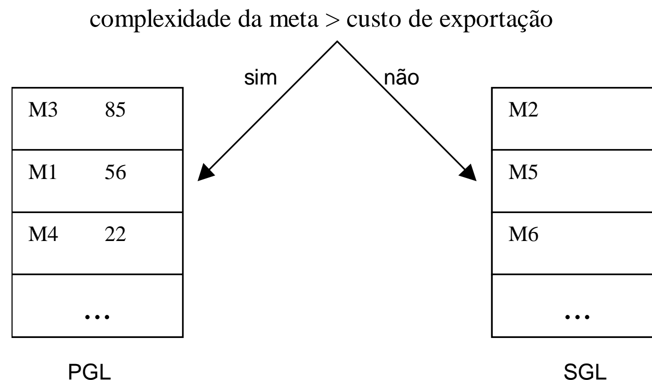

FIGURA 6.23 - Uso da Granulosidade no Trabalhador E

No mesmo processador do trabalhador E existe um processo espião (*spy*). Este processo periodicamente avalia a complexidade da PGL e mantém o escalonador informado das mudanças de estado. É utilizado um processo *spy* pois o trabalhador E não encontra-se no mesmo processador que o escalonador. Com isso, só são informadas ao escalonador mudanças significativas de carga, não onerando o sistema com mensagens desnecessárias.

6.3.5 Espião (*spy*)

Espião é o processo responsável por manter o escalonador informado das variações de carga do trabalhador E. Este processo está presente em todos os processadores que possuem trabalhadores E.

O processo *spy* fica inativo enquanto o trabalhador E está em estado *idle*.

Quando o trabalhador recebe trabalho e passa para *quiet*, o espião começa a monitorar o processo. O espião fica inativo um tempo determinado pelo valor definido na variável *spy sleep time* da tabela ACT. Esta inatividade, faz com que o número de troca de mensagens entre o espião e o trabalhador E não seja muito grande.

O processo *spy* verifica a complexidade da lista de objetivos paralelos do trabalhador E. Sempre que existir uma alteração na complexidade que altere o estado do trabalhador, este estado é propagado ao escalonador. Quando o trabalhador está em estado *overloaded*, mudanças na carga maiores que o valor da variável *load range* da tabela ACT também são propagadas ao escalonador. Inalterações no estado ou mudanças pouco significativas na carga não são enviadas para não onerar os custos de comunicação do sistema.

6.4 Extensões ao DSLP

O presente modelo é a base para o desenvolvimento de diversos trabalhos futuros. Nesta seção serão destacadas algumas possíveis extensões ao DSLP:

- **determinação automática dos nodos:** através de uma análise da aplicação e do sistema é possível que o DSLP determine a melhor configuração possível, isto é, a quantidade de escalonadores e trabalhadores no sistema bem como em quais processadores cada um residirá. As informações geradas pelo GRANLOG servem de base para a determinação de trabalhadores, uma vez que é possível saber a quantidade de paralelismo E Independente e OU existente na aplicação. No momento da criação da arquitetura o ambiente pode realizar testes para determinar automaticamente as velocidades de comunicação e os poderes computacionais dos processadores. Com isso, o DSLP poderia decidir por uma configuração adequada para cada problema, liberando o usuário desta tarefa;

- **reconfiguração dinâmica**: determinada a configuração inicial, o DSLP não necessitaria manter esta configuração durante todo o processamento do problema. O escalonador poderia alterar a quantidade de trabalhadores E Independente e OU no sistema. Caso em determinado momento seja necessária uma vasta exploração do paralelismo OU, um número maior de processadores seriam alocados para esta tarefa e vice-versa. Além disso, trabalhadores vinculados a um escalonador poderiam migrar para outro com mais trabalho a ser realizado. Vários estudos vem sendo realizados neste sentido. É possível destacar [DUT 95];

- **tolerância a falhas:** no DSLP podem ser empregados protocolos que permitam a detecção automática de falhas. Isso é facilitado uma vez que não é utilizada uma política de escalonamento centralizada. Um nodo pode facilmente verificar a falha de outro reorganizando o sistema para que continue a resolução do problema;

- **outros modelos de execução:** podem ser aplicados ao DSLP novos modelos de execução que suportem a exploração simultânea de paralelismo E Independente e OU, bem como a exploração de novas fontes de paralelismo (paralelismo E Dependente, por exemplo). Existem vários outros modelos de execução implementados para a exploração do paralelismo na programação

em lógica. Para uma visão detalhada destes modelos consultar [COS 96a].

6.5 Conclusões

O presente capítulo apresentou o DSLP – *Distributed Scheduler for Logic Programming*. O DSLP é um escalonador para programação em lógica que integra o paralelismo E Independente e OU. O modelo utiliza informações de granulosidade da aplicação Prolog no auxílio ao escalonamento. É uma proposta hierárquica, dinâmica e que não faz uso de primitivas de *broadcast*.

Inicialmente foram apresentados os princípios que nortearam a criação do modelo bem como uma visão geral de seu funcionamento. A arquitetura e o funcionamento do modelo foram detalhados, destacando as duas fases de escalonamento.

Os principais algoritmos do sistema tanto para o escalonamento OU, como para o escalonamento E foram apresentados. Na seqüência, o uso das informações de granulosidade foi detalhado, bem como o funcionamento de cada um dos componentes do modelo.

Finalmente, foram propostas extensões ao sistema, que poderão gerar trabalhos futuros.

7 Protótipo do Escalonador DSLP

Este capítulo apresenta o protótipo implementado. Parte de sua implementação foi realizada como Trabalho de Conclusão de Curso de Ciência da Computação da UFRGS. Para mais informações sobre o protótipo consultar [TRE 97].

7.1 Descrição do Protótipo

O protótipo implementado é uma simplificação do modelo DSLP, estando limitado à exploração apenas do paralelismo E Independente. Esta opção por dedicação ao paralelismo E Independente ocorreu basicamente pelos seguintes motivos:

- o paralelismo E é mais presente nas aplicações da programação em lógica do que o paralelismo OU;

- o projeto OPERA na UFRGS, ao qual o trabalho encontra-se inserido, dedica-se principalmente ao estudo do paralelismo E ([WER 94a] e [YAM 94]);

- a implementação do GRANLOG disponível somente fornece informações de granulosidade para o paralelismo E;

- o modelo de escalonamento proposto pelo DSLP, para o paralelismo OU, é também empregado na exploração do paralelismo E.

Os nodos OU do modelo DSLP são dedicados exclusivamente ao escalonamento de tarefas no protótipo (não existem trabalhadores OU) e portanto, são chamados de *nodos Escalonadores*. O protótipo permite que um mesmo processador possua simultaneamente um nodo E e um nodo Escalonador. Cabe ressaltar que apesar dos processos poderem compartilhar a mesma UCP, eles comunicam-se através de troca de mensagem.

O protótipo do DSLP é um simulador de programas em lógica. No momento da implementação do mesmo, o OPERA E não estava operacional e por isso sua máquina abstrata não pôde ser utilizada. Este problema foi solucionado com a criação de um simulador para execução de programas Prolog (descrito na seção 7.2). O simulador faz o uso das informações de granulosidade geradas pelo GRANLOG.

7.1.1 Ferramentas Utilizadas

Para a implementação do protótipo foi utilizado o ambiente *Parallel Virtual Machine* - PVM ([GEI 94]). O PVM é um ambiente que permite a utilização de uma coleção de computadores heterogêneos como uma máquina paralela. Os computadores individuais podem ser multiprocessadores com memória compartilhada ou distribuída, supercomputadores vetoriais, arquiteturas superescalares, etc., interconectados por redes *ethernet*, ATM etc.

Através do PVM um programa pode executar em cada computador da máquina virtual paralela. Programas escritos em C ou em FORTRAN acessam a plataforma paralela através do uso de uma biblioteca com as funções PVM, tais como: enviar e receber mensagens, sincronizar, dentre outras. Através de uma interface gráfica,

chamada XPVM, é possível a visualização de toda a dinâmica do sistema, bem como o controle das tarefas paralelas e do ambiente.

Para a implementação dos programas utilizando a biblioteca PVM foi escolhida a linguagem C. Sua ampla utilização aliada à facilidade de manutenção e uso em desenvolvimento de software básico foram os motivos que levaram ao seu emprego no projeto.

7.1.2 Organização Básica

O protótipo organiza a arquitetura em dois tipos de nodos, a saber: nodos E (trabalhadores) e nodos Escalonadores. A tabela ACT define quais processadores contêm cada um dos nodos, além de outras configurações previamente descritas (seção 6.3.1). A opção de escolha automática pelo sistema da quantidade de nodos escalonadores não foi descrita. A tabela SRT descreve as máquinas que podem ser empregadas na computação e seus respectivos poderes computacionais. A especificação da velocidade de rede não foi implementada, uma vez que de forma análoga a UFRGS muitas instituições de ensino trabalham com redes homogêneas (via de regra *ethernet*). Exemplos de tabelas SRT e ACT são apresentadas nos Anexos 1 e 2 respectivamente.

A figura 7.1 apresenta a arquitetura do protótipo DSLP.

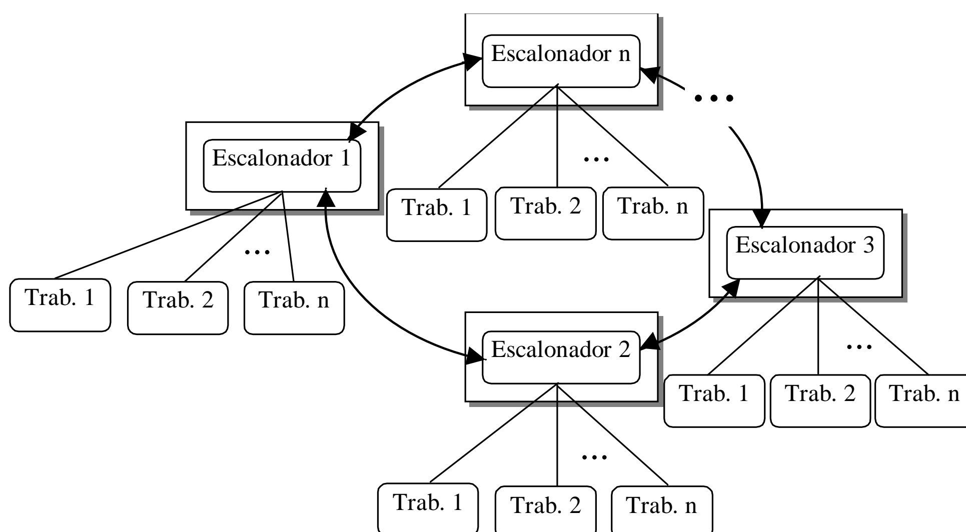

FIGURA 7.1 - Arquitetura do Protótipo DSLP

Cada trabalhador é composto de dois processos: o *solver* e o *spy*. O processo *solver* é o simulador de programas em lógica. Já o *spy,* é o responsável pela avaliação de carga do trabalhador e informação ao escalonador (seção 6.3.5). Foram implementados também os programas para manipulação da arquitetura *createarch* e *startarch* previamente descritos (seção 6.3). O módulo *changearch* não foi implementado por não ter sido considerado essencial ao funcionamento do sistema.

Para a execução de um programa no DSLP, além das tabelas ACT e SRT utilizadas para a criação da arquitetura, é necessário o programa Prolog convertido para o código do simulador. O programa simulado deve estar presente em todos os

processadores que contêm trabalhadores E na arquitetura do DSLP, seja no sistema de arquivos local ou através de *Network File System* (NFS).

Para executar o programa inicialmente deve-se criar a arquitetura com o módulo *createArch*. O módulo deve receber como argumento o nome do arquivo que contêm as tabelas ACT e SRT. Caso seja especificada a diretiva *-g*, o escalonador não fará o uso das informações de granulosidade.

Com a arquitetura montada para executar a aplicação basta chamar o módulo *startArch*. Este módulo recebe como parâmetro o nome do arquivo a ser executado. O código simulado e as tabelas da aplicação e do sistema utilizam as extensões .sim, .act e .srt respectivamente.

7.2 Simulador de Programas Prolog

Para que a eficiência do escalonador possa ser avaliada o simulador utiliza programas sintéticos. Estes programas são basicamente arquivos que contêm todo o percurso da árvore de execução Prolog, acrescidos de algumas instruções necessárias. A figura 7.2 apresenta um código do simulador para execução do fibo de 5.

```
CODE 0
  FIBO5N
  M>1
  M1IS4
  M2IS3
  PARALLEL 1
    FIBO4N1
    ETL 9.0681
  FIBO3N2
  M>1
  M1IS2
  M2IS1
  PARALLEL 2
    FIBO2N1
    ETL 3.0168
  FIBO1N2
  FIBO11
  RESULT 2
  NIS2
  RESULT 1
  NIS5
ENDCODE 0
CODE 1
  M>1
  M1IS3
  M1IS2
  PARALLEL 3
    FIBO3N1
    ETL 5.0336
  FIBO2N2
  M>1
  M1IS2
  M2IS1
  PARALLEL 4
    FIBO2N1
    ETL 3.0168
  FIBO1N2
  FIBO11
  RESULT 4
  NIS1
  RESULT 3
  NIS3
ENDCODE 1
CODE 2
  M>1
  M1IS1
  M2IS0
  PARALLEL 5
    FIBO1N1
    ETL 1.0080
  FIBO0N2
  FIBO00
  RESULT 5
  NIS1
ENDCODE 2
CODE 3
  M>1
  M1IS2
  M1IS1
  PARALLEL 6
    FIBO2N1
    ETL 3.0168
  FIBO1N2
  FIBO11
  RESULT 6
  NIS2
ENDCODE 3
CODE 4
  M>1
  M1IS1
  M2IS0
  PARALLEL 7
    FIBO1N1
    ETL 1.0080
  FIBO0N2
  FIBO00
  RESULT 7
  NIS1
ENDCODE 4
CODE 5
  FIBO11
ENDCODE 5
CODE 6
  M>1
  M1IS1
  M2IS0
  PARALLEL 8
    FIBO1N1
    ETL 1.0080
  FIBO0N2
  FIBO00
  RESULT 8
  NIS1
ENDCODE 6
CODE 7
  FIBO11
ENDCODE 7
CODE 8
  FIBO11
ENDCODE 8
```

FIGURA 7.2 - Código do Fibo de 5 para o Simulador

As metas a serem processadas pelo simulador são representadas por uma *string* qualquer, diferente das palavras-reservadas (em negrito no exemplo). Sempre que o simulador encontrar uma meta é esperado o tempo equivalente à execução de um resolução. Este tempo é configurado pelo usuário, podendo refletir a velocidade de execução da WAM desejada.

As palavras-reservadas utilizadas no código do simulador são as seguintes instruções: *CODE*, *ENDCODE*, *PARALLEL*, *ETL* e *RESULT*. Estas instruções podem receber de um a três argumentos, de acordo com o caso. Os argumentos são separados por espaços e/ou avanços de linha. A seguir cada uma delas é descrita.

- *CODE*: indica o início de um segmento de código. Deve ser sempre seguido de um número. O início da simulação começa no *CODE 0*. Cada segmento de código deve ser encerrado com a palavra-reservada complementar *ENDCODE*;

- *ENDCODE*: representa o término de um segmento de código. Para facilitar a leitura do programa, é necessário acrescentar o número do segmento de código finalizado. Sendo assim, a simulação termina no *ENDCODE 0*;

- *PARALLEL*: informa que determinada meta é candidata à paralelização. É seguida do número do segmento de código que contém a árvore de execução da meta. Além deste argumento é passado também a meta a ser paralelizada e as informações de granulosidade (*ETL*);

- *ETL*: possui como argumento a informação de granulosidade de uma meta, isto é, a complexidade da meta a ser paralelizada. Durante a execução será decidido se a meta será armazenada na PGL ou SGL de acordo com estas informações;

- *RESULT*: indica que determinado resultado deve estar disponível para a continuidade da execução do programa. Como argumento recebe o número do segmento de código que deverá prover o resultado.

Durante a execução do simulador podem ser recebidas ou enviadas mensagens de importação ou exportação de trabalho, pedido de situação de carga do *spy,* envio de resultados e informação de término da computação. Entre a execução de cada uma das instruções do código simulado é feita uma verificação nas mensagens recebidas. A figura 7.3 apresenta o algoritmo do simulador.

```
endComp := falso;
addrSent := 0;
tag := 0;
local := 0;
export := 0;
import := 0;
granuPGL := 0;
mainWork := falso;
recebe (addrSent, ,tag);
se tag = STARTCOMPUTATION então
  abrir (arqSim);
ler (arqSim, word);
enquanto word <> "CODE" faça
```

```
  ler (arqSim, word);
ler (arqSim, numCode);
enquanto endComp <> verdadeiro faça
início
se temMensagem ()= verdadeiro então
início
  addrSent := 0;
  tag := 0;
  recebe (addrSent, info ,tag);
  caso tag de
    PROBE:     envia(addrSent, granuPGL , PROBEANSWER);
    PRINCIPAL:início
                 putList (EGL, numCode 0 LOCAL 0 e 0);
                 ler (arqSim, word);
                 local := local + 1;
                 mainWork := verdadeiro;
               fim
    AND_WORK:  se getList (PGL, auxCode granu) = verdadeiro então
               início
                 granuPGL := granuPGL – granu;
                 envia (addrSent, auxCode, CONFIRM);
                 putList (EGL, auxCode 0 EXPORT addrSent e numCode);
                 export := export + 1;
               fim
               senão
                 envia (addrSent, , CANCEL);
    IMPORT:    início {info contém o TID do exportador}
                 tidImp = info;
                 envia (tidImp, ,AND_WORK);
                 tag := 0;
                 recebe (tidImp,auxCode,tag);
                 se tag = CONFIRM então
                 início
                   arqPos := posição(arqSim);
                   putList(EGL, auxCode arqPos IMPORT tidImp numCode);
                   tmpCode = -1;
                   enquanto tmpCode <> auxCode faça
                   início
                     enquanto word <> "CODE" faça
                       ler (arqSim, word);
                     ler (arqSim, tmpCode);
                   fim
                   numCode := auxCode;
                   import := import + 1;
                 fim
               fim
    RESULT:    putList(LR, info); {coloca resultado recebido na lista}
    ENDAPPS:   início
                 endComp := verdadeiro;
                 word := "CODE";
               fim
  fim
  caso word de
    "CODE":      parar
    "PARALLEL":  início
                   ler (arqSim, auxCode); {algumas leituras do arquivo}
                   ler (arqSim, granu);   {foram eliminadas}
                   se granularity = verdadeiro então
                     se granu > cost então
                     início
                       granuPGL := granuPGL + granu;
```

```
                    putList (PGL, auxCode granu);
                  fim
                  senão
                    putList (SGL, auxCode granu);
                fim
                senão
                  putList(PGL, numcode 0);
                ler (arqSim, word);
              fim
"RESULT":     início
                ler (arqSim, numCode);
                word := "WAIT";
              fim
"WAIT":       se getList(LR, numCode) = verdadeiro então
              início
                getList (EGL, numCode arqPos type tidImp auxcode);
                se type = LOCAL então
                  local := local - 1
                senão
                  export := export - 1;
                ler (arqSim, word);
                numCode = auxCode;
              fim
              senão
              início
                haveWork := false;
                se getList (SGL, auxCode) = verdadeiro então
                  haveWork := true
                senão
                  se getList (PGL, auxCode granu) = verdadeiro então
                  início
                    haveWork := true;
                    granuPGL := granuPGL – granu;
                  fim
                se haveWork = true então
                início
                  arqPos := posicao (arqSim);
                  putList (EGL, auxCode arqPos LOCAL 0 numCode);
                  tmpCode = -1;
                  enquanto tmpCode <> auxCode faça
                  início
                    enquanto word <> "CODE" faça
                      ler (arqSim, word);
                    ler (arqSim, tmpCode);
                  fim
                  numCode := auxCode;
                fim
              fim
"ENDCODE":    início
                ler (arqSim, numCode);
                se consultList(EGL, numCode, numCode arqPos type
                             tidImp auxCode) = verdadeiro então
                início
                  se type = LOCAL então
                  início
                    putList (LR, numCode);
                    posição (arqSim) := arqPos;
                    word := "WAIT";
                    numCode := auxCode;
                  fim
                  se type = IMPORT então
```

```
                     início
                       import := import - 1;
                       envia (TidImp, numCode , RESULT);
                       posição (arqSim) := arqPos;
                       word := "WAIT";
                       numCode := auxCode;
                     fim
                     se (local = 0 E export = 0 E import = 0 E
                         mainWork = verdadeiro) então
                     início
                       endComp := verdadeiro;
                       envia (TidDSLP, , ENDAPPS);
                     fim
                 fim
    outrocaso:   início
                   espera (timeResolution);
                   ler (arqSim, word);
                 fim
  fim
fim
fechar (arqSim);
```

FIGURA 7.3 - Algoritmo do Simulador no Protótipo DSLP

O algoritmo apresentado utiliza as mesmas instruções já descritas anteriormente na seção 6.3.2. Foram acrescentadas instruções para abrir, ler, alterar posição e fechar um arquivo. São utilizados pelo algoritmo também módulos implementados para trabalhar com listas encadeadas, tais como putList, getList e consultList. O primeiro argumento destes métodos é a lista a ser manipulada. O último argumento são os dados a serem inseridos/removidos. O retorno de um valor lógico verdadeiro é a indicação de sucesso na operação.

O simulador possui como estrutura de dados principais, além da lista de objetivos paralelos (PGL) e lista de objetivos seqüenciais (SGL), a lista de objetivos em execução – *Executing Goals List* (EGL) e a Lista de Resultados – *List of Results* (LR). A EGL armazena informações pertinentes às metas em execução, tais como: trecho de código em execução, posição atual do ponteiro de arquivo, tipo de execução, endereço do nodo que originou o trabalho e código anteriormente em execução. O tipo de execução indica se a meta foi gerada localmente (*LOCAL*), remotamente (*IMPORT*) ou está sendo executada por outro nodo (*EXPORT*). A LR por sua vez, guarda os resultados de metas já processadas.

O algoritmo do simulador pode ser dividido basicamente em duas etapas distintas, quais sejam: tratamento de mensagens e simulação do código. O tratamento de mensagens é feito sempre que uma nova mensagem chega ao nodo. Caso contrário, a simulação do código é realizada.

A computação inicia por um simulador que recebe a mensagem *PRINCIPAL*. Neste momento o primeiro trecho de código é armazenado na EGL e a simulação inicia. O nodo principal é o responsável também pela informação do término da computação ao escalonador.

A mensagem *PROBE* é enviada pelo *spy* para que o nodo informe a sua carga. Quando isto ocorre, o simulador envia a soma das complexidades armazenadas na PGL

(granuPGL). O recebimento da mensagem *AND_WORK* significa a solicitação de exportação de trabalho por um outro nodo E do sistema. Caso o nodo não possua metas na PGL, uma mensagem *CANCEL* é enviada. Caso contrário, a meta é enviada (*CONFIRM*) e granuPGL é decrementado com a granulosidade do trabalho removido.

Quando o escalonador determina que um nodo deve importar trabalho a mensagem *IMPORT* é enviada ao nodo. Essa mensagem faz com que o nodo inicie o processo de importação de uma meta. Caso a importação ocorra, a simulação inicia imediatamente. Quando um nodo termina de executar um trabalho importado ele envia o resultado (*RESULT*) para o seu originador.

A simulação do código começa quando o simulador encontra a palavra *CODE*. A partir deste momento qualquer palavra lida do arquivo que não seja uma instrução ao simulador faz com que o simulador espere um tempo predeterminado (*timeResolution*). A instrução *PARALLEL* é a indicação de que um trecho de código pode ser executado em paralelo. A granulosidade deste trecho é avaliada e dependendo do caso a meta é colocada da PGL ou na SGL.

Ao encontrar uma instrução *RESULT* o simulador entra no estado *WAIT*. Neste estado é feito uma verificação inicial na LR. Se o resultado desejado já encontra-se disponível, a computação prossegue normalmente. Porém, caso o resultado ainda não tenha chegado, isto é, a meta está sendo computada por outro nodo, uma outra meta é designada para processamento imediato. Essa meta é obtida preferencialmente da SGL. Uma vez que a SGL esteja vazia, a meta é obtida da PGL.

No estado *WAIT* pode ocorrer de não existirem mais metas disponíveis localmente para execução. Neste caso, o nodo entra em estado *idle* e aguarda ou novos trabalhos a serem determinados pelo escalonador ou o resultado desejado.

Quando a instrução *ENDCODE* é encontrada o trabalho encerrado é retirado da EGL. Caso seja um trabalho com o tipo *LOCAL*, seu resultado é colocado na LR. Caso seja um trabalha com o tipo *IMPORT*, seu resultado é enviado ao nodo que o originou. Se o escalonador em questão for o principal, é feita a verificação do término da execução da aplicação. Caso o número de trabalhos locais acrescido dos remotos seja zero, a computação está encerrada e a mensagem *ENDAPPS* é enviada ao escalonador.

7.3 Resultados Obtidos

Nesta seção são apresentados os resultados obtidos a partir da submissão de diversos testes ao protótipo. Os testes foram realizados no Instituto de Informática da UFRGS em uma rede *ethernet* com estações de trabalho SUN conectadas. Sempre que possível foi buscado o uso de máquinas com poderes computacionais idênticos. Como não estavam disponíveis máquinas em número suficiente com características semelhantes, foi buscada uma justa distribuição de maneira a influenciar o menos possível no resultado final.

Todos os testes foram realizados dez vezes em horário de baixa utilização das máquinas (normalmente durante a noite). Após, uma média dos tempos obtidos em cada execução foi calculada. Nesta seção, os valores apresentados são as médias das dez execuções realizadas. Os valores apresentados estão em segundos. O tempo de execução de uma resolução configurado no escalonador foi de 1 segundo. Caso seja necessário

comparar os tempos de execução no DSLP com os de outro ambiente, basta multiplicar os tempos aqui apresentados pelo tempo de execução de uma resolução neste outro ambiente.

Foram realizadas três categorias principais de testes. Na primeira, foi mantido constante o número de escalonadores e foram variados os valores de entrada da aplicação, a quantidade de trabalhadores e o tempo de inatividade do espião. O objetivo é mostrar a eficiência do protótipo, bem como a influência dos parâmetros da ACT. Na segunda categoria de testes ocorreu uma execução de uma mesma aplicação no DSLP com e sem o uso das informações de granulosidade. O objetivo é apresentar a influência destas informações. Finalmente, a terceira categoria variou o número de escalonadores e trabalhadores para uma mesma aplicação. O objetivo principal é mostrar o impacto da configuração da arquitetura no tempo de execução do sistema.

Nas duas primeiras categorias de testes foram utilizados programas para cálculo da série de *fibonacci* de 5, 6, 7 e 8 elementos. Na terceira categoria de testes foi utilizado um programa que possui cinco metas paralelizáveis com granulosidade alta (20 resoluções). O código deste programa, bem como o código para o cálculo do fibo[4] são apresentados no Anexo 3.

7.3.1 Quantidade de Trabalhadores e Tempo de Inatividade do Espião

Nesta categoria foi utilizado o cálculo do fibo de 5, 7 e 8 elementos. A arquitetura do DSLP foi montada com um escalonador e o número de trabalhadores variou de 3 a 6 máquinas. O tempo da execução seqüencial também foi obtido. A tabela 7.1 apresenta os resultados obtidos com o tempo de inatividade do espião de 6 segundos.

TABELA 7.1 - Tempos de Execução com Espião Inativo por 6s

	Seqüencial	3 máquinas	4 máquinas	5 máquinas	6 máquinas
FIBO 5	51,9	51,2	52	52	52,1
FIBO 7	134,7	132,4	91,5	92,7	92,5
FIBO 8	209,3	123,1	130,3	116,4	115,4

Conforme pode ser observado para o fibo de 5 elementos o melhor tempo obtido foi com três máquinas (*speedup* de 1,01). Já o fibo de 7 elementos obteve um desempenho melhor com quatro máquinas (*speedup* de 1,47), uma vez que o número de metas paralelizáveis é maior. O fibo de 8 obteve o menor tempo de execução com 6 máquinas (*speedup* de 1,81). Na tabela 7.2 são apresentados os resultados obtidos com o tempo de inatividade do espião de 3 segundos.

[4] O anexo apresenta apenas o código para o fibo de 5. Os outros programas para cálculo da série de *fibonacci* de 6,7 e 8 elementos foram suprimidos do anexo por sua semelhança com o de 5 elementos;

TABELA 7.2 - Tempos de Execução com Espião Inativo por 3s

	Seqüencial	3 máquinas	4 máquinas	5 máquinas	6 máquinas
FIBO 5	51,9	51,4	51,4	52,3	52,3
FIBO 7	134,7	91,1	88,9	89,5	90,3
FIBO 8	209,3	127,3	114	117	117,1

Nestes testes foi observada uma melhora nos tempos de execução para o fibo de 7 (*speedup* de 1,51) e de 8 elementos (*speedup* de 1,84). O fibo de 5 sofreu uma degradação no tempo de execução, por apresentar uma baixa granulosidade, fazendo com que as interferências adicionais do espião sobrecarregassem o sistema. A tabela 7.3 apresenta os resultados dos testes com tempo de inatividade do espião de 1s.

TABELA 7.3 - Tempos de Execução com Espião Inativo por 1s

	Seqüencial	3 máquinas	4 máquinas	5 máquinas	6 máquinas
FIBO 5	51,9	51,5	51,6	53	53,1
FIBO 7	134,7	132,2	88,6	89,4	90,7
FIBO 8	209,3	127,4	114,6	116,6	116,6

Na tabela é observada uma pequena melhora no tempo de execução do fibo de 7 (*speedup* de 1,52) e uma ligeira degradação no tempo de execução do fibo de 8 (*speedup* de 1,83). É importante observar que a variação de 3s para 1s não alterou significativamente os resultados dos testes. A figura 7.4 apresenta os tempos obtidos para o fibo de 5, 6 e 7 elementos com o melhor tempo de inatividade do espião em cada caso (6s, 1s e 3s respectivamente).

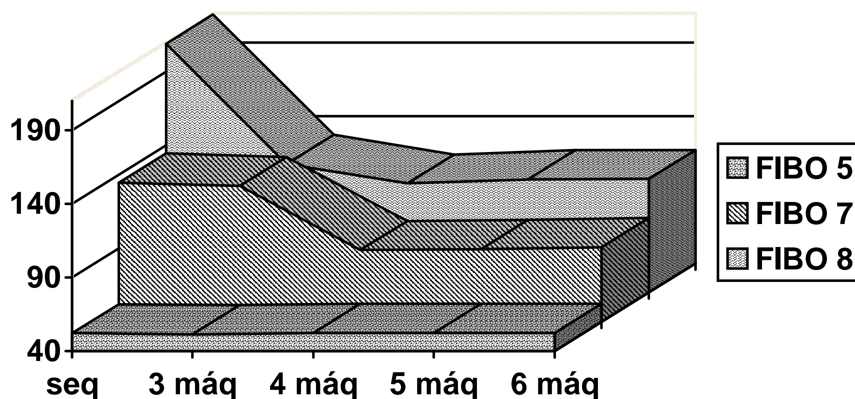

FIGURA 7.4 - Comparação no Tempo de Execução das Séries de *Fibonacci*

O menor tempo de execução do fibo de 5 elementos foi de 51,2 segundos, com três máquinas e inatividade do espião de 6 segundos. Para o cálculo do fibo de 7

elementos o menor tempo foi 88,6 segundos, com quatro máquinas e tempo de inatividade do espião de 1 segundo. Finalmente, o fibo de 8 elementos teve um desempenho melhor (114 segundos) com 4 máquinas e tempo de inatividade do espião 3s.

Devido as baixas granulosidades das metas paralelizáveis existentes no cálculo das séries de *fibonacci* de 5, 7 e 8 elementos, os tempos de execução com mais de 3 máquinas mantiveram-se praticamente constantes. O parâmetro tempo de inatividade do espião utilizado apresentou um comportamento inversamente proporcional a granulosidade das metas paralelizáveis, isto é, quanto mais trabalho E possuir o programa menor deve ser o tempo de inatividade do espião.

7.3.2 Uso das Informações de Granulosidade

Os testes da influência das informações de granulosidade foram feitos em uma arquitetura do DSLP com um escalonador e dois trabalhadores. O programa submetido foi o do cálculo da série de *fibonacci* com 5, 6, 7 e 8 elementos. Os resultados obtidos são apresentados na tabela 7.4 e na figura 7.5.

TABELA 7.4 -Tempos de Execução sem e com o Uso das Informações de Granulosidade

	sem informações de Granulosidade	com informações de granulosidade
FIBO 5	51,11	50,87
FIBO 6	66,75	63,95
FIBO 7	118,35	113,53
FIBO 8	174,38	167,21

Conforme pode ser observado os tempos de execução dos programas melhoraram com o uso das informações de granulosidade. No caso da série de 6 elementos a melhora foi de 4,22% e no caso do fibo de 7 e 8 elementos foi de 4,1%. O fibo de 5, por sua baixa granulosidade teve uma melhora de apenas 0,47%.

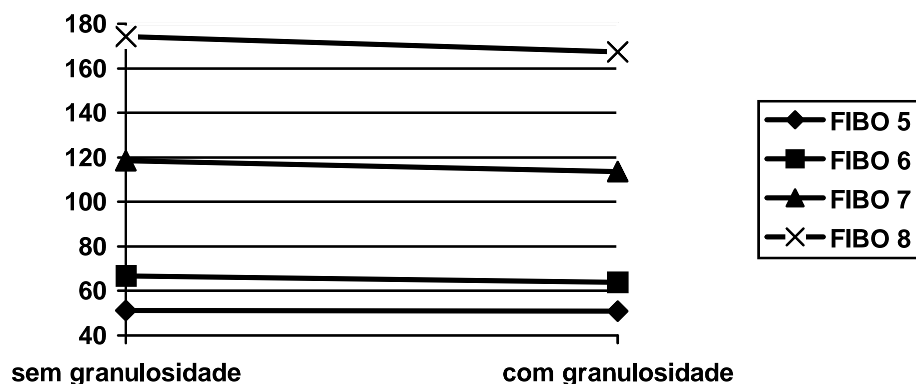

FIGURA 7.5 - Influência das Informações de Granulosidade

7.3.3 Quantidade de Escalonadores e Trabalhadores

Nesta última categoria de testes foi utilizado um programa com metas paralelizáveis de alta granulosidade (20 resoluções). A análise de granulosidade foi ativada, porém as configurações da arquitetura do DSLP variaram. Foram configuradas seis situações diferentes, quais sejam:

- **situação 1:** um escalonador e um trabalhador vinculado a ele (duas máquinas);
- **situação 2:** dois escalonadores e um trabalhador vinculados a cada um deles (quatro máquinas);
- **situação 3:** um escalonador e dois trabalhadores vinculados a ele (três máquinas);
- **situação 4:** três escalonadores e um trabalhador vinculado a cada um deles (seis máquinas);
- **situação 5:** um escalonador com três trabalhadores vinculados (quatro máquinas);
- **situação 6:** dois escalonadores com dois trabalhadores vinculado a cada um (seis máquinas).

A tabela 7.5 sintetiza os resultados obtidos.

TABELA 7.5 - Tempos de Execução com Diversas Configurações Diferentes

Configuração	Tempo de execução
Situação 1	133,1
Situação 2	112,92
Situação 3	112,64
Situação 4	93,01
Situação 5	92,31
Situação 6	72,13

Os resultados obtidos mostram que o tempo de execução foi diminuindo progressivamente. A medida que o número de trabalhadores aumenta, aumenta também o desempenho do sistema. Porém, quando é alterado o número de escalonadores e o número de trabalhadores é mantido constante, o tempo total de execução sofre uma pequena diminuição (como por exemplo, da situação 2 para 3). A situação que apresentou o melhor desempenho foi com dois escalonadores e dois trabalhadores. A figura 7.6 apresenta o resultado graficamente.

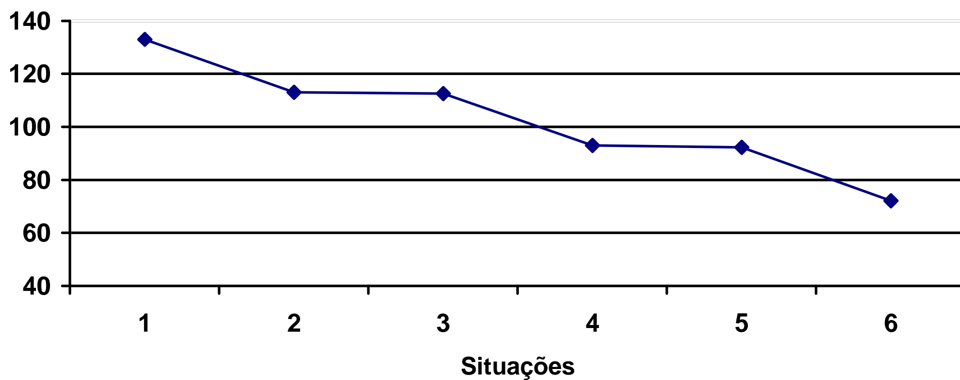

FIGURA 7.6 - Diversas Configurações de Trabalhadores e Escalonadores

Conforme os resultados obtidos nestes testes observa-se que o uso de um configurador automático é uma importante adição ao sistema, uma vez que libera o usuário de definir a melhor arquitetura manualmente para cada aplicação.

7.4 Conclusões

O presente capítulo descreveu o protótipo implementado do *Distributed Scheduler for Logic Programming*. O protótipo implementa apenas a fase E do escalonamento e simula a execução de programas Prolog. Para tal, um modelo de programas sintéticos e um simulador foram implementados.

Inicialmente o protótipo foi descrito, com o detalhamento de seus componentes e organização básica. Em seguida, o simulador implementado foi apresentado. O algoritmo do simulador foi descrito, bem como exemplos de programas sintéticos para execução neste.

Por fim, foram apresentados resultados obtidos com testes submetidos ao protótipo. Os resultados mostraram a eficiência do protótipo, bem como a influência das informações de granulosidade, do parâmetro inatividade do espião e das configurações de arquiteturas do ambiente DSLP.

8 Conclusões

O processamento paralelo tem sido vastamente empregado devido à busca constante de um desempenho melhor. Para dar suporte ao paralelismo, existem dois tipos de linguagens, a saber: as que exploram o paralelismo de forma explícita, e as que o fazem de maneira implícita.

A programação em lógica, pela sua característica de separar a lógica do controle, permite a exploração do paralelismo implícito. Deste modo é possível paralelizar cláusulas de um mesmo procedimento, Paralelismo OU, ou metas de uma cláusula, Paralelismo E. Com a exploração conjunta das duas fontes de paralelismo, é possível obter um desempenho melhor com uma vasta gama de aplicações ([KER 94]).

Vários ambientes para exploração do paralelismo na programação em lógica estão atualmente disponíveis. Dentre estes é possível destacar o Andorra I, ACE, ROPM e IDIOM. Como principais características destes ambientes são destacadas a exploração do paralelismo E e OU, o uso em arquiteturas com memória compartilhada (multiprocessadores), o escalonamento em duas fases e a análise em tempo de compilação.

O projeto OPERA ([GEY 92]) é um ambiente para exploração do paralelismo na programação em lógica que está em desenvolvimento na UFRGS. Como premissa básica, o OPERA executa em arquiteturas com memória distribuída (multicomputadores). O presente projeto é enquadrado no OPERA.

Até então, o OPERA realiza separadamente a exploração do paralelismo E e OU. O PloSys é o ambiente responsável pela exploração do paralelismo OU. O Opera E, encontra-se atualmente em desenvolvimento, e executa programas em lógica utilizando o paralelismo E. Outro componente do projeto, é o GRANLOG ([BAR 96a]). O GRANLOG gera informações de granulosidade para programas em lógica. As informações geradas são para ambas as fontes de paralelismo.

O escalonamento é uma das tarefas mais importantes em ambientes que exploram o processamento paralelo. Através dele é determinado em quais máquinas e em que ordem os trabalhos serão computados. Várias propostas de escalonamento são encontradas na literatura.

No escalonamento distribuído existe mais de um processador responsável pela tarefa de escalonar. Dentre as várias propostas estudadas é destacado o escalonamento hierárquico. Nesta proposta é obtido um desempenho satisfatório, sem problemas de escalabilidade, excesso de troca de mensagens e tolerância a falhas.

O presente trabalho propõe um escalonador hierárquico para a programação em lógica (DSLP – *Distributed Scheduler for Logic Programming*) que integra o paralelismo E e OU. O escalonador utiliza informações de granulosidade e informações do sistema para determinar os nodos exportadores e importadores. O DSLP é totalmente compatível com o OPERA e GRANLOG.

O escalonador é composto por nodos E e nodos OU. Os nodos OU são responsáveis pela exploração do paralelismo OU, bem como do escalonamento de

tarefas no sistema. Os nodos E executam em paralelo metas independentes e são vinculados aos nodos OU. Entre os nodos OU é formado um anel lógico bi-direcional.

Grande parte dos escalonadores para programas em lógica utilizam como heurística para decidir o nodo exportador o número de metas ou cláusulas disponíveis neste. O DSLP utiliza como heurística a complexidade total das metas ou cláusulas, isto é, o número de resoluções máximas necessários para executar as tarefas. Claramente, a complexidade é uma informação mais precisa.

O modelo DSLP permite várias extensões que poderão dar origem a trabalhos futuros. É possível destacar: criação de um reconfigurador dinâmico, determinação automática da quantidade de nodos E e OU e emprego de políticas de tolerância a falhas. Estas extensões foram brevemente descritas no trabalho.

Um protótipo do modelo DSLP foi implementado para a validação do trabalho. Sua implementação foi feita na UFRGS, utilizando a linguagem C e o ambiente PVM. O protótipo é limitado à exploração apenas do paralelismo E. Ele é um simulador de programas Prolog. Programas sintéticos contendo a árvore de execução do programa e mais algumas instruções adicionais são submetidos ao simulador. Vários testes foram feitos no protótipo, e resultados satisfatórios foram obtidos.

O presente trabalho é o ponto de partida para a utilização de propostas de escalonamento distribuídas no projeto OPERA. É também uma das primeiras propostas a fazer uso de informações de granulosidade como auxílio ao escalonamento. Além disso, vários outros trabalhos podem ser originados deste. É possível destacar dentre outras: modelagem e implementação das extensões propostas (seção 6.4), integração com o Opera E, implementação da fase OU de escalonamento e integração com o PloSys ([MOR 96]).

É muito difícil implementar uma política de escalonamento que atenda satisfatoriamente a todas as naturezas de programas existentes. Aliado a isto, está o fato de já existirem disponíveis diversas propostas diferentes de escalonadores. O objetivo deste trabalho não é a criação de uma proposta ideal. Nem tampouco, de mais uma proposta de escalonamento. O objetivo deste trabalho é sim o estudo do escalonamento na programação em lógica, na integração do paralelismo E e OU e no uso das informações de granulosidade. Neste particular, seu objetivo foi alcançado.

Anexo 1 *System Resource Table*

```
##############################################################################
#
#                           System Resource Table
#
##############################################################################
#
# DSLP
#
# Projeto OPERA - Prolog Paralelo
#
# Version : 1.0
#
# Date    : 20/07/97
#
# Author  : Cristiano Costa
#
#
#
#
# Copyright (c) 1997 - Curso de Pos-Graduacao em Ciencia da Computacao
#
#                       Universidade Federal do Rio Grande do Sul
#
#                       Brasil
#
##############################################################################
#
# Sparc Station 2  - 28 Mips
# Sparc Station 1+ - 16 Mips
# Sparc Station 1  - 12 Mips
# Sparc Station IPC - 16 Mips
# Sparc Station SLC - 12 Mips

minuano - 500
bolicho - 221
cuia - 100
pala - 100
espora - 110
mate - 120
pampa - 111
poncho - 110
guaiaca - 110
coxilha - 111
guria - 130
pingo - 150
pilcha - 50
prenda - 200
piratini - 500
```

Anexo 2 *Application Characteristic Table*

```
##############################################################################
#
#                    Application Characteristic Table
#
##############################################################################
#
# DSLP
#
# Projeto OPERA - Prolog Paralelo
#
# Version : 1.0
#
# Date    : 20/07/97
#
# Author  : Cristiano Costa
#
#
#
#
# Copyright (c) 1997 - Curso de Pos-Graduacao em Ciencia da Computacao
#
#                      Universidade Federal do Rio Grande do Sul
#
#                      Brasil
#
##############################################################################
#
# Machines used in the execution:
m pala
m espora
m pingo
m espora
m mate
m guaiaca
m poncho
m pampa
m cuia

# Machines used as schedulers
p cuia
p pala
p bolicho

# Spy sleep time (microseconds)
s 500

# Overloaded level (resolutions)
o 10

# Range of load flutuation (resolutions)
r 10

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

Anexo 3 Programas Utilizados nos Testes

Programa para cálculo da série de Fibonnaci (para 5 elementos)

```
CODE 0
  FIBO5N
  M>1
  M1IS4
  M2IS3
  PARALLEL 1
    FIBO4N1
    ETL 9.0681
  FIBO3N2
  M>1
  M1IS2
  M2IS1
  PARALLEL 2
    FIBO2N1
    ETL 3.0168
  FIBO1N2
  FIBO11
  RESULT 2
  NIS2
  RESULT 1
  NIS5
ENDCODE 0
CODE 1
  M>1
  M1IS3
  M1IS2
  PARALLEL 3
    FIBO3N1
    ETL 5.0336
  FIBO2N2
  M>1
  M1IS2
  M2IS1
  PARALLEL 4
    FIBO2N1
    ETL 3.0168
  FIBO1N2
  FIBO11
  RESULT 4
  NIS1
  RESULT 3
  NIS3
ENDCODE 1
CODE 2
  M>1
  M1IS1
  M2IS0
  PARALLEL 5
    FIBO1N1
    ETL 1.0080
  FIBO0N2
  FIBO00
  RESULT 5
  NIS1
ENDCODE 2
CODE 3
  M>1
  M1IS2
  M1IS1
  PARALLEL 6
    FIBO2N1
    ETL 3.0168
  FIBO1N2
  FIBO11
  RESULT 6
  NIS2
ENDCODE 3
CODE 4
  M>1
  M1IS1
  M2IS0
  PARALLEL 7
    FIBO1N1
    ETL 1.0080
  FIBO0N2
  FIBO00
  RESULT 7
  NIS1
ENDCODE 4
CODE 5
  FIBO11
ENDCODE 5
CODE 6
  M>1
  M1IS1
  M2IS0
  PARALLEL 8
    FIBO1N1
    ETL 1.0080
  FIBO0N2
  FIBO00
  RESULT 8
  NIS1
ENDCODE 6
CODE 7
  FIBO11
ENDCODE 7
CODE 8
  FIBO11
ENDCODE 8
```

Programa de Alta Granulosidade

```
CODE 0
  R1
  R2
  R3
  R4
  R5
  R6
  R7
  R8
  R9
  R10
  PARALLEL 1
    R11
    ETL 20
  R12
  R13
  R14
  PARALLEL 2
    R15
    ETL 20
  R16
  R17
  R18
  R19
  PARALLEL 3
    R20
    ETL 20
  R21
  R22
  R23
  R24
  PARALLEL 4
    R25
    ETL 20
  R26
  R27
  R28
  R29
  PARALLEL 5
    R30
    ETL 20
  R31
  R32
  R33
  R34
  R35
  RESULT 5
  RESULT 4
  RESULT 3
  RESULT 2
  RESULT 1
ENDCODE 0

CODE 1
P1
P2
P3
P4
P5
P6
P7
P8
P9
P10
P11
P12
P13
P14
P15
P16
P17
P18
P19
P20
ENDCODE 1

CODE 2
P1
P2
P3
P4
P5
P6
P7
P8
P9
P10
P11
P12
P13
P14
P15
P16
P17
P18
P19
P20
ENDCODE 2

CODE 3
P1
P2
P3
P4
P5
P6
P7
P8
P9
P10
P11
P12
P13
P14
P15
P16
P17
P18
P19
P20
ENDCODE 3

CODE 4
P1
P2
P3
P4
P5
P6
P7
P8
P9
P10
P11
P12
P13
P14
P15
P16
P17
P18
P19
P20
ENDCODE 4

CODE 5
P1
P2
P3
P4
P5
P6
P7
P8
P9
P10
P11
P12
P13
P14
P15
P16
P17
P18
P19
P20
ENDCODE 5
```

Bibliografia

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

[ARA 94] ARAUJO, Lourdes; RUZ, Jose. PDP: Prolog distributed processor for independent AND\OR parallel execution of Prolog. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 1994. **Proceedings...** Cambridge: MIT Press, 1994. p.142-156.

[BAR 95a] BARBOSA, Jorge V.; GEYER, C. GRANLOG: um modelo para análise automática de granulosidade na programação em lógica. In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES – PROCESSAMENTO DE ALTO DESEMPENHO, 10., 1995, Canela. **Anais...** Porto Alegre: SBC, 1995.

[BAR 95b] BARBOSA, Jorge V.; GEYER, C. Integração OPERA-GRANLOG: Aplicação da Análise de Granulosidade na Execução Paralela de Programas em Lógica. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 15., 1995, Canela. **Anais...** Porto Alegre: SBC, 1995. p. 61-75.

[BAR 96a] BARBOSA, Jorge V. **GRANLOG: um modelo para análise automática de granulosidade na programação em lógica**. Porto Alegre: CPGCC-UFRGS, 1996. Dissertação de Mestrado.

[BAR 96b] BARBOSA, Jorge L. V.; GEYER, Cláudio F. R. Análise de Grãos na Programação em Lógica. In: SEMINÁRIO INTEGRADO DE SOFTWARE E HARDWARE, 1996, Recife. **Anais...** Recife: SBC, 1996. p. 345-356.

[BAR 96c] BARBOSA, Jorge L. V.; GEYER, Cláudio F. R. Análise Global na Programação em Lógica. In: SIMPÓSIO BRASILEIRO DE LINGUAGENS DE PROGRAMAÇÃO, 1996, Belo Horizonte. **Anais...** Belo Horizonte: SBC, 1996. p. 89-102.

[BRI 96] BRIAT, Jacques; GINZBURG, Ilan; PASIN, Marcelo. **Athapascan0b – User Manual.** Grenoble: Université Joseph Fourier, 1996.

[BUE 93] BUENO, F. et al. **The AND-Prolog compiler system - automatic parallelization tools for LP.** Madrid: Universidad Politécnica de Madrid, June 1993. 40p. (Technical Report DIA/CLIP5/93.0).

[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, Feb. 1988.

[CER 97] CERVO, L. et al. Implementação de um Escalonador Distribuído para o Projeto OPERA. In: SALÃO DE INICIAÇÃO CIENTÍFICA, 9., 1997, Porto Alegre. **Anais...** Porto Alegre: UFRGS, 1997. p. 33.

[CIE 91] CIEPIELEWSKI, Andrzej. Scheduling in Or-Parallel Prolog Systems: Survey and Open Problems. **International Journal of Parallel Programming**, [S.l.], v.20, n. 6, p. 421-451, 1991.

[CLO 81] CLOCKSIN, W.F.; MELLISH, C.S. **Programming in Prolog.** Berlin: Springer-Verlag, 1981. 189p.

[CLO 88] CLOCKSIN, W. F.; ALSHAWI, H. A method for efficiently executing horn clause programs using multiple processors. **New Generation Computing**, Berlin, v.3, n.5, p.361-376, 1988.

[COS 91a] COSTA, V. S.; WARREN D. H. D.; YANG, R. The Andorra-I Engine: a parallel implementation of the basic andorra model. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 8., Aug. 1991. **Proceedings...** Cambridge: MIT Press, 1991. p.825-839.

[COS 91b] COSTA, V. S.; WARREN D. H. D.; YANG, R. The Andorra-I Preprocessor: supporting full prolog on the basic andorra model. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 8., Aug. 1991. **Proceedings...** Cambridge: MIT Press, 1991. p.443-456.

[COS 96a] COSTA, Cristiano A. da. **Um estudo das propostas que integram o paralelismo E/OU na Programação em Lógica**. Porto Alegre: CPGCC-UFRGS, 1996. 64p. (TI-493).

[COS 96b] COSTA, Cristiano A. da. Uma proposta de escalonamento distribuído para a esploração do paralelismo na programação em Lógica. In: SEMANA ACADÊMICA DA COMPUTAÇÃO, Porto Alegre. **Anais...** Porto Alegre: CPGCC-UFRGS, p.287-290. 1996.

[DAN 96a] DANDAMUDI S. et al. **Hierarchical load sharing policies for distributed systems**. Ottawa: Carleton University, 1996. 17 p. (Technical Report SCS – 96 – 1).

[DAN 96b] DANDAMUDI S. **The effect of scheduling discipline on dynamic load sharing in heterogeneous distributed systems**. Ottawa: Carleton University, 1996. 17 p. (Technical Report TR– 96 – 26).

[DAN 96c] DANDAMUDI S. et al. **Performance of hierarchical processor scheduling in shared-memory multiprocessor systems**. Ottawa: Carleton University, 1996. 17 p. (Technical Report TR– 96 – 21).

[DAN 97a] DANDAMUDI S. et al. **Performance comparison of processor scheduling strategies in a distributed-memory multicomputer system**. Ottawa: Carleton University, 1997. 15 p. (Technical Report TR – 97 – 1).

[DAN 97b] DANDAMUDI S. **Sensitivity evaluation of dynamic load sharing in distributed systems**. Ottawa: Carleton University, 1997. 19 p. (Technical Report TR – 97 – 12).

[DEB 93] DEBRAY, S. K.; LIN, N. Cost Analysis of Logic Programs. **ACM Transactions on Programming Languages and Systems**, New York, v.15, n.5, p.826-875, Nov. 1993.

[DER 96] DERANSART, P. et al. **PROLOG:** the standart**.** Berlin: Springer-Verlag, 1996. 272 p.

[DIA 94] DIAZ, D. **Wamcc Prolog User's Manual**. Domaine de Voluceau: INRIA-Rocquencourt, 1994.

[DIK 89] DIKSHIT, P. et al. SAHAYOG: a test bed for evaluating dynamic load-sharing policies. **Software - Practice and Experience**, New York, v. 19, n. 5, p. 411-435, May 1989.

[DUT 95] DUTRA, Inês de C. Performance analysis of a strategy to distribute and-work and or-work in parallel logic programming systems. In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES, 7., 1995. **Anais...** Canela: SBC, 1995. p.449-463.

[DUT 97] DUTRA, Inês C. **Comunicação pessoal numa visita ao II/UFRGS**, maio 1997.

[EAG 86] EAGER, D. et al. Adaptive load sharing in homogeneous distributed systems. **IEEE Transactions on Software Engineering**, New York, v. SE-12, n. 5, p. 662-675, May 1986.

[ELR 94] EL-REWINI et al. **Task scheduling in parallel and distributed systems**. Englewood Cliffs: Prentice-Hall, 1994. 290p.

[EVA 92] EVANS, D. J. et al. Dynamic load balancing using task-transfer probabilities. **Parallel Computing**, New York, v.19, p. 897-916, 1992.

[EVA 94] EVANS, D. J. et al. Load balancing with network partitioning using host groups. **Parallel Computing**, New York, v. 20, p. 325-345, 1994.

[FEI 95] FEITELSON, Dror G. and RUDOLPH, Larry. Coscheduling based on runtime identification of activity working sets. **International Journal of Parallel Programming**, [S.l.], v.23, n.2, 1995.

[GEI 94] GEIST, Al et al. **PVM 3 user's guide and reference manual version 3.3**. Disponível por FTP anônimo em netlib2.cs.utk.edu, 1994.

[GEY 91] GEYER, Cláudio F. R. **Une Contribution a L'Etude du Parallelisme OU en Prolog sur des Machines sans Mémoire Commune**. Grenoble: Université Joseph Fourier, 1991. 166 p. PHD Thesis.

[GEY 92] GEYER, Cláudio F. R. et al. Projeto Opera: um modelo E/OU para Prolog. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 1992, Rio de Janeiro. **Anais...** Rio de Janeiro: SBC, 1992. 390p. p. 269-281.

[GUP 91] GUPTA, G.; COSTA, V. S.; YANG, R. IDIOM. Integrating Dependent and-, Independent and-, and Or-parallelism. In: INTERNATIONAL SYMPOSIUM ON LOGIC PROGRAMMING, 4., Oct. 1991. **Proceedings...** Cambridge: MIT Press, 1991. p.152-166.

[GUP 93] GUPTA, C.; HERMENEGILDO, M. PONTELLI, E.; COSTA, V. S. **ACE: and/or-parallel copying-based execution of logic programs**. Las Cruces: New Mexico State University, 1993. 16p. (Technical Report).

[GUP 95] GUPTA, G. et al. **Parallel Execution of Prolog Programs: a survey**. Disponível por WWW em http://www.cs.nmsu.edu/lldap/pub_para/survey.html, 1995. (set. 95).

[HEI 96] HEISS, Hans. **Comunicação pessoal numa visita ao II/UFRGS**, agosto 1996.

[HER 91] HERMENEGILDO, M.V.; GREENE, K. J. The &-Prolog System: exploiting independent and-parallelism. **New Generation Computing**, Berlin, v.9, n.3,4, p.233-256, 1991.

[HOG 84] HOGGER, C. J. **Introduction to Logic Programming.** London: Academic Press, 1984.

[KAL 87] KALÉ, L. V. Completeness and full parallelism of parallel logic programming schemes. In: IEEE SYMPOSIUM ON LOGIC PROGRAMMING, 4., San Francisco, 1987. **Proceedings...** San Francisco: IEEE, 1987, p.125-133.

[KAY 95] KAYSER, Patrícia; GEYER, Cláudio. **Implementação de um analisador de granulosidade para Prolog.** Porto Alegre: CPGCC-UFRGS, 1995. 71p. Projeto de Diplomação.

[KER 94] KERGOMMEAUX, J.C.; CODOGNET, Philippe. **Parallel Logic Programming Systems.** Grenoble: Universite Joseph Fourier-Grenoble I, 1994. 52p. Technical Report.

[KER 96] KERGOMMEAUX, J.C. **Comunicação pessoal numa visita ao II/UFRGS**, dezembro 1996.

[KUC 90] KUCHEN, Herbert and WAGENER, Andreas. **Comparison of dynamic load balancing strategies**. Disponível por WWW em http://fiachra.ucd.ie/~david/loadbalancingpapers.html, 1990 (abr. 1995).

[KUN 91] KUNZ, T. The influence of different workload descriptions on a heuristic load balancing scheme. **IEEE Transactions on Software Engineering**, New York, v. 17, n. 7, p. 725-734, July 1991.

[LIN 91] LIN, Zheng. A distributed fair polling scheme applied to or-parallel logic programming. **International Journal of Parallel Programming**, [S.l.], v.20, n. 4, p. 315-339, 1991.

[LOD 96] LO, Michael; DANDAMUDI S. **Performance of hierarchical load sharing in heterogeneous distributed systems**. Ottawa: Carleton University, 1996. 8 p. (Technical Report TR – 96 – 22).

[MOR 96] MOREL, Éric et al. Side-effects in PloSys or-parallel Prolog on distributed memory machines. In: COMPULOG NET MEETING ON PARALLELISM AND IMPLEMENTATION TECHNOLOGY, 1996. **Proceedings...** [S.l.:s.n.], 1996.

[MUL 93] Mullender, S. **Distributed Systems**. New York: ACM Press, 1993. 601p.

[NIL 85] NI, Lionel M. et al. A distributed drafting algorithm for load balancing. **IEEE Transactions on Software Engineering**, New York, v. SE-11, n. 10, p. 1153-1161, Oct. 1985.

[OKE 94] O’KEEFE, Richard A. **The Craft of Prolog.** 2. ed. Cambridge: MIT Press, 1994. 387 p.

[PON 94] PONTELLI, E.; GUPTA, C.; HERMENEGILDO, M. &ACE: a high-performance parallel prolog system. In: INTERNATONAL SYMPOSIUM ON PARALLEL SYMBOLIC COMPUTATION, 1, 1994. **Proceedings...** [S.l.:s.n.], 1994.

[PON 95] PONTELLI, E.; GUPTA, C.; HERMENEGILDO, M. **Implementation and Performance of &ACE:** an independent and-parallel system. Las Cruces: New Mexico State University, 1995. Technical Report.

[RAM 89] RAMKUMAR, B.; KALÉ, L. V. **Compiled execution of the REDUCE-OR process model on multiprocessor.** Urbana-Champaing: University of Illinois, May 1989. 22p. (Technical Report UIUCDCS-R-89-1513).

[RAM 90] RAMKUMAR, B.; KALÉ, L. V. A Chare Kernel Implementation of a Parallel Prolog Compiler. INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, 1990. **Proceedings...** [S.l.:s.n.], 1990.

[RAM 92] RAMKUMAR, B.; KALÉ, L. V. A Join algorithm for combining and parallel solutions in AND/OR parallel systems. **International Journal of Parallel Programming**, [S.l.], v.21, n.1, 1992.

[SER 94] SERGENT, Thierry Le; BERTHOMIEU, Bernard. **Balancing load under large and fast load changes in distributed computing systems - a case study**. Disponível por WWW em http://fiachra.ucd.ie/~david/loadbalancingpapers.html, 1994 (abr. 1995).

[SHA 89] SHAPIRO, E. The Family of Concurrent Logic Programming Languages. **ACM Computing Surveys**, New York, v.21, n.3, p.413-510, Sept. 1989.

[SHE 97] SHEN, Kish. **A New Implementation Scheme for Combining And/Or Parallelism**. Disponível por WWW em http://www.sics.se/isl/sicstus.html, 1997. (mar. 1997)

[SHI 92] SHIVARATI, N. G.; KRUEGER P. Load Distributing for Locally Distributed Systems. **IEEE Computer**, New York, p.33-44, Dec. 1992.

[SON 94] SONG, Jianjian. A partially asynchronous and iterative algorithm for distributed load balancing. **Parallel Computing**, New York, v. 20, 1994. p. 853-868.

[STA 85] STANKOVIC, John A. An application of bayesian decision theory to decentralized control of job scheduling. **IEEE Transactions on Computers**, New York, v. C-34, n. 2, p. 117-130, Feb. 1985.

[STE 94] STERLING, L; SHAPIRO, E. **The Art of Prolog.** 2. ed. Cambridge: MIT Press, 1994. 509 p.

[SUE 92] SUEN, Tony T. Y.; WONG, J. K. Efficient task migration algorithm for distributed systems. **IEEE Transactions on Parallel and Distributed Systems**, New York, v. 3, n. 4, p.488-499, July 1992.

[SWE 95] SWEDISH INSTITUTE OF COMPUTER SCIENCE. **SICStus Prolog User's Manual**. Disponível por WWW em http://www.sics.se/isl/sicstus.html, 1995. (mar. 1996).

[TOW 94] TOWSLEY, Donald et al. A Comparison of sender-initiated and receiver-initiated reliable multicast protocols. **SIGMETRICS Performance Evaluation Review**, Nashville, v. 22, n. 1, p. 221-230, May 1994.

[TRE 97] TREIN, Charles L. **Implementação de um escalonador distribuído para a Programação em Lógica**. Porto Alegre: CPGCC – UFRGS, 1997. 63 p. Trabalho de Conclusão de Curso.

[WAR 83] WARREN, David D. H. **An abstract Prolog instruction set.** Manchester: SRI International, Oct. 1983. (Technical Note, 309).

[WAR 87] WARREN, David. H.D. The SRI model for or-parallel execution of Prolog - abstract design and implementation issues. In: SYMPOSIUM ON LOGIC PROGRAMMING, 1987, San Francisco. **Proceedings...** New York: IEEE Press, 1987. p.92-102.

[WER 94a] WERNER, Otília. **Uma máquina abstrata estendida para o paralelismo E na Programação em Lógica**. Porto Alegre: CPGCC-UFRGS, 1994. Dissertação de Mestrado.

[WER 94b] WERNER, O. et al. Opera Project: an approach towards parallelism exploitation on logic programming. In: LOGIC PROGRAMMING WORKSHOP, 10., 1994. **Proceedings...** Zurich: University of Zurich, 1994.

[YAM 92] YAMIN, Adenauer C. **Modelos de Implementação do Paralelismo OU na Programação em Lógica**. Porto Alegre: CPGCC-UFRGS, 1992. 114p. (TI-280).

[YAM 94] YAMIN, Adenauer C. **Um ambiente para a exploração de paralelismo na Programação em Lógica**. Porto Alegre: CPGCC-UFRGS, 1994. Dissertação de Mestrado.

[YAN 93] YANG, R. et al. Performance of the compiler-based Andorra-I system. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 10., 1993. **Proceedings...** Cambridge: MIT Press, 1993. p.150-166.

[ZAK 95] ZAKI, M. J. et al. **Customized dynamic load balancing for a network of workstations**. Rochester: The University of Rochester, 1995. 21p. (Technical Report 602).

[ZHO 88] ZHOU, Songnian. A trace-driven simulation study of dynamic load balancing. **IEEE Transactions on Software Engineering**, New York, v. 14, n. 9, p. 1327-1341, Sept. 1988.

[ZHO 93] ZHOU, Songnian et al. Utopia: a load sharing facility for large, heterogeneous distributed computer systems. **Software - Practice and Experience**, New York, v. 23, n. 12, p. 1305-1336, Dec. 1993.