100897-8

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMATICA
CURSO DE PÓS-GRADUAÇAO EM CIENCIA DA COMPUTAÇAO

MAQUINA DE CLAUSULAS : Arquitetura e modelo de execução de cláusulas Prolog

por

José Carlos Bins Filho

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

Prof. Philippe Navaux
Orientador

Prof. Antônio Carlos da Rocha Costa
Co-orientador

Porto Alegre, maio de 1990.

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

CIP = CATALOGAÇÃO NA PUBLICAÇÃO.

Bins Filho, Jose Carlos

Máquina de Cláusulas: Arquitetura e modelo de execução de cláusulas Prolog./José Carlos Bins Filho. - Porto Alegre: CPGCC UFRGS, 1990.

115p. il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Sul, Curso de Pós-graduação em Ciência da Computação, Porto Alegre, 1990. Orientador : Navaux, Philippe.

Dissertação: Arquiteturas paralelas; Máquinas Prolog; Modelos computacionais; Redes de Petri.

AGRADECIMENTOS

A minha mãe Leda e ao meu pai José Carlos pela educação e incentivo aos meus estudos.

Aos Profs. Philippe Navaux e Antônio Carlos da Rocha Costa pela orientação, ensinamentos, sugestões e paciência durante a elaboração desta dissertação.

Aos colegas e amigos pelo apoio, incentivo e companherismo.

A todos os demais que contribuiram na execução deste trabalho, de maneira especial aos funcionários do CPGCC-UFRGS.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Sistema de Biblioteca da UFRGS

30729

681.32.02(043)
B614M

INF
1995/100857-8
1995/07/10

MOD. 2.3.2

A memória de
Arildo Silva, tio
e amigo.

SUMARIO

LISTA DE FIGURAS 008
LISTA DE TABELAS 009
LISTA DOS ALGORITMOS 010
RESUMO 011
ABSTRACT 013

1 INTRODUÇAO 015

2 PROGRAMAÇAO EM LOGICA
2.1 Introdução 017
2.2 Comparação entre linguagens Convencionais
e linguagens para IA 017
2.3 Programação em Lógica 019
2.3.1 Cláusulas 020
2.3.2 Regra de resolução 021
2.4 Prolog 021
2.4.1 Unificação 022
2.4.2 Backtraking 023
2.5 AND / OR Paralelismo 024
2.6 Prolog concorrente 024
2.7 Conclusão 025

3 MAQUINAS PROLOG
3.1 Introdução 027
3.2 Classificação de Máquinas para IA 030
3.3 Máquinas Prolog 031
3.4 Exemplo de Máquinas Prolog 033
3.5 Conclusão 036

4 MODELO DE EXECUÇAO
4.1 Introdução 037
4.1.1 Redes de Petry 037
4.1.2 Máquinas de Cláusulas 038

4.2 Descrição do modelo de Execução 038
4.2.1 Redes Predicado Transição 038
4.2.2 Predicados 039
4.2.3 Arcos 041
4.2.4 Cláusulas 042
4.2.5 Propagação de valores 044
4.2.6 Pesquisa na tupla 048
4.2.7 Inconsistência 049
4.3 Conclusão 050

5 ARQUITETURA PROPOSTA
5.1 Introdução 051
5.2 Escolha da arquitetura básica 052
5.2.1 Memória local 052
5.2.2 Memória global dividida 054
5.2.3 Memória global partilhada 055
5.3 Descrição da Arquitetura 056
5.3.1 Arquitetura Básica 056
5.3.2 Arquitetura de um Bloco 059
5.3.3 Estrutura dos dados 060
5.3.3.1 Predicados 061
5.3.3.2 Tuplas 063
5.3.3.3 Argumento de tupla 066
5.3.4 Interface de memória 069
5.3.4.1 Arquitetura 070
5.3.4.2 Lay-out das Instrucõoes................. 070
5.3.4.3 Instruções 071
5.3.5 Processador 078
5.3.5.1 Arquitetura 079
5.3.5.2 Fila de tuplas à processar 080
5.3.5.3 Estrutura dos dados 081
5.3.6 Rede de interconexão 084
5.3.6.1 Combinação 085
5.3.6.2 Geometria da rede de interconexão 085

5.3.6.3 Descrição de um nodo da rede 088
5.3.6.4 Combinações possíveis 092
5.3.7 Predicados Pré-definidos 094
5.4 Conclusão 094
6 VALIDAÇÃO 095
7 **CONCLUSÃO** 100
ANEXO - ALGORITMO DE UNIFICAÇÃO 101
BIBLIOGRAFIA 108

LISTA DE FIGURAS

Figura 2.1 Processo de backtracking 023
Figura 3.1 Tipos de enfoques no desenvolvimento de um computador para IA 030
Figura 4.1 Exemplo de uma cláusula do modelo 039
Figura 4.2 Exemplo de arcos 042
Figura 4.3 Rede para cálculo do fatorial 044
Figura 4.4 Rede para dedução do predicado avô 046
Figura 5.1 Modelo de arquitetura com memória local 053
Figura 5.2 Modelo de arquitetura com memória global dividida 055
Figura 5.3 Modelo de arquitetura com memória global partilhada 056
Figura 5.4 Arquitetura geral proposta 057
Figura 5.5 Arquitetura de um bloco 059
Figura 5.6 Representação em arvore de uma lista . 069
Figura 5.7 Arquitetura do processador 079
Figura 5.8 Rede Omega 8 x 8 com nodos 2 x 2 086
Figura 5.9 Arquitetura de um nodo da rede de interconexão 090
Figura 5.10 Arquitetura da fila de combinação 091

LISTA DE TABELAS.

Tabela 2.1 Características de linguagens 017
Tabela 3.1 Máquinas Prolog 033
Tabela 4.1 Procedimentos para inclusão ou alteração 045
Tabela 4.2 Execução do predicado Fat 049
Tabela 6.1 Número de requisições 096
Tabela 6.2 Largura de banda 097
Tabela 6.3 Comparação entre 01 e N PEs 099

LISTA DE ALGORITMOS.

4.1 Propagação do valor lógico 045
4.2 Teste dos valores lógicos das pré-condições.. 047
4.3 Teste dos valores lógico das tuplas 047
4.4 Tratamento de inconsistência 050
5.1 Leitura e marcação de tuplas 072
5.2 Leitura de palavra 073
5.3 Alocação de tupla 074
5.4 Gravação de palavra 075
5.5 Remoção de palavra 076
5.6 Libera tupla 077
5.7 Unificação 101

Resumo.

Este trabalho define um modelo de execução para cláusulas Prolog, a partir do modelo abstrato de Máquinas de Cláusulas, e o Projeto de uma arquitetura paralela que suporte o modelo proposto. São também introduzidos alguns aspectos sobre as linguagens Lógicas e as máquinas Prolog visto que estes elementos estão relacionados intimamente tanto com o modelo quanto com a arquitetura propostos.

Na proposta do modelo de execução são definidos uma representação para os elementos do modelo abstrato (predicados, arcos e cláusulas) e um conjunto de algoritmos que permitem a operacionalização do modelo de forma a que tanto o paralelismo como a concorrência inerentes ao modelo abstrato sejam exploradas de forma integral.

Na proposta da arquitetura são, primeiramente, discutidas algumas opções de arquitetura básica e, posteriormente, descrita a arquitetura escolhida tanto a nível de blocos bem como dos seus componentes principais, a saber: interface de memória, processador e rede de interconexão. Para cada um destes componentes são descritas as principais intruções e são apresentados os algoritmos que as implementam.

Junto com a descrição da arquitetura é definida uma estrutura de dados que permite a implementação da representação descrita no modelo de execução e é definido também o algoritmo de unificacão que percorre a estrututa proposta.

Na validação é feito o cálculo da largura de banda máxima alcançada pela arquitetura proposta, cálculo este baseado no algoritmo de unificação descrito. E também feita uma avaliação do ganho de performance da arquitetura

proposta em relação a um processador, bem como é justificado o número de processadores escolhidos comparando a performance alcançada na arquitetura proposta com a performance alcancada por conjuntos maiores e menores de processadores.

Por fim na conclusão são feitos comentários sobre os objetivos atingidos e sobre possíveis extensões a este trabalho.

PALAVRAS-CHAVE: Arquiteturas Paralelas; Máquinas Prolog; Modelos Computacionais; Redes de Petri.

TITLE : "Clause Machines : Architecture and Prolog Clauses Execution Model".

Abstract

The present work defines a execution model for Prolog clauses based on the clause machines abstract model and then proposes a parallel architecture for the execution model. Some topics about Logic languages and Prolog machines were therefore introduced because they are closely related with, both, the model and the architecture proposed.

In the execution model the representation of the abstract model elements (predicates, arcs and clauses) and the set of algoritms that allow the operation of the model were defined so that the parallelism of the model can be integraly achieved.

In the architecture proposal, first some options for the basic architecture were discussed and then the chosen architecture is describeb at block level as much as at its components level. The most importants components reported are the memory interface, the processor and the interconection net, for each one of them the possible instructions were describeb as well as their algoritms.

Together with the especification of the architecture, the data estructure that allows the implementation of the execution model representation and the concerning unification algorit that scans the proposed representation were especified too.

In the validation the thoughtput permited by the proposal architecture is calculated based on the unification algoritm earlier described. Besides that the performance gain compared with an architecture with only one processor

was estimated, as much as the confrontation of the performance of lesser and greater sets of processors elements were made in order to validate the chossen number.

At last, in the conclusion, some coments about the fulfilled goals and about eventual extends for the work.

KEY-WORDS : Parallel Architectures; Prolog Machines; Compu-tational Models; Petri Nets.

1 INTRODUÇAO.

Nos últimos anos Prolog tem sido considerado como uma das mais promissoras linguagens para o desenvolvimento de aplicações em Inteligência Artificial , destacando-se entre estas os chamados sistemas especialistas.

Contudo a execução de programas em Prolog exige uma elevada utilização de recursos computacionais, o que tem limitado o seu uso, em computadores convencionais, basicamente a programas experimentais. Apesar disto, Prolog tem gerado muitos estudos com vistas a explorar o seu potencial.

O potencial de Prolog advém do alto grau de paralelismo inerente ao modelo no qual está baseado : a Programação em Lógica. Com o intuito de explorar esta característica muitos enfoques tem sido tentados. Dentre estes os principais são :

- Definição de modelos computacionais paralelos
- Definição de linguagens de programação paralelas mais poderosas
- Desenvolvimento de arquiteturas paralelas que executem os modelos propostos.
- Desenvolvimento de software que suporte o modelo e a arquitetura
- Desenvolvimento de novas tecnologias de "hardware"

Para alguns destes tópicos muito já foi feito. Quanto as linguagens paralelas podemos citar o exemplo de Parlog [CLA86] entre outros.

Já no caso da modelagem os progressos tem sido mais lentos. A maioria das máquinas que executam linguagens lógicas utilizam arquiteturas paralelas e unidades de

unificação em "hardware" sem no entanto utilizar modelos novos.

O trabalho atual pretende criar um modelo de execução para o modelo abstrato de máquinas de cláusulas posteriormente propondo uma arquitetura paralela que permita a implementação do modelo.

A organização do texto ficou assim dividida :

- Capítulo 1 - Introdução
- Capítulo 2 - Introduz o tema Programação em Lógica no qual está baseado o modelo e o Prolog.
- Capítulo 3 - Mostra as principais características das máquinas Prolog, que são máquinas projetadas para executar diretamente o Prolog.
- Capítulo 4 - Descreve o modelo de execução proposto para a máquina de cláusulas.
- Capítulo 5 - Descreve a arquitetura proposta para a máquina e a estrutura de dados que permite a execução do modelo.
- Capítulo 6 - Avalia os níveis de desempenho do sistema.
- Capítulo 7 - Conclusão.

2 PROGRAMAÇAO EM LOGICA.

2.1 Introdução.

As primeiras linguagens de programação eram projetadas pensando-se exclusivamente nas máquinas nas quais seriam usadas e na performance desejada, sem levar em conta a facilidade de programação nem aplicações especificas.

As dificuldades inerentes a este tipo de programação levaram ao desenvolvimento de linguagens que, cada vez mais, facilitassem ao homem a descrição e resolução de problemas.

Atualmente a programação da maioria das linguagens de alto nível ainda continua muito voltada para as máquinas, embora estas sejam muito mais fáceis de programar e muito mais poderosas que as anteriores.

2.2 Comparação entre linguagens convencionais e linguagens para IA.

A tabela 2.1 mostra as principais características das linguagens convencionais e compara a forma de implementação destas caracteristicas nos dois paradigmas de programação.

Tipo de processamento.

O tipo de processamento difere muito nos dois tipos de linguagens. Nas linguagens ditas convencionais o processamento é essencialmente numérico, ou seja , executa operações entre números e movimentação de dados. Já nas linguagens voltadas para IA o processamento é feito sobre símbolos, dentre os quais os números.

Tab 2.1 : Comparação entre Linguagens convencionais e Linguagens de Inteligência Artificial.

Característica	Ling. Conv.	Ling. de I. A.
Tipo processam.	Numérico	Simbólico
Técnica	Algoritmica	Proc. Heurística
Def. dos passos da solução	Precisa	Não explícita
Controle e dados	Separados	Interligados
Conhecimento	Preciso	Impreciso
Modificações	Raras	Frequentes

Técnica de execução e passos na solução.

As linguagens convencionais são linguagens procedurais, ou seja, a solução do problema é descrita na forma de um algoritmo determinista que é seguido passo a passo pelo interpretador da linguagem. Já nas linguagens de IA o usuário descreve a estrutura do problema e a sequência de execução é dada pela procura de satisfação dos objetivos pretendidos.

Controle e dados.

Nas linguagens convencionais existe uma clara separação entre programa e dados. No caso das linguagens de IA a distinção entre dado e programa é decorrente mais de uma convenção do programador do que da estrutura dos mesmos, para o compilador não existe diferença entre os dois.

Conhecimento.

Nas linguagens convencionais os dados tratados são dados precisos e completos. No caso de linguagens de IA muitas vezes não se tem informações precisas sobre um

determinado problema. Esta capacidade de tratamento de informações imprecisas é uma das maiores vantagens das linguagens de IA em relação as linguagens convencionais, embora isto traga um grande aumento no tempo de execução.

Modificações.

Nas linguagens convencionais os dados são precisos o que implica em que eles só serão alterados quando houver alguma alteração no universo que se está modelando por exemplo a inclusão de mais um objeto. No caso de linguagens de IA como as informações são imprecisas e interligadas, uma modificação qualquer pode acarretar várias outras alterações nas informações armazenadas no sistema, o que faz com que as modificações nos dados seja grande e constante.

2.3 Programação em Lógica.

A idéia essencial por trás da Programação em Lógica é a de que lógica pode ser usada como uma linguagem de programação. Isto pode ser extendido no sentido do uso da lógica para representar programas e no uso da dedução para executar computações. A idéia difere fundamentalmente da programação convencional no fato de que devemos descrever a estrutura lógica dos problemas em vez de descrever como o computador deve solucioná-los. A filosofia por trás deste formalismo está baseada na crença de que em vez do ser humano aprender a pensar em termos de operações de computador, o computador deve executar instruções que são fáceis para o ser humano especificar. E portanto um método de programação inteiramente voltado ao usuário. Além disto, por ser baseada em lógica de predicados é essencialmente não-determinista e paralela.

2.3.1 Cláusulas.

A programação em lógica está baseada em sentenças em formas de cláusulas que permitem uma sintaxe simples sem perder o poder do cálculo de predicados [KOW74]. Uma cláusula é um par de conjuntos de fórmulas atômicas A_i e B_i que pode ser expresso por :

B1, ... , Bm <-- A1, ... , An

Uma fórmula atômica tem a forma **P(x1, ... , xk)** onde **P** é o símbolo de um predicado de aridade **k** e **xi** é um termo. Um termo é uma variável ou expressão.

A interpretação de uma cláusula fica assim formulada : para todo x1, ... , xk ; B1 ou ... ou Bm é implicado por A1 e ... e An.

No entanto, a grande maioria das relações computáveis pode ser adequadamente definida usando um tipo especial de cláusula chamada cláusula de Horn. Uma cláusula de Horn é uma cláusula do tipo :

B1, ... , Bm <-- A1, ... , An com m <= 1

Existem portanto quatro tipos de cláusulas de Horn :

- m > 0 e n > 0 : este tipo define uma regra de inferência e pode ser representado por :

B <-- A1, ... , An

- m > 0 e n = 0 : este tipo define um fato e pode ser representado por :

B <---

- m = 0 e n > 0 : este tipo define um conjunto de objetivos a serem satisfeitos representado por :

<--- A1, ... , An

- m = 0 e n = 0 : este tipo define uma clausula nula, que é interpretada como uma sentença inválida.

2.3.2 Regra da Resolução.

A regra da resolução pode ser usada para a criação de sistemas de inferência usando cláusulas. Neste caso cada objetivo pode ser interpretado como a chamada de um conjunto de procedimentos (uma cláusula ou conjunto de cláusulas para cada procedimento). As cláusulas que serão executadas são aquelas cuja fórmula atômica **B** unificarem com alguma fórmula atômica que integra o objetivo. As fórmulas atômicas **Ai**, integrantes das cláusulas chamadas, passam então a ser vistas como novos objetivos a serem alcançados e o processo continua recursivamente. Um objetivo é considerado alcançado apenas quando unifica com um fato.

2.4 Prolog.

O Prolog é uma restrição da Programação em Lógica e é a principal das linguagens lógicas em uso atualmente na programação de sistemas de Inteligência Artificial. O Prolog padrão se diferencia da Programação em Lógica pela sintaxe e pelo uso.

Em Prolog cláusulas de Horn são definidas como :

```
K(X,Y) :- F(X) , G(X,Y) , F(Y).
```

Onde o símbolo **':-'** representa o operador de implicação e o símbolo ',' o operador AND.

Fatos.

Quando o lado direito de uma cláusula de Horn tem o valor lógico verdadeiro (t) ela é chamada de Fato e é representada conforme abaixo.

F(a,b).

A interpretação deste fato é dada por : F (a.b) é sempre verdadeiro.

Regras.

Ao contrário dos fatos uma regra é definida por uma cláusula de Horn onde o lado direito da implicação tem um corpo.

Predicados.

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

```
P(a,b).
P(c,d).
P(X,Y) :- G(X,Z) , H(Y).
```

No exemplo acima o predicado **P** é definido por dois fatos e uma regra.

2.4.1 Unificação.

Unificação é o processo de tornar dois termos iguais pela substituição de suas variáveis livres da maneira a mais geral possível. Por exemplo na unificação de F(X,a,Y,Y) e F(b,a,c,Z) são unificados como F(b,a,c,c).

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

De forma geral, um termo é unificado com uma cláusula pela unificação do termo com a cabeça da cláusula.

2.4.2 Backtracking.

Chama-se "bactracking" ao processo de resatisfação de cláusulas já satisfeitas com o intuito de achar outros valores que satisfaçam os objetivos . Este é um processo muito importante em Prolog mas é ao mesmo tempo o principal motivo do alto gasto de processamento apresentado por Prolog e linguagens semelhantes.

A figura 2.1 permite uma melhor visualizacão do processo. Nela é mostrada a execução da cláusula H para todos os seus valores. As barras usadas simbolizam falhas na satisfação de um objetivo.

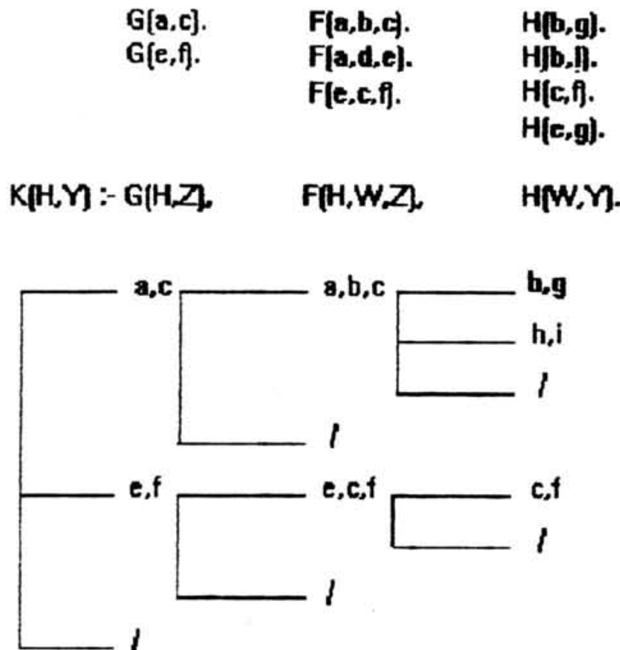


Fig 2.1 : Processo de backtracking

2.5 Paralelismo AND/OR.

Uma das principais características da Programação em Lógica é o alto grau de paralelismo inerente ao modelo. Este paralelismo é expresso basicamente de duas formas diferentes nas cláusulas de Horn. O exemplo abaixo ilustra estes dois tipos de paralelismo.

```
H(X,Y) :-  F(X,Z) , G(Z,Y).
H(X,Y) :-  F(X,Z) , I(X,Y,Z).
```

No exemplo acima existem duas cláusulas para a dedução do predicado **H**. Este tipo de paralelismo entre cláusulas é chamado de paralelismo **OR**, pois apenas uma das cláusulas sendo verdadeira torna o predicado cabeça de cláusula verdadeiro. Este tipo de predicado é de simples implementação pois não existe interação entre as diversas cláusulas de um predicado.

Já o paralelismo **AND** é o paralelismo entre os predicados do corpo de uma cláusula. No exemplo acima **F(X,Z)** e **G(Z,Y)** definem um paralelismo do tipo AND. Neste tipo de paralelismo os processos paralelos podem ter, e normalmente tem, variáveis em comum. Isto implica em que os processos devem manter um controle das variáveis partilhadas o que dificulta bastante a implementação deste tipo de paralelismo.

2.6 Prolog Concorrente.

Embora a Programação em Lógica traga inerente a sí o paralelismo, as primeiras linguagens lógicas não permitiam a exploração deste paralelismo. Com o surgimento de novas máquinas capazes de executar algorítmos paralelos foi possível a criação de linguagens que explorassem esta característica do modelo lógico. O PARLOG [CLA86] foi uma

das primeiras linguagens lógicas paralelas e até hoje serve de modelo para muitas outras.

As principais diferencas entre o PARLOG e o Prolog são : Guardas de cláusulas e Declaração de modo.

Guarda de cláusula.

As guardas de cláusula são predicados que devem ser verdadeiros para que a cláusula possa ser avaliada. No exemplo abaixo **F** e **G** só serão avaliadas quando **C** for verdadeira. **C** difere dos outros predicados porque sua falha não implica na falha do predicado cabeça da cláusula mas simplesmente faz com que a cláusula aguarde que ele seja verdadeiro para continuar avaliando os outros predicados.

```
H(X,Y) :- C(X) : F(X,Y) , G(X,Y).
```

Declaração de modo (Mode)

Esta declaração é usada para especificar o tipo de unificação que deve ser feito entre os argumentos dos predicados.

```
Mode H(X?,Y?,Z^).
```

A declaração acima mostra um exemplo de especificação da unificação dos argumentos. A '**?**' indica que a variável só pode ser usada para a unificação com um argumento de entrada e o '^' para unificação com argumento de saída.

2.7 Conclusão.

Neste capítulo foi dada uma breve introdução das linguagens Lógicas e de Prolog, visto que o modelo

apresentado no capítulo 4 busca modelar cláusulas destas linguagens. Além disto mostrou algumas das características que fazem do Prolog uma das linguagens mais importantes da atualidade o que fez com que fosse escolhida para o projeto de 5º geração do Japão e tantos outros.

3 MAQUINAS PROLOG.

3.1 Introdução.

As primeiras máquinas projetadas para suportar processos de Inteligência Artificial foram as implementações de Lisp no PDP-6 e posteriormente nos seus sucessores PDP-10 e PDP-20 (todos da Digital Equipment Corporation). A partir daquela época foi grande a proliferação de computadores capazes de prover um processamento simbólico, contudo muito ainda resta a ser estudado nesta área. Os principais esforços estão voltados para : implementação em hardware das principais operações primitivas em aplicações de IA, o projeto de arquiteturas micro-programadas que suportam funções mais complexas de IA e o projeto de arquiteturas voltadas para linguagens e esquemas de representação do conhecimento .

No entanto para que um computador suporte o processamento de aplicações em IA de maneira eficiente ele deve explorar ao máximo as características destas aplicações que são: processamento simbólico, algorítmos não deterministas, alocação dinâmica da memória, alto grau de paralelismo, tratamento do conhecimento e sistemas abertos.

Processamento Simbólico

A aquisição, representação e o inteligente uso da informação e do conhecimento são fundamentais no processamento de IA. Isto implica no uso de operações simbólicas poderosas tais como comparação, seleção, reconhecimento de padrões e operações lógicas.

No caso específico de Máquinas Prolog a comparação e a unificação são os mecanismos básicos para a

implementação de inferências utilizando a regra de resolução.

Processamento não determinista.

Muitos dos algorítmos usados em IA são não deterministas. Isto significa que muitas vezes é impossível, com as informações disponíveis, definir qual processo deve ser executado num determinado momento. Isto advém da falta de um conhecimento completo do problema e resulta na necessidade de explorar várias (ou mesmo todas) as possibilidades de forma exaustiva.

Embora esta caracterÍstica seja importante para o tratamento de aplicações de IA ela acarreta uma explosão no tempo de processamento, o que torna imprescindível que o computador tenha uma alta capacidade de processamento.

Particularmente no caso da linguagem Prolog a Programação em Lógica, que é o modelo usado para a sua definição, é um modelo não determinista.

Alocação dinâmica da memória.

Uma das grandes vantagens da IA é a possibilidade de tratar problemas sobre os quais não se tem um conhecimento completo, o que confere um certo grau de incerteza ao processamento. Com isto a definição das estruturas de dados e funções bem como a criação de novas estruturas e funções devem poder ser feitas durante a execução do processo. Disto resulta que, seja qual for o tamanho da área alocada a uma determinada estrutura quando do início do processo, ela pode vir a ser insuficiente ou a gerar um grande desperdício de área. Para contornar este problema a alocação de espaços na memória deve ser dinâmica.

Alto grau de paralelismo

Em algorítmos deterministas existem conjuntos de tarefas que necessitam obrigatoriamente serem executadas, mas que no entanto podem ser executadas de forma independente. Este tipo de paralelismo é chamado de paralelismo AND.

No entanto algorítmos de IA, por serem não deterministas, contêm conjuntos de tarefas que são executadas de forma alternativa. Isto permite um outro nível de paralelismo não encontrado em processos comuns que é o paralelismo OR.

Portanto fica claro que algorítmos de IA permitem um alto grau de paralelismo e que uma implementação eficiente tem que saber explorar ao máximo este paralelismo.

Tratamento do conhecimento.

O conhecimento é um importante componente para reduzir o complexidade da solução de um problema. No entanto muitos problemas em IA tem um alto grau de complexidade o que leva ao tratamento de uma grande quantidade de conhecimento. Além disto, este conhecimento é muitas vezes incompleto ou incerto. Logo o tratamento do conhecimento é uma parte muito importante a ser levada em conta em aplicações de IA.

Sistemas Abertos.

Em muitas aplicações de IA o conhecimento que se tem para a solução de um problema pode ser incompleto ou este conhecimento pode variar no tempo tendo como causa desta variação mudanças no ambiente no qual está inserido.

Isto implica que sistemas de IA devem ser projetados de forma a permitir um refinamento e atualização contínuos do conhecimento.

3.2 **Classificação de máquinas para IA**

Conforme a figura 3.1 , o desenvolvimento de um computador voltado a IA pode ser encarado de duas formas: "Bottom-up" e "Top-down" [HWA87].

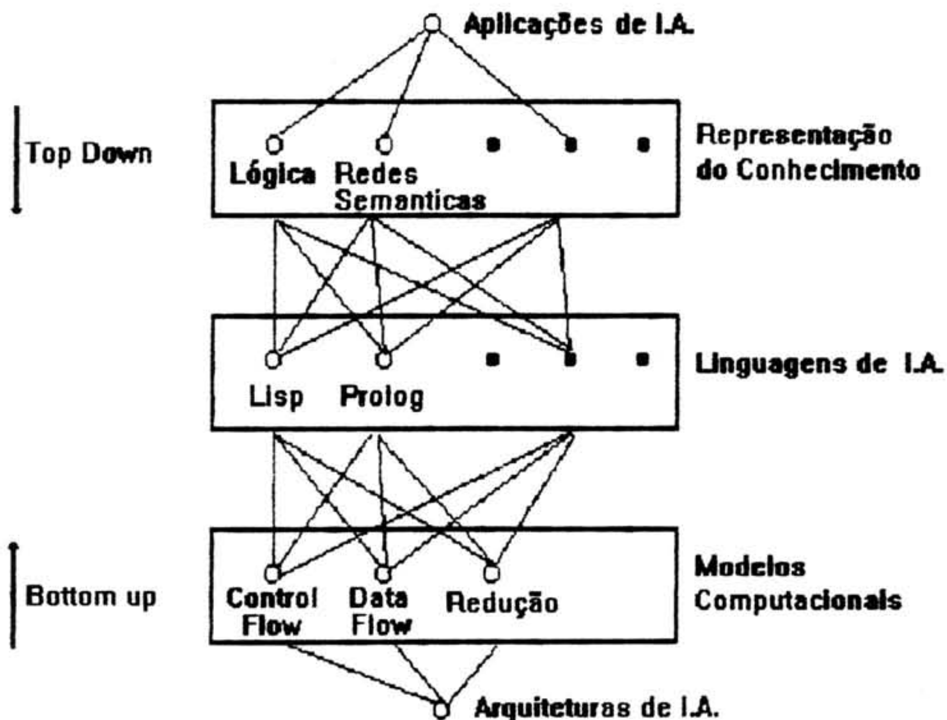


Fig 3.1 : Enfoques no desenvolvimento de um computador para IA.

No caso do desenvolvimento "Bottom-up" começa-se projetando uma arquitetura com características específicas e passa-se a projetar sucessivamente os outros níveis até o nível das aplicações. Exemplos deste tipo de enfoque são

computadores com "hardware" especiais para acesso aos dados, reconhecimento de padrões, unificação etc....

Já no caso do enfoque "top-down" primeiro define-se uma aplicação e depois passa-se a projetar os níveis inferiores até chegar a arquitetura.

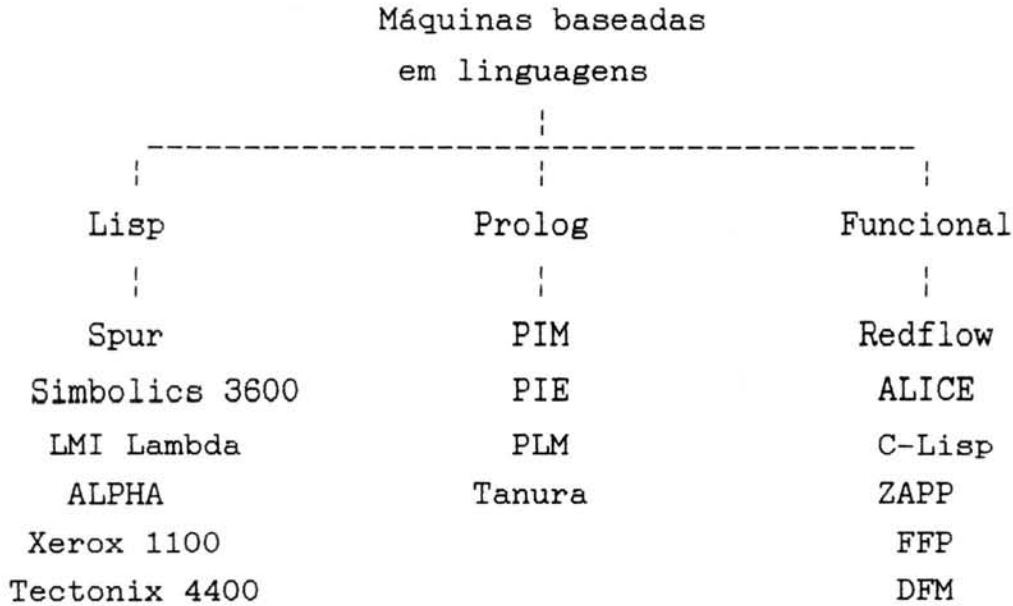
O esquema abaixo mostra que as máquinas voltadas a IA podem ser divididas em três categorias principais: máquinas baseadas em linguagens, máquinas baseadas em conhecimento e máquinas que provêem uma "interface" inteligente.

Arquiteturas para IA	Máquinas baseadas em linguagens
	Máquinas baseadas em conhecimento
	Máquinas p/ interface inteligente

Máquinas baseadas em linguagens.

Máquinas baseadas em linguagens são projetadas de forma a executar eficientemente um modelo computacional de alguma linguagem de alto nível de IA. Normalmente estas máquinas implementam operações primitivas da linguagem em "hardware".

O esquema na página a seguir mostra os principais tipos de máquinas baseadas em linguagens dando exemplos para cada tipo de máquina que está em desenvolvimento. [HWA87]



Máquinas baseadas em conhecimento.

Máquinas baseadas em conhecimento tentam prover uma execução eficiente de modelos poderosos de representação e manipulação de objetos.

Máquinas que provêem uma "interface" inteligente

Estas são máquinas que tentam prover uma melhor comunicação com o usuário ou com o ambiente no qual estão inseridas. Exemplos típicos são máquinas que interpretam a fala, máquinas que entendem linguagem natural, máquinas voltadas a visão robótica, etc.

3.3 Máquinas Prolog.

Máquinas Prolog são máquinas que executam diretamente as regras de resolução da programação lógica através de programas escritos em uma linguagem lógica, das quais Prolog é a principal.

A tabela 3.1 mostra as principais máquinas Prolog

que estão sendo desenvolvidas, com suas respectivas arquiteturas e modelos computacionais. [HWA87]

Tab 3.1 : Máquinas Prolog

Máquina	Arquitetura	Paralelismo	Status
Parallel Inference Machine (PIM)	Fracamente acoplada	AND, OR, Unificação	Em construção
Parallel Inference Engine (PIE)	Fracamente Acoplada	OR , Unificação	Em construção
Programmed Logic Machine (PLM)	Coprocessador	OR , Unificação	Em construção
Tamura Machine	Multiproces. master slave	AND	Pronta

3.4 Exemplo de máquinas Prolog.

Trataremos neste item de dar algumas noções básicas de uma máquina Prolog. Para tal escolhemos como exemplo a máquina PIM-D da I.C.O.T. do Japão por ser uma das que permite um alto grau de paralelismo (Três tipos de paralelismo: AND, OR e Unificação).

A máquina PIM-D é baseada no modelo computacional de máquinas de fluxo de dados que é um modelo propício para processamento paralelo. Programas neste modelo são representados por grafos de fluxo de dados onde cada nodo corresponde a operações e arcos direcionados correspondem ao caminho de dados pelos quais os operadores são enviados.

A execução de programas em Lógica (Prolog concorrente) é feita através de um processamento do tipo "goal-driven". Uma cláusula de um programa é iniciada quando um objetivo é dado e termina quando retorna as soluções do objetivo. A função básica neste processo é a unificação de operandos. Quando uma cláusula inicia ela trata o seu corpo como novos objetivos a serem atendidos e espera pelos resultados.

A arquitetura básica da máquina PIM-D é composta por um conjunto de elementos de processamento e elementos de memória ligados entre si por uma rede de interconexão.

O paralelismo OR é implementado através da alocação das cláusulas concorrentes uma para cada PE e com os dados de chamada duplicados. Isto permite um fácil controle do processo já que não haverá variáveis compartilhadas.

Já no caso do paralelismo AND deve-se prover o sistema de uma verificação de consistência que permite que dados compartilhados sejam eficientemente tratados. Isto implica em maior complexidade do controle e um maior tempo de execução.

A máquina usa um esquema de etiquetagem (TAG) no qual cada operando tem um campo de valor e um campo de etiqueta especificando o tipo de dado do operando. Se o operando consiste de dados estruturados o campo de valor tem um apontador e a etiqueta contém além do tipo de estrutura a informação que diz se a estrutura tem ou não variáveis livres (unbound). Quando da unificação a máquina reconhece a etiqueta do campo e desvia o controle para a rotina apropriada do "Hardware".

A estrutura de dados é armazenada e distribuída para a estrutura de memórias e é compartilhada por todos os

elementos de processamento. Isto elimina a redundância de dados mas aumenta a latência do acesso. Para diminuir este problema os elementos devem fazer múltiplos pedidos de acesso a memória sem esperar pelas respostas. Neste caso pedidos e respostas devem ser identificados, uma vez que as respostas podem ser devolvidas em ordem diferente dos pedidos.

A unificação é feita através da combinação das primitivas unificação, checagem e consistência, partilhamento, decomposição e substituição.

Primitivas.

- Unificação : E alocada uma primitiva deste tipo para cada argumento da cabeça de cláusula que se deseja unificar. Esta primitiva unifica o argumento recebido e propaga o valor das instâncias e o ambiente para outras primitivas.

- Decomposição : Esta primitiva é usada para fazer a decomposição de argumentos complexos e chamar novas primitivas de unificação para os novos argumentos.

- Checagem e consistência : Esta primitiva recebe informações das primitivas de unificação e faz a checagem e consistencia destes valores.

- Partilhamento : Esta primitiva faz o partilhamento das variáveis que foram checadas e consistidas entre as primitivas de unificação.

- Substituição : Esta primitiva faz a substituição das variáveis unificadas no corpo da cláusula e reinicia o processo.

3.5 Conclusão.

Este capítulo mostrou em que contexto estão situadas as máquinas Prolog entre as máquinas de IA e quais características principais estas máquinas devem ter para efetivamente explorarem todas as potencialidades das linguagens lógicas.

Vimos também que a maioria destas máquinas ainda estão em desenvolvimento. Isto advém das grandes dificuldades encontradas numa implementação que permita uma efetiva exploração de todo o paralelismo inerente as linguagens Lógicas. A maioria das máquinas citadas permite apenas um paralelismo OR e em alguns casos da unificação. Quando o paralelismo AND é usado ele trás consigo um grande "overhead" de controle. Outro grande problema destas máquinas é o acesso a memória, que na maioria dos casos é o gargalo do sistema.

4 MODELO DE EXECUÇÃO.

4.1 Introdução.

Neste capítulo será descrito o modelo de execução proposto para o modelo abstrato de Máquinas de Cláusulas. No entanto é necessário que primeiro se introduza os modelos de redes de Petri e Máquinas de Cláusulas.

4.1.1 Redes de Petri.

A teoria de redes de Petri foi criada, e é usada, para modelagem e análise de sistemas e das informações que fluem através deles. A teoria introduz uma linguagem formal, muitas vezes em termos matemáticos, que permite descrever os objetos e eventos modelados de uma forma exata e uniforme.

Uma das características principais das redes de Petri é a independência temporal existente entre eventos, o que permite que esta teoria seja especialmente útil na modelagem de sistemas concorrentes, assíncronos e não-deterministas.

Concomitantemente, a teoria permite a obtenção de informações sobre a estrutura e o comportamento dinâmico do sistema e possibilita a verificação de propriedades do sistema além de provas de correção. Consequentemente pode-se operar com informações deste tipo de forma a propor modificações e melhorias no sistema.

Esta flexibilidade e precisão são alguns dos principais motivos que levaram as redes de Petri a serem usadas em aplicações as mais variadas e terem sido feitas várias versões e extensões a mesma.

4.1.2 Máquinas de Cláusulas.

Uma máquina de cláusulas, conforme descrito em [COS85], é um modelo abstrato para a execução concorrente de programas em lógica. Este modelo se baseia num tipo especial de redes de Petri de alto nível, cujo princípio de operação é o da equilibração de redes. O modelo permite a execução direta de programas em lógica além de possibilitar a exploração de todo o paralelismo e concorrência inerente a este tipo de programação.

4.2 Descrição do Modelo de Execução.

O modelo de execução aqui proposto baseia-se no modelo de Máquinas de Cláusulas e tenta extendê-lo de forma a definir um modelo de execução de programas em lógica utilizando para a equilibração das redes o princípio da propagação de valores.

4.2.1 Redes Predicado-Transição.

O modelo consiste de um conjunto de cláusulas (transições) onde a cada cláusula está associado um conjunto de predicados através de um conjunto de arcos, e onde as cláusulas se comunicam através de predicados comuns.

Em termos de rede de Predicado_Transição a estrutura do modelo é definida por :

R = (P, C, A).

Onde **P** é o conjunto de todos os predicados, **C** é o conjunto de todas as cláusulas e **A** é o conjunto de todos os arcos.

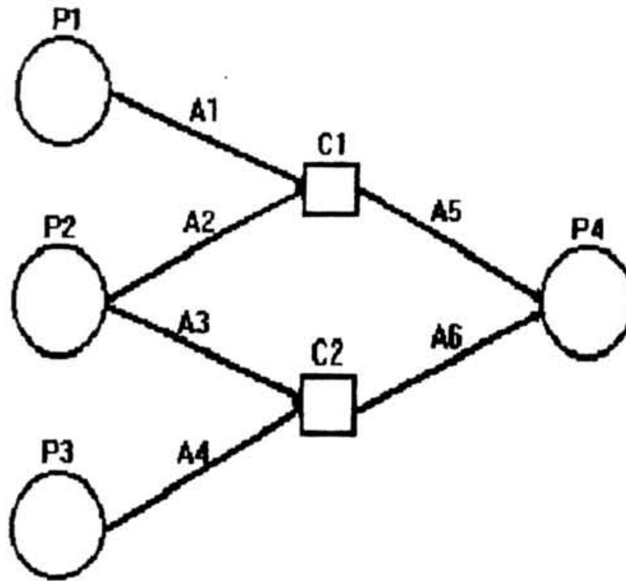


Fig 4.1 : Exemplo de uma cláusula do modelo

Onde P = { P1, P2, P3, P4 };
C = { C1, C2 };
A = { A1, A2, A3, A4, A5, A6 };

4.2.2 Predicados.

Os predicados são estruturas passivas no modelo e armazenam tuplas de valores. Cada uma destas tuplas é um conjunto ordenado de constantes e/ou variáveis e tem associadas a si várias informações, dentre as quais as mais importantes são : valor lógico da tupla, justificativa do valor lógico da tupla e estado da tupla. Desta forma uma tupla pode ser representada por :

<T1, ... , Tn : V, J, E >

Onde **T1** a **Tn** representam os termos da tupla e **V**, **J** e **E** são respectivamente valor lógico, justificativa e estado da tupla.

Valor lógico da tupla .

Uma tupla pode apresentar um de três valores lógicos : **v** (Verdadeiro), **f** (Falso), **i** (Indeterminado). Os valores **v** e **f** expressam informações conhecidas pelo sistema, já o valor **i** indica que não se sabe o valor lógico da tupla. Tuplas não explicitadas no sistema são assumidas como tendo valor lógico **i**.

Justificativa do valor lógico da tupla.

Uma tupla pode ter seu valor lógico definido das seguintes formas : **e** (Explicitado), **d** (Deduzido), **a** (Assumido). A justificativa **e** indica que o valor lógico da tupla foi explicitado pelo usuário. A justificativa **d** indica que o valor foi deduzido pela execução do modelo. A justificativa **a** também indica um valor explicitado pelo usuário mas do qual o usuário não tem certeza.

Estados da tupla.

As tuplas apresentam a cada instante da execução do modelo um de cinco estados que são : **a** (Atualizada), **e** (Em espera), **i** (Inconsistente), **p** (Procura do primeiro valor) e **t** (Procura de todos os valores).

- Estado **e** : Neste estado a tupla não está sendo usada para nenhuma atividade no modelo.

- Estado **a** : Uma tupla só pode ter seu valor lógico alterado se ela estiver no estado **e** ou quando da sua criação. Caso a atualização seja efetuada, a tupla passa para o estado **a** e assim se mantém até que todas as cláusulas que usam o predicado tenham tratado a atualização, ou seja

constatada uma inconsistência.

- Estado **i** : A tupla passará para o estado **i** quando for constatada alguma inconsistência na atualização do valor lógico.

- Estados **p** e **t** : Quando uma cláusula desejar satisfazer um determinado objetivo, isto será explicitado criando-se uma tupla especial com estado **p** ou **t**. Estes estados acarretarão numa pesquisa tipo "demand-driven" que tentará satisfazer o objetivo. Caso o estado seja **p** a pesquisa parará quando o primeiro resultado for encontrado, caso o estado seja **t** a pesquisa tentará encontrar todas as respostas possíveis.

Exemplo :

Um exemplo de um conjunto de tuplas válido para o predicado **é-casado-com(X,Y)** é mostrado abaixo :

```
é-casado-com =
     {<João,Maria :v,e,e>, <Paulo,Carla :v,e,a>,
      <Pedro,Rosa :v,e,e>, <Carlos,Silvia :f,e,e>};
```

4.2.3 Arcos.

Os arcos são utilizados como ligação entre uma cláusula e seus predicados, e a cada arco é associada uma tupla de termos que expressa a maneira pela qual as tuplas dos predicados serão vistas na propagação de valores. Todas as variáveis explicitadas nos arcos são variáveis locais à cláusula a qual está ligado o arco.

Os arcos também definem as pré-condições e pós-condições de cada cláusula usando o sentido dos arcos de tal forma que : arcos no sentido predicado-cláusula indicam pré-condições e arcos no sentido cláusula-predicado indicam

pós-condições.

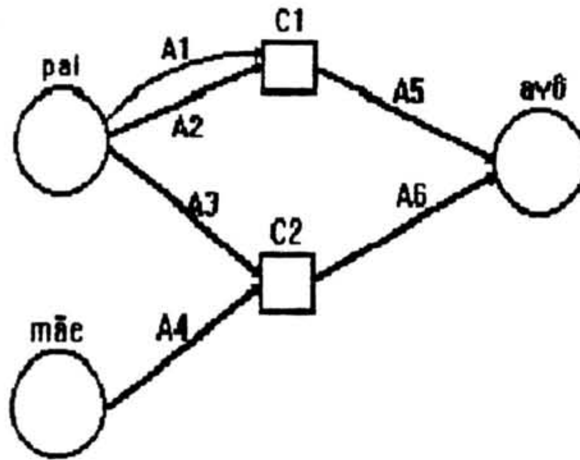


Fig 4.2 Exemplo de arcos

Onde para a cláusula **C1** temos :

- 2 pré-condições : Predicado **pai** através do arco **A1** e predicado **pai** através do arco **A2**.
- 1 pós-condição : Predicado **avô** através do arco **A5**.

E para a cláusula **C2** temos :

- 2 pré-condições : Predicado **pai** através do arco **A3** e predicado **mãe** através do arco **A4**.
- 1 pós-condição : Predicado **avô** através do arco **A6**.

4.2.4 Cláusulas.

As cláusulas são os elementos responsáveis pela execução do processo de dedução. Para isto utilizam os dados armazenados nos predicados e as informações dadas pelos arcos, propagando os valores de forma a equilibrar a

rede. No entanto, em uma execução de cláusulas de forma não-determinista como o modelo se propõe, torna-se as vezes necessário um mecanismo que impeça a execução incondicional de todas as pré-condições simultaneamente. Para prover esta capacidade adotou-se a noção de guardas de cláusulas [CLA86] no modelo.

Cada cláusula executa individualmente o mesmo processo sem levar em conta a existência de outras cláusulas.

O processo básico é o seguinte :

1 Para todas as cláusulas.
  1.1 Para cada predicado da cláusula.
    1.1.1 Se alguma tupla foi incluída ou teve seu valor lógico alterado então executa o procedimento adequado. Os procedimentos possíveis são : Propagação de valores, Pesquisa de tupla e Tratamento de inconsistência.

Este "loop" deve ser executado ininterruptamente enquanto o processo dedutivo estiver em execução.

Exemplo:

Uma rede que calcule o fatorial pode servir de exemplo para mostrar esta necessidade.

No caso de uma linguagem de programação em lógica que seja determinista, um programa que execute o fatorial pode ser escrito da seguinte forma :

```
fat(1,1).
fat(X,F) :- fat(X - 1,F1), F = X.F1.
```

Mas no caso de uma linguagem onde não haja

ordenação na execução das cláusula não se pode garantir que o programa interromperá a execução. Neste caso torna-se necessário um mecanismo que diga quando uma cláusula deve ser executada ou não. Um programa para calcular fatorial usando guardas de cláusulas pode ser dado por :

```
fat(X,1) :- X = 1:.
fat(X,F) :- X # 1: fat(X - 1, F1), F = X.F1.
```

A rede para este programa fica então :

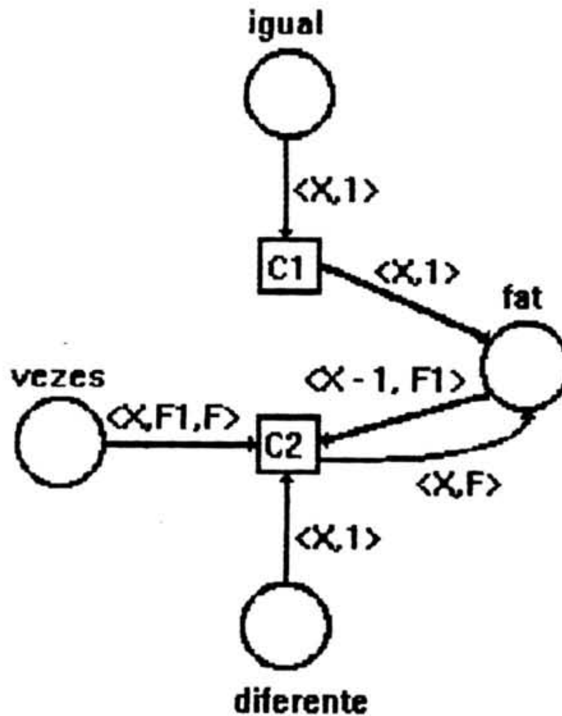


Fig 4.3 Rede para cálculo do Fatorial

Na rede da figura 4.3, **diferente** é a guarda da cláusula C2, o que significa que numa consulta do tipo "demand-driven" as pré-condições **vezes** e **fat** só serão avaliadas caso a pré-condição **diferente** seja satisfeita.

4.2.5 Propagação de valores.

Uma cláusula pode ser entendida como uma implica-

ção e pode ser representada como a seguir :

P => Q

Onde P é a conjunção de todas as pré-condições e Q é o conjunto das pós-condições. Portanto, uma cláusula terá valor verdadeiro se a implicação tiver valor verdadeiro. Com isto pode-se construir uma tabela de valores para pré e pós-condições nas quais a cláusula tem valor verdadeiro.

P	Q
V	V
F	V
F	F

Com base nesta tabela pode-se criar outra tabela que mostre os procedimentos que devem ser tomados quando da inclusão/alteração de valores para que a propagação seja feita corretamente.

Tab 4.1 : Tabela de Propagação.

		Pre-condição	Pós-condição
Inclusão	V	A1	--
	F	--	A3
	I	--	A4
Alteração	V	A1	--
	F	A2	A3
	I	A2	A4

A1: Propagação do valor lógico : Este procedimento compõe-se dos seguintes passos :

1. Achar nas outras pré-condições os conjuntos de tuplas que instanciem a cláusula para os valores dos argumentos constantes na tupla para a qual foi fixado o valor **v**.

2. Para cada pós-condição gerar um conjunto de todas as tuplas possíveis a partir das tuplas encontradas em 1.

3. Incluir as tuplas nas suas respectivas pós-condições colocando junto o valor logico **v** e a justificativa **d**.

   Caso no momento da inclusão seja constatado que a tupla já existe na pós-condição com um valor lógico diferente deve-se executar um dos procedimentos a seguir :

   a) no caso de tupla ter valor lógico **i**, alterar o valor da tupla para **v**.
   b) no caso da tupla ter valor lógico **f**, se a justificativa atual for **a** então alterar o valor lógico para **v** e a justificativa para **d**, caso contrario acusar inconsistência nos dados do predicado.

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

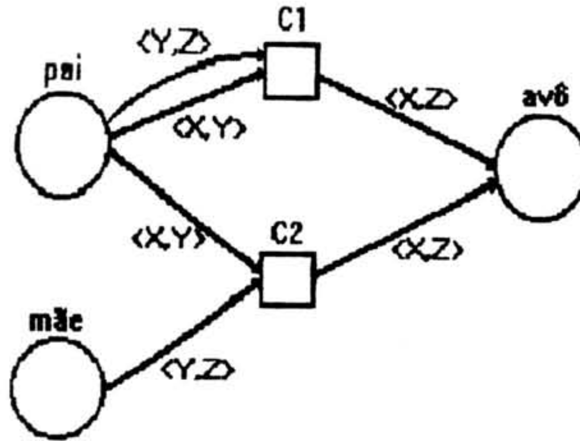


Fig 4.4 : Rede para cálculo do predicado avô.

Na rede antecedente, dados os seguintes conjuntos de tuplas para os predicados:

pai = { <João,Paulo>, <Paulo,Pedro>,
<Carlos,Marcos>, <Marcos,Maria>,
<Alvaro,Beatriz> };
mãe = { <Beatriz,José>, <Ana,Ivo> };
avô = { <João, Pedro>, <Carlos,Maria>,
<Alvaro,José> };

Se for acrescentada a tupla <Marcos,Ana> no predicado **pai**, esta tupla será propagada para a pós-condição gerando no predicado **avô** duas novas tuplas que são : <Carlos,Ana> e <Marcos,Ivo>.

A2 : Teste do valor lógico das pós-condições : Neste caso os passos são os seguintes:

1. Determinar para cada pós-condição todas as tuplas com valor verdadeiro que possam

ter sido deduzidas em função da tupla cujo valor foi alterado.

2. Para cada uma destas tuplas encontradas verificar se ainda existe algum conjunto de tuplas nas pré-condições que a torne verdadeira.

3. Caso não haja, trocar o valor da tupla para **i** e a justificativa para **d**.

A3 : Teste do valor lógico da tupla : Neste caso devem ser executados os seguintes passos :

1. Verificar para a tupla que teve o valor lógico alterado, se ainda existe algum conjunto de tuplas nas pré-condições que indiquem sua dedução como verdadeira.
2. Caso haja, acusar inconsistência.

A4 : Teste do valor lógico da tupla : Neste caso devem ser executados os seguintes passos :

1. Verificar para a tupla que teve o valor lógico alterado, se ainda existe algum conjunto de tuplas nas pré-condições que indiquem sua dedução como verdadeira.
2. Caso haja trocar o valor lógico da tupla para **v** e a justificativa para **d**.

4.2.6 Pesquisa de tupla.

O modelo aqui proposto baseia-se na propagação de valores o que permite uma dedução do tipo "bottom-up" que parte das pré-condições para deduzir as pós-condições.

No entanto para que o modelo seja útil ele necessita também de uma dedução do tipo "top-down" que

permita a satisfação de objetivos explicitados pelo programa.

Para que isto seja possível, foi criado um processo de pesquisa do tipo "demand-driven" que permite este tipo de dedução. Esta pesquisa baseia-se nos estados **p** e **t** das tuplas. Quando uma cláusula detecta uma tupla em um dos estados acima nas suas pós-condições, passa a executar o seguinte procedimento :

1. Testar se existe algum conjunto de tuplas nas pré-condições que permita a dedução pedida.
2. Se houver criar a tupla na pós-condição.
3. Caso contrário, propagar o pedido para as pré-condições.

Por exemplo podemos mostrar a execução da rede da figura 4.3 imaginando a requisição da satisfação de um objetivo e seguindo o desenvolvimento do processo de pesquisa e propagação.

Incluindo-se no predicado **Fat** a tupla <3,F:_,_,P>.

Tab 4.2 : Tabela de execução do predicado FAT.

Passos	Cláusula C1		Cláusula C2			Procedimento
	Igual	Fat.	Dif.	Fat.	Vezes	
1		<3,F>				Pesq.
2	F		V	<2,F>	<3,F1,F>	Pesq.
3	F		V	<1,F>	<2,F1,F>	Pesq.
4	V	<1,1>	F			Pesq/Prop.
5					<2,1,2>	Prop.
6				<2,2>		Prop.
7					<3,2,6>	Prop.
8				<3,6>		Prop./Resp.

4.2.7 Inconsistencia.

A inconsistência é tratada no modelo através de uma interrupção para o programa indicando que houve inconsistência em uma determinada cláusula e marcando os fatos geradores da inconsistência. O programa deve então intervir no processo, alterando os fatos ou se omitindo. No caso de omissão a inconsistência permanecerá no sistema e qualquer referência futura a uma tupla em estado de inconsistência resultará em nova interrupção. No entanto o restante do sistema, que independa das tuplas que causam inconsistência, pode ser usado de forma normal.

O procedimento quando da constatação de inconsistência é o seguinte :

1 Caso seja constatado inconsistência numa tupla na pós-condição
  1.1 Determinar todos os conjuntos possíveis de tuplas que permitem a dedução da tupla inconsistente.
  1.2 Para cada uma das tuplas fazer :
    1.2.1 Se a tupla tem justificativa e ou a mostrar a tupla ao usuário.
    1.2.2 Caso contrario trocar o estado da tupla para inconsistente.

4.3 Conclusão.

Neste capítulo foi mostrado o modelo de execução para cláusulas Prolog baseado no modelo abstrato de máquinas de cláusulas. O modelo proposto permite um alto grau de paralelismo e tem como característica importante a inexistência de comunicação direta entre cláusulas.

No capítulo a seguir será proposta uma arquitetura que possibilite a execução do modelo e que mantenha as suas caracteristicas.

5 ARQUITETURA PROPOSTA.

5.1 Introdução.

A máquina de cláusulas aqui descrita é uma proposta de implementação do modelo de execução de cláusulas lógicas visto anteriormente.

Isto implica que a arquitetura deverá espelhar as principais características do modelo, a saber:

- Permitir o aproveitamento do paralelismo inerente a programação em Lógica.
- Cada cláusula é um processo independente dos outros.
- As cláusulas se comunicam apenas através de seus predicados.
- Todas as cláusulas executam processos idênticos.
- Existe grande concorrência no acesso a mesma informação.

A máquina deverá ser ligada a um hospedeiro que executará as funções de interface com o usuário e o partilhamento das cláusulas e predicados entre os processadores e as memórias.

Hospedeiro.

O processador hospedeiro será descrito apenas através das funções que deve desempenhar para um bom funcionamento da máquina de cláusulas.

Funções do hospedeiro.

O hospedeiro deve prover toda a interface com o

usuário tal como: entrada e saída de dados, compilação e utilitários. Além disto é necessário que o hospedeiro realize o partilhamento de cláusulas e predicados entre processadores e memórias. Isto deve ser feito de forma a que, preferencialmente, as cláusulas que utilizam predicados iguais fiquem alocadas no mesmo bloco. Consequentemente minimizando sobremaneira a comunicação entre blocos. O hospedeiro deve também alocar processadores para as cláusulas criadas durante o processo de execução.

De forma a diminuir o tempo gasto com sucessivos partilhamentos de cláusulas e predicados durante a execução de um programa é recomendável que o número máximo de cláusulas de um programa seja menor ou igual ao número de PEs (Processor Elements) da máquina de cláusulas. Caso isto não seja possível o programador deve dividir o programa em módulos que serão executados separadamente.

5.2 Escolha do modelo básico de arquitetura.

5.2.1 Memória local.

As características do modelo de execução nos trazem diretamente a uma arquitetura onde cada cláusula é tratada por um processador e os predicados a ela acoplados são armazenados em memórias locais ao processador (conforme figura 5.1).

Esta arquitetura é tentadora pois permite o total aproveitamento de todas as vantagens do modelo de execução além de permitir um alto grau de integração. No entanto o modelo prevê que várias cláusulas possam usar o mesmo predicado, já que esta é a única via de comunicação entre as cláusulas. Isto acarretaria dois grandes problemas que sao: duplicação dos dados e coerência de dados.

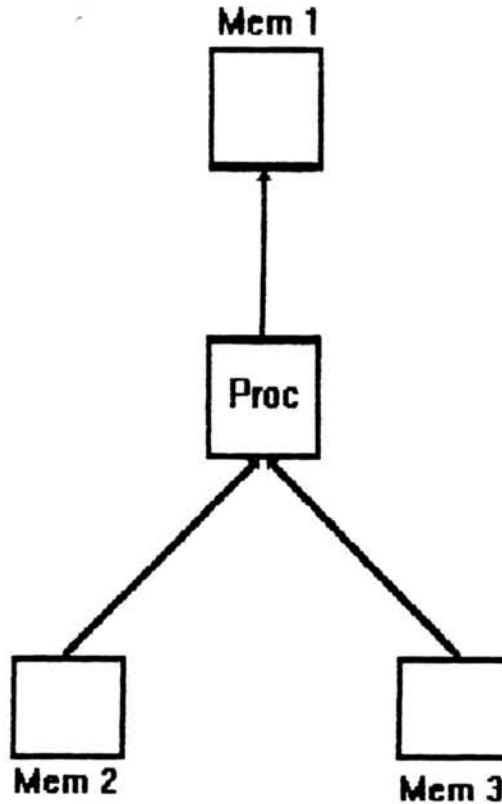


Fig 5.1 : Modelo de arquitetura com memória local

Duplicação dos dados.

Deve-se notar que uma das características das linguagens lógicas e do modelo de execução é o compartilhamento de predicados, via de regra, por várias cláusulas. Isto acarretaria que houvessem várias cópias de um mesmo predicado tornando a redundância muito grande.

Coerência de dados.

Com uma tão grande redundância o processo de manter a coerência dos dados seria extremamente trabalhoso e demandaria uma grande rede de interconexão entre todos os processadores.

Tamanho da memória local.

Outro problema crucial nesta arquitetura seria o tamanho da memória. Como cada predicado ocuparia uma memória não importando o tamanho do predicado, o tamanho das memórias deveria ser grande o que acarretaria extrema ineficiência no uso das memorias. Com isto, predicados pequenos deixariam a maior parte da memória inaproveitada enquanto que para predicados muito grandes torna-se-ia necessário prever um esquema de "overflow".

Este problema poderia ser minimizado se em vez de três memórias locais tivessemos apenas uma memória local que armazenasse os três predicados.

No entanto os problemas acima citados, principalmente a duplicação dos dados, inviabilizam a arquitetura.

5.2.2 Memória global dividida.

A próxima arquitetura a ser cogitada foi a de uma memória global dividida entre os processadores (conforme figura 5.2). Ou seja cada processador está associado a uma memória que é acessada diretamente apenas por ele. A memória global é, neste caso, formada pela soma das memórias locais.

Com isto o problema da duplicação de dados não existe já que só existe uma cópia de cada predicado. O problema que ocorre aqui é que os processadores vão passar a maior parte do tempo lendo a sua memória para fornecer dados aos outros processadores já que devido as características do modelo de execução cada vez que uma cláusula recebe uma nova informação o processador deve percorrer integralmente todos os predicados a ela ligados para fazer a propagação da informação.

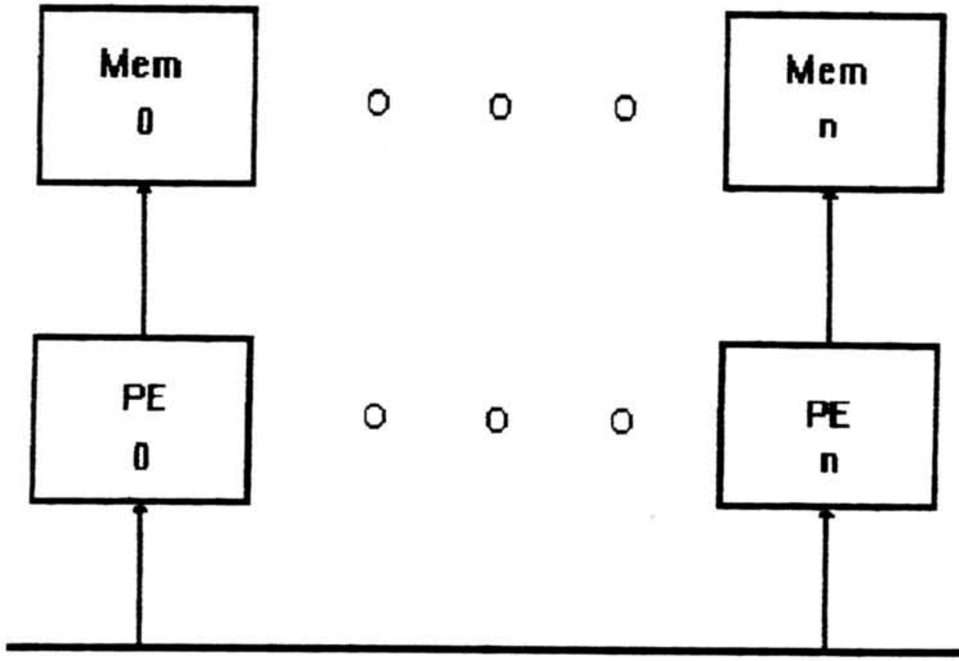


Fig 5.2 : Modelo de arquitetura com memória global dividida

5.2.3 Memória global partilhada.

A opção escolhida para o modelo de arquitetura foi o de uma memória global partilhada (conforme figura 5.3).

Esta arquitetura elimina os problemas de duplicação de dados e tamanho de memória e reduz o problema de comunicação entre os processos. Em contrapartida cria um gargalo de acesso a memória. No entanto com o uso de duas técnicas, que serão descritas posteriormente, este problema é sensivelmente reduzido.

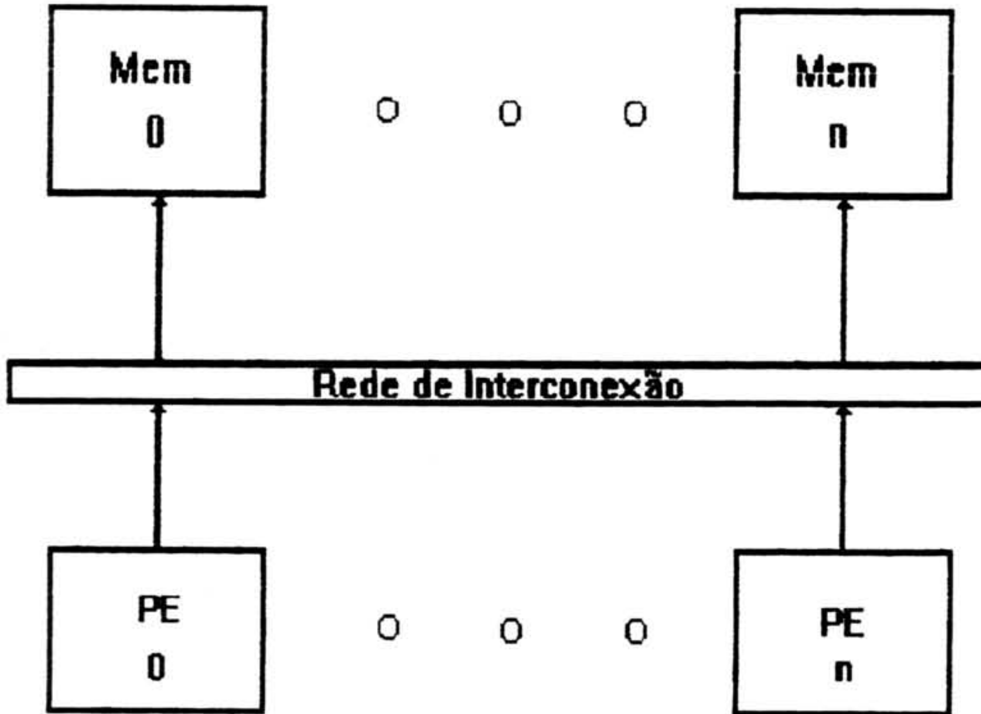


Fig 5.3 : Modelo de arquitetura com memória global partilhada

5.3 Descrição da arquitetura.

5.3.1 Arquitetura geral.

A arquitetura geral da máquina de cláusulas proposta é composta por um conjunto de blocos de PEs e memórias interligados por três barramentos e ligados ao Hospedeiro (conforme a figura 5.4).

Acesso entre blocos

Quando um **PE** necessita acessar um predicado que não esteja armazenado no seu bloco, ele endereça o seu

pedido de acesso para a interface de dados como se esta fosse um módulo de memória normal.

A interface quando recebe o pedido do **PE** monta a mensagem de pedido de acesso ao módulo de memória do outro bloco e a envia pelo barramento de dados (**BDD**) quando este estiver livre. Enquanto a interface não estiver enviando nenhum pedido de acesso a memória ela fica "ouvindo" o barramento de dados em busca de alguma mensagem endereçada aos módulos de memória ou aos **PEs** do seu bloco . Caso algum pedido seja feito ela o envia ao seu destino por intermédio da rede de interconexão. Portanto a interface de dados é usada tanto para mandar mensagens para o exterior do bloco como para receber mensagens do exterior. Desta maneira os módulos de memória "vêem" a interface como um PE e os PEs "vêem" a interface como uma memória.

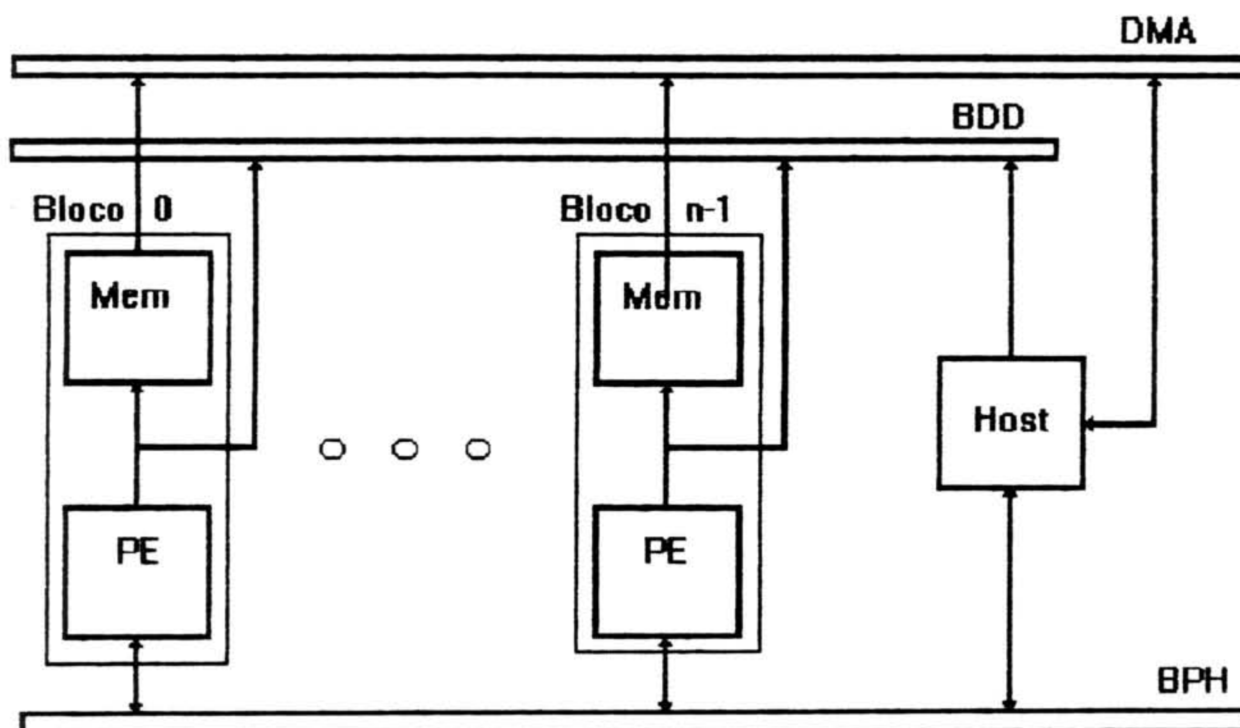


Fig 5.4 : Arquitetura geral

Acesso direto a memória.

O acesso direto a memória (DMA) é usado pelo hospedeiro como forma de comunicação com os PEs e para o partilhamento dos predicados. No início do processamento o hospedeiro partilha os predicados e envia para cada módulo de memória as informações correspondentes. Além disto são também armazenados nos módulos os dados usados para a inicialização dos PEs tais como : os predicados associados, quais são pré-condições, quais pós-condições, quais "guard clauses", o apontador para a última tupla pesquisada de cada predicado e etc...

Comunicação PE-hospedeiro.

O hospedeiro utiliza o barramento PE-Hospedeiro (BPH) para a troca de mensagens com os PEs . As mensagens possíveis são :

- Do hospedeiro para os PEs :

- Início : Ordena aos PEs que iniciem o processo de dedução. Quando o processador recebe a mensagem de início de processamento ele acessa a memória que lhe é correspondente (para i = 0 até n - 2 ; PE_i acessa MEM_i) inicializa sua memória local e inicia o processamento.

- Reinício : Ordena aos PEs que reiniciem o processo de dedução. Neste caso os processadores continuam de onde pararam.

- Fim : Ordena aos PEs que encerrem a dedução e salvem seus estados na memória.

- Dos PEs para os outros PEs e o hospedeiro.

- Inconsistência : Indica aos outros PEs e ao hospedeiro que houve inconsistência na dedução de algum predicado. Os PEs ficam lendo os seus predicados esperando

a inclusão de alguma tupla que solicite a pesquisa da inconsistência ou até que o hospedeiro mande reinicializar o processo. O hospedeiro lê a memória através do DMA e mostra ao usuário os fatos que causaram a inconsistência.

5.3.2 Arquitetura de um bloco.

Os blocos são compostos por **n - 1** PEs idênticos ligados a uma memória dividida em **n - 1** módulos e que é partilhada por todos os PEs através de **n - 1** modulos de interface (conforme figura 5.5).

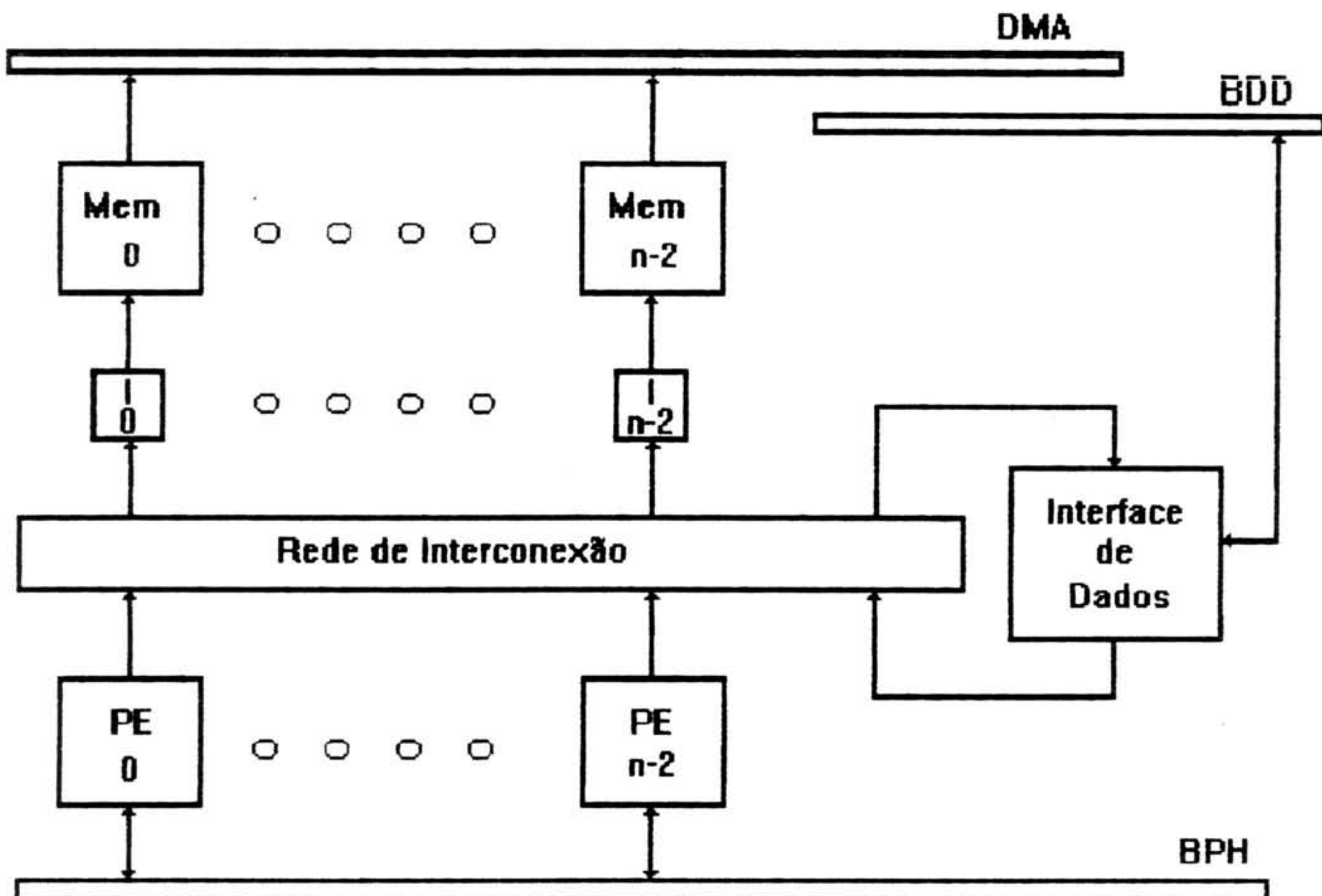


Fig 5.5 : Arquitetura de um bloco.

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

Memória.

A memória está dividida entre os vários módulos e todos os módulos são acessados por cada um dos PEs do bloco pela rede de interconexão ou por PEs de outros blocos através do Barramento De Dados (BDD). Um predicado qualquer armazenado na memória está distribuído pelos módulos em tuplas. A área ocupada por cada tupla varia de acordo com os dados nela armazenados, sendo que ela deve ser armazenada de forma contínua dentro de um módulo.

5.3.3 Estrutura de dados.

E´ muito importante para a boa performance de uma arquitetura que os dados estejam estruturados na memória de uma maneira eficiente e simples. Portanto passaremos aqui a mostrar a estrutura proposta para a nossa máquina.

Existem básicamente dois tipos de dados armazenados nos módulos de memória de cada bloco que são : descritor de predicados e tuplas. Embora estas informações estejam encadeadas numa lista, cada descritor ou tupla deve ser armazenado numa área contínua.

Bits de uso.

Todas as palavras de memória dos módulos contêm 32 bits para a informação e dois bits extras que identificam se a palavra está em uso.

B 33	B 32		
0	0	-	Palavra livre.
0	1	-	Informação foi removida mas continua em uso por algum processador.

1 0 - Palavra contém informações ativas mas não está sendo usada por nenhum processador.

1 1 - Palavra com informações ativas e em uso.

Com isto o processador que acessa a memória sabe se pode usar a palavra de memória para armazenar outra tupla ou não. (A palavra só poderá ser usada caso esteja livre (bits 33-32 = 00)).

5.3.3.1 **Predicados.**

Os descritores dos predicados são os registros iniciais de uma lista encadeada que liga todas as tuplas de um mesmo predicado. Preferencialmente cada predicado terá seu descritor armazenado num módulo diferente de memória de maneira a diminuir a concorrência aos módulos.

Cada descritor de predicado é composto por três palavras de memória contendo :

Palavra 01 :

B 31 - B 16 : Código do predicado

B 15 - B 12 : Aridade do predicado

B 11 - B 00 : Livre

Palavra 02 : Endereço da primeira tupla

B 31 - B 28 : Bloco

B 27 - B 22 : Módulo do bloco

B 21 - B 00 : Endereço no módulo

Palavra 03 : Endereço da última tupla

B 31 - B **28** : Bloco

B 27 - B 22 : Módulo do bloco

B 21 - B 00 : Endereço no módulo

Código do predicado.

O código do predicado é um número associado a cada predicado quando do partilhamento dos predicados e que identifica internamente o predicado. Isto é feito para padronizar o tamanho do nome e diminuir a área ocupada no armazenamento dos descritores.

Aridade

A aridade indica o número de argumentos do predicado. Por exemplo, o predicado **P1(X,a,[b,[c,d]])** tem aridade 3 pois possui três argumentos que são respectivamente : **X** (variável livre), **a** (constante string) e [b,[c,d]] (lista).

Endereço da primeira tupla

Esta palavra indica o endereço onde está armazenada a primeira tupla do predicado e está dividida em três campos distintos que são : bloco, módulo e endereço.

O bloco indica em qual bloco da máquina está a tupla. O módulo indica qual dos módulos dentro deste bloco e o endereço é o endereço dentro do módulo. No caso do endereço da primeira tupla o bloco será sempre o mesmo do descritor do predicado, mas esta informação foi assim mesmo incluída de forma a manter o padrão com os outros endereços.

Endereço da última tupla.

Esta palavra contém o endereço da última tupla incluída na lista do predicado. O formato é o mesmo do endereço da primeira tupla. No entanto aqui o bloco é uma informação necessária já que a tupla poderá estar armazenada em outro bloco.

5.3.3.2 Tuplas.

A estrutura de dados de uma tupla é variável e é composta por duas partes distintas que são : descritor da tupla e argumentos.

Descritor da tupla.

O descritor da tupla é composto por duas palavras contendo as seguintes informações :

Descritor da tupla:
Palavra 01 :

B 31 - B 26	:	Número de processadores acessando a tupla
B 25 - B 24	:	Valor lógico da tupla
B 23 - B 22	:	Justificativa do valor
B 21 - B 20	:	Valor lógico anterior
B 19 - B 18	:	Estado da tupla
B 17	:	Flag de variável livre
B 16 - B 07	:	Tamanho em palavras da tupla
B 06 - B 00	:	Livre

Palavra 02 : Endereço da próxima tupla

B 31 - B 28 : Bloco

B 27 - B 22 : Módulo do bloco

B 21 - B 00 : Endereço no módulo

Número de processadores.

Este campo contém o número de processadores que estão acessando a tupla. Quando este valor é zero o bit 32 deve ser setado para zero, quando diferente de zero o bit 32 deve ser setado para 1. Isto deve ser feito para todas as palavras que pertencem a tupla.

Valor lógico da tupla.

O valor lógico da tupla é composto por dois bits. o primeiro indica se o valor lógico da tupla é determinado ou não e o segundo especifica este valor.

B_{25}	B_{24}	
0	x	- Indeterminado (I)
1	x	- Determinado
1	0	- Falso (F)
1	1	- Verdadeiro (V)

Como é mostrado acima o segundo bit só tem sentido quando o primeiro for 1.

Justificativa.

A justificativa do valor da tupla também é composto por dois bits onde o primeiro indica se o valor da

tupla foi explicitado pelo usuário ou não e o segundo se o valor é exato ou se foi assumido embora restem dúvidas quanto a sua exatidão.

B 23	- B 21	
0	x	- Explicitado
0	0	- Exato (E)
0	1	- Assumido (A)
1	x	- Deduzido (D)

Valor lógico anterior.

E o valor lógico anterior da tupla. Os valores possíveis são os mesmos do valor lógico. Se a tupla estiver sendo incluída o valor lógico anterior é colocado como indeterminado.

Estado da tupla.

Os estados da tupla indicam o que está acontecendo com a tupla no momento. Os estados possíveis são :

B 19	B 18	
0	0	- Espera (E)
0	1	- Inconsistente (I)
1	x	- Em Pesquisa
1	0	- Pesquisa pelo primeiro resultado (P)
1	1	- Pesquisa todos os resultados (T)

O estado **A**, descrito no modelo de execução, é implementado tirando-se a tupla do local onde ela se encontra e colocando-a no fim da lista de tuplas do predicado. Isto fará com que a tupla seja revisitada por todos os PEs que utilizam o predicado.

Flag de variável livre.

Este campo especifica se a tupla contém variáveis livres. Ele é usado na unificação para simplificar o processo.

Tamanho da tupla.

Este campo especifica o tamanho em palavras da tupla e só pode ser alterado quando a tupla for removida e incluída em outro ponto da lista.

Endereço da próxima tupla.

Esta palavra contém o endereço da próxima tupla da lista de tuplas do predicado.

5.3.3.3 Argumento da tupla.

Os argumentos de uma tupla são representados por uma palavra de memória se forem curtos ou duas ou mais palavras se forem longos. Além disto as listas podem ocupar várias palavras.

Argumento:
Palavra 01 :

B 31 - B 28 : Tipo do argumento

B 27 - B 21 : Tamanho do argumento
B 20 - B 16 : Livre

B 15 - B 00 : Valor

Palavra 02 :

B 31 - B 00 : Continuação do valor

Tipo.

Este campo contém o tipo do argumento. Os tipos possíveis estão relacionados abaixo.

B 31	B 30	B 29	B 28	
0	0	0	0	- Variável livre
0	0	0	1	- Nil
0	0	1	0	- Livre
0	0	1	1	- Livre
0	1	0	0	- Constante string curta
0	1	0	1	- Constante real curta
0	1	1	0	- Constante inteira curta
0	1	1	1	- Constante booleana curta
1	0	0	0	- Livre
1	0	0	1	- Livre
1	0	1	0	- Livre
1	0	1	1	- Livre
1	1	0	0	- Constante string longa
1	1	0	1	- Constante real longa
1	1	1	0	- Livre
1	1	1	1	- Lista

Os valores livres permitem uma eventual inclusao de tipos novos.

Tamanho do argumento

Este campo especifica o tamanho do argumento e os valores que pode assumir são dados abaixo.

Valor

Zero - Variável livre (0000)
Nil (0001)

		Lista (1111)
1	-	Constantes curtas (01xx)
2	-	Constantes numéricas longas (110x)
1 - 128	-	Constante string longa

Valor e continuação

Estes campos contêm o valor do argumento. Se a contante for curta só o campo de valor será usado.

Exemplo de representação de uma tupla.

Vamos mostrar aqui como seria representada uma tupla de um predicado com os seus argumentos.

Tupla : <a,b,[cdef,1,[gh,i,j],k] : V,D,E>

Esta é uma tupla com três argumentos (**a**, **b** e **[cdef,1,[gh,i,j],k]**) e cujo valor lógico é verdadeiro, a justificativa é deduzido e o estado é espera. A representação na memoria ficaria conforme abaixo.

```
                 Nro  V   J   E  Tamanho
Desc Tupla :   | 0 | V | D | E |        |
               |    End. Prox Tupla     |
                   Tipo     Tamanho  Valor
Arg 1      :   |String Curto |  1   | a |   |
Arg 2      :   |String Curto |  1   | b |   |
Arg 3      :   |Lista        |  0   |       |
               |String Longo |  2   | c | d |
               |   e   |  f    |            |
               |Lista        |  0   |       |
               |Inteiro Curto|  1   |   1   |
               |Lista        |  0   |       |
               |Lista        |  0   |       |
```

String Curto	1	g	h
Lista	0		
String Curto	1	i	
Lista	0		
String Curto	1	j	
Nil	0		
Lista	0		
String Curto	1	k	
Nil	0		

Onde a lista e representada como um conjunto formado por um item seguido de uma lista. Pode-se fazer uma árvore que ilustra este tipo de representação. A lista do exemplo anterior ficaria conforme mostrado a seguir.

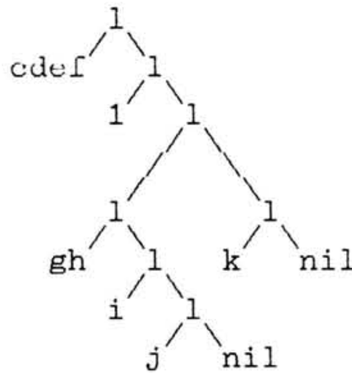


Fig 5.6 : Representação em árvore de uma lista

5.3.4 Interface de memória.

Esta unidade é a responsável pela leitura e gravação dos dados no módulo de memória. Ela executa instruções enviadas pelos PEs e foi colocada na arquitetura como forma de garantir que algumas operações sobre a memória (como por exemplo alocação de área para uma tupla) sejam feitas sem interrupção de outros processadores.

5.3.4.1 Arquitetura.

Existem quatro registradores que são usados na execução das instruçEes que são :

RI - Registrador onde é colocada a instrução que chega da rede de interconexão .

Rend - Registrador onde é colocado o endereço a ser acessado na memória.

Rdado - Registrador onde é colocado o dado lido ou a gravar na memória.

Rcont - Registrador auxiliar para contagem.

Além disto existem duas filas, uma para entrada de instruçEes e outra de saída.

5.3.4.2 Lay-out das InstruçEes.

As instruçEes que a interface de memória aceita são os seguintes : Leitura, Alocação, Remoção, Escrita, Liberação. Cada uma destas instruçEes utiliza 48 bits e pode usar uma ou mais palavras. O "lay-out" de cada palavra é o seguinte:

B 47 - B 42 - Processador (Proc)
B 41 - B 39 - Código da instrução (Cod)
B 38 - Tamanho da instrução (T)
B 37 - B 32 - Nro de PEs (Npe)
B 31 - B 00 - Complemento

A unidade devolve uma resposta para cada instrução informando o "status" da execução da instrução além de outros valores. O "lay-out" da mensagem de resposta é o seguinte :

B 47 - B 42 - Processador (Proc)
B 41 - B 39 - Código da instrução (Cod)
B 38 - Tamanho da instrução (T)

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

B 37 - "status" da execução

B 36 - B 32 - Livre

B 31 - B 00 - Complemento

A seguir são descritos os itens citados acima.

Processador.

Especifica o processador que enviou a instrução.

Código da instrução.

Especifica a instrução que deve ser executada. Os valores possíveis são os seguintes:

B 41	B 40	B 39		
0	0	0	-	Leitura e marcação de tupla
0	0	1	-	Leitura de uma palavra
0	1	0	-	Alocação de área
0	1	1	-	Gravação
1	0	0	-	Remoção de tupla
1	0	1	-	Libera tupla
1	1	0	-	Livre
1	1	1	-	Livre

Tamanho da instrução.

Este campo especifica se a instrução usa uma ou duas palavras de memória.

B 38		
0	-	Instrução Simples
1	-	Instrução Dupla

Número de PEs .

Este campo contém o número de PEs que enviaram a intrução.

"Status" da Instrução.

Este campo é usado para retornar o "status" da execucão.

B 37

0 - Execução não terminou OK
1 - Execução terminou OK

5.3.4.3 Instruções.

Aqui são descritas as instruções aceitas pela interface de memória.

Leitura e marcação de tupla.

Esta instrução lê o descritor de uma tupla e soma ao número de processadores acessando a tupla o número de PEs que enviaram a instrução. Nesta instrução o complemento tem a seguinte forma.

Complemento:

B 31 - B 22 - Não usado
B 21 - B 00 - Endereço da palavra (End)

```
Alg_Le_Marca:
    Início
    RI <- Instrução
    Rend <- RI 21-00
```

```
Rdado <- Mem(Rend)
Se Rdado 33  = 0                % Informação removida

Então : RI 37  <- 0             % Execução com erro

Senão : Início
            Rdado 31-26  <- Rdado 31-26  + RI 37-32
                            % Soma ao nro de  PEs  aces-
                            % sando a tupla o nro de PEs
                            % que enviaram a instrução.
            RI 31-00  <- Rdado

            Mem(End) <- Rdado
            RI 37  <- 1      % Execução Ok

            Fim
Retorna(RI)
Fim
```

Leitura de palavra.

Esta instrução lê uma palavra da memória. Nesta instrução o complemento tem a seguinte forma.

Complemento:

B 31 - B 22 - Não usado
B 21 - B 00 - Endereço da palavra (End)

```
Alg_Le_Pal:
     Início
     RI <- Instrução
     Rend <- RI 21-00
     Rdado <- Mem(Rend)
     RI 31-00  <- Rdado
     RI 37  <- 1                % Execução Ok
     Retorna(RI)
     Fim
```

Alocação de área.

Esta instrução lê a memória até encontrar uma área com o tamanho especificado na instrução e reserva esta área. Nesta instrução o complemento tem a seguinte forma:

Complemento:

B 31 - B 22 - Tamanho a alocar

B 21 - B 00 - Não Usado

```
Alg_Aloca :
     Início
     RI <- Instrução
     Rend <- InícioMemória       % Inicializa endereço de busca
     Rcont <- 0                  % Inicializa contador de área
                                 % contigua
     Enquanto (RCont < RI 31-22 ) e (Rend + Rcont < FimMem)
                                 % Enquanto  RCONT  menor  que
                                 % o tamanho da área à alocar e
                                 % ainda houver área na memória
          Início
          Rdado <- Mem(Rend + Rcont)
          Rcont <- Rcont + 1
          Se  Rdado 33  = 1 ou   % Testa  os bits de uso para
                                 % ver se a palavra está livre
             Rdado 32  = 1

          Então : Início         % Reinicia a busca
                     Rend <- Rend + Rcont
                     Rcont <- 0
                     Início
          Fim
     Se Rcont = RI 31-22

     Então : Início              % Aloca área encontrada
                Rcont <- 0
                Faça RI 31-22  vezes

                   Início
                   Rdado <- Mem(Rend + Rcont)
```

```
            Rdado 33  <- 1          % Liga o bit de uso

            Rcont < Rcont + 1
            Fim
        RI 37  <- 1        % Execução Ok

        Fim
Senão : RI 37  <- 0        % Execução com erro

Retorna(RI)
Fim
```

Gravação.

Esta instrução grava uma palavra de memória e devolve ao processador a informação que estava armazenada na palavra. E a única instrução que usa mais de uma palavra. Os complementos têm os seguintes "lay-outs":

Complemento 1:

B 31 - B 22 - Não usado
B 21 - B 00 - Endereço da palavra (End)

Complemento 2:

B 31 - B 00 - Conteúdo a gravar

```
Alg_Grav_Pal:
    Início
    RI <- Instrução
    Rend <- RI 21-00

    Rdado <- Mem(Rend)
    RI 31-00  <- Rdado

    Rdado <- Instrução 31-22

    Mem(Rend) <- Rdado
```

```
RI 37  <- 1                    % Execução Ok

Retorna(RI)
Fim
```

Remoção de tupla.

Esta instrução marca uma tupla como removida. A tupla só estará disponível para ser usada em nova alocação quando além de removida, todos os PEs que a estiverem usando a liberarem. A instrução devolve o valor do apontador para a próxima tupla. Nesta instrução o complemento tem a seguinte forma.

Complemento:

B 31 - B 22 - Não usado
B 21 - B 00 - Endereço da palavra (End)

```
Alg_Rem_Tupla:
    Início
    RI <- Instrução
    Rend <- RI 21-00

    Rdado <- Mem(Rend)
    Rdado 33  <- 0              % Marca tupla como removida

    Rcont  <- Rdado 17-08       % Rcont  recebe o tamanho da
                                % tupla
    Enquanto Rcont > 1
        Início                  % Marca  todas as palavras  da
                                % tupla  como removidas
        Rdado <- Mem(Rend + Rcont)
        Rdado 33  <- 0

        Rcont <- Rcont - 1
        Fim
    Rdado <- Mem(Rend + Rcont)
    Rdado 33  <- 0
```

```
RI 31-00  <- Rdado

RI 37  <- 1                    % Execução Ok

Retorna(RI)
Fim
```

Libera tupla

Decrementa o número de PEs que estão usando a tupla. Nesta instrução o complemento tem a seguinte forma.

Complemento:

B 31 - B 22 - Não usado
B 21 - B 00 - Endereço da palavra (End)

```
Alg_Lib_Tupla:
    Início
    RI <- Instrução
    Rend <- RI 21-00
    Rdado <- Mem(Rend)
    Rdado 31-26  <- Rdado 31-26  - RI 37-32
                              % Diminui do nro de PEs  aces-
                              % sando a tupla o  nro  de PEs
                              % que enviaram a instrução
    Se Rdado 31-26  = 0       %Se todos já liberaram a tupla

    Então : Início            % Marca a tupla como fora  de
                              % uso
            RI 32  <- 0

            Rcont <- Rdado 17-08

            Enquanto Rcont > 0
                Início
                Rdado <- Mem(Rend + Rcont)
                Rdado 32  <- 0
                Rcont <- Rcont - 1
```

```
            Fim
        Fim
RI 37  <- 1                  % Execução Ok

Retorna(RI)
Fim
```

5.3.5 Processador.

O processador (PE) é o responsavel pela execução dos processos de propagação dos valores de uma tupla e pela pesquisa de tupla descritos no modelo de execução. A principal função para a execução destes procedimentos é a unificação. Logo o processador aqui descrito é uma máquina que basicamente controla o processo de unificação.

5.3.5.1 Arquitetura do processador.

A arquitetura básica do processador é dada pela figura 5.7.

Registradores

Os registradores do processador são :

- **Mar** : Registrador endereçador da memória local
- **Mbr** : Registrador de dados da memória local
- **Aux** : Registrador de saída da ULA.
- **Aux1** : Registrador de entrada da ULA.
- **Aux2** : Registrador de entrada da ULA.
- **Status** : Registrador de "status" da ULA.
- **RI** : Registrador onde é montada a instrução a enviada para a rede de interconexão.
- **RD** : Registrador de dados que recebe as respostas às instruçEes.

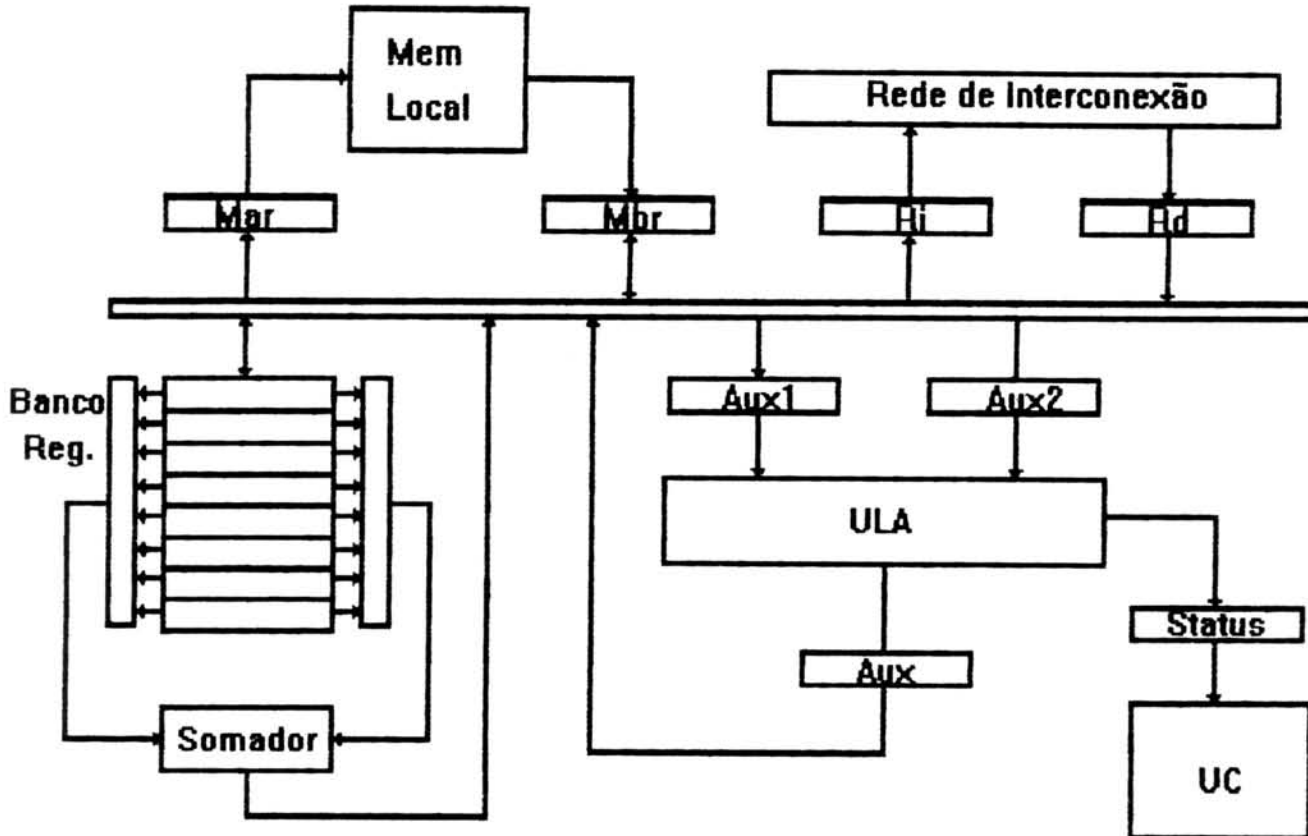


Fig 5.7 : Arquitetura do processador

Bloco de registradores.

São registradores de uso geral que são usados pela máquina. Estão ligados diretamente a um somador o que lhes permite serem usados como índices sem que tenham que ser levados a ULA.

Memória local.

A memória local é usada para armazenar as informações referentes aos predicados que estão associados ao processador. Está memória utiliza palavras de 32 bits e deve ter uma capacidade de 0.5K words (512 x 32 bits). Isto advém da necessidade da armazenamento da estrutura de dados.

Unidade Lógica e Aritmética

Uma das caracteristicas que diferencia este processador de uma máquina comum é a capacidade da ULA (Unidade Lógica e aritmética) . Esta deve ser capaz não só de realizar cálculos e operacões lógicas mas principalmente de fazer a comparação da unificaçõ .

Dois dados podem ser unificados em um dos casos abaixo :

- Se forem dados simples e do mesmo tipo e contiverem o mesmo valor.
- se forem estruturados e do mesmo tipo
- se um dos dados for uma variável livre.

Somador.

O somador é uma pequena ULA que faz as operações de soma e subtração entre registradores. Ela foi incluida na arquitetura para possibilitar o uso dos registradores como indices sem necessidade de levar os registradores a ULA. Com isto reduz-se o tráfego de registradores pelo barramento e diminui-se o tempo de processamento.

5.3.5.2 Fila de tuplas à processar.

No modelo de execução proposto anteriormente

existe um estado da tupla (estado **a**) que não foi inserido explicitamente na estrutura de dados da memória global. Em vez disto ele é simulado pelo processador através da criação de uma fila de tuplas a processar.

No modelo, este estado foi definido como sendo o estado na qual uma tupla, tendo sido alterada ou incluída, não foi ainda processada por todos os PEs que a utilizam.

Numa implementação mais comum, as tuplas teriam um contador que controlaria o número de predicados que ainda não tivessem processado a nova alteração . Isto controlaria o estado da tupla, mas teria o inconveniente de obrigar que os PEs trocassem mensagens entre si avisando as novas alterações além de criar mais uma tarefa para os PEs que seria o tratamento da fila de mensagens.

Com o uso do apontador para a próxima tupla à processar criou-se uma divisão entre as tuplas de um predicado. As tuplas anteriores ao apontador são as que já foram processadas e as posteriores as que ainda não o foram. Tuplas que tiverem o seu valor alterado são removidas do seu local na lista e incluídas no fim da lista. Com isto cria-se um pequeno aumento no acesso a memória mas permite-se que cada PE cuide apenas do seu processamento, sem troca de mensagens com os outros PEs o que está mais de acordo com o modelo e simplifica bastante o fluxo de mensagens do sistema.

5.3.5.3 Estrutura dos dados da memória local.

A estrutura dos dados na memória local é outra característica importante deste processador. E esta estrutura que permite que o PE realize as funções do modelo lógico apresentado anteriormente.

Os dados estão armazenados por predicado sendo que cada predicado utiliza no mínimo 7 e no máximo 64 palavras

de memória. O número máximo de predicados que uma cláusula pode conter é de 16.

A seguir são mostradas as palavras com o seu conteúdo correspondente.

Palavra 1 :

B 31 - B 16 : Código do Predicado
B 15 - B 12 : Aridade
B 11 - B 09 : Tipo
B 08 - B 03 : Tamanho da àrea do predicado
B 02 - B 00 : Livre

Palavra 2 :

B 31 - B 00 : Endereço do descritor do Predicado

Palavra 3 :

B 31 - B 00 : Apontador para a próxima tupla a ser processada.

Palavra 4 :

B 31 - B 00 : Endereço da tupla atual.

Palavra 5 :

B 31 - B 00 : Buffer do descritor de tuplas.

Palavra 6 :

B 31 - B 00 : Buffer de leitura.

Palavras 7 : Número variável de palavras.

B 31 - B 28 : Número do argumento
B 27 - B 24 : Número do argumento à unificar
B 23 - B 15 ¦ Endereço do Predicado
B 14 ¦ Flag de unificação
B 13 - B 10 : Número do argumento à unificar
B 09 - B 01 ¦ Endereço do Predicado
B 00 ¦ Flag de unificação

Palavra 1

Esta palavra contêm informações que descrevem o predicado e a área que ocupará na memoria local.

Palavra 2.

Esta palavra contêm o endereço do registro descritor do predicado na memória.

Palavra 3.

Esta palavra contêm o apontador para a próxima tupla a ser processada da fila de tuplas a processar.

Palavra 4

Esta palavra endereça a tupla que está sendo processada no predicado.

Palavra 5.

Esta palavra guarda o descritor da tupla lido da memória global.

Palavra 6.

Esta palavra guarda a última palavra lida da memória global.

Palavra 7 em diante.

Estas palavras guardam as informações dos arcos da cláusula, ou seja a associação entre os argumentos dos predicados. O primeiro valor é o número do argumento do

predicado e após vem o número do argumento de outro predicado que está relacionado com ele, o endereço deste predicado e um "flag" que indica se a unificação foi OK.

5.3.6 Rede de interconexão.

Em sistemas multiprocessadores com memória partilhada acessos a variáveis partilhadas ("hot spot") criam uma contenção em alguns módulos de memória. Embora estes acessos sejam uma pequena percentagem do número total de acessos (na maioria das aplicações menos de 10 % [YEW87]) a contenção de memória pode criar um fenômeno chamado de "tree saturation" o que causa uma severa diminuição no tráfego da rede de interconexão. Este problema é o principal fator limitante da maioria dos sistemas multiprocessadores com memória partilhada.

No caso do sistema proposto, este problema é ainda mais grave que na maioria dos casos, já que o acesso simultâneo dos processadores aos predicados em busca de inclusão (ou alteração) de tuplas faz esperar um número maior de "hot spots" do que os 10 % acima citados.

Para solucionar o problema de saturação da rede na presença dos "hot spots" foi proposto [GOT83] um esquema chamado de combinação de mensagens, onde a idéia básica é a incorporação de um "hardware" extra a cada nodo da rede de interconexão com o intuito de combinar as requisicões de acesso a uma mesma variável num mesmo módulo de memória. Embora este esquema requeira um custo bem elevado (estima-se um acréscimo de 6 a 32 vezes no custo da rede de interconexão [PFI85b]) e além disto aumente o tempo de tramitação de pedidos de acesso normais devido ao "hardware" extra, o aumento de performance ganho na maioria dos casos justifica o uso do esquema.

5.3.6.1 Combinação.

A combinação de mensagens funciona através da detecção de pedidos de acesso a memória para uma mesma variável quando da passagem destas mensagens por um nodo da rede de interconexão. Estas mensagens são combinadas no nodo de chaveamento numa única mensagem que é posteriormente passada para o próximo nodo da rede. Quando a resposta da memória retorna ao módulo esta é retransmitida para todos os processadores que fizeram a requisição. Desde que requisições já combinadas podem ser recombinadas, a rede satisfaz a propriedade de que qualquer número de referências concorrentes de memória podem ser satisfeitas no tempo requerido para apenas um acesso.

No entanto em [GIN85] prova-se através de testes em redes com combinação que este esquema só é eficiente para redes pequenas (menos de 9 níveis de chaveamento) o que limita o número de módulos de memória e processadores que podem ser ligados à rede. No caso da arquitetura aqui proposta utilizou-se uma rede com seis níveis de chaveamento (Pfister em [PFI85b] provou através de testes numa rede de seis níveis que para este caso a combinação mostrou-se eficiente).

5.3.6.2 Geometria da rede de interconexão.

A técnica da combinação de mensagens pode ser implementada em qualquer geometria que se deseje. No entanto é importante notar que quanto maior for o número de portas de cada módulo mais filas de combinação serão necessárias e mais complexo será o controle interno de cada módulo.

A geometria escolhida para a rede de interconexão proposta foi a Omega com nodos de 2x2 (Figura 5.8). Esta

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

escolha se deveu a simplicidade da sua geometria e porque com a "pipelinização" da rede, o fator limitador da rede Omega, que era a impossibilidade de que mais de um processador utilizar um mesmo nodo como caminho simultaneamente, não existe mais.

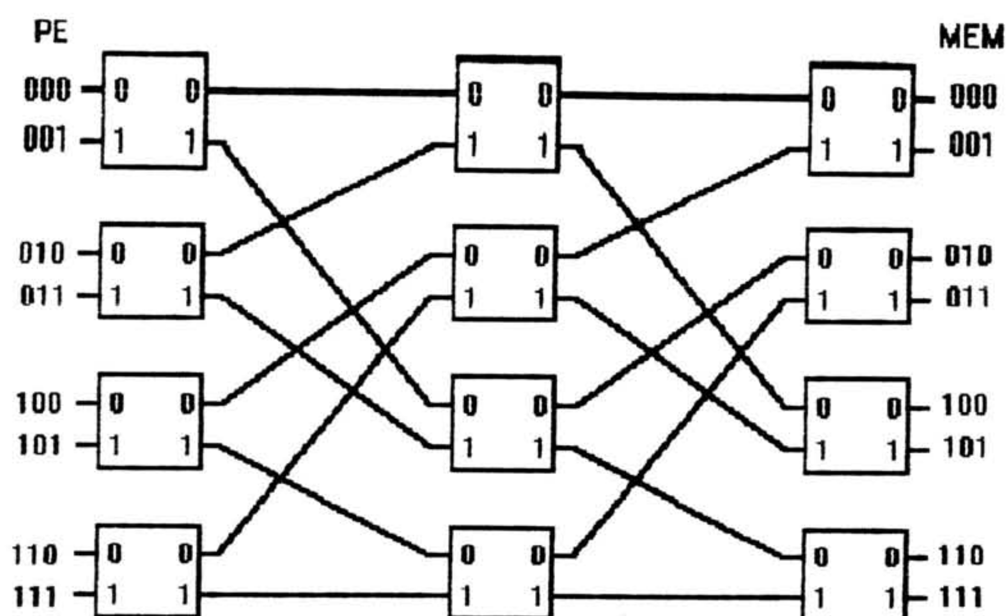


Fig 5.8 : Geometria da rede de interconexão

Características

As principais características das redes com geometria Omega são :

- Largura de banda linear ao número de PEs .
- Tempo de latência igual a $Log_2 N$.
- $N \, Log_2 N$ chaves idênticas.
- A decisão de roteamento em cada chave.

Roteamento numa rede Omega.

A técnica usada para fazer o roteamento numa rede Omega é bem conhecido e está descrita abaixo.

Dada uma rede que permita conectar **n** PEs com **n** módulos de memória onde :

- PEs e Mem´s são numerados de 0 a n - 1 em binário.
- Nas chaves as portas superiores são numeradas com 0 e as inferiores com 1.
- Mensagens de PEs para Mem´s transitam da esquerda para direita e mensagens de Mem´s para PEs no sentido inverso.
- Existem $k = \mathrm{Log}_2 N$ níveis de chaveamento.

A mensagem é transmitida do PE de endereco $(p_k \ldots p_1)$ para a memória de endereço $(m_k \ldots m_1)$ através da porta m_i quando sair da chave de nível **i** e pela porta p_i quando transmitida da memória para o PE.

Características adicionais.

Algumas características devem ser adicionadas numa rede Omega de forma a permitir a combinação de mensagens.

- A rede é "pipelinizada" ou seja o tempo de transmissão das mensagens pelos PEs é igual ao ciclo de um nodo da rede.
- A rede é chaveada pela própria mensagem o que permite que o chaveamento seja descartado enquanto uma resposta é esperada.
- E acrescentado um "hardware" extra ao nodo para fazer a combinação das mensagens.

Chaveamento das mensagens.

Desde que se proponha a utilizar o chaveamento pela própria mensagem pode parecer que tanto o endereço destino como o de retorno devem ser transmitidos junto com a mensagem.

No entanto isto pode ser evitado usando uma técnica bastante simples. Quando uma mensagem entra na rede, o seu destino é determinado pelas portas de saída das chaves, ou seja os bits endereçadores da memória $(m_k \ldots m_1)$. Após ser feito o roteamento dentro de uma chave de nível **i** baseado no bit m_i do endereço este bit deve ser substituido pelo número da porta pela qual a mensagem chegou (p_i). Com isto quando a mensagem chegar ao destino o endereço da mensagem será o do PE que a originou.

5.3.6.3 Descrição de um nodo da rede.

O nodo utilizado na rede aqui proposta é bidirecional e é dividido internamente em dois nodos distintos : um na direção PE-Mem, que faz a combinação das mensagens e posterior transmissão ao próximo nível e o outro na direção Mem-PE, que recebe as respostas divide-as e as retransmite para os processadores que fizeram a requisição de acesso a memória. A arquitetura interna de cada nodo é mostrada na figura 5.9.

Subnodo PE-Mem

Este nodo tem as seguintes unidades principais :

- Quatro "buffers" de entrada (dois em cada entrada) : Cada um destes "buffers" e capaz de armazenar uma mensagem de acesso a memória (48 bits). Eles são usados para

reter mensagens cujas filas de saída estejam cheias. Isto evita que o nodo seja obrigado a assinalar para o nodo anterior quando uma das filas tiver menos de uma vaga livre. São usados dois "buffers" em cada saída porque a mensagem de gravação utiliza duas palavras de instrução.

- Dois "buffers" de entrada das filas de saída : Estes "buffers" são usados quando do chaveamento da saída da mensagem para direcionar a mensagem a fila de saída adequada.

- Duas filas de saída : Estas filas são usadas para armazenar as mensagens que não puderem ser transmitidas à saída diretamente, no caso do próximo nodo estar com sua fila completa. Cada fila tem capacidade de armazenar duas mensagens a cada ciclo de execução da rede já que o nodo pode receber nas suas entradas duas mensagens para a mesma saída. A combinação das mensagens é feita quando estas estão nas filas de saída o que faz com que mensagens que não entrarem na fila não serão combinadas.

Subnodo Men-PE.

Este nodo tem as seguintes unidades principais:

- Dois buffers de entrada : Cada "buffer" com capacidade de armazenar uma resposta à requisição de acesso a memória.

- Um "buffer" de espera : Este "buffer" contém as informações sobre as mensagens combinadas de forma que quando a resposta chegar esta possa ser duplicada e mandada para os endereços correspondentes.

- Dois "buffers" de entrada das filas de saída : Estes "buffers" são usados quando do chaveamento da saída da mensagem para direcionar a mensagem a fila de saída adequada.

- Duas filas de saída : Estas filas são

usadas para armazenar as mensagens que não puderem ser transmitidas á saída diretamente, no caso do próximo nodo estar com a sua fila completa. Cada fila tem capacidade de armazenar quatro mensagens a cada ciclo de execução da rede já que o nodo pode receber nas suas entradas duas mensagens para a mesma saída e cada uma delas pode ser duplicada.

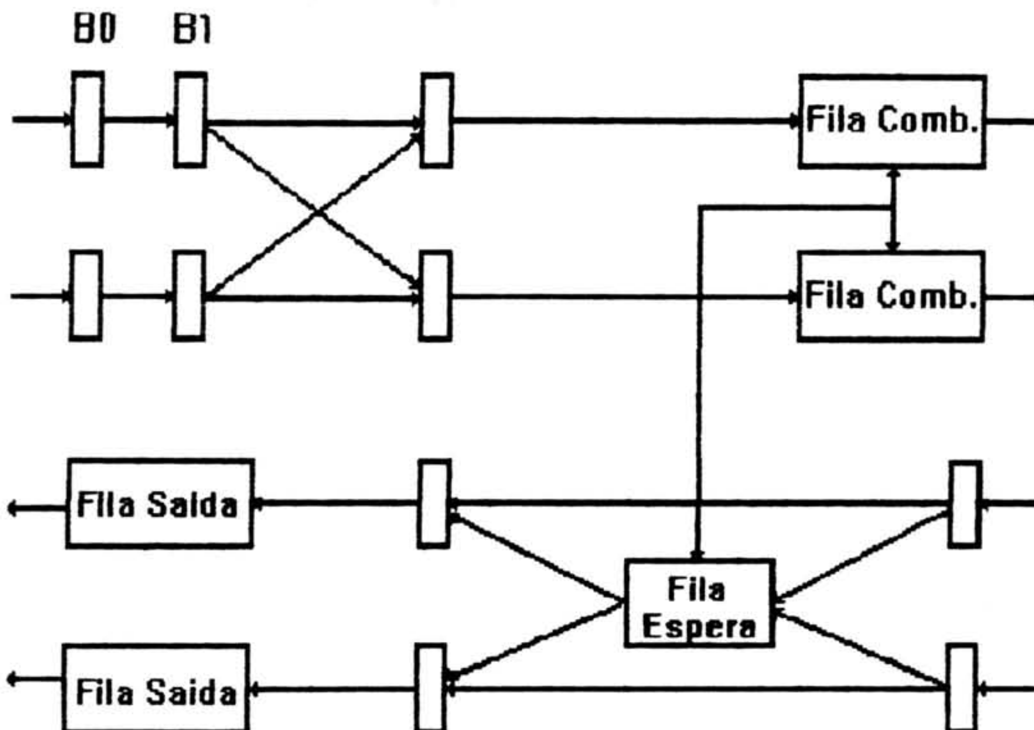


Fig 5.9 : Arquitetura de um nodo da rede de interconexão.

Características do nodo

E importante salientar também algumas características dos nodos.

- As filas de saída são utilizadas apenas quando o próximo estágio da rede sinalizar indicando que não pode receber mais mensagens.

- Combinação só ocorre na fila de saída do subnodo PE-Mem, portanto nenhuma combinação ocorrerá se o tráfego

pela rede for suficientemente baixo.

- Com o uso dos "buffers" de entrada o nodo só precisa sinalizar para o nodo anterior não enviar mais mensagens quando alguma das filas de saída estiver cheia e já existir uma mensagem no "buffer" de entrada para aquela fila.

Fila de Combinação de mensagens.

Cada fila de combinação de mensagens, conforme proposto por [GOT87] é formada por três filas interligadas.(Figura 5.10). Na fila mais a direita estão as mensagens que estão esperando para sair, na fila mais a esquerda são colocadas as mensagens que foram combinadas e a fila central é usada para fazer a combinação .

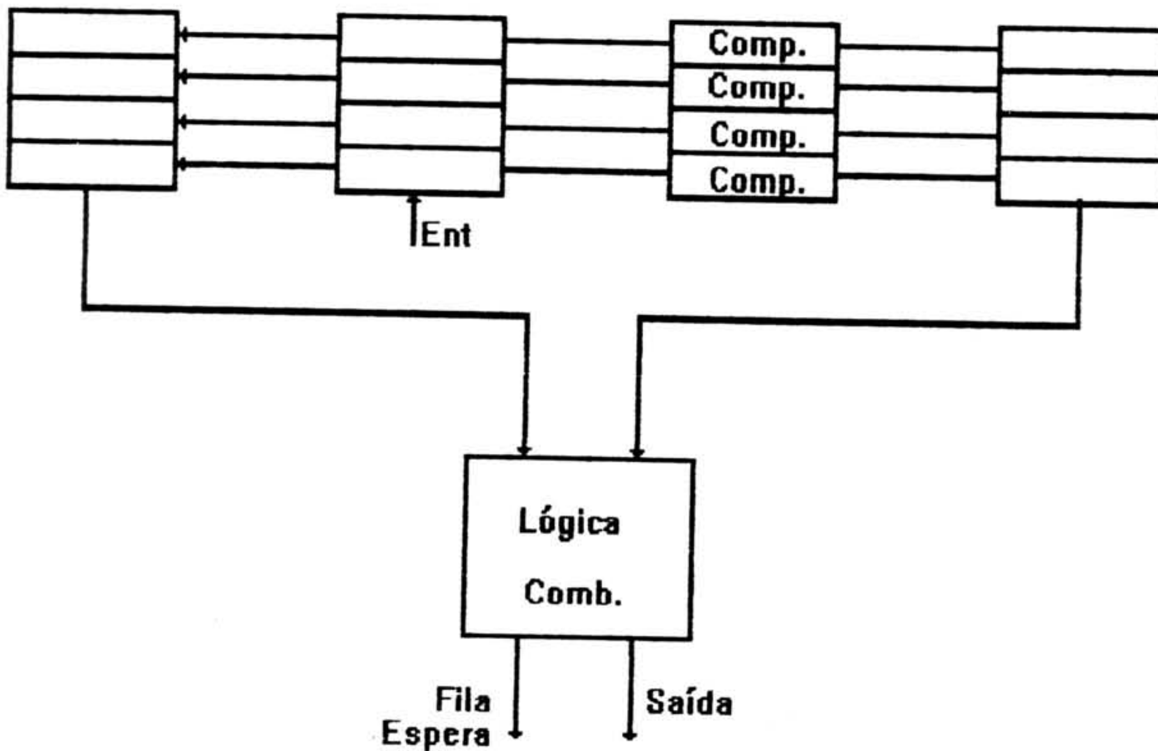


Fig 5.10 : Arquitetura da fila de combinação.

Quando uma mensagem é colocada na fila de combinação ela é inserida na primeira posição da fila central. A seguir é feito um teste para verificar se o elemento de mesma posição na fila da direita está vazio, caso isto seja verdadeiro a mensagem é colocada na fila da direita na posição correspondente a que se encontrava. Caso isto não seja possível é feito uma comparação entre os dois elementos para ver se são possíveis de combinar. Se a combinação for possível então o elemento da fila central é colocado na fila da esquerda na posição correspondente. Se nenhum dos testes for verdadeiro então avança a mensagem para a próxima posição da fila central e recomeça o processo.

A combinação propriamente dita ocorrerá quando as duas mensagens estiverem saindo da fila para um próximo nodo. Neste momento é feita a combinação da mensagem que vem da fila da esquerda com a mensagem que vem da fila a direita e armazenando na fila de espera os endereços de retorno.

Fila de espera.

A fila de espera contém os endereços de retorno das requisições combinadas. Quando uma resposta chega o seu endereço de retorno é comparado com o endereço armazenado na fila de espera. Se o valor coincidir é feita a duplicação do valor e colocado nas filas de saída para os PEs .

5.3.6.4 Combinações possíveis.

Nem todas as requisições de acesso a uma mesma variável podem ser combinadas e as que podem geram respostas muitas vezes diferentes. A seguir serão mostradas todas as possibilidades de combinação de mensagens com as suas respectivas consequências sobre as respostas geradas.

Leitura tupla X Leitura tupla

Neste caso existem dois pedidos de leitura de uma tupla. A combinação é feita e apenas um dos pedidos é enviado mas este tem o campo, que especifica o número de PEs que enviaram a requisião, somado com o mesmo campo do outro pedido. O endereço do outro pedido é colocado na fila de espera junto com o código da operação.

Leitura tupla X Remoção

Neste caso foi pedido a leitura de uma tupla que está sendo removida. Apenas o pedido de remoção é enviado enquanto o de leitura é devolvido com o bit de "status" zerado (instrução não funcionou ok).

Leitura de tupla X Liberação

Neste caso alguns PEs estão querendo ler e consequentemente marcar a tupla como estando em uso enquanto outros PEs querem liberar a tupla da sua marcação. Portanto apenas o pedido de leitura é enviado, mas com o campo que especifica o número de PEs que enviaram o pedido diminuido do mesmo campo do pedido de liberação. Mesmo que este valor se torne negativo (mais PEs estão liberando a tupla do que PEs estão marcando) quando for feito a soma deste valor com o campo da tupla o resultado sempre será maior que zero.

Leitura palavra X Leitura palavra.

Neste caso existe mais de um PE querendo ler uma mesma palavra de memória . Com isto apenas um pedido é enviado enquanto o outro é colocado na fila de espera.

5.3.7 Predicados Pré-definidos.

A linguagem Prolog utiliza predicados pré-definidos para a implementação das funções de interface com o usuário (entrada e saída). Estes predicados além de executarem a regra da resolução (unificação e substituição) executam comandos de entrada e saída que não estão definidos no modelo da Programação em Lógica.

Da mesma maneira o modelo de execução proposto não preve comandos de entrada e saída de dados, portanto estes comandos são implementados a partir de predicados pré-definidos.

O procedimento a ser adotado quando da execução de um predicado é enviá-lo para o endereço associado ao mesmo. Este endereço indicará o bloco de I/O. Isto quer dizer que para o PE ele estará endereçando uma memória mas na verdade o endereço será o de um processador de I/O ou o endereço do próprio hospedeiro no caso de não haver um processador específico para I/O.

5.4 Conclusão.

Neste capítulo foi descrita a arquitetura proposta para o modelo de execução de cláusulas Prolog. A arquitetura resultante foi a de um multicomputador dividido em blocos e cada bloco dividido em varios módulos de memória e processadores.

As principais características do modelo foram conservadas e o principal gargalo do sistema que é o acesso a memória foi resolvido usando-se a técnica de combinação de mensagens.

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

6 VALIDAÇÃO.

O modelo proposto é baseado em dois algorítmos básicos : a propagação de valores e a pesquisa de tupla. O primeiro implementa uma execução do tipo "data driven" e o segundo uma execução "demand driven". No entanto ambos os algorítmos baseiam-se principalmente no processo de unificação . Em [WO085] estima-se que as operações de unificação gastem de 55 a 75 % do tempo total de processamento em execucões de programas de linguagens lógicas. Isto indica que o processo de unificação é um bom parametro para medirmos o tempo de processamento do nosso sistema.

Dadas as seguintes tuplas, P1(a,[b,cd,X,e]) e P2([b,Y,[a,d,f],e],c) desejava-se unificar o primeiro argumento do predicado 1 com o segundo argumento do predicado 2 . Usando o algorítmo de unificação (Anexo 1) sobre o exemplo acima chegou-se ao resultado que apenas 2 % das instruções realizadas são instruções de leitura na memória global. Deve-se juntar a estas as intruções de gravação e teremos aproximadamente 3 % das operações do processador acessando a memória global.

No entanto, o processador envia as intruções de acesso em 6 ciclos (48 bits 8 a 8) logo o ciclo de um nodo da rede é 6 vezes mais lento que o do processador. Com isto podemos dizer que o processador envia em média 0,18 comandos de memória por ciclo do nodo.

Com esta informação pode-se fazer uma comparação entre um único processador e vários e de que forma a inclusão de novos processadores alterará o desempenho da máquina.

Largura de banda do sistema.

Dados : **N** - o número de processadores do sistema

r - o número de mensagens enviadas pelos processadores por ciclo da rede de interconexão ($0 \le r \le 1$).

h - o percentual de requisições para uma mesma variável .(hot spot).

Então para cada ciclo haverão **Nrh** requisições que causarão colisão no sistema e **r(1 - h)** requisições para o módulo de memória para o qual estão endereçadas as colisões. Portanto o total de requisições que são feitas para um módulo de memória onde houve colisão é dado pela expressão:

$$R = Nrh + r(1 - h)$$

O valor de **r**, conforme citado, foi estimado através do algorítmo de unificação em 0,18 (18 %). Com isto pode-se calcular uma tabela com o número de acessos requisitados para um módulo para vários valores de h e N. A tabela abaixo mostra a tabulação destes valores.

Tab 6.1 : Valores de R para N e h variados

N / h	10	50	100	500
0.05	0.26	0.62	1.07	4.67
0.10	0.34	1.06	1.96	9.16
0.15	0.42	1.50	2.85	13.65
0.20	0.50	1.94	3.74	18.14
0.25	0.58	2.38	4.63	21.63
0.30	0.66	2.82	5.52	27.12
0.35	0.74	3.26	6.41	31.61
0.40	0.82	3.70	7.30	36.10
0.45	0.90	4.14	8.19	40.50
0.50	0.99	4.59	9.09	45.09

Se cada modulo pode tratar de 1 requisição (valor máximo) por ciclo da rede o máximo "throughput" atingido por cada processador é dado por :

$$T = 1 / (Nrh + r(1 - h)$$

E a largura de banda máxima para o sistema é dado por :

$$L = N / (Nrh + r(1 - h)$$

A tabela 6.2 mostra o valor de L para vários valores de N e h.

Tab 6.2 : Valores de L para N e h váriados

N h	10	50	100	500
0.05	38.4	80.6	93.4	107.0
0.10	29,4	47.1	51.0	54.5
0.15	23.8	33.3	35.0	36.6
0.20	20.0	25.7	26.7	27.5
0.25	17.2	21.0	21.5	22.0
0.30	15.1	17.7	18.1	18.4
0.35	13.5	15.3	15.6	15.8
0.40	12.1	13.5	13.6	13.8
0.45	11.1	12.0	12.2	12.3
0.50	10.1	10.8	11.0	11.0

Como pode ser visto na tabela acima a largura de banda do sistema aumenta muito pouco acima dos 100 processadores. A arquitetura proposta utiliza 64 processadores para um h estimado em torno de 25 %. Isto da uma largura de banda razoável ao sistema sem o uso de muitos processadores. O número de processadores não poderia ser muito pequeno pois isto acarretaria num acréscimo muito

grande de acessos a outros blocos o que tornaria o sistema muito lento.

Os valores da tabela 6.2 maiores que o respectivo **N** indicam que a memoria aguentaria um maior numero de PEs .

Com estes dados podemos comparar um PE com N PEs .

O tempo total de execução de um processador é dado então por :

$$T_t = T_p + T_m$$

Onde T_p é o tempo total de processamento e T_m é o tempo total de leitura da memória. O tempo de comunicação entre Processadores não existe já que eles não se comunicam assim como o tempo de tramitação pela rede de interconexão também é nulo já que a rede é "pipelinizada".

Com isto para um PE que execute **m** unificações o tempo é dado por :

$$T_1 = m\,(T_p + T_m)$$

$$T_p = K1 * C_p$$

$$T_m = K2 * C_m$$

Onde K1 e o numero de comandos do processador

C_p é o tempo de ciclo do processador

K2 e o numero de comandos de acesso a memoria

C_m é o tempo de ciclo da memória

Além disto podemos dizer que o tempo de ciclo da memória é K3 vezes o ciclo do processador . Com isto temos que o T_1 é dado por:

$$T_1 = m(K1\ C_p + K2\ K3\ C_p)$$

e o tempo médio de uma unificaçãO vai ser dado por:

$$T_1/m = ((K1 + K2) / m) (0,82\ C_c + 0,18\ K3\ C_p)$$

$$= ((K1 + K2) / m) (0,82 + 0,18\ K3)\ C_p \qquad [1]$$

Podemos agora fazer o mesmo cálculo para N processadores e chegaremos a:

$$T_N /m = ((k1 + K2) / m) (0,82/N + 0,18*K3/L) * C_p \qquad [2]$$

Onde L <= N

Dividindo agora 1 por 2 nos termos a relação de desempenho entre um PE e N PEs e as variáveis K1, K2 e m são eliminadas. No entanto K3 ainda continua e para podermos fazer a tabela devemos arbitrar um valor para K3. A tabela abaixo foi construída para K3 = 10.

Tab 6.3 : Quantas vezes o sistema é mais rápido do que um único PE, para vários valores de h e N.

N / h	10	50	100	500
0.05	10.0	50.0	95.3	142.2
0.10	10.0	48.3	60.2	75.6
0.15	10.0	37.3	45.1	51.5
0.20	10.0	30.5	34,6	39.1
0.25	10.0	25.7	28.6	31.5
0.30	10.0	22.2	24.3	26.3
0.35	10.0	19.6	21.2	22.6
0.40	10.0	17.5	18.6	19.8
0.45	10.0	15.8	16.8	17.7
0.50	10.0	14.3	15.2	15.8

7 CONCLUSÃO

O presente trabalho objetivou definir um modelo de execução para cláusulas Prolog usando Redes de Petri e posteriormente criar uma arquitetura capaz de suportar o modelo. O modelo proposto foi capaz de explorar muito do paralelismo inerente a Programação em Lógica principalmente o paralelismo AND que é o que tem causado maiores dificuldades nas implementações até aqui propostas. Isto levou a uma arquitetura que permitiu que os processadores executassem as suas operações de forma totalmente independente eliminando assim o tempo de comunicação entre os processadores. A maior dificuldade na arquitetura foi o gargalo de memória já que vários processadores usam um mesmo predicado simultaneamente. A solução encontrada foi a utilizacão de uma rede de interconexão capaz de fazer a combinação de mensagens . Isto conseguiu diminuir sobremaneira o tráfego na rede conforme mostrado na validação. Um ponto importante foi a estrutura de dados utilizada que permitiu um fácil acesso aos dados . Foi também usada uma interface de memória o que permitiu que alguns acessos a memória fossem feitos initerruptamente o que eliminou o problema de controle das áreas de memória.

Como continuação do trabalho poderia-se estudar o uso de memória associativas para a unificação de predicados, o partihamento do controle em dois permitindo que a unidade de unificação fique junto a memória o que diminuiria bastante o fluxo de informações através da rede de interconexão e o software necessário para suportar a arquitetura.

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

Anexo - Algorítmo de Unificação.

Este capítulo mostra o algorítmo para a unificação de dois argumentos na máquina proposta.

Definições

Aqui são mostradas os registradores e variáveis usadas no algorítmo.

PreX - Registrador que contém o endereço (na memória local) do predicado X que se deseja unificar .

ArgX - Registrador usado como variável de controle para achar o argumento desejado do predicado X.

AruX - Registrador que contém o número do argumento do predicado X à unificar.

EndX - Registrador usado para armazenar o endereço que se deseja acessar na memória global para o predicado X.

ConX - Registrador usado para percorrer os itens.

BFP - Constante que indica o deslocamento na estrutura de dados na memória local entre o descritor do predicado e o apontador para o buffer de palavra.(BFP = 5).

ATU - Constante que indica o deslocamento na estrutura de dados na memória local entre o descritor do predicado e o apontador para a tupla atual.(ATU = 3).

CLP - Constante para a instrução de leitura de palavra (indica o código(001), o tamanho da instrução (0) e o número de PEs (000001)).

Lst - Código de tipo de variável lista (Lst = 1111).

Val - Código de variável livre (Val = 0000).

Ctd(X) - Função que endereça os bits de 31 a 00 de X.

Mod(X) - Função que endereça os bits de 27 a 22 de X.

Mem(X) - Função que endereça os bits de 41 a 32 de X.

Tam(X) - Função que endereça os bits de 27 a 21 de X.

Quando o algorítmo inicia os registradores Pre1, Pre2, Aru1 e Aru2 já vem inicializados.

Alg_Unif.

```
¦ Bloco que inicializa o predicado 1

        Arg1 <- 1

        Mar  <- Pre1 + Atu
        Le Mem Local
        End1 <- Mbr

        End1 <- End1 + 2

¦ Bloco que inicializa o predicado 2

        Arg2 <- 1

        Mar  <- Pre2 + Atu
        Le Mem Local
        End2 <- Mbr

        End2 <- End2 + 2

¦ Bloco que percore os argumentos do predicado 1 até achar o
¦ que deve ser unificadob

     1: Aux1 <- Arg1
        Aux2 <- Aru1
        If Aux1 = Aux2 Goto 6
        Con1 <- 1
     2: Mem(RI) <- Mod(End1)
        Cod(RI) <- CLP
        Ctd(RI) <- End1

        Le Mem Global
```

```
       Mar <- Pre1 + BFP
       Mbr <- Ctd(RD)
       Grava Mem Local

       Con1 <- Con1 + 1

       Aux1 <- Ctd(RD)
       If Tam(Rd) = 0 Goto 3

       Aux2 <- End1
       Aux  <- Aux1 + Tam(Aux2)
       End1 <- Aux
       Goto 5

    3: If Tip(Aux1) <> Lst Goto 4
       Con1 <- Con1 + 2

    4: End1 <- End1 + 1

    5: Aux1 <- Con1
       If Aux1 <> 0 Goto 2

       Arg1 <- Arg1 + 1
       Goto 1

; Bloco que percore os argumentos do predicado 2 até achar o
; que deve ser unificadob
    6: Aux1 <- Arg2
       Aux2 <- Aru2
       If Aux1 = Aux2 Goto 11

       Con2 <- 1

    7: Mem(RI) <- Mod(End2)
       Cod(RI) <- CLP
       Ctd(RI) <- End2

       Le Mem Global
```

```
      Mar <- Pre2 + BFP
      Mbr <- Ctd(RD)
      Grava Mem Local

      Con2 <- Con2 + 1

      Aux1 <- Ctd(RD)
      If Tam(Rd) = 0 Goto 8

      Aux2 <- End2
      Aux  <- Aux1 + Tam(Aux2)
      End2 <- Aux
      Goto 10

   8: If Tip(Aux1) <> Lst Goto 9
      Con2 <- Con2 + 2

   9: End2 <- End2 + 1

  10: Aux1 <- Con2
      If Aux1 <> 0 Goto 7

      Arg2 <- Arg2 + 1
      Goto 6

; Bloco que lê a palavra dos argumentos a unificar

  11: Con1 <- 1
      Con2 <- 1

  12: Mem(RI) <- Mod(End1)
      Cod(RI) <- CLP
      Ctd(RI) <- End1

      Le Mem Global

      Mar <- Pre1 + BFP
```

```
     Mbr <- Ctd(RD)
     Grava Mem Local

     Con1 <- Con1 + 1

     End1 <- End1 + 1

     Aux1 <- Ctd(RD)
     If Tipo(Aux1) <> Lst Goto 13
     Con1 <- Con1 + 1

 13: Mem(RI) <- Mod(End2)
     Cod(RI) <- CLP
     Ctd(RI) <- End2

     Le Mem Global

     Mar <- Pre2 + BFP
     Mbr <- Ctd(RD)
     Grava Mem Local

     Con2 <- Con2 + 1

     End2 <- End2 + 1

     Aux2 <- Ctd(RD)
     If Tipo(Aux2) <> Lst Goto 14
     Con2 <- Con2 + 1

| Bloco que testa a unificação

 14: If Unif = 0 Goto 26

     Aux1 <- Con1
     Aux2 <- Con2
     If Con1 = 0 Goto 15
     Goto 16

 15: If Con2 = 0 Goto 17
```

```
    Goto 26

16: If Cond2 = 0 Goto 26

17: Mar <- Pre1 + BFP
    Le Mem Local
    Aux1 <- Mbr

    Mar <- Pre2 + Bfd
    Le Mem Local
    Aux2 <- Mbr

    If Tipo(Aux1) = Tipo(Aux2) Goto 13

    Con3 <- 1
    If Tipo(Aux1) <> Val Goto 22

| Bloco que lê um item do argumento do predicado 1

18: Mem(RI) <- Mod(End1)
    Cod(RI) <- CLP
    Ctd(RI) <- End1

    Le Mem Global

    Mar <- Pre1 + BFP
    Mbr <- Ctd(RD)
    Grava Mem Local

    Con3 <- Con3 + 1

    Aux1 <- Ctd(RD)
    If Tam(Rd) = 0 Goto 19

    Aux2 <- End1
    Aux  <- Aux1 + Tam(Aux2)
    End1 <- Aux
    Goto 21

19: If Tip(Aux1) <> Lst Goto 20
```

```
        Con3 <- Con3 + 2

    20: End1 <- End1 + 1

    21: Aux1 <- Con3
        If Aux1 <> 0 Goto 18

¦ Bloco que lê um item do argumento do predicado 2

    22: Mem(RI) <- Mod(End2)
        Cod(RI) <- CLP
        Ctd(RI) <- End2

        Le Mem Global

        Mar <- Pre2 + BFP
        Mbr <- Ctd(RD)
        Grava Mem Local

        Con3 <- Con3 + 1
        Aux1 <- Ctd(RD)
        If Tam(Rd) = 0 Goto 23

        Aux2 <- End2
        Aux  <- Aux1 + Tam(Aux2)
        End2 <- Aux
        Goto 25

    23: If Tip(Aux1) <> Lst Goto 24
        Con3 <- Con3 + 2

    24: End2 <- End2 + 1

    25: Aux1 <- Con3
        If Aux1 <> 0 Goto 22

        Goto 13
¦ Fim do programa
    26: Stop
```

UFRGS
INSTITUTO DE INFORMÁTICA

BIBLIOGRAFIA

[BIA87] BIANCHINI Jr., Ronald; SHEN, John Paul. Inter processor traffic scheduling algoritHm for multiple processor networks. **IEEE Trans= actions on computers**, New York, c-36, n. 4, p. 369-409, Apr. 1987.

[BRA85] BRANTLEY, W. C. et al. RP3 processor memory element. In : INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, Aug. 20-23, 1985, St. Charles. **Proceedings...** New York : IEEE, 1985. p. 782-797.

[CHA85] CHANG, Jung-Herng et al. AND-parallelism of logic programs based on a static data dependency analisis. In : COMPCON 85 SPRING TECNOLOGICAL LEVERAGE : A COMPETITIVE NECESSITI, 30. San Francisco, Feb. 25-28, 1985. **Digest of Papers...** Los Alamos : IEEE, 1985. p. 218-225.

[CLA84] CLARK, Keith; GREGORY, Steve. Notes on systems programming in Parlog. In : INTER- NATIONAL CONFERENCE ON FIFTH GENERATION COMPUTERS SYSTEMS, Nov. 6-9, 1984, Tokio. Tokio : OHM, 1984. p. 295-305.

[CLA86] CLARK, Keith; GREGORY, Steve. PARLOG : parallel programming in logic. **ACM Trans- actions on programming languages and sys- tems,** New York, v. 8, n. 1, p. 1-49, Jan. 1986.

[COS85] COSTA, Antonio C. da Rocha. **Clause Machines.** Porto Alegre, CPGCC/UFRGS, 1985. (Rela- tório Tecnico, n. 14)

[COR89] CORSINI, P. et al. The parallel interpretation of logic programs in distributed architectures. **The Computer Journal**, New York, v. 32, n. 1, p. 29-35, Feb. 1989.

[CRA85] CRAMMOND, Jim. A comparative study of unification algoritms for OR-parallel execution of logic languages. **IEEE Transactions on computers**, New York, v. c-34, n. 10, p. 911-917, Oct. 1985.

[DEC88] DECHTER, Rina; PEARL, Judea. Network-based Heuristics for constraint satisfaction problems. **Artificial Inteligence**, Amsterdam, v. 34, n. 1, p. 1-38, Jan. 1988.

[DEG84] DEGROOT, Doug. Restricted AND-parallelism,In: INTERNATIONAL CONFERENCE ON FIFTH GENERATION COMPUTER SYSTEMS, Tokyo, Nov. 6-9, 1984. **Fifth generation computer systems**, Tokyo : OHM, 1984. p. 471-478.

[DEG85] DEGROOT, Doug. Alternate graph expressions for restricted AND-paralelism. In : COMPCON 85 SPRING TECNOLOGICAL LEVERAGE : A COMPETITIVE NECESSITI, 30, Feb. 25-28, 1985. San Francisco. **Digest of Papers...** Los Alamos : IEEE, 1985. p. 206-210.

[DOB84] DOBRY, T. P. et al. Design decisions influencing the microarchitecture for a prolog machine. **SIGMICRO NEWSLETTER**, v. 15, n. 4, p. 217-237, Dec. 1984. Trabalho apresentado no "ANNUAL MICROPROGRAMMING WORKSHOP, Oct. 30-Nov. 02, 1982, New Orleans. Micro 17"

[DOB85] DOBRY, T. P. et al. Performance studies of a Prolog machine architecture. In : ANNUAL INTERNATIONAL SIMPOSYUM ON COMPUTER ARCHITECTURE, Jun. 17-19, 1985, Boston. **Proceedings...** New York : IEEE. 1985. p. 180-190.

[DOY79] DOYLE, Jon. A truth maintenance system. **Artificial inteligence**, Amsterdam, V. 12, n.3, p. 231-272, Nov. 1979.

[DUP88] DUPRAT, Jean. **LAIOS: un reseau multiprocesseur oriente vers des applications d'intelligence artificielle.** Glenoble, Institut National Polytechinique de Grenoble, 1988. Tese de Doutorado.

[GAL87] GALLARD, R. An extension in the definition of a Petri net Execution. **Computer**, Los Alamitos, v. 20, n. 1, p. 16-34, Jan. 1987.

[GAJ85] GAJSKI, Daniel; PEIR, Jih-kwon. Essencial issues in multiprocessor systems. **Computer**, Los Alamitos, v. 18, n.6, p. 9-27, Jun. 1985.

[GIN85] GINOSAR, Ran; HILL, Dwight. Design and implementation of switching systems for parallel processors. In : INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, Aug. 20-23, 1985, St. Charles. **Proceedings...** New York : IEEE, 1985. p. 674-680.

[GOT83] GOTTLIEB, Allan et al. The NYU ultracomputer - Designing and MIND shared memory parallel computer. **IEEE Transactions on computers**, New York, v. c-32, n. 2, p. 175-189, Feb. 1983.

[HWA87] HWANG, Kai et al. Computer arquitectures for artifitial intelligence processing computers. **Computer**, Los Alamitos, v. 20, n. 5, p. 19-27, Jan. 1987.

[KAR87] KARP, Alan. Programming for parallelism. **Computer**, Los Alamitos, v. 20, n. 5, p. 43-57, May 1987.

[KNO86] KNODLER, Brigitte; ROSENTIEL, Wolfgang. A Prolog preprocessor for Warren's abstract instruction set. **Microprocessing and microprogramming**, Amsterdam, v. 18, p. 71-80, 1986.

[KOW74] KOWALSKI, Robert. Predicate logic as programming language. In : IFIP CONGRESS 74, Aug. 5-10, 1974, Stockholm. **Proceedings...** Amsterdam: North-Holland, 1974. p. 569-574.

[LEB86] LEBLANC, Thomas. Shared memory versus message passing in a tightly-coupled multiprocessor : a case study. In : INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, Aug. 19-22, 1986, University Park. **Proceedings...** New York : IEEE, 1986. p 463-466.

[LEE86] LEE, Gyungho et al. The effectiveness of combining in shared memory parallel computers in the presence of HOT SPOTS. In : INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, University Park, Aug. 19-22, 1986, University Park. **Proceedings...** New York : IEEE, 1986. p. 35-41.

[LIG85] LI, Guo-jie; WAH, Benjamin. MANIP-2 a multicomputer architecture for evoluating logic programs. In : INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, Aug. 20-23, 1985. St. Charles. **Proceedings...** New York: IEEE, 1985. p. 123-130.

[MAC77] MACKWORTH, Alan. Consistency in networks of relations, **Artificial Inteligence**, Amsterdam, v. 8, n. 1, p. 99-118, Feb. 1977.

[MOT84] MOTO-OKA, Tohru et al. The architecture of a parallel inference engine - PIE, In : INTERNATIONAL CONFERENCE ON FIFTH GENERATION COMPUTER SYSTEMS, Tokyo, Nov. 6-9, 1984. **Fifth generation computer systems**. Tokyo : OHM, 1984, p. 479-488.

[MUD84] MUDGE, T. N. et al. Analisis of multiple bus interconnection networks. In : INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, Aug. 21-24, 1984, Bellaire. **Proceedings...** Silver Spring : IEEE, 1984. p. 228-232.

[MUR85] MURAKAMI, Kunio et al. Research on parallel machime architecture for fifth-generation computer systems. **Computer**, Los Alamitos, v. 18, n. 6, p. 76-92, June 1985.

[NAK85] NAKAZAKI, Ryosel et al. Design of a high-speed Prolog machine (HPM). In : ANNUAL INTERNATIONAL SIMPOSYUM ON COMPUTER ARCHITECTURE, 12, Boston, 17-19 June 17-19, 1985. **Proceedings...** New York : IEEE, 1985. p. 191-197.

[PAT85] PATTON, Peter. Multiprocessors: architecture and aplications. **Computer**, Los Alamitos, v. 18, n. 6, p. 29-40, June, 1985.

[PET86] PETRIE Jr., Charles. A diffusing computation for truth maintenance. In : INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, Aug. 19-22, 1986, University Park. **Proceedings...** New York : IEEE, 1986. p. 19-22.

[PFI85a] PFISTER, G. F. et al. The IBM research parallel processor prototype (RP3) introduction and architecture. In : INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, Aug. 20-23, 1985, ST. Charles. **Proceedings...** New York : IEEE, 1985. p. 764-781.

[PFI85b] PFISTER, Gregory; NORTON, V. Allan. HOT SPOTS contention and combining in multistage interconnection networks.**IEEE Transactions on computers**, New York, v. c-34, n. 10, p. 943-948, Oct. 1985.

[PLE87] PLESZKUN, Andrew; THAZTUTHAVEETIL, Matthew. The architecture of Lisp machines, **Computer**, Los Alamitos, v. 20, n. 3, p. 35-44, Mar. 1987.

[PON84] PONDER, Carl; PATT, Yale. Alternative proposes for implementing Prolog concurrently and implications regarding their respective architectures.**SIGMICRO NEWSLETTER**, v. 15, n. 4, Dec. 1984. p. 192-203. Trabalho publicado no "ANNUAL MICROPROGRAMMING WORKSHOP, Oct. 30- Nov. 02, 1984. Micro 17"

[RAM87] RAMAMOORTHY, C. V. et al. Software development support for AI programs. **Computer**, Los Alamitos, v. 20, n. 1, p. 30-40, Jan. 1987.

[TIC84] TICK, Evan. Sequential prolog machine : image and host architectures. In : ANNUAL INTERNATIONAL SIMPOSYUM ON COMPUTER ARCHITECTURE, 12, June 17-19, 1985, Boston. **Proceedings...** New York : IEEE, 1985. p. 204-216.

[TRE82] TRELEAVEN, Philip et al. Data-driven and Demand-driven computer architecture. **Computing surveys**, New York, v. 14, n. 1, p. 93-143, Mar. 1982.

[UCH83] UCHIDA, Shunichi. **Inference machine:** from sequential to parallel. Tokyo : ICOT, [198_], (TR - 011)

[VAR85] VARMA, A.; RAGHAVENDRA, G. S. Performance analysis of a redundant-path interconnection network. In : INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, Aug. 1985, St. Charles, **Proceedings...** New York: IEEE, 1985. p. 474-479.

[YAL85] YALAMANCHILI, Sudhakar; AGGARWAL, J. K. Reconfiguration strategies for parallel architectures. **Computer**, Dec. 1985, p. 44-61.

[YEN85] YEN, Wei et al. Data coherence problem in a multicache system. **IEEE Transactions on computers**, New York, v. c-34, n. 1, p. 56-65, jan. 1985.

[YEW87] Yew, Pen-Chung et al. Distributing HOT SPOT adressing in large-scale multiprocessors. **IEEE Transactions on computers**, New York, v. c-36, n. 4, p. 388-395, Apr. 1987.

[YOK83] YOKOTA, Minoru et al. The design and implementation of a personal sequential inference machine : PSI. **New generation computing**, Berlin, v. 1, p. 125-144, 1983.

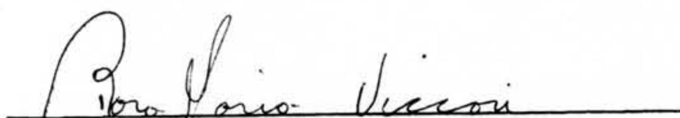
[WAH87] WAH, Benjamin. New computers for artificial intelligence processing. **Computer**, Los Alamitos, v. 20, n. 1, p. 10-13, Jan. 1987.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

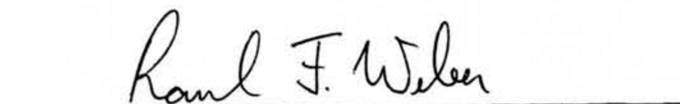
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Máquinas de cláusulas:
Arquitetura e Modelo de
execução de cláusulas PROLOG.

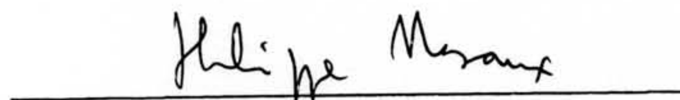
Dissertação apresentada aos Srs.



Profa. Rosa Maria Viccari



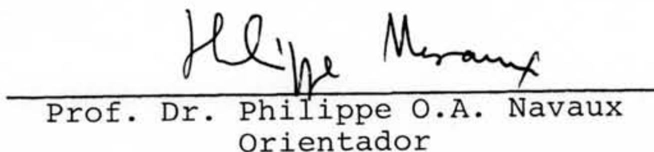
Prof. Dr. Raul Fernando Weber



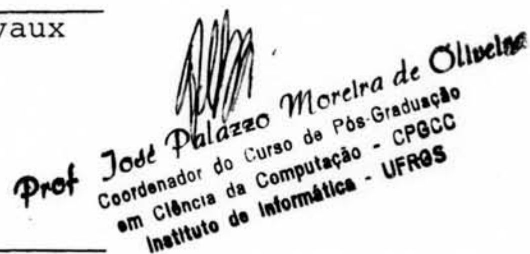
Prof. Dr. Philippe O.A. Navaux

Profa. Dra. Taisy Silva Weber

Visto e permitida a impressão.
Porto Alegre, 22/06/95



Prof. Dr. Philippe O.A. Navaux
Orientador



Prof. Ricardo A. da L. Reis,
Coordenador do Curso de Pós-
-Graduação em Ciência
da Computação