Um Novo Modelo de Execução Paralela de Programas Lógicos

Ricardo Gouvêa Bianchini

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Claudio L. de Amorim, Ph.D.
(Presidente)

Profa. Leila M. R. Eizirik, M.Sc.

Profa. Sueli B. T. Mendes, Ph.D.

Prof. Maria Carolina Monard, D.Sc.

RIO DE JANEIRO, RJ – BRASIL

JULHO DE 1990

BIANCHINI, RICARDO GOUVÊA

Um Novo Modelo de Execução Paralela de Programas Lógicos [Rio de Janeiro] 1990

IX, 95 p., 29.7 cm, (COPPE/UFRJ, M.Sc., ENGENHARIA DE SISTEMAS E COMPUTAÇÃO, 1990)

TESE – Universidade Federal do Rio de Janeiro, COPPE

1 – Programação Lógica 2 – Prolog 3 – Processamento Paralelo

I. COPPE/UFRJ II. Título(Série).

Ao meu avô Candido Álvaro de Gouvêa

Agradecimentos

Ao meu orientador, Claudio Amorim, e co-orientadora, Leila Eizirik, pela orientação correta durante a pesquisa, pelo incentivo à realização deste trabalho e pela sua amizade nos momentos difíceis.

Ao amigo Ruy Marinho pela enorme colaboração, na implementação do modelo Gol-Log, sem a qual teria sido impossível realizar este trabalho.

Ao amigo Paulo Cesar 'Gatorade', pela enorme ajuda na confecção dos gráficos incluídos na tese.

Aos Profs. Valmir Barbosa e Sueli Mendes, pela preocupação constante com o andamento da minha tese, e pelas cartas de recomendação.

À amiga Inês, que, mesmo de longe, ajudou na implementação do modelo Gol-Log.

Aos amigos Luiz Adauto e Juarez, pela força que me deram, principalmente, durante o processo de aplicação para o doutorado.

Às amigas da secretaria, Denise, Cláudia e Suely, sem as quais eu não teria conseguido nem terminar minhas matérias no mestrado.

E, finalmente, à minha futura esposa, Marcia, pelo seu carinho e atenção intermináveis.

Resumo da Tese apresentada à COPPE como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M. Sc.)

Um Novo Modelo de Execução Paralela de Programas Lógicos

Ricardo Gouvêa Bianchini

Julho de 1990

Orientador: Claudio Luis de Amorim

Programa: Engenharia de Sistemas e Computação

Nesta tese apresentamos um novo modelo de execução paralela de programas lógicos, denominado Gol-Log.

A principal contribuição desta tese está relacionada com a proposta, descrição, implementação e avaliação do modelo Gol-Log. Outras contribuições importantes são as soluções dadas aos problemas comuns aos modelos paralelos de execução, e algumas inovações trazidas por Gol-Log.

Abstract of Thesis presented to COPPE as partial fulfillment of the requirements for the degree of Master of Science (M. Sc.)

A New Model of Parallel Execution of Logic Programs

Ricardo Gouvêa Bianchini

July, 1990

Thesis Supervisor: Claudio Luis de Amorim

Department: Programa de Engenharia de Sistemas e Computação

In this thesis a new model of parallel execution of logic programs, called Gol-Log, is presented.

The main contibution of this thesis is the proposal, description, implementation and evaluation of the Gol-Log model. Other important contributions are the solutions given to the common problems of the parallel execution models and the inovations included in Gol-Log.

Índice

CAPÍTULO IV

CAPÍTULO V

CAPÍTULO I

Introdução

Há vários anos a programação lógica vem gerando um interesse crescente por parte da comunidade de computação. O que poderíamos chamar de 'boom' da programação lógica ocorreu a partir da criação da linguagem Prolog (CLOCKSIN, 1981), no início dos anos setenta. Os primeiros anos de experiência com Prolog serviram para que fossem detectados alguns problemas com a linguagem, tais como:

- a grande necessidade de memória; e
- a ineficiência de execução.

Tais problemas desencadearam, então, um grande esforço que procurava obter implementações (seqüenciais) eficientes para Prolog. Como resultados deste esforço, temos, por exemplo, os trabalhos sobre compilação de Prolog, desenvolvidos por Warren (WARREN, 1977 e WARREN, 1983a), e por nós (BIANCHINI, 1988 e DUTRA, 1988b).

Atualmente, o que vemos é a utilização de alguns resultados obtidos naquele "esforço seqüencial", aliados à exploração de paralelismo, na tentativa de alcançar uma maior eficiência de execução. Um outro objetivo, além deste, é o de conseguir implementar sistemas chamados reativos ('reactive'), tais como sistemas operacionais e bancos de dados.

A partir das metas citadas, foram criadas duas linhas de pesquisa distintas: a que visa executar programas lógicos eficientemente, sem a utilização de recursos "impuros" (diretivas de processamento concorrente), e a que pretende utilizar recursos que permitam a programação lógica concorrente.

Nesta tese apresentamos um novo modelo de execução paralela de programas lógicos, chamado Gol-Log, o qual pertence à primeira das linhas de pesquisa descritas acima.

A principal contribuição desta tese está relacionada com a proposta, descrição, implementação e avaliação do modelo Gol-Log. Outras contribuições importantes são as soluções dadas aos problemas comuns aos modelos paralelos de execução, e algumas inovações trazidas por Gol-Log.

Este trabalho supõe algum conhecimento de programação lógica (CASANOVA, 1987) e Prolog (CLOCKSIN, 1981). Na parte de paralelismo, apesar de não apresentarmos uma introdução completa, o material aqui contido é suficiente para a compreensão dos modelos que descrevemos e da proposta de Gol-Log. Uma introdução bastante clara e abrangente à execução paralela de programas lógicos pode ser encontrada em (BIANCHINI, 1989).

A dissertação transcorre da seguinte forma: o próximo capítulo apresenta uma rápida introdução à execução paralela de programas lógicos. O capítulo III faz uma descrição suscinta de alguns importantes modelos de execução, que apresentam pontos em comum com o Gol-Log. A seguir, é descrito o modelo Gol-Log, ressaltando a sua implementação e suas principais inovações. O capítulo V apresenta uma implementação do modelo e os resultados obtidos, em uma rede de processadores Transputer. Finalmente, o capítulo VI descreve as nossas conclusões sobre o trabalho.

CAPÍTULO II

Execução Paralela de Programas Lógicos

II.1 - Conceitos Básicos

Nesta seção apresentamos alguns conceitos básicos, que, por serem de grande importância para a compreensão do resto da dissertação, precisam ser revistos.

II.1.1 - Árvore de Objetivos

Os programas lógicos podem ser descritos operacionalmente como exploradores de uma árvore de objetivos. A árvore de objetivos de um programa é aquela que apresenta como nós, o conjunto de objetivos que devem ser executados. A raiz da árvore é a consulta ao programa e as folhas são sucessos ou falhas.

Para ilustrar, apresentaremos a árvore de objetivos de um programa simples (figura II.1), considerando a estratégia de Prolog: seleção de literais da esquerda para a direita. Desta forma, cada nó interno da árvore é obtido pela substituição do literal mais a esquerda dos objetivos, pelo corpo de uma das cláusulas para tal literal.

```
:- p(X) & r(X).

p(X) :- a(X).
p(X) :- b(X).
p(X) :- c(X).

r(2).

a(1).
a(2).

b(2).

c(3).
```

Podemos observar que, partindo da consulta ao programa, a primeira geração de filhos na árvore é obtida pela substituição de *p(X)* pelos corpos da cláusulas do predicado *p*. A partir do objetivo *:- a(X) & r(X)*, observamos a geração de dois filhos: *:- r(1)* e *:- r(2)*. Tais filhos foram, também, obtidos pela substituição do literal mais a esquerda no objetivo pai (*a(X)*).

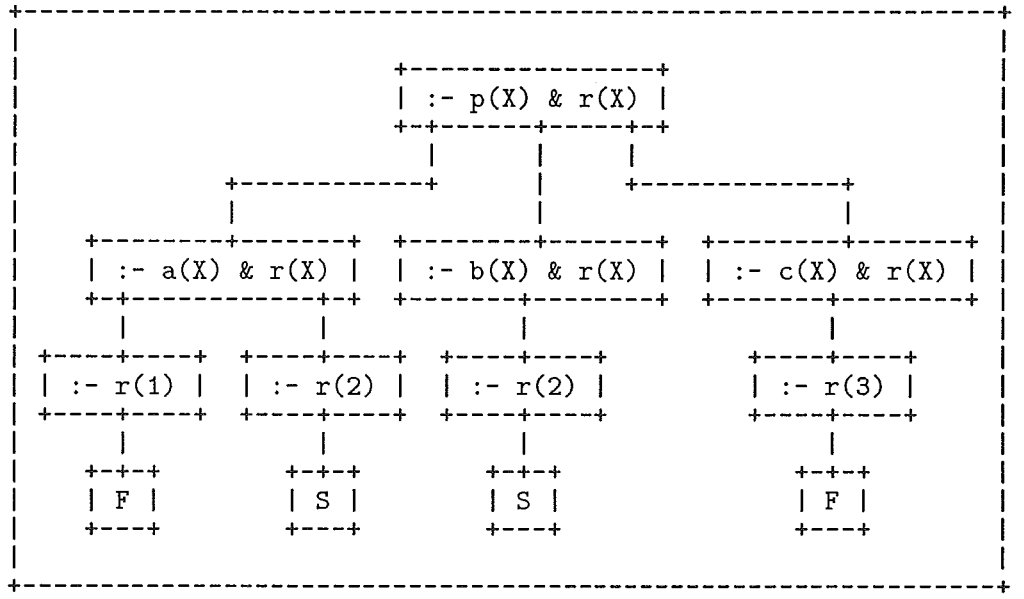


Figura II.1: Exemplo de Árvore de Objetivos

II.1.2 - Árvore AND/OR

Uma outra forma operacional de ver os programas lógicos é considerar que estes definem uma árvore AND/OR de pesquisa. A árvore AND/OR de um programa apresenta dois tipos de nós (AND e OR), com um nó AND na raiz (consulta) e nós AND nas folhas (sucesso ou falha).

Assim, utilizando o exemplo da seção anterior, teremos a árvore AND/OR apresentada na figura II.2.

As árvores AND/OR são conceitualmente muito simples. No exemplo, observamos que existe um nó AND para a consulta ao programa (*:- p(X) & r(X)*), com nós OR (filhos) para cada um dos literais do nó AND (*p(X)* e *r(X)*). Estes nós OR terão um filho AND para cada cláusula correspondente ao seu literal. Assim, temos, por exemplo, o nó OR para *p(X)* gerando os filhos AND *p(X)*, para *p(X) :- a(X)*, *p(X) :- b(X)*, e *p(X) :- c(X)*.

II.1.3 - Busca em Profundidade e Outras Estratégias

Como sabemos, a estratégia de controle de Prolog combina a exploração do literal mais a esquerda nos objetivos, com uma busca em profundidade sobre a árvore de objetivos

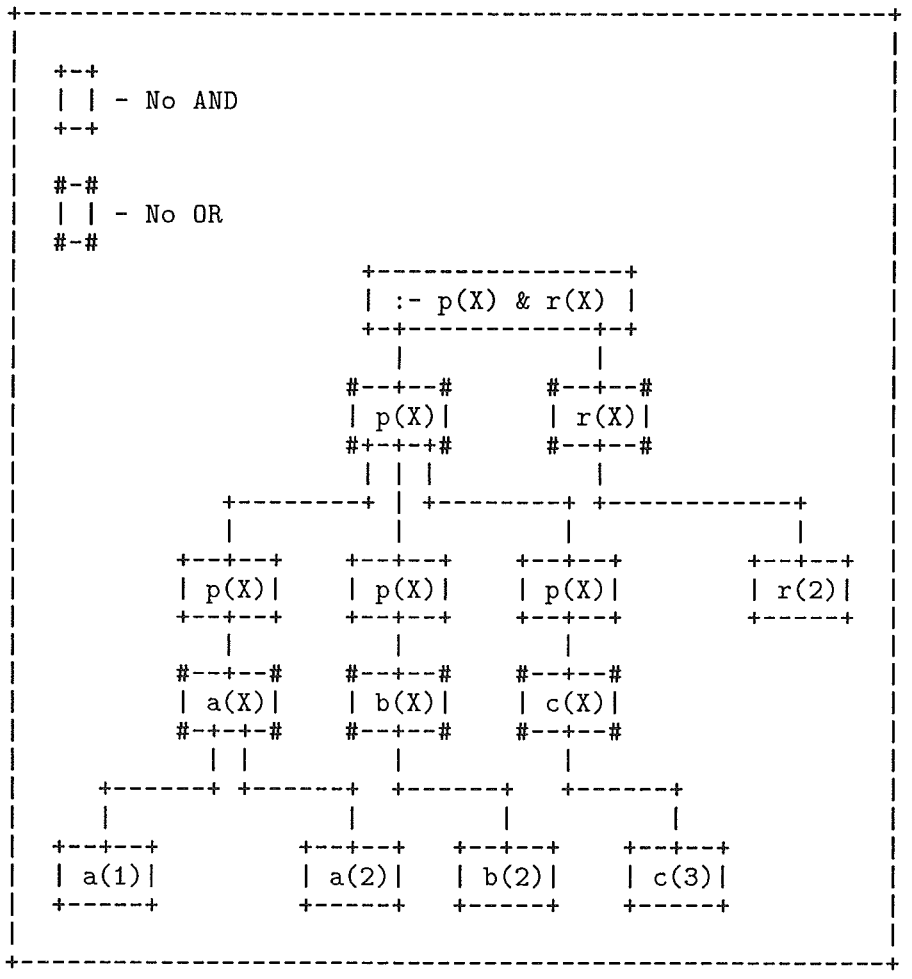


Figura II.2: Exemplo de Árvore AND/OR

dos programas. Desta forma, quando o Prolog encontra um nó de falha, ele retrocede (realiza um 'backtracking') até o último nó que ainda não tenha tido todos os seus filhos explorados e prossegue a busca.

No entanto, esta não é a única forma possível de percorrer a árvore para os programas lógicos. Podemos utilizar uma busca em largura, por exemplo, a qual escolhe os objetivos de forma a explorar a árvore de objetivos de forma "horizontal", isto é, só descendo um nível na árvore quando todos os objetivos do nível anterior já tiverem sido escolhidos para substituição. Na verdade, seria interessante poder realizar uma busca que escolhesse sempre o melhor caminho a seguir, ou seja, aquele que mais rapidamente nos levasse à(s) solução(ões) do programa. Esta estratégia pode ser chamada 'best-first', e precisa de algum critério para definir qual o melhor caminho a seguir a cada momento.

Existem, ainda, outros pontos onde podemos ter estratégias diferentes das de Prolog:

- na escolha de literais a serem resolvidos, podemos utilizar critérios que levem em conta o número de variáveis instanciadas;
- podemos permitir ao programador que modifique a estratégia de controle; e
- podemos eliminar a computação repetida de certas partes do programa.

II.1.4 - Implementação de Prolog

A estrutura geral de Prolog (CLOCKSIN, 1981) já é bastante conhecida, por isso, nesta seção, nos limitaremos apenas a citar alguns tópicos considerados mais relevantes.

As principais características do Prolog seqüencial, como sabemos, são:

- estratégia de seleção de objetivos da esquerda para a direita;
- estratégia de pesquisa para escolha de cláusulas para satisfação de objetivos de cima para baixo;
- utilização de 'backtracking' para geração de todas as soluções dos programas; e
- utilização de predicados que provocam efeitos colaterais ('cut', 'assert', 'retract').

As três primeiras características compõem a estratégia de controle chamada LRDF ('left-rigth depth-first').

Para a proposta do novo modelo seria interessante observar alguns princípios relacionados à implementação de Prolog.

Os interpretadores Prolog utilizam estruturas de dados estáticas e dinâmicas, e ainda, alguns registradores auxiliares (BRUYNOOGHE, 1982). Em geral, as estruturas estáticas necessárias são:

- tabela de definições: cada item guarda o nome do predicado, o seu número de argumentos e um ponteiro para a primeira cláusula;

- tabela de cabeças de cláusulas: cada item desta tabela é uma lista de descritores de cabeças, composta pelo número de variáveis, pela lista de argumentos, pelo ponteiro para o corpo da cláusula e pelo ponteiro para a próxima cláusula;
- tabela de corpos de cláusulas: cada item desta tabela é uma lista de descritores de objetivos no mesmo corpo de cláusula. Cada descritor de objetivos é composto pelo ponteiro descritor de definição, pelo número de argumentos, pela lista de argumentos e pelo ponteiro para o próximo objetivo; e
- tabela de termos: cada item guarda o nome do termo, o número de argumentos e a lista de argumentos.

As estruturas dinâmicas são:

- pilha de ambientes: controla o estado corrente da avaliação;
- pilha de variáveis: guarda os 'bindings' das variáveis; e
- trilha: guarda informações que permitem realizar o 'backtracking'.

Os registradores principais são responsáveis por:

- apontar a chamada corrente;
- apontar a próxima cláusula alternativa à chamada de objetivo corrente;
- apontar para o último ponto de escolha;
- apontar para a moldura corrente na pilha de ambientes;
- apontar para a moldura corrente na pilha de variáveis; e
- apontar para a moldura corrente na trilha.

Quanto aos compiladores Prolog (WARREN, 1977, WARREN, 1983a e WARREN, 1983b), a estratégia de controle e as estruturas de dados estáticas estão definidas no código gerado. As estruturas dinâmicas são muito semelhantes às dos interpretadores. As formas de tratamento de estruturas nas implementações Prolog (KACSUK, 1990) se dividem em:

- substituição de literais: neste método, em cada novo objetivo, as variáveis são substituídas pelos seus 'bindings';
- compartilhamento de estruturas: esta forma representa 'bindings' de variáveis como dois ponteiros. Um ponteiro para o código e o outro para uma nova pilha, chamada pilha de moléculas; e
- cópia de estruturas: esta forma é uma combinação das outras. Um 'binding' é representado por um ponteiro para o código (se no código da estrutura não existirem variáveis) ou para uma nova pilha, chamada pilha de cópias, a qual conterá uma cópia do termo.

II.2 - Formas de Exploração de Paralelismo em Programas Lógicos

Existem duas grandes fontes de exploração de paralelismo em programação lógica, chamadas paralelismo AND e paralelismo OR. No paralelismo AND teremos os objetivos em uma regra podendo ser executados paralelamente, enquanto no paralelismo OR teremos as cláusulas de um predicado executadas em paralelo.

A figura II.3 mostra os paralelismos AND e OR na árvore de objetivos dos programas.

Com esta figura queremos dizer que o paralelismo OR procura realizar uma busca paralela dos diversos caminhos na árvore de soluções. Já o paralelismo AND procura acelerar os "passos" dentro de cada caminho.

II.2.1 - Paralelismo OR

O paralelismo OR utiliza-se das diferentes cláusulas de um certo predicado concorrentemente, procurando soluções para um mesmo objetivo.

Exemplificando:

```
:- p(X).

p(X) :- p1(X).
p(X) :- p2(X).
```

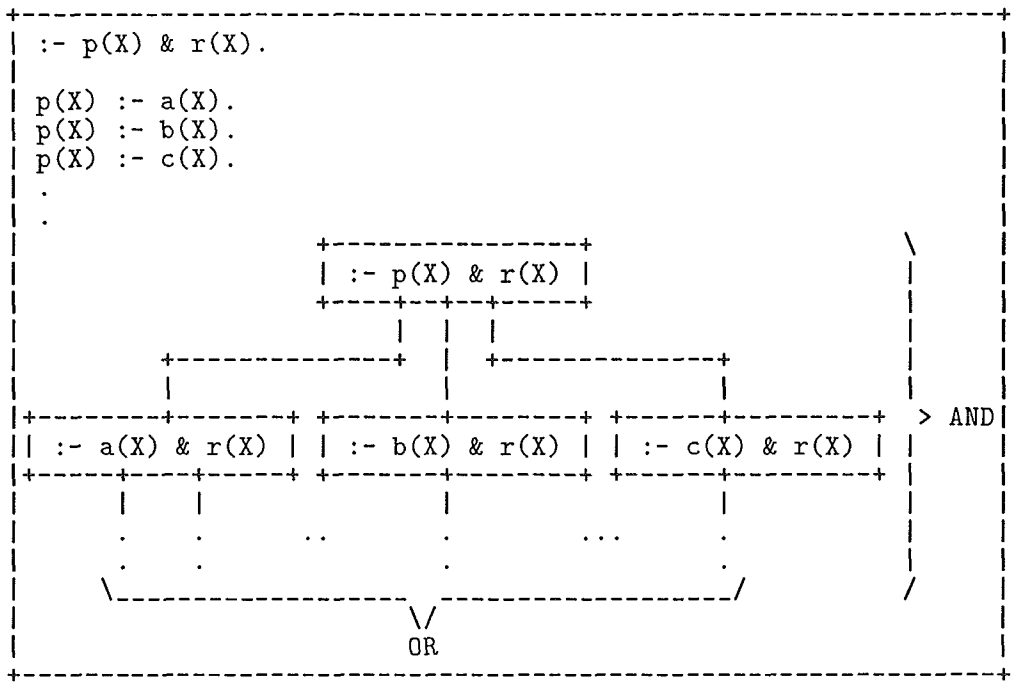


Figura II.3: Representação dos paralelismos AND e OR

Na tentativa de solução do objetivo *p(X)*, as duas cláusulas para *p* são chamadas em paralelo e as suas soluções são "combinadas". Esta "combinação" de resultados é necessária, pois podemos ter diferentes conjuntos de soluções para cada cláusula utilizada, podendo inclusive ter 'bindings' diferentes para a mesma variável.

O paralelismo OR relaciona-se com a busca em largura, enquanto em Prolog temos a busca em profundidade. Desta forma, modelos utilizando esta forma de paralelismo são mais "completos" que Prolog, uma vez que soluções a direita de ramos infinitos, na árvore de busca, podem ser computadas.

Existem dois grandes problemas com o paralelismo OR: a explosão do número de processos concorrentes e a manutenção dos ambientes de variáveis.

A explosão do número de processos OR é um problema fundamental: a característica de não-determinismo dos programas lógicos pode gerar uma sobrecarga de concorrência, isto é um número tão grande de processos que o 'overhead' causado pelo escalonamento dos mesmos seja excessivo.

O gerenciamento de ambientes é, também, uma questão importante, uma vez

que os processos OR podem, inadvertidamente, gerar 'bindings' incompatíveis para certas variáveis.

Um exemplo simples deste problema é o seguinte:

```
:- p(X).

p(X) :- p1(X).
p(X) :- p2(X).

p1(1).

p2(2).
```

Neste exemplo, dependendo da forma com que se gerencie os ambientes de variáveis, podemos ter a variável X recebendo dois valores diferentes (1 e 2). Em outras palavras, se utilizarmos uma única posição de memória para armazenar X, poderá ocorrer que X fique com apenas um dos dois valores na solução da consulta ao programa (o valor que lhe for atribuído por último).

II.2.2 - Paralelismo AND

O paralelismo AND procura acelerar a busca por cada solução, executando paralelamente os objetivos de uma mesma cláusula.

Observando-se a árvore de objetivos, o paralelismo AND faz com que o caminho (número de nós) percorrido até as folhas seja reduzido.

Podemos utilizar duas formas de paralelismo AND: o AND Restrito e o Stream AND. A primeira forma pode ser utilizada quando temos objetivos independentes (sem variáveis compartilhadas, no momento da sua execução) no corpo da cláusula, e a segunda quando existem objetivos dependentes.

Exemplificando o paralelismo AND Restrito:

Exemplo 1:

```
p(X,Y) :- q(X) & r(Y).
```

Neste caso, apesar de aparentemente independentes, os objetivos *q(X)* e *r(Y)* somente o serão se o objetivo que causou a ativação desta cláusula não for do tipo *p(X,X)*,

por exemplo.

Exemplo 2:

p(X,Y) :- q(X) & r(X) & s(Y).

Uma análise rápida deste exemplo poderia considerar que os objetivos *q(X)* e *r(X)* são necessariamente dependentes. Na verdade, eles somente o são se a variável X não estiver "totalmente instanciada" no momento da chamada desta cláusula.

Exemplo 3:

p(X,Y) :- q(X,Z) & q(Y,Z) & r(X,Y).

Já neste caso, observamos que os objetivos *q(X,Z)* e *q(Y,Z)* não são independentes, sendo necessário portanto que exista uma ordem que especifique a execução seqüencial destes objetivos.

A outra forma de paralelismo AND, citada acima, é o paralelismo Stream AND. Este permite a avaliação concorrente de objetivos que compartilham variáveis.

Nesta forma de paralelismo, os objetivos comunicam-se incrementalmente através de 'bindings' de variáveis compartilhadas. Podemos imaginar este tipo de comunicação como sendo um esquema produtor-consumidor, em que alguns objetivos produzem valores para variáveis e outros os consomem.

O último exemplo apresentado poderia utilizar-se deste paralelismo:

p(X,Y) :- q(X,Z) & q(Y,Z) & r(X,Y).

Poderíamos ter o segundo objetivo do corpo da cláusula produzindo valores para a variável Z, os quais são consumidos pelo primeiro objetivo.

É fácil observar que esta forma de paralelismo é adequada quando temos programas lógicos determinísticos. Quando este não é o caso, é necessária uma forma de restringir o não-determinismo para que se possa alcançar alguma eficiência. A maneira de restringir o não-determinismo é fazer com que cada objetivo só possa computar uma solução (a conseqüência desta restrição é análoga à utilização do operador corte em Prolog).

Os grandes problemas do paralelismo AND são: lixo na pilha de execução e objetivos que ficam bloqueados. Quando os processos AND paralelos são alocados a um

mesmo processador, os problemas podem aparecer, dependendo da ordem de criação dos processos, da alocação dos registros dos processos na pilha de execução, e da ordem com que certos processos falham (KACSUK, 1990).

II.3 - Modelos de Execução Paralela

Como já mencionado, as propostas de modelos utilizando-se de estratégias de controle paralelas para programas lógicos dividem-se em dois grandes grupos: as que utilizam programas "puros" de cláusulas de Horn, e as que incluem uma linguagem de controle (operadores, guardas, etc) para guiar a avaliação paralela (tais propostas são, em geral, orientadas à programação concorrente).

Tais grupos podem ainda ser subdivididos em outros dois: aqueles que são projetados para um tipo específico de arquitetura, e aqueles mais "abstratos", no sentido de não serem propostos para uma determinada máquina.

As propostas pertencentes a cada um destes grupos se diferenciam principalmente pelo tipo de programas (programas Prolog, programas lógicos "puros", programas concorrentes, etc) que elas procuram implementar, pelas formas de paralelismo que elas utilizam, e pelo nível de preocupação que os programadores devem ter com a forma com que o processamento se desenvolve.

Pelo exposto, as questões fundamentais que devem ser abordadas nos modelos de execução paralela são:

- as formas de paralelismo que o modelo pode utilizar;
- a forma de prover e controlar o paralelismo (através do interpretador/compilador ou da linguagem de programação);
- a forma de gerenciar os ambientes de variáveis, principalmente em modelos que utilizam paralelismo OR;
- a explosão do número de processos, principalmente quando é utilizado o paralelismo OR;
- a forma de evitar os problemas do paralelismo AND (lixo na pilha de execução e objetivos bloqueados), caso este seja utilizado; e

- o método de manipular estruturas.

Além destas questões, temos ainda, por exemplo:

- a questão do balanceamento de carga entre os processadores; e
- a maneira de alocar o programa lógico e os processos pelos processadores.

CAPÍTULO III

Alguns Modelos Importantes

Nesta seção apresentaremos as principais características de modelos que têm pontos em comum com Gol-Log: o modelo B-Log, o KABU-WAKE, o KABU-WAKE Alterado, o PRISM, o PIE, e o Modelo de Processos AND/OR.

No final da descrição de cada um dos modelos paralelos, serão indicados os tipos específicos de paralelismos AND e OR que cada um deles utiliza, segundo as classificações de Conery (CONERY, 1987) e Gregory (GREGORY, 1987).

III.1 - Modelo B-Log

B-Log (LIPOVSKI, 1985) é uma metodologia para a avaliação de programas lógicos que inclui a utilização de algoritmos 'branch and bound', procurando realizar uma busca 'best-first' em árvores OR, com pesos para cada ramo. É descrita também, na referência citada, uma arquitetura, chamada máquina B-Log, para implementar tal metodologia.

O paralelismo deste modelo está no fato de que vários processadores têm os pedaços de árvore de melhor 'bound' para processar ao mesmo tempo. Cada processador deve, é claro, ter conhecimento dos pesos dos caminhos explorados nos outros processadores.

Dentre os conceitos principais relacionados a B-Log estão:

- a representação do banco de cláusulas como grafos;
- um esquema de manipulação de pesos que guia o processo de busca de soluções (caminhos de menor peso são explorados primeiro);
- a idéia de "sessão", a qual procura acelerar a busca através de atualizações locais (dentro das sessões) e globais (entre sessões) dos pesos de cada ramo. A idéia é fazer com que os sucessos tenham os menores 'bounds'; e
- a utilização de 'semantic paging disks', os quais organizam os dados em páginas

definidas semântica e dinamicamente (as páginas não têm tamanho fixo), com base no grafo que representa o banco de cláusulas e o estado da busca.

Os componentes da máquina B-Log são:

- os já citados, 'semantic paging disks';
- processadores com memórias locais: pequenos pedaços do grafo global ficam nestas memórias locais. Os processadores usam tais pedaços para trabalhar nas suas partes da árvore de pesquisa;
- uma rede de chaveamento de pacotes e circuitos: por onde passam os pedaços do grafo global, entre os 'semantic paging disks' e os processadores; e
- um circuito de 'seek' mínimo e de prioridade: responsável por difundir os 'bounds' mínimos dos caminhos já percorridos e atribuir os mínimos aos processadores.

O problema da explosão de processos OR, neste modelo, é contornado, uma vez que apenas o melhor caminho a cada momento é percorrido (o melhor caminho é o de menor peso total).

Não há informações em (LIPOVSKI, 1985) sobre como são gerenciados os ambientes de variáveis, nem como são manipuladas as estruturas.

Segundo Gregory, podemos classificar B-Log como um modelo que utiliza o paralelismo OR. Na classificação de Conery, o paralelismo encontrado em B-Log se enquadra no paralelismo OR puro e na busca distribuída (pois temos vários 'semantic paging disks', nos quais está distribuído o banco de cláusulas, atuando ao mesmo tempo).

III.2 - Modelo KABU-WAKE

O objetivo básico do modelo KABU-WAKE (SOHMA, 1986) é utilizar a otimização dos métodos seqüenciais de inferência e o paralelismo OR, na execução de programas.

A idéia é fazer com que cada elemento processador processe seqüencialmente a árvore de soluções, realizando uma busca em profundidade, até que um outro elemento

processador solicite um pedaço de árvore para executar. Neste momento, o elemento processador que recebeu o pedido divide a sua árvore de forma a prover paralelismo OR à avaliação e a transfere para o processador que pediu.

Utilizaremos um exemplo simples, na sintaxe do modelo, para ilustrar (figura III.1) a execução de um programa, no modelo KABU-WAKE:

```
?- a.

a :- b, c.

b :- e, f.
b :- g.

c :- h.

e.

f.
f.

h.
```

Neste exemplo, temos o ramo mais a esquerda da árvore de objetivos sendo executado no processador 0, enquanto os objetivos *?-* $g \wedge c$, e *?-* c já foram transferidos para os processadores 1 e 2, respectivamente.

Segundo (SOHMA, 1986), as vantagens deste modelo são: pequeno 'overhead' de execução em cada elemento processador (inferência seqüencial), pouca comunicação entre processadores (somente quando existe pedido de trabalho), granularidade dos pedaços de árvore pode ser mantida tão alta quanto possível (a árvore é partida o mais próximo da raiz possível), número de processos OR é limitado (o limite é o número de elementos processadores).

A máquina experimental, baseada em memória distribuída, utilizada na avaliação deste método de inferência é composta basicamente por:

- Dezesseis elementos processadores (um deles para entrada e saída), cada um com uma cópia do programa e tendo uma capacidade de processamento de 1 Klips (Lips - Logical Inferences per Second);

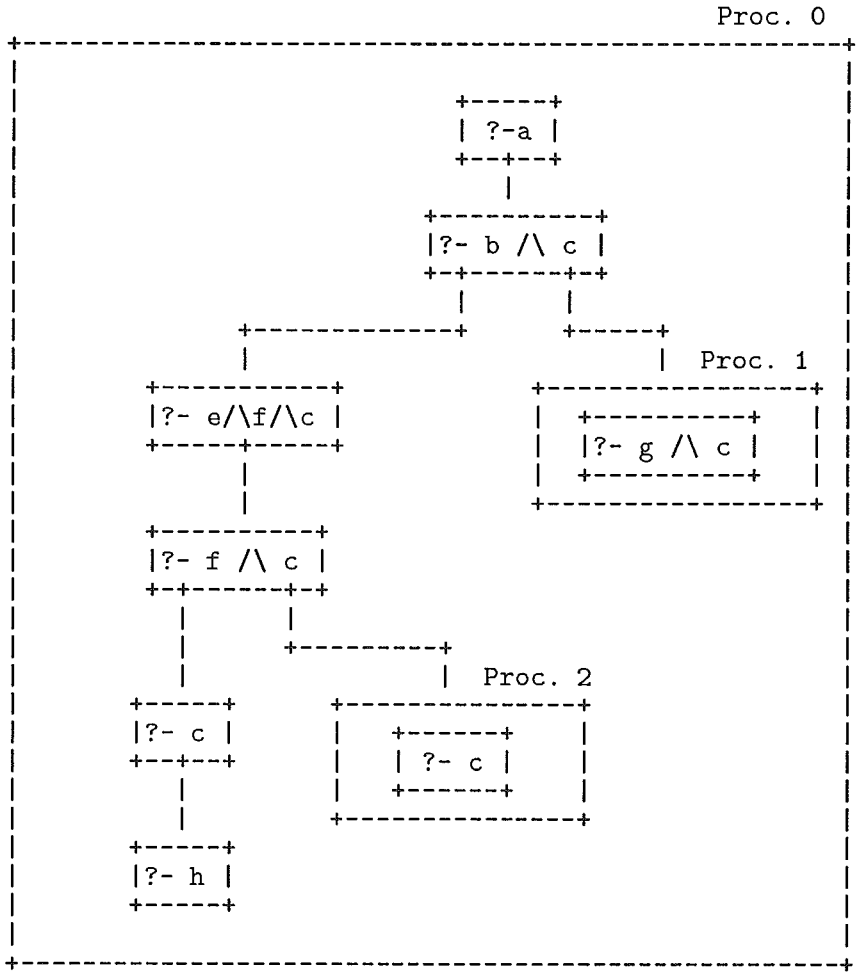


Figura III.1: Exemplo de execução no modelo KABU-WAKE

- Uma rede de pedido de trabalho (CONT NETWORK), onde são colocadas informações sobre o estado dos processadores; e
- Uma rede de dados (DATA NETWORK), por onde passam os "pedaços de árvore".

A representação gráfica da arquitetura KABU-WAKE (retirada de SOHMA, 1986) se encontra na figura III.2.

Os resultados da avaliação do modelo, apresentados na referência citada, são muito restritos, uma vez que só foram avaliadas variações do problema das rainhas.

Observando a classificação de formas de paralelismo de Conery, este modelo se enquadra no paralelismo OR puro. Na classificação de Gregory, podemos dizer que este modelo utiliza o paralelismo OR.

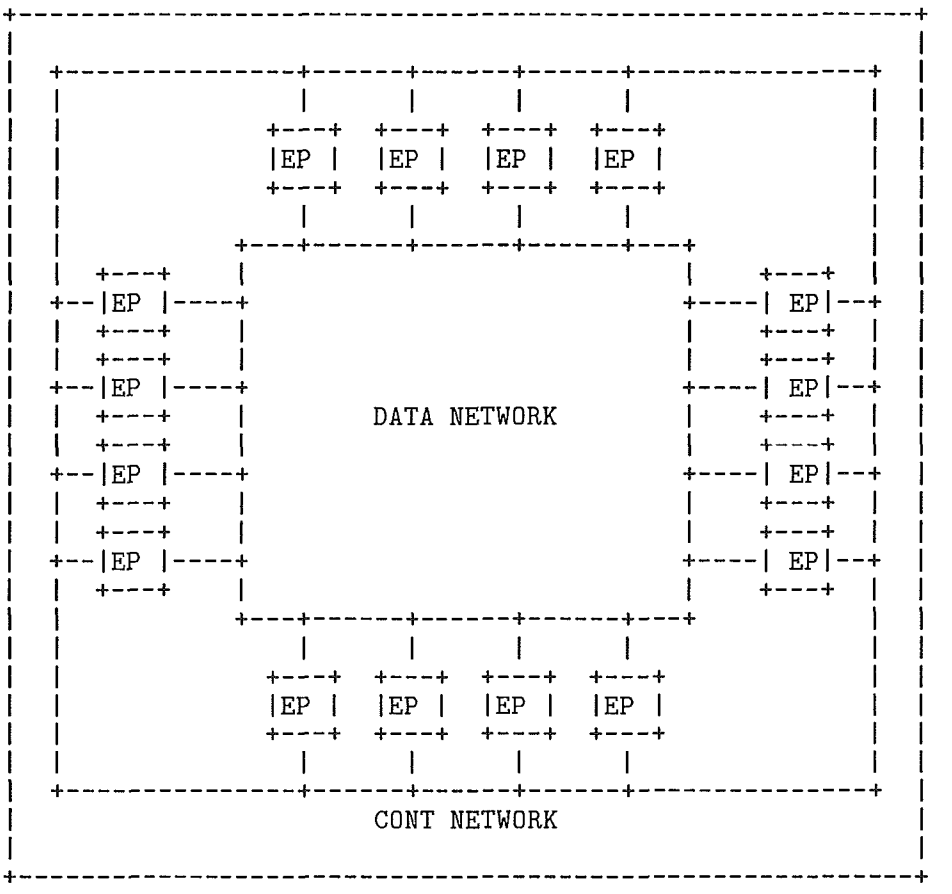


Figura III.2: Arquitetura para o modelo KABU-WAKE

III.3 - O Modelo KABU-WAKE Alterado

Algumas otimizações, em certos modelos de execução propostos, nos parecem claras. Nesta seção discutimos o modelo KABU-WAKE com alterações, feitas por nós, que devem melhorar o seu desempenho. O objetivo de tais modificações é diminuir o tempo de execução dos programas, sem prejudicar outras características, tais como a pouca necessidade de comunicação entre os processadores.

Assim, foram realizadas as seguintes alterações:

- passamos a utilizar além do paralelismo OR, já explorado pelo modelo original, o paralelismo AND Restrito (segundo a classificação de Gregory); e
- passamos a permitir que outras estratégias de inferência fossem utilizadas, não apenas a busca em profundidade (pode ser usada a busca em largura, por exemplo).

Existe uma grande vantagem conceitual na realização da primeira modificação citada: a utilização do paralelismo AND Restrito fará com que não haja computação repetida em certos casos. Para ilustrar este conceito apresentaremos um exemplo, na sintaxe aceita pelo modelo, mais simples que o da seção III.2:

```
?- r(X,Y).

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

p(1).
p(2).
p(3).

q(4).
q(5).
```

Se não for utilizado o paralelismo AND Restrito, teremos a árvore apresentada na figura III.3.

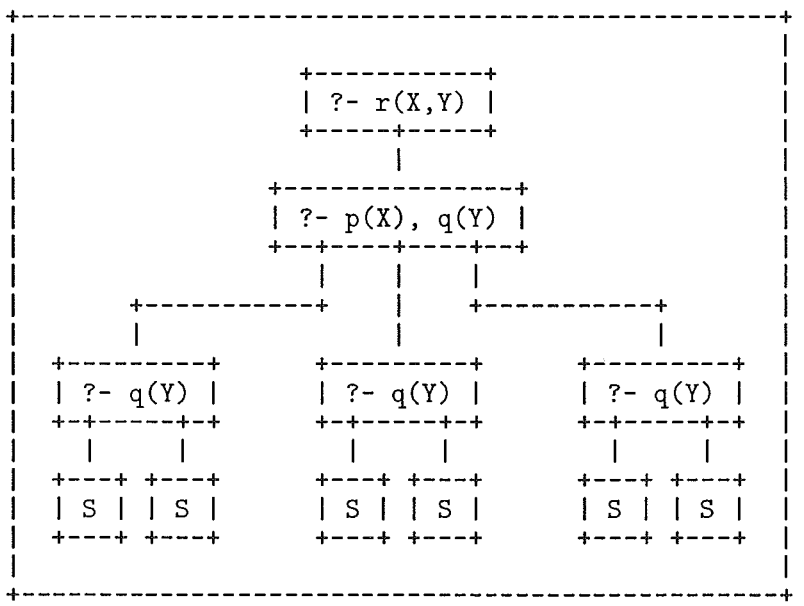


Figura III.3: Árvore de Objetivos sem o uso de paralelismo AND

Desta forma, estaremos computando *?- q(Y)* tantas vezes quantas forem as soluções de *?- p(X)*. No entanto, é importante notar que estes dois objetivos são independentes entre si, podendo, portanto, ser executados em paralelo (paralelismo AND Restrito). Assim, para obter as soluções de *?- p(X), q(Y)* basta combinar as soluções de *?- p(X)* e *?- q(Y)*. Isto eliminará a computação repetida, como mostra a figura III.4.

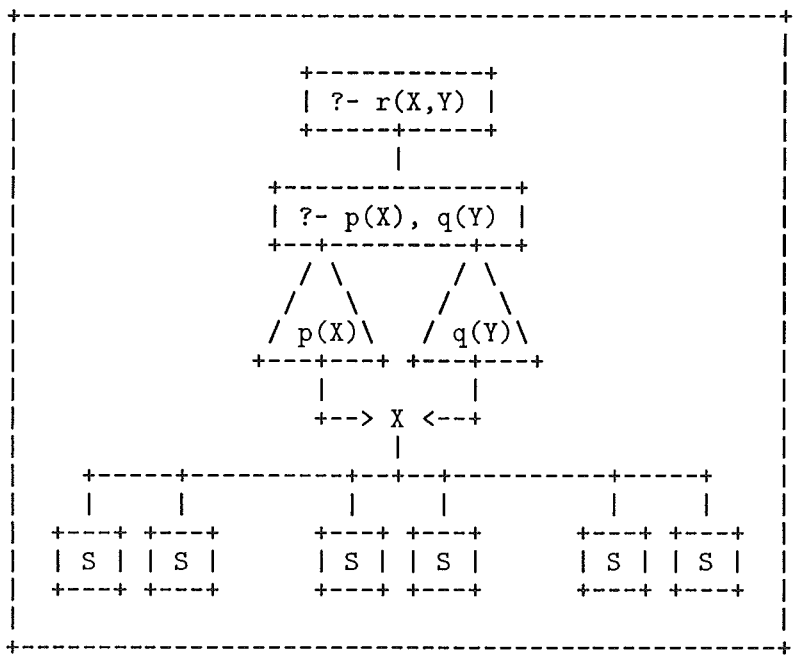


Figura III.4: Árvore de objetivos usando o paralelismo AND

Como pudemos observar pela figura, a computação de *?- q(Y)* não mais se repete (o processamento deste objetivo, assim como de *?- p(X)*, está representado como um triângulo com o respectivo objetivo no centro).

A arquitetura projetada na proposta original não precisa ser alterada em conseqüência das modificações realizadas. Apenas na parte de software são necessárias algumas alterações:

- algumas informações a mais devem ser trocadas na hora de dividir o processamento entre processadores; e
- um controle um pouco mais elaborado nos interpretadores para que estes sejam capazes de definir quais objetivos serão processados com paralelismo AND Restrito e combinar as soluções de tais objetivos.

As outras características do modelo KABU-WAKE original se mantém inalteradas.

Este modelo foi simulado e os resultados obtidos inicialmente são promissores.

Observando as classificações de Gregory e Conery, podemos dizer que, de acordo

com o primeiro, o KABU-WAKE alterado apresenta o paralelismo OR e o paralelismo AND Restrito. De acordo com Conery, o modelo utiliza o paralelismo OR puro e o paralelismo AND na árvore de objetivos. O paralelismo Stream AND é evitado fazendo com que literais dependentes sejam executados seqüencialmente da esquerda para a direita.

III.4 - Modelo PRISM

O modelo PRISM (KASIF, 1983 e WISE, 1986) apresenta uma concepção muito simples: a busca por soluções pode ser definida como uma exploração direta da árvore de objetivos, em paralelo.

Os objetivos (chamados resolventes) são representados por listas de literais. Quando um literal é escolhido para expansão, e é unificado com uma cabeça de cláusula, os literais do corpo da cláusula são incluídos no objetivo (no lugar do literal inicial), com todos os 'bindings' sendo aplicados ao novo objetivo gerado. No caso de um literal escolhido para expansão ter N cláusulas cujas cabeças sejam unificadas, são gerados N filhos (novos objetivos).

Para ilustrar, digamos que o objetivo pai seja:

```
:- r(X) & q(X,Y)
```

Se expandirmos *r(X)* utilizando as cláusulas:

```
r(1).
r(Z) :- p(Z).
```

Teremos os seguintes objetivos filhos:

```
:- q(1,Y)
:- p(Z) & q(Z,Y)
```

O modelo PRISM foi implementado na arquitetura ZMOB (KASIF, 1983 e WISE, 1986), a qual é composta por 256 microprocessadores interligados sob a forma de anel. Cada nó do anel tem três componentes:

- uma 'extensional data base machine': guarda as cláusulas unitárias sem variáveis;

- uma 'intensional data base machine': guarda as cláusulas não unitárias e as unitárias com variáveis; e
- um solucionador de problemas: define qual literal, em que objetivo, será mandado aos bancos de dados.

Cada solucionador de problemas escolhe um literal e o envia aos seus bancos de dados, para que estes tentem realizar unificações com as suas cláusulas. As unificações que tiverem sucesso são então retornadas ao solucionador que enviou o literal.

Desta forma, o problema do número de processos OR é resolvido, uma vez que o número de processos fica limitado pelo número de solucionadores de problemas.

Existem ainda, no modelo PRISM, os chamados nós AND, os quais são criados para resolver o problema dos 'bindings' de literais que compartilham variáveis. Os literais que têm variáveis em comum são os filhos do nó AND. Tais literais são executados seqüencialmente, até que as variáveis compartilhadas se tornem 'bound'. A partir dai, os literais que sobraram podem ser executados em paralelo.

O paralelismo de busca seria, segundo Conery, a forma de paralelismo que PRISM utiliza. Para Gregory, este modelo apresenta os paralelismos: All-solutions AND 1, OR e AND Restrito.

III.5 - Modelo Parallel Inference Engine (PIE)

O modelo PIE (MOTO-OKA, 1984 e WISE, 1986) segue a mesma idéia que a 'OR Parallel Token Machine' (CIEPIELEWSKI, 1984 e WISE, 1986) e o modelo PRISM, apresentado na seção anterior: utilizar os corpos de cláusulas para literais de um objetivo para gerar novos objetivos filhos. No entanto, existe uma diferença, entre os dois primeiros modelos e o último, na manutenção dos 'bindings' de variáveis. Na PIE e na 'Token Machine' os 'bindings' realizados nas expansões de literais não são aplicados aos outros literais dos objetivos.

Assim, observando o exemplo da seção anterior, teríamos os seguintes objetivos filhos:

```
:- q(X,Y)
:- p(Z) & q(X,Y)
```

No modelo PIE, existem os chamados 'goal frames', os quais são compostos pelos objetivos e pela lista de 'bindings' que foram feitos nos mesmos. O gerenciamento dos 'goal frames' utiliza o compartilhamento de dados para ambientes de pais e filhos, e procura reduzir tais ambientes eliminando informações desnecessárias. Este método é chamado modelo de re-escrita de objetivos.

A arquitetura da máquina PIE (MOTO-OKA, 1984) apresenta 1024 processadores, e é definida em dois níveis. O nível 2 gerencia as atividades dos seus componentes, os sistemas de nível 1. Cada sistema de nível 1 contém sete modulos e três redes:

- módulos de memória: onde os 'goal frames' ficam armazenados;
- memórias de definição: guardam cópias completas do programa;
- memória de estruturas: guarda estruturas que contém apenas termos 'ground';
- processadores de unificação: realizam as expansões de literais;
- 'lazy-fetch buffers': são buffers utilizados para reduzir o número de acessos às memórias de estruturas;
- controladores de atividade: coordenam as atividades dos processadores de unificação;
- gerente de atividade: fica acima dos controladores de atividade, gerenciando o fluxo de objetivos no sistema;
- rede de distribuição: por onde passam os objetivos transferidos entre módulos de memória;
- rede de 'lazy-fetch': por onde passam os dados relativos às memórias de estruturas; e
- rede de comandos: liga os controladores de atividade ao gerente de atividade.

Um outro ponto importante deste modelo é a política de retirada e inclusão de objetivos. A estratégia utilizada faz com que os processadores de unificação retirem

objetivos dos seus próprios módulos de memória, e incluam os novos objetivos de volta nos mesmos, sempre que possível. Será possível incluir um objetivo num módulo de memória, se o número de objetivos já existentes no módulo específico for menor que um certo limite. Caso não seja, será escolhido randomicamente outro módulo de memória que possa receber o objetivo.

Da mesma forma que nos modelos anteriores, o número de processos é controlado pelo número de processadores (processadores de unificação).

Na classificação de Gregory, o paralelismo utilizado neste modelo é o paralelismo OR. Para Conery, esta forma de paralelismo se chama paralelismo OR puro.

III.6 - Modelo de Processos AND/OR

Este modelo foi proposto por Conery (CONERY, 1987 e WISE, 1986). O Modelo de Processos AND/OR é bastante simples. Como seu nome diz, existem apenas dois tipos de processos, chamados AND e OR. Um processo AND é criado para resolver o corpo de uma cláusula, criando por sua vez um processo OR para cada literal do corpo da cláusula. Caso um processo OR encontre uma regra na solução de um literal, ele criará um processo AND para a regra. Desta forma, a computação se resume a um conjunto de processos AND e OR organizados numa árvore AND/OR, resolvendo pequenos pedaços do programa lógico. A comunicação entre os processos é realizada através de mensagens, as quais são usadas na criação, cancelamento, pedido e retorno de resultados de processos.

As principais características deste modelo são (DUTRA, 1989):

- Não utiliza diretivas para processamento concorrente no programa fonte (sintaxe semelhante a de Prolog).
- Utiliza geradores ('generators') para variáveis compartilhadas por literais no corpo das regras. Existem algoritmos que especificam literais geradores de variáveis, os quais são executados antes dos literais que consomem tais variáveis.
- Permite a utilização de diretivas de modo ('mode') no programa fonte. Tais diretivas facilitam a escolha de geradores para as variáveis.
- Pode alcançar um bom grau de paralelismo.

- Não utiliza o operador corte (DUTRA, 1989).

As maiores críticas feitas a este modelo se concentram na incorreção do mecanismo de 'backtracking' inteligente e na ineficiência do algoritmo dinâmico de ordenação de literais (determinação de geradores). Algoritmos para solucionar o primeiro problema são encontrados em Woo e Choe (WOO, 1986), e Chang, Despain e DeGroot (CHANG, 1985b). Como solução para a ineficiência da ordenação de literais encontramos as propostas de Chang e Despain (CHANG, 1985a) e DeGroot (DEGROOT, 1984).

Segundo a definição de Conery, este modelo utiliza os processos OR e os processos AND. No contexto das formas de paralelismo descritas por Gregory, pode-se dizer que este modelo utiliza o paralelismo OR, o paralelismo All-solutions AND 1 e o paralelismo AND Restrito. O paralelismo Stream AND é evitado através da utilização de geradores para as variáveis compartilhadas.

CAPÍTULO IV

O Modelo Gol-Log

Este capítulo descreve um novo modelo de execução paralela, chamado Gol-Log. As principais características do modelo são:

- orientado à execução de programas lógicos puros, isto é sem a utilização de recursos como diretivas de processamento concorrente;
- utiliza uma estratégia de controle "inteligente" (padrão); e
- utiliza o paralelismo OR.

Chamamos a estratégia de "inteligente", pois ela tenta, sempre com uma visão global da árvore de objetivos, descobrir qual o melhor caminho (aquele que mais rapidamente levará a uma solução – sucesso ou falha) a seguir.

A descrição de Gol-Log se concentra nos objetivos que nortearam a sua criação, no modelo abstrato de Gol-Log, na proposta de como implementá-lo e no que julgamos serem as inovações trazidas por Gol-Log.

IV.1 - Objetivos do Projeto Gol-Log

O principal objetivo de qualquer proposta de modelo de execução paralela é obter eficiência de execução, não apenas em comparação às implementações seqüenciais, como também, em relação às outras implementações de modelos paralelos. Entretanto, é fundamental, na nossa opinião, que a busca de alto desempenho não traga novos problemas, tais como a dificuldade de leitura dos programas e a sua utilização em pequena escala.

Um outro objetivo que deve ser perseguido em qualquer proposta é a simplicidade conceitual. A concepção de Gol-Log segue obstinadamente este objetivo.

Também de grande importância para a proposta de Gol-Log, foi o objetivo de especificar a sua implementação com base numa arquitetura de computador paralelo já existente. Nossa intenção com isto é, não só facilitar a sua implementação, como também, permitir a avaliação do seu desempenho em situações reais.

Os outros objetivos que nortearam a proposta de Gol-Log já foram citados acima, são eles:

- aplicar uma estratégia de controle com uma visão global da árvore de objetivos; e
- encontrar soluções viáveis para os problemas do paralelismo OR (explosão do número de processos OR e gerenciamento dos ambientes de variáveis).

IV.2 - O Modelo Abstrato

A representação gráfica do modelo abstrato de Gol-Log é apresentada na figura IV.1.

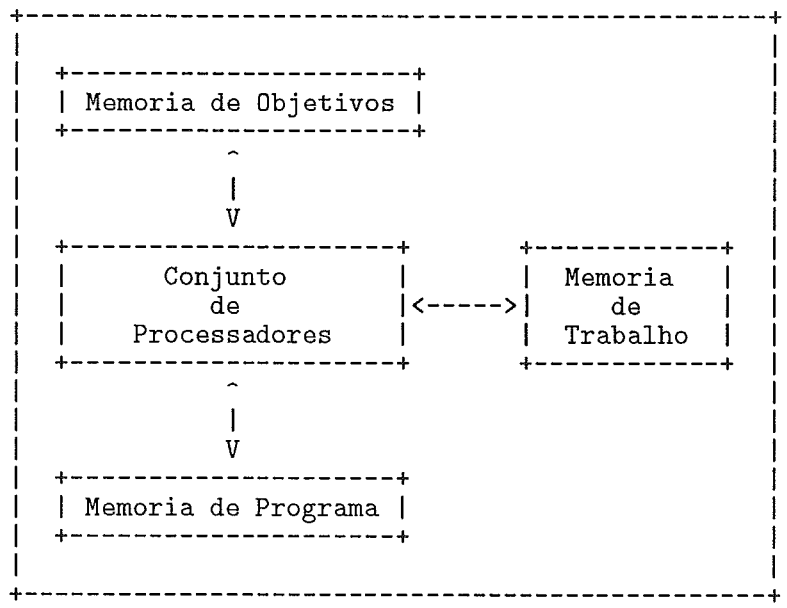


Figura IV.1: Modelo Lógico de Gol-Log

A memória de objetivos guarda os nós da árvore de objetivos que ainda não foram expandidos. Os processadores retiram objetivos desta memória, expandem os mesmos com base no programa, e colocam os novos objetivos de volta. Os novos objetivos são formados pelo objetivo inicial, com um literal substituído pelo corpo da cláusula que foi usada na sua expansão, e pelos 'bindings' de variáveis realizados na expansão. Desta forma, observamos o paralelismo OR na execução dos programas.

A exploração direta da árvore de objetivos, como em Gol-Log, é uma carac-

terística de alguns dos modelos descritos no capítulo III: KABU-WAKE (seção III.2), KABU-WAKE Alterado (seção III.3), PRISM (seção III.4) e PIE (seção III.5).

Algumas características de Gol-Log tornam-se aparentes desde já:

- o objetivo da simplicidade conceitual foi plenamente alcançado;
- o número de processos OR é limitado pelo número de processadores no conjunto de processadores;
- a carga está sempre balanceada entre os processadores; e
- pode ocorrer alguma computação repetida pela não utilização de paralelismo AND.

IV.3 - Estratégias de Controle

IV.3.1 - A Estratégia "Inteligente" (Padrão)

Um dos principais objetivos de Gol-Log é reduzir a necessidade de computação para alcançar as soluções. Para isto, procura-se adotar uma estratégia de controle "inteligente", a qual será implementada através de heurísticas que tentarão escolher o melhor caminho a seguir, a cada momento, com base em informações globais sobre o estado da busca.

A utilização de informações globais para guiar a avaliação é também uma característica do modelo B-Log (seção III.1).

Em Gol-Log usaremos dois níveis de heurísticas:

- nível de escolha de objetivo a expandir, que denominaremos primeiro nível; e
- nível de escolha de literal a expandir dentro do objetivo escolhido para expansão, que denominaremos segundo nível.

O primeiro nível de heurísticas permite que seja escolhido o objetivo que mais rapidamente deverá chegar a uma solução (sucesso ou falha). Já o segundo nível, permite que se escolha um literal que deverá gerar um menor espaço de busca.

Como exemplo de modelo que utiliza o primeiro nível de heurísticas, temos o modelo PIE (seção III.5), com a estratégia chamada 'conclusion-precedence'. O Modelo de Processos AND/OR (seção III.6) usa uma heurística de segundo nível, no procedimento de ordenação de literais.

Para propor as heurísticas utilizadas no modelo Gol-Log, foram consideradas as seguintes premissas:

- objetivos com maior porcentagem de variáveis instanciadas levarão às folhas (soluções) mais rapidamente;
- objetivos com menor número de literais levarão mais rapidamente às folhas;
- literais com variáveis compartilhadas, se executados inicialmente, reduzirão o espaço de busca; e
- literais com menor número de cláusulas aplicáveis à sua solução podem potencialmente gerar um espaço de busca menor.

Para o primeiro nível de heurísticas temos as seguintes possibilidades:

- busca em profundidade;
- busca em largura;
- busca pelo melhor ('best-first search'); e
- busca aleatória.

A heurística definida para o primeiro nível de Gol-Log, chamada heurística 1 ou H1, é uma mistura de busca em largura com uma busca pelo melhor e segue os seguintes passos:

Passo 1) Escolher o objetivo com menor número de variáveis não instanciadas;

Passo 2) Em caso de empate no passo 1), escolher o objetivo com menor número de literais; e

Passo 3) Em caso de empate no passo 2), escolher o objetivo que proporcionar uma busca em largura.

Para o segundo nível de heurísticas temos as seguintes possibilidades:

- escolha do literal mais a esquerda ('leftmost');
- escolha do literal mais a direita ('rigthmost');
- escolha do melhor literal; e
- escolha aleatória.

A heurística definida para o segundo nível do modelo, chamada heurística 2 ou H2, é uma mistura da escolha do literal mais a esquerda e a escolha do melhor e obedece aos passos seguintes:

Passo 1) Escolher o literal com menor número de variáveis não instanciadas;

Passo 2) Em caso de empate no passo 1), escolher o literal com menor número de cláusulas aplicáveis; e

Passo 3) Em caso de empate no passo 2), escolher o literal mais a esquerda.

É importante notar que H2 só deve ser utilizada quando os literais do objetivo forem independentes, isto é, não tiverem variáveis compartilhadas. Caso contrário, os literais dependentes devem ser escolhidos, inicialmente, da esquerda para a direita.

Note-se que o aumento da eficiência de execução obtido com a utilização de heurísticas pode não ser alcançado, uma vez que:

- as heurísticas podem não ser boas para uma classe de aplicações particular;
- existe a necessidade de processar tais heurísticas, isto é, pode-se perder o que foi ganho pelo melhor percurso na árvore de busca se o tempo de computação das heurísticas não for pequeno o suficiente; e
- a implementação torna-se um pouco mais complexa.

IV.3.2 - Outras Estratégias

Vale ressaltar que as heurísticas descritas na seção anterior especificam a estratégia de controle padrão de Gol-Log. Existem outras estratégias mais simples que também podem ser utilizadas de acordo com a conveniência do usuário:

- busca em largura para escolha de objetivos e mais a esquerda para escolha de literais;
- a estratégia LRDF do Prolog seqüencial; e
- busca "mista" em profundidade e largura para escolha de objetivos e mais a esquerda para escolha de literais.

Chamamos busca "mista" à estratégia de controle que procura realizar uma busca em profundidade, mas, quando existem processadores ociosos, permite alguma busca em largura.

Existe uma característica comum a todas as estratégias adotadas pelo modelo Gol-Log: os sucessos são apresentados assim que estejam disponíveis, independentemente da seqüência estabelecida pela estratégia de controle.

A possibilidade de escolher entre várias estratégias de controle é uma característica também apresentada pelo modelo KABU-WAKE Alterado (seção III.3).

IV.3.3 - Controle de Ramos Infinitos da Árvore de Busca

Uma propriedade das estratégias de controle de Gol-Log é a capacidade de evitar que a exploração de possíveis ramos infinitos na árvore de busca prejudique significativamente o desempenho do modelo, na solução dos programas.

A forma de evitar que se siga indefinidamente um ramo infinito da árvore é implementada, em Gol-Log, fazendo com que a todo o literal expandido (e aos valores dos seus argumentos) se aplique uma função de 'hash'. O valor gerado por esta função de 'hash' deve, então, ser guardado para que quando um novo literal for escolhido para expansão, seu 'hash' seja computado, visando verificar se o literal já foi expandido. Caso ele já tenha sido expandido, deve-se incrementar o número de vezes que o literal foi expandido, até que, ao se ultrapassar um limite pré-estabelecido, se considere que existe um ramo infinito

na árvore. Para sair do ramo infinito, é suficiente abandonar o objetivo de onde veio o literal.

É importante observar que esta aproximação não é capaz de detectar qualquer ramo infinito na árvore de busca. Existem situações onde, as vezes por um erro de programação, seja criado um ramo infinito que Gol-Log não tem condições de descobrir. Para ilustrar esta questão, apresentamos alguns exemplos de programas que geram ramos infinitos de busca:

Exemplo 1:

```
? p(X).

p(X) :- a(X).

a(X) :- p(X).
```

Exemplo 2:

```
? fat(-1,X).

fat(0,1).
fat(X,Y) :- X1 is X - 1, fat(X1,Z), Y is Z * X.
```

Gol-Log pode detectar um ramo infinito no primeiro exemplo, uma vez que os objetivos (e os valores de seus argumentos) se repetem. No entanto, o segundo exemplo não permite que as estratégias de controle de Gol-Log reconheçam a computação infinita, uma vez que os objetivos sempre mudam: *fat(-1,X)*, *fat(-2,Z)*, *fat(-3,Z)*, etc.

IV.4 - Problemas com o Paralelismo OR

Como observado na seção IV.2, o primeiro dos problemas com o paralelismo OR foi naturalmente resolvido. O número de processos OR é limitado pelo número de processadores utilizados (seção V.2).

Tal característica é verificada também nos modelos KABU-WAKE (seção III.2), KABU-WAKE Alterado (seção III.3), PRISM (seção III.4) e PIE (seção III.5).

Quanto ao segundo problema, o gerenciamento de ambientes de variáveis, Gol-Log usa uma estratégia de cópia de dados entre ambientes de objetivos pais e filhos, com os 'bindings' já realizados sendo aplicados ao resto dos objetivos.

Um objetivo qualquer na memória de objetivos é acompanhado por um ambiente de variáveis que será copiado para os seus filhos (sem os 'bindings' desnecessários). Um objetivo que tenha sido expandido pode ter seu ambiente apagado sem problemas. Desta forma, não é possível a geração de 'bindings' conflitantes para variáveis. No entanto, existem duas desvantagens nesta aproximação:

- há desperdício de memória pelo não compartilhamento de dados dos ambientes. Cada objetivo terá um ambiente para as variáveis da consulta ao programa; e
- há uma grande necessidade de transferência de dados para executar um objetivo, uma vez que é necessário transferir o ambiente da consulta para o processador que realizará a expansão de um dos literais do objetivo.

As vantagens obtidas com a não utilização do compartilhamento de dados, além da não geração de 'bindings' conflitantes, são:

- a maior simplicidade de implementação; e
- uma maior independência entre os objetivos.

A aplicação de 'bindings' já realizados, nos objetivos restantes, também é uma característica do modelo PRISM (seção III.4). O modelo PIE (seção III.5) usa uma forma de gerenciamento de ambientes semelhante a Gol-Log, no entanto, a nossa proposta utiliza a cópia de dados entre ambientes, enquanto a PIE usa o compartilhamento de dados.

IV.5 - Implementação do Modelo

IV.5.1 - Arquitetura Paralela

Uma arquitetura paralela para implementar o modelo abstrato de Gol-Log deve ter as seguintes características básicas:

- memória global compartilhada, onde deve ficar a memória de objetivos;
- memórias locais, onde devem estar cópias inteiras do programa;

- uma topologia que não cause uma contenção excessiva no acesso à memória compartilhada; e
- estar implementada e facilmente disponível.

Assim, uma possibilidade seria o computador paralelo BBN Butterfly (LEBLANC, 1988), que possui até 256 nós conectados por uma rede de chaveamento de alta velocidade. Toda a memória é distribuída pelos nós (até 4 Mbytes em cada nó), mas qualquer memória pode ser acessada por qualquer processador, através da rede (figura IV.2).

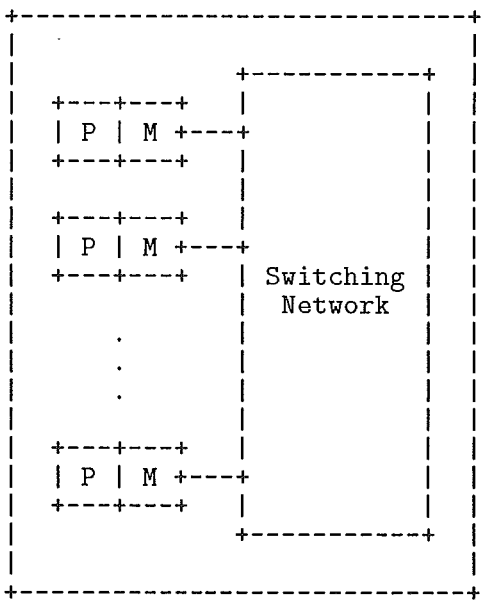


Figura IV.2: Arquitetura da BBN Butterfly

Como vantagens na utilização desta arquitetura paralela, temos:

- possibilidade de uso do modelo em larga escala; e
- possibilidade de chegarmos a um ambiente com grande número de processadores e grande quantidade de memória.

Existem, no entanto, duas desvantagens:

- possibilidade de acontecer alguma contenção no acesso à memória de objetivos; e
- o número de passos, na rede de chaveamento, necessários para alcançar as memórias remotas a um processador.

IV.5.2 - Estrutura Geral da Implementação

O algoritmo que deve ser executado nos processadores pode ser descrito, em alto nível, da seguinte forma:

```
{
    Aplicar H1 para escolher o objetivo a expandir;
    Retira o objetivo da memória de objetivos;
    Verificar a existência de literais dependentes;
    Se (existirem literais dependentes)
    {
        Escolher o mais a esquerda;
    }
    senão
    {
        Aplicar H2 para escolher o literal a expandir;
    }
    Aplicar função de 'hash' ao literal escolhido;
    Se (já foi expandido)
    {
        Se (número de expansões + 1 > LIMITE)
        {
            Aborta;
        }
        Acumula novo número de expansões;
    }
    Enquanto (houver cláusulas para o literal escolhido)
    {
        Expande;
        Se (não for falha)
        {
            Armazena o novo objetivo;
        }
```

}

Retorna os objetivos novos para a memória de objetivos;

}

IV.5.3 - Estruturas de Dados

Memória de Objetivos

A estrutura da memória de objetivos procura reduzir a possível contenção gerada pelos vários processadores ao tentar obter objetivos para expandir, na arquitetura proposta.

A memória de objetivos pode ser implementada como uma matriz de ponteiros para listas encadeadas de objetivos. As listas encadeadas armazenam objetivos (e seus 'bindings' de variáveis) com as mesmas características. Para referenciar o inicio de uma destas listas deve-se usar:

matriz (< no. de vars. não instanciadas >, < no. de literais >)

É importante citar duas das características desta forma de implementação:

- é indicada para a estratégia padrão de Gol-Log. No caso da utilização uma das outras estratégias de controle é necessária outra estrutura de implementação; e
- ela dificulta muito a utilização do terceiro critério de escolha de objetivos em H1.

Para ilustrar a matriz de objetivos, apresentamos a figura IV.3.

As posições marcadas com "X" devem conter informações descrevendo o estado dos ponteiros da sua linha. Cada linha da matriz deve se encontrar em uma memória local diferente.

Para as outras estratégias de controle, que não a padrão, deve ser utilizada uma lista encadeada simples, para o armazenamento dos objetivos, como apresentado na figura IV.4.

Memória de Programas

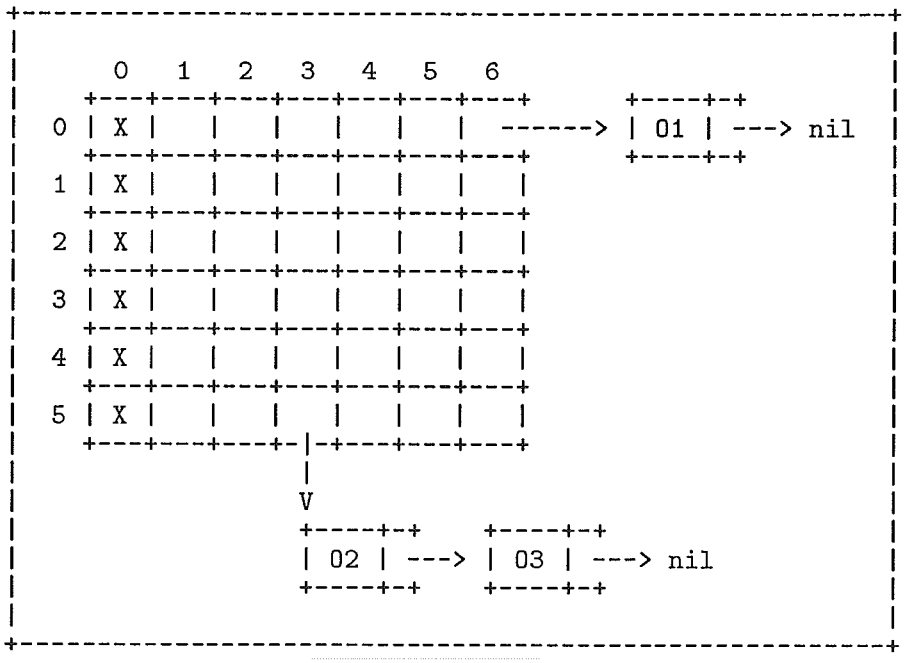


Figura IV.3: Matriz de objetivos

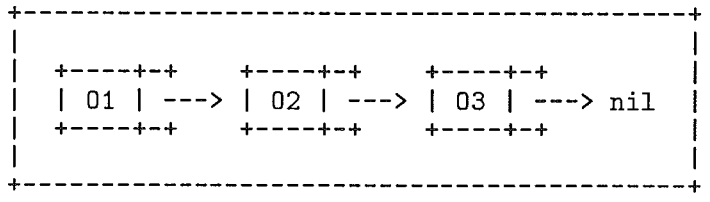


Figura IV.4: Lista simples de objetivos

A estrutura dos programas é, basicamente, semelhante a que é utilizada nas implementações de Prolog seqüencial (seção II.1.4).

As seguintes estruturas compõem a representação dos programas:

- lista de cabeças de cláusulas: cada elemento desta lista inclui a aridade da cláusula, a lista de argumentos da cláusula, um ponteiro para o corpo da cláusula e um ponteiro para a próxima cláusula do predicado;
- lista de literais de corpo de cláusula: cada elemento desta lista inclui um ponteiro para a definição do literal, o número de argumentos do literal, um ponteiro para a lista de argumentos e um ponteiro para o próximo literal;

- tabela de definições de literais: composta pelo nome do literal, o número de argumentos do literal, pelo número de cláusulas aplicáveis ao literal, e pelo ponteiro para a primeira cláusula do literal;
- estruturas para os termos: composta pelo nome do termo, pelo seu número de argumentos e pelo ponteiro para a sua lista de argumentos; e
- estruturas para os 'bindings' de variáveis: composta pela identificação da variável, e por um ponteiro para o termo que representa o seu valor.

Desta forma, o que chamamos objetivo (On) tem uma representação semelhante a um corpo de cláusula. As consultas ao programa são representadas como cláusulas especiais, também ligadas em forma de lista.

Para ilustrar a representação dos programas, apresentamos na figura IV.5, uma estrutura de dados simplificada, que representa um programa com duas regras (cada uma com dois literais no corpo), um fato e uma consulta (com apenas um literal).

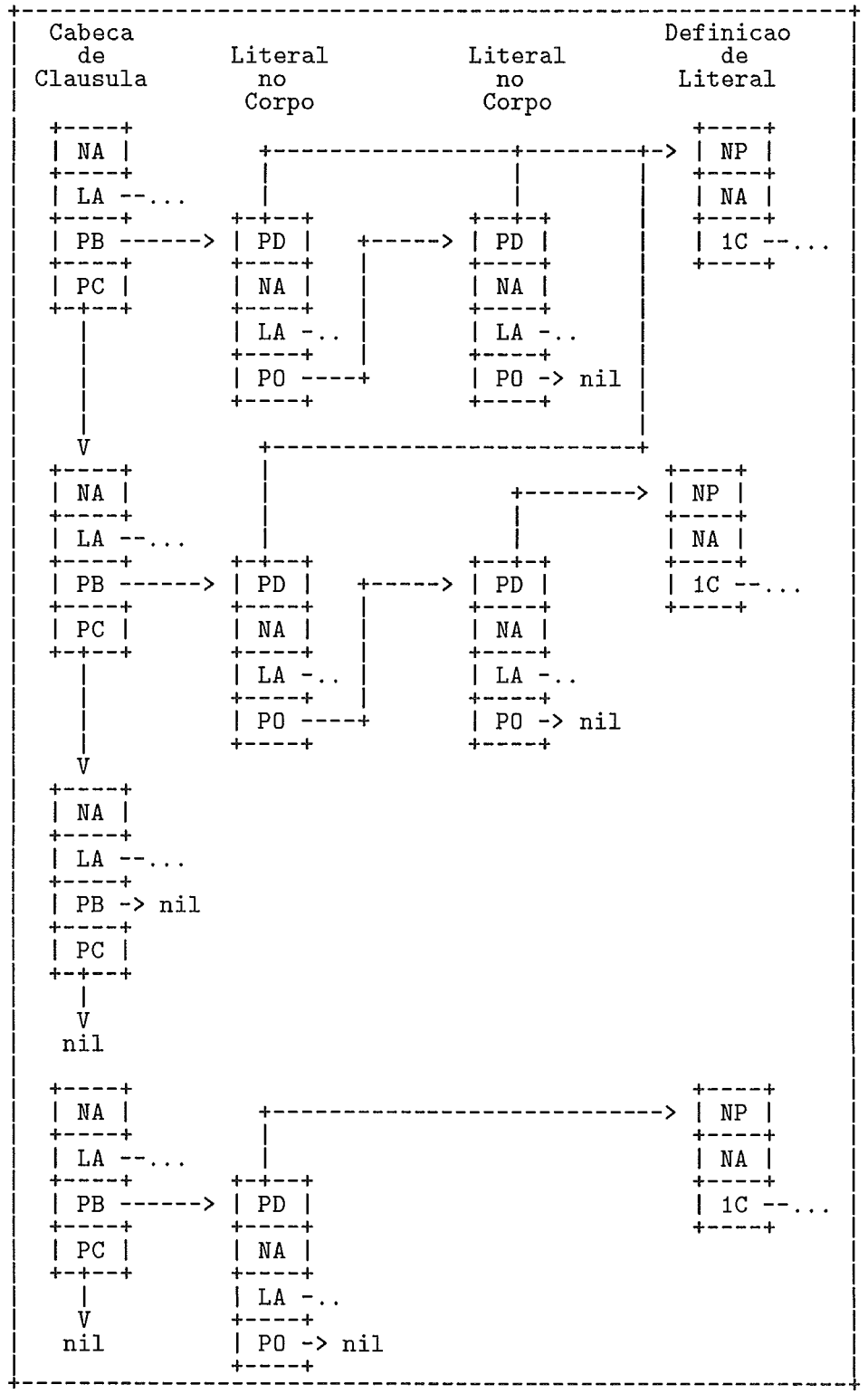
IV.6 - Inovações do Modelo Gol-Log

Consideramos, com Gol-Log, ter produzido algumas inovações, em relação às características encontradas nos modelos de execução paralela de programas lógicos atuais.

É importante notar que os conceitos que consideramos "inovações" não necessariamente são características nunca apresentadas por nenhum modelo. Na verdade, são inovações até onde a literatura consultada possa indicar.

Na parte da estratégia de controle "inteligente", não há indicações de outro modelo que utilize dois níveis de heurísticas. Em geral, os modelos usam apenas heurísticas a nível de objetivos, como na PIE (seção III.5), ou a nível de literais de um objetivo, como no Modelo de Processos AND/OR (seção III.6). As heurísticas H1 e H2 (seção IV.3) também nos parecem novas.

A possibilidade de o usuário escolher a estratégia de controle a usar é também uma característica que nos parece inovadora, uma vez que o outro modelo que sabemos permitir isto é o KABU-WAKE Alterado, também proposto nesta tese, mas ainda não implementado (apenas simulado).



NA - Núm. de Argumentos LA - Lista de Argumentos PB - Pont. para o Corpo

PC - Ponteiro para a Próxima Clausula PO - Ponteiro para o Próximo Objetivo

PD - Pont. para Definição de Literal 1C - Pont. para a 1a. Cláusula para o Literal

Figura IV.5: Exemplo de estrutura de dados para programas

Outra característica da estratégia de controle é o tratamento dado aos possíveis ramos infinitos da árvore de objetivos. Tal preocupação não parece ser comum aos outros modelos até agora propostos.

A proposta de implementação em computadores paralelos disponíveis não é propriamente uma inovação, porém é uma vantagem em relação a maior parte dos modelos, cujas compatibilidades com computadores paralelos existentes não são avaliadas.

Na parte de implementação consideramos que a estrutura da memória de objetivos é completamente nova, tendo sido motivada pelas heurísticas propostas e pela arquitetura da máquina alvo.

CAPÍTULO V

Avaliação Experimental do Modelo Gol-Log

Neste capítulo apresentamos uma implementação de Gol-Log, realizada no Laboratório de Computação Paralela da COPPE/UFRJ.

Devido à indisponibilidade de uma arquitetura paralela de memória compartilhada, como a descrita no capítulo anterior, foram feitas algumas alterações na forma de implementar o modelo. Tais alterações não mudam significativamente as características principais de Gol-Log, e serão apresentadas nas seções seguintes.

V.1 - Implementação numa Arquitetura de Memória Distribuída

O ambiente de processamento que utilizamos nesta implementação de Gol-Log é composto por: 4 processadores Transputer T-800 (10 MIPS cada), com 1 Mbyte de memória cada, ligados em forma de estrela, e utilizados através de um micro PC (figura V.1). A linguagem usada para a implementação do modelo foi o Parallel C da 3L.

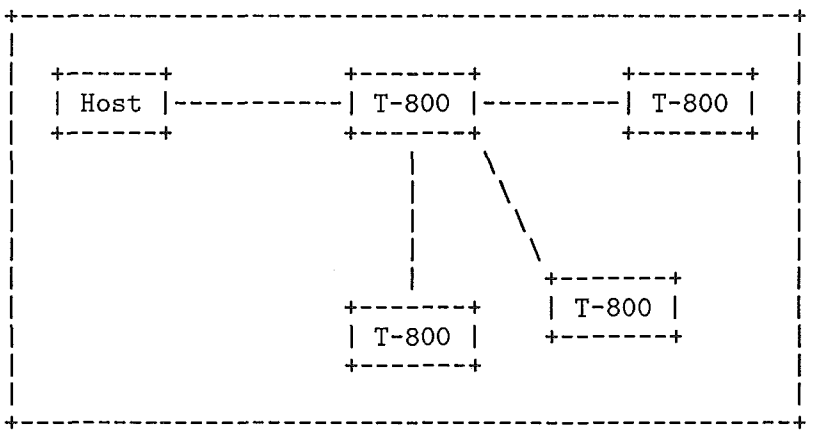


Figura V.1: Topologia de ligação da implementação

Vale notar que o número de Transputers utilizados não precisa ser 4, necessariamente. Podemos aumentar o número de processadores facilmente, sendo preciso apenas reconfigurar o sistema. Esta reconfiguração não pode, entretanto, modificar a topologia em forma de estrela.

V.2 - Alterações no Modelo

Algumas alterações tiveram que ser feitas em Gol-Log, devido à utilização de uma arquitetura totalmente diferente da proposta. Outras alterações foram realizadas para facilitar a depuração do sistema. É importante notar, no entanto, que todas as alterações foram realizadas procurando manter as características originais de Gol-Log.

A principal alteração realizada foi a utilização de um processador, chamado 'pool manager' (PM), para controlar o acesso à memória de objetivos. É importante observar que uma alteração deste tipo era inevitável, uma vez que não existe memória compartilhada na arquitetura apresentada na seção anterior, e não queríamos deixar de utilizar uma estratégia de controle global.

Desta forma, para acessar a memória de objetivos é necessário que os processadores, chamados 'workers' (WP), se comuniquem com o PM, pedindo objetivos a expandir. Assim, o PM recebe pedidos de trabalho, executa H1 e envia objetivos aos WP. Os WP recebem um objetivo a cada momento, executam H2 e retornam os objetivos filhos ao PM, pedindo mais objetivos a seguir.

A estrutura da implementação fica, então, como mostrado na Figura V.2.

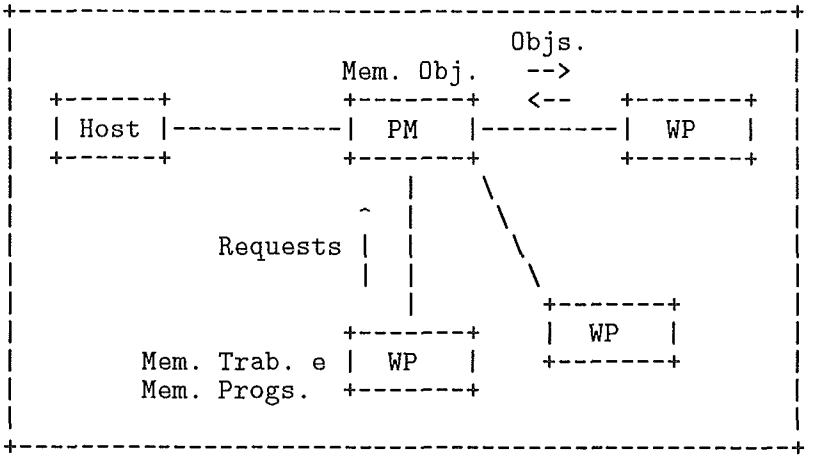


Figura V.2: Esquema da Implementação Distribuída

Como conseqüências desta alteração destacam-se:

- a grande necessidade de memória no processador PM; e

- a grande possibilidade de contenção devido a necessidade de comunicação de vários WP com um PM. Observe que a expansão de um literal de um objetivo causa três comunicações (WP → PM → WP → PM).

Para o caso de termos um PM com pouca memória, em relação aos programas executados, teremos um problema equivalente ao estouro do número de processos, em sistemas utilizando paralelismo OR. Tal problema deve ser bastante reduzido pela utilização de H2, mas mesmo assim pode existir um "estouro de memória".

Uma forma de minimizar este problema, e reduzir a comunicação necessária para a solução dos programas, seria repartir a memória de objetivos por todos os processadores e utilizar uma estratégia de controle local, a partir do momento que se chegasse à saturação da memória do PM.

Outra diferença em relação à proposta inicial de Gol-Log é a necessidade de modificar o algoritmo apresentado anteriormente, e de criar um novo algoritmo para o PM, o qual será descrito na seção seguinte.

V.3 - Implementação

Como mencionado acima foram necessárias alterações no algoritmo apresentado na seção IV.5.2. Assim, aquele algoritmo, o qual será executado nos WP, fica:

```
{
    Pedir um objetivo ao PM;
    Verificar a existência de literais dependentes;
    Se (existirem literais dependentes)
    {
        Escolher o mais a esquerda;
    }
    senão
    {
        Aplicar H2 para escolher o literal a expandir;
    }
    Aplicar função de 'hash' ao literal escolhido;
```

```
    Se (já foi expandido)
    {
        Se (número de expansões + 1 > LIMITE)
        {
            Aborta;
        }
        Acumula novo número de expansões;
    }
    Enquanto (houver cláusulas para o literal escolhido)
    {
        Expande;
        Se (não for falha)
        {
            Retorna o novo objetivo ao PM;
        }
    }
}
```

O novo algoritmo, executado no PM, é o seguinte:

```
{
    Enquanto (memória de objetivos não vazia OU
                                    algum WP estiver trabalhando)
    {
        Se (houver pedido de objetivo)
        {
            Recebe pedido;
            Aplicar H1 para escolher o objetivo;
            Retira o objetivo da memória de objetivos;
            Retorna o objetivo escolhido ao WP que pediu;
        }
        Se (houver novo objetivo sendo retornado)
        {
```

```
            Recebe novo objetivo;
            Inclui novo objetivo na memória de objetivos;
        }
        Se (houver sucesso sendo retornado)
        {
            Imprimir solução;
        }
    }
}
```

Quanto as estruturas de dados apresentadas na seção IV.5, algumas alterações foram feitas na nossa implementação de Gol-Log. Basicamente, as alterações foram:

- a não utilização de uma tabela única de definições (podem existir definições repetidas);
- o uso de uma estrutura um pouco maior para os termos, incluindo informações como o tipo do termo (variável, lista, etc), por exemplo;
- a utilização de um campo de definição e a não utilização do ponteiro para a próxima cláusula do predicado, na estrutura das cabeças de cláusulas; e
- a criação de uma lista de 'headers' para as cláusulas (esta lista liga os 'headers' de todas as cláusulas do programa).

As outras características de implementação do modelo (seção IV.5.3) ficam mantidas.

V.4 - Resultados

Nesta seção procuramos avaliar o desempenho do modelo Gol-Log, através dos resultados de execução de vários programas na implementação que desenvolvemos.

O que queremos avaliar, principalmente, é a qualidade da estratégia de controle padrão de Gol-Log e o desempenho da nossa implementação, em comparação às outras estratégias de controle fornecidas. Desejamos também estudar o provável comportamento

do sistema, quando implementado no tipo de arquitetura que consideramos ideal para suportar o modelo.

Os resultados que utilizamos foram obtidos através da execução de um 'benchmark' contendo 10 programas, divididos em cinco classes distintas, assim como apresentado por Ciepielewski (CIEPIELEWSKI, 1986).

Os programas utilizados foram retirados do livro texto de Prolog (CLOCKSIN, 1981), do artigo citado acima (CIEPIELEWSKI, 1986), e de uma das teses sobre implementação de Prolog, realizada na COPPE Sistemas/URFJ (DUTRA, 1988a).

Os programas do 'benchmark' estão divididos nas seguintes classes:

- programas básicos: tais programas são, geralmente, partes de outros programas maiores;
- programas seqüenciais: programas que realizam computações intrinsicamente seqüenciais;
- programas paralelisados: são programas obtidos pela paralelização de programas seqüenciais;
- programas de busca com uma solução: programas que realizam busca extensiva para encontrar uma única solução; e
- programas de busca com mais de uma solução: também apresentam busca extensiva sobre um banco de dados, mas têm mais de uma solução.

As medidas fornecidas pela nossa implementação de Gol-Log são as seguintes:

- o tempo, em pulsos de relógio, gasto para encontrar cada sucesso (incluindo a saída de tais sucessos e das medidas);
- o número de unificações, de literais com cabeças de cláusulas e de termos em literais com termos em cabeças de claúsulas, realizadas a cada sucesso;
- o tamanho máximo da memória de objetivos, em número de objetivos, a cada sucesso;
- o tamanho máximo de objetivos, em número de literais, a cada sucesso;

- o número de sucessos já alcançados, a cada sucesso;
- o número de falhas já alcançadas, até cada sucesso;
- o número de comunicações necessárias para encontrar cada sucesso;
- o número de bytes transferidos nas comunicações, a cada sucesso;
- o número de acessos à memória dinâmica (memória de objetivos), a cada sucesso; e
- o número de acessos à memória estática (programa), a cada sucesso.

É importante notar que os sucessos podem aparecer em ordem diferente para cada estratégia de controle.

Dentre tais medidas comentaremos alguns pontos fundamentais. O tempo que medimos não representa muita coisa como medida absoluta, uma vez que: inclui o tempo de saída de sucessos, o sistema ainda contém recursos fornecidos visando apenas a depuração mais simples, e a própria computação das medidas toma tempo. A medida de tempo só nos será útil de forma relativa, comparando-se as várias estratégias de controle utilizadas.

O número de unificações é a nossa principal fonte de avaliação da computação necessária para encontrar cada sucesso, e da computação que deve ser realizada para achar todos os sucessos. O número de unificações nos será bastante útil, na avaliação das heurísticas de Gol-Log, uma vez que nos dará uma medida do tamanho do espaço de busca que se explorou para encontrar os sucessos. É importante citar que, tanto a busca em profundidade, quanto a busca mista, na nossa implementação, percorrem o banco de cláusulas de baixo para cima. Tal fato pode, portanto, causar pequenas diferenças no número de unificações, em relação ao esperado.

O tamanho máximo da memória de objetivos, o tamanho máximo de objetivos, o número de acessos à memória dinâmica e o número de acessos à memória estática, nos permitem antecipar certas características da execução dos programas numa arquitetura paralela de memória compartilhada distribuída ('distributed shared memory parallel architecture').

O número de comunicações e o número de bytes transferidos entre processadores não nos dão informações "limpas" sobre o modelo, uma vez que tais medidas são bastante

influenciadas pelas alterações realizadas na implementação que fizemos.

Note-se ainda, que, com a busca em largura executando com apenas um WP, além do PM, o tamanho máximo da memória de objetivos nos dá a informação do potencial máximo de exploração de paralelismo OR (largura máxima da árvore). E, com a busca em profundidade, também executando com apenas um WP, o tamanho máximo de objetivos nos informa sobre o potencial máximo para o paralelismo AND.

A avaliação de Gol-Log será feita em três fases. Na primeira será avaliada a estratégia "inteligente" do modelo, através da comparação do número de unificações realizadas pelas diversas estratégias de controle (busca em profundidade, busca em largura e busca "inteligente"). A obtenção destas medidas foi feita utilizando-se apenas um WP, além do PM, para que elas não fossem distorcidas pelo paralelismo.

A segunda fase de avaliação de Gol-Log, basicamente, compara os resultados das medidas de cada programa, executados em paralelo (com vários WPs) com cada uma das estratégias de controle permitidas pela implementação do modelo (busca em largura, busca em profundidade de Prolog, busca mista em profundidade e largura, e busca "inteligente").

Na terceira fase de avaliação procuraremos tirar conclusões sobre o potencial de contenção na execução dos programas, na arquitetura que consideramos mais adequada ao modelo Gol-Log. Tais conclusões serão baseadas na análise dos resultados de execução utilizando mais de um WP, mais especificamente, na relação entre o número de acessos à memória dinâmica, por unificação.

A partir de agora, estaremos apresentando os programas que compõem o 'benchmark', e o resultado das comparações que fizemos.

Programas Básicos

Nesta classe estaremos avaliando o comportamento dos programas *concat* e *permute*. Tais programas, assim como outros, são fontes de paralelismo OR em aplicações reais.

```
concat ([ ], L, L).
concat ([X|Y], Z, [X|L]) :- concat (Y, Z, L).
```

? concat (X, Y, [a, b, c, d]).

O programa *concat* deve fornecer todos os conjuntos de listas, que quando concatenados formam a lista fornecida como terceiro argumento da consulta.

Os resultados de execução deste programa somente confirmaram nossas expectativas, para a primeira fase de avaliação (gráfico V.1). A busca "inteligente" se comportou exatamente como uma busca em largura (o número de unificações para a busca em profundidade é maior, pois a busca é feita de baixo para cima). Este fato já era esperado, uma vez que o número máximo de objetivos na memória de objetivos é 1 (H1 não tem escolha) e o tamanho máximo de objetivos é 1 (H2 não tem escolha).

Analisando os resultados de tempo de execução (gráfico V.2) das implementações com mais de um WP (segunda fase de avaliação), podemos observar que, das quatro estratégias de controle, a busca em largura e a busca mista acabam de fornecer os sucessos mais rapidamente.

A terceira fase de avaliação mostra que a busca em profundidade realiza um menor número de acessos à memória dinâmica por unificação que as outras estratégias (tabela V.1). Desta forma, aquela busca tem um potencial menor para causar contenção no acesso à memoria compartilhada, na execução deste programa numa arquitetura de memória compartilhada.

Grafico V.1 - Resultados de Execucao de concat (1a. fase)

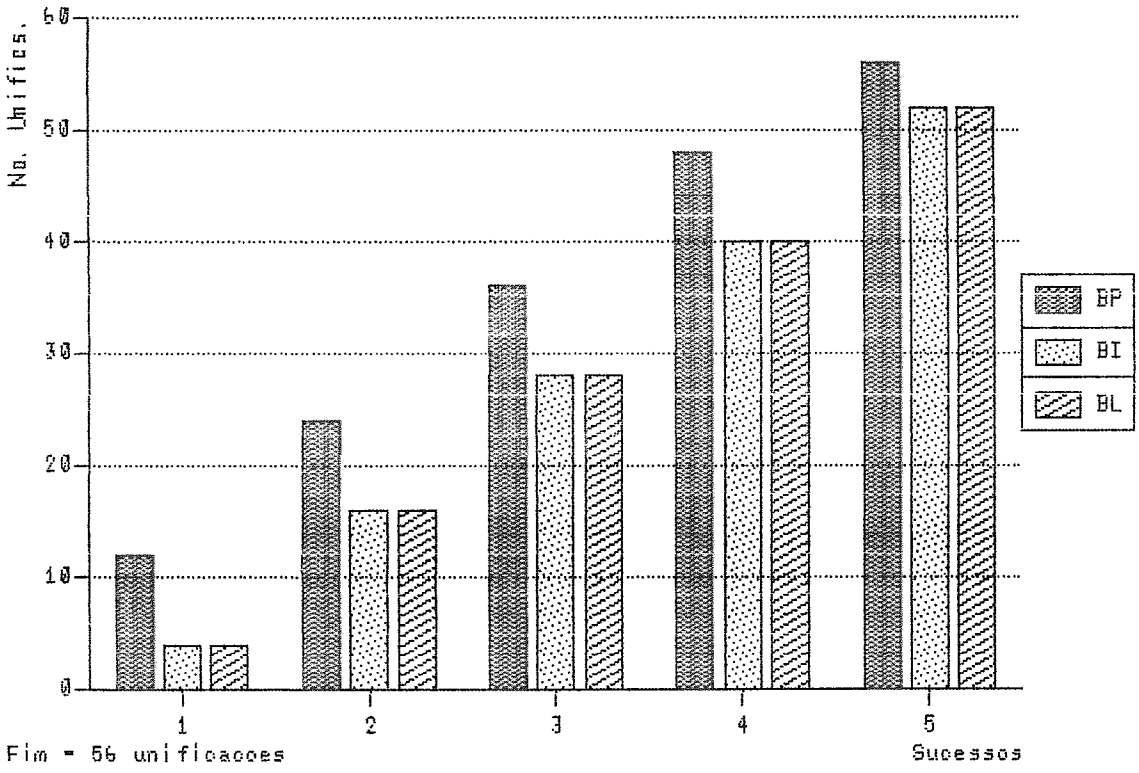
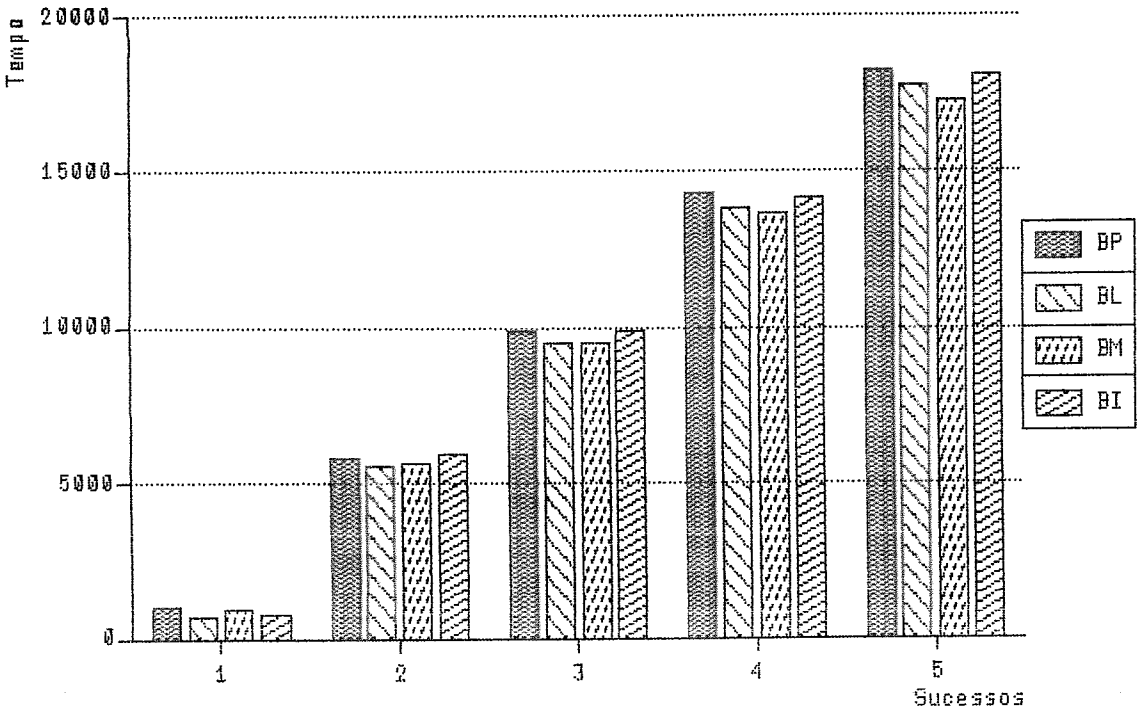


Grafico V.2 - Resultados de Execucao de concat (2a. fase)



Fim BP = 21385 Ticks
Fim BL = 20901 Ticks
Fim BM = 20490 Ticks
Fim BI = 21205 Ticks

	Sucesso	Busca em Prof.	Busca em Larg.	Busca Mista	Busca Padrao
Tempo	1	983	688	950	794
	3	9866	9478	9462	9874
	5	18179	17694	17276	18051
Numero de Unifics	1	12	4	12	4
	3	36	28	36	28
	5	56	52	56	52
Tamanho Memoria Objs	1	1	1	1	1
	3	1	1	1	1
	5	1	1	1	1
Tamanho de Obj	1	1	1	1	1
	3	1	1	1	1
	5	1	1	1	1
Numero de Falhas	1	0	0	0	0
	3	0	0	0	0
	5	0	0	0	0
Numero de Comunics	1	84	55	124	55
	3	268	239	308	239
	5	417	423	421	423
Numero de Bytes	1	1072	1077	1831	1077
	3	3076	3081	3835	3081
	5	4697	5085	5061	5085
Acessos Memoria Dinamica	1	4	5	8	5
	3	10	13	16	13
	5	15	21	20	21
Acessos Memoria Estatica	1	2	1	2	1
	3	6	7	6	7
	5	10	13	10	13

Tabela V.1: Avaliação de Desempenho com concat (vários WP)

O outro programa básico, *permute*, é o seguinte:

```
? permute([a, b, c, d], X).

permute([ ], [ ]).
permute([X|Y], [U|V]) :- delete(U, [X|Y], Z), permute(Z, V).

delete(X, [X|Y], Y).
delete(X, [Y|Z], [Y|W]) :- delete(X, Z, W).
```

Este programa gera permutações dos elementos da lista fornecida como primeiro argumento da consulta.

O programa *permute* apresentou uma grande vantagem, em número de unificações até os primeiros sucessos, para a busca em profundidade (gráfico V.3). As outras estratégias realizaram bem mais unificações para encontrar os primeiros sucessos, no entanto, é preciso notar que o último sucesso foi encontrado, pela busca "inteligente", com um número de unificações semelhante ao da busca em profundidade.

A segunda fase de avaliação (gráfico V.4) apresentou, para este programa, um resultado interessante: o primeiro sucesso foi encontrado mais rapidamente pela estratégia seqüencial de Prolog. Os outros sucessos foram obtidos bem mais rapidamente pela busca mista.

A tabela V.2 mostra que as estratégias de controle de Gol-Log são equivalentes em matéria de potencial para contenção, numa arquitetura com memória compartilhada, para este programa.

Grafico V.3 - Resultados de Execucao de permute (1a. fase)

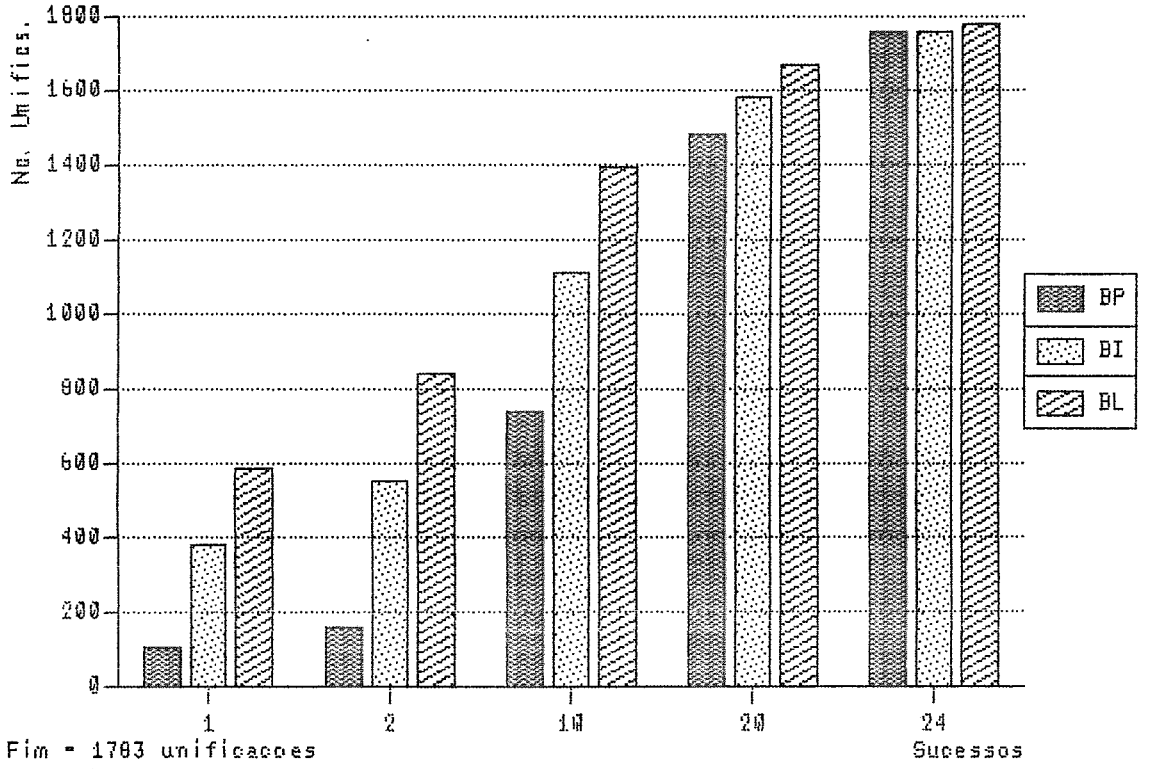
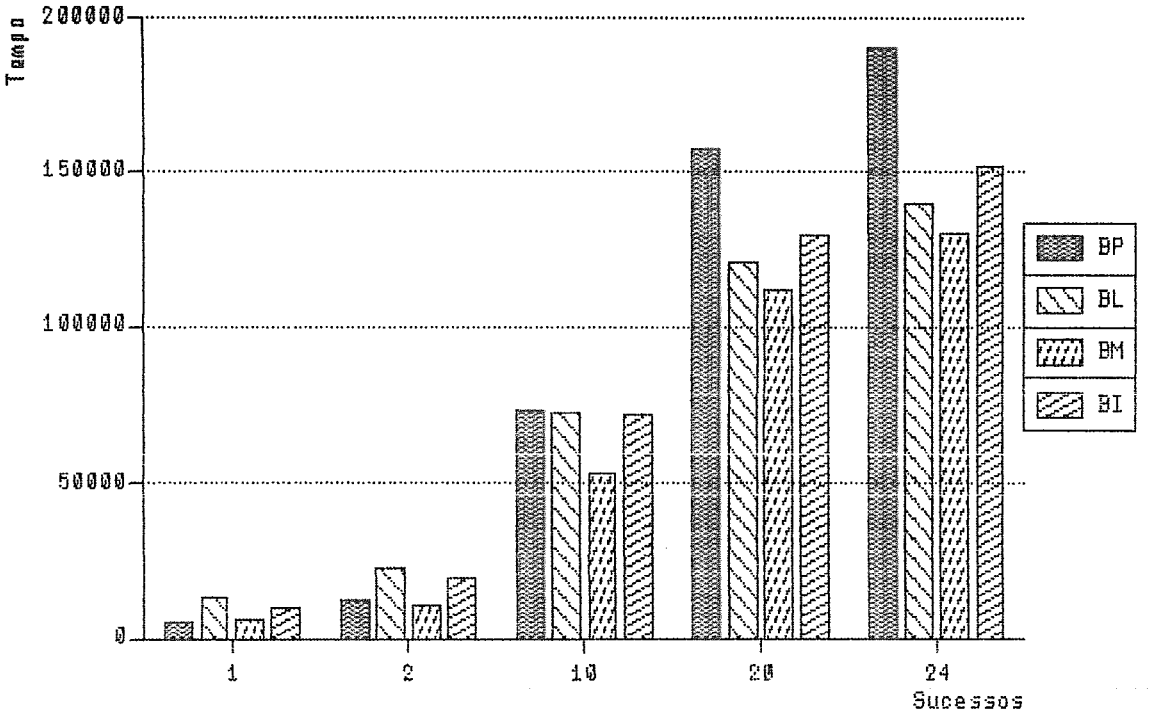


Grafico V.4 - Resultados de Execucao de permute (2a. fase)



	Sucesso	Busca em Prof.	Busca em Larg.	Busca Mista	Busca Padrao
Tempo	1	5107	12846	6138	10202
	12	88956	81592	64101	84322
	24	190135	139387	130385	151974
Numero de Unifics	1	106	508	266	379
	12	875	1433	949	1199
	24	1759	1767	1765	1749
Tamanho Memoria Objs	1	5	19	9	16
	12	5	26	11	25
	24	5	26	11	25
Tamanho de Obj	1	2	2	2	2
	12	2	2	2	2
	24	2	2	2	2
Numero de Falhas	1	0	4	2	0
	12	17	25	21	13
	24	37	40	38	37
Numero de Comunics	1	780	3898	2019	2900
	12	6533	10771	7181	9027
	24	13145	13242	13193	13148
Numero de Bytes	1	8855	43817	23005	32763
	12	72461	119828	80095	100500
	24	145573	147101	146554	146055
Acessos Memoria Dinamica	1	31	150	80	113
	12	250	411	279	345
	24	502	512	507	506
Acessos Memoria Estatica	1	26	124	64	91
	12	234	385	258	315
	24	474	503	477	494

Tabela V.2: Avaliação de Desempenho com permute (vários WP)

Programas Seqüenciais

Desta classe utilizaremos como exemplo os programas *fatorial* e *quicksort.*

O programa *fatorial* é o seguinte:

```
fatorial (0, 1).
fatorial (X, Y) :- gt (X, 0), is (Z, sub (X, 1)), fatorial (Z, T), is (Y, mult (X, T)).

? fatorial (10, X).
```

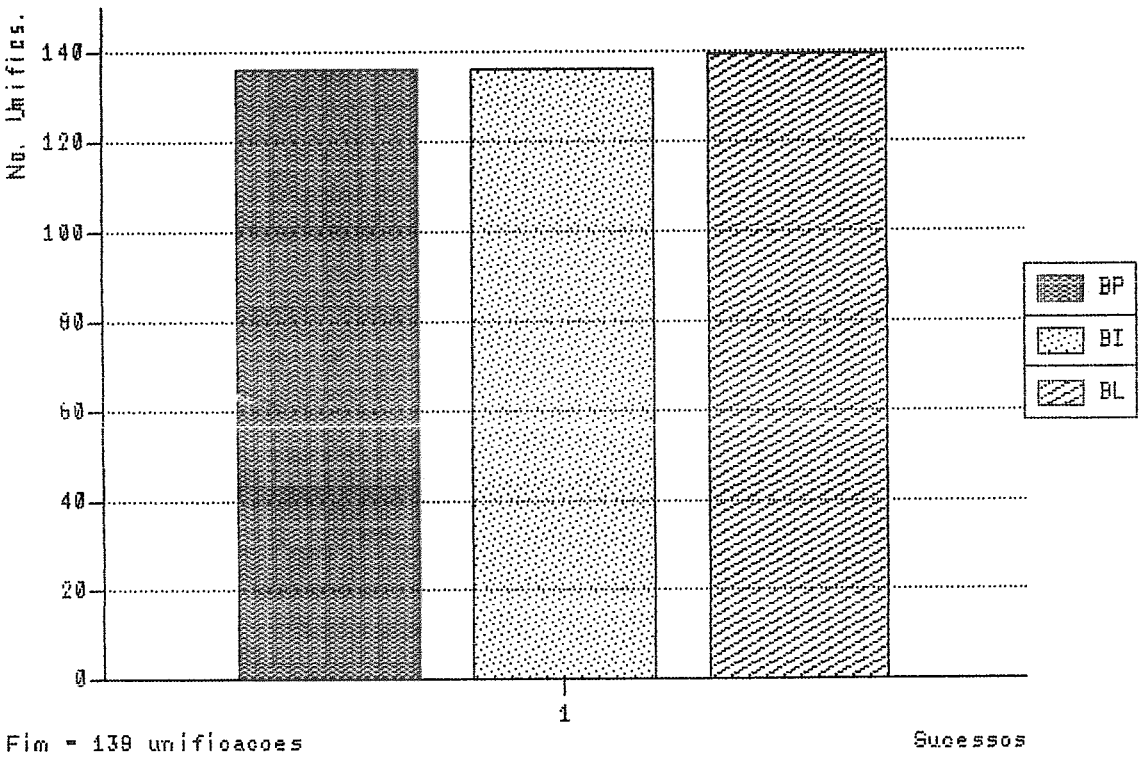
Este programa calcula o fatorial de 10.

A estratégia de controle "inteligente" de Gol-Log se comporta, neste caso, exatamente como uma busca em profundidade de Prolog (gráfico V.5). Tal fato é simples de explicar, já que, como os literais dos objetivos são todos dependentes, sempre será escolhido o mais a esquerda, assim como em Prolog. A busca em largura apresenta um número de unificações realizadas um pouco maior.

Quanto ao desempenho de execução das estratégias (gráfico V.6), observamos a busca mista e a busca em largura encontrando a solução mais rapidamente, com a busca "inteligente" um pouco mais lenta. A busca seqüencial de Prolog é a mais lenta de todas.

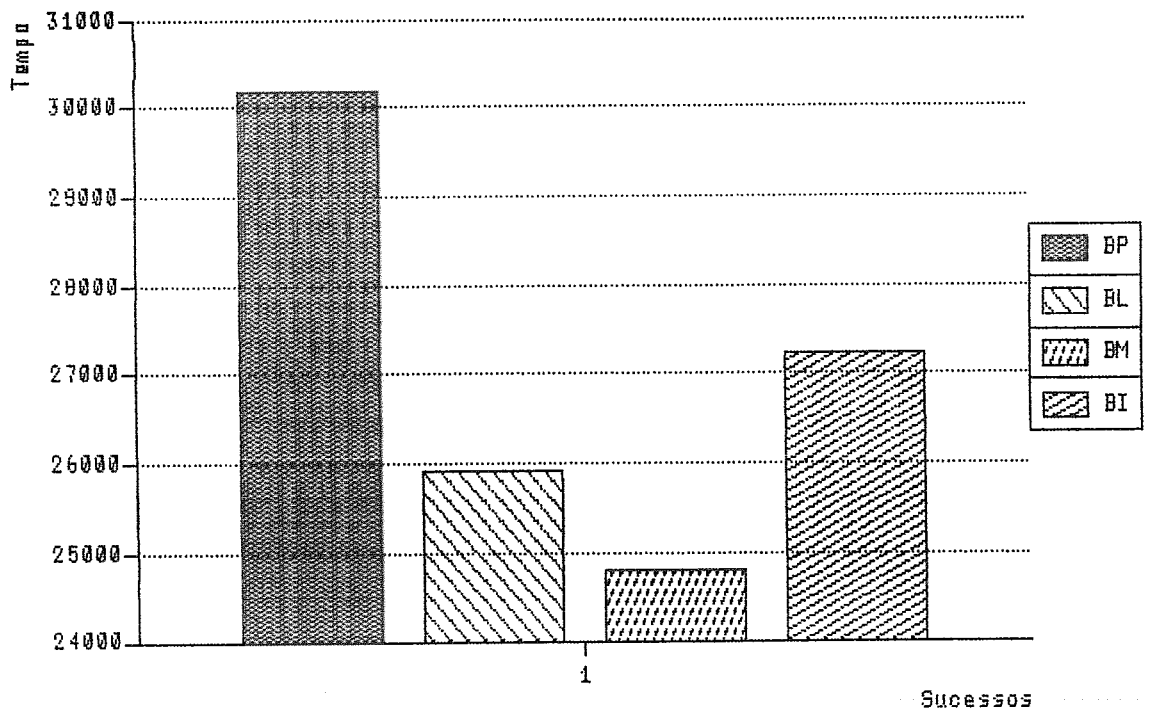
A busca em profundidade mostra, segundo a tabela V.3, o menor potencial para contenção de todas as estratégias de controle, para este programa. As outras formas de busca são equivalentes neste ponto.

Grafico V.5 - Resultados de Execucao de fatorial (1a. fase)



Fim = 139 unificacoes

Grafico V.6 - Resultados de Execucao de fatorial (2a. fase)



Fim BP = 35557 Ticks
Fim BL = 30116 Ticks
Fim BM = 29404 Ticks
Fim BI = 31053 Ticks

	Sucesso	Busca em Prof.	Busca em Larg.	Busca Mista	Busca Padrao
Tempo	1 -	30180	25912	24798	27237
Numero de Unifics	1 -	136	139	139	139
Tamanho Memoria Objs	1 -	2	1	1	1
Tamanho de Obj	1 -	14	14	14	14
Numero de Falhas	1 -	0	1	1	1
Numero de Comunics	1 -	7680	7865	7865	7865
Numero de Bytes	1 -	88564	91096	91096	91096
Acessos Memoria Dinamica	1 -	124	168	169	168
Acessos Memoria Estatica	1 -	91	94	93	94

Tabela V.3: Avaliação de Desempenho com fatorial (vários WP)

O outro programa que apresentamos nesta classe é o *quicksort*:

```
? qsort([0, 2, 3, 1], X).

qsort([ ], [ ]).
qsort([H|U], S) :- split(H, U, P, G), qsort(P, S1), qsort(G, S2), app(S1, [H|S2], S).

split(X, [ ], [ ], [ ]).
split(X, [H|T1], [H|U1], U2) :- le(H, X), split(X, T1, U1, U2).
split(X, [H|T1], U1, [H|U2]) :- gt(H, X), split(X, T1, U1, U2).

app([ ], L, L).
app([X|Y], Z, [X|L]) :- app(Y, Z, L).
```

Com este programa queremos ordenar os elementos da lista fornecida como primeiro argumento da consulta.

Este programa é muito importante para esta avaliação, pois apresenta um exemplo de que, às vezes, a "inteligência" da estratégia de Gol-Log pode não estar ao lado do usuário. Isto é, algumas vezes as heurísticas propostas podem tomar decisões que vão prejudicar a busca, aumentando, e não reduzindo, o espaço de busca.

Na primeira fase de avaliação (gráfico V.7) aparece o problema: o número de unificações realizado pela busca "inteligente" é quase 2 vezes maior, no momento do sucesso, e quase 4 vezes maior no final da execução, que as outras duas estratégias. Para ilustrar esta constatação observe o número de falhas encontradas pelo programa, quando executado com a busca "inteligente".

Os tempos de execução para 4 processadores apresentam problema semelhante (gráfico V.8), a busca "inteligente" é mais lenta até que o Prolog seqüencial.

Na terceira fase de avaliação (tabela V.4), observamos que o número de acessos à memória dinâmica feito pela busca padrão é muito maior que o das outras formas de busca. No entanto, é interessante observar que a relação entre tal número de acessos e o número de unificações não é má, sendo semelhante à relação encontrada pela busca em profundidade.

Grafico V.7 - Resultados de Execucao de quicksort (1a. fase)

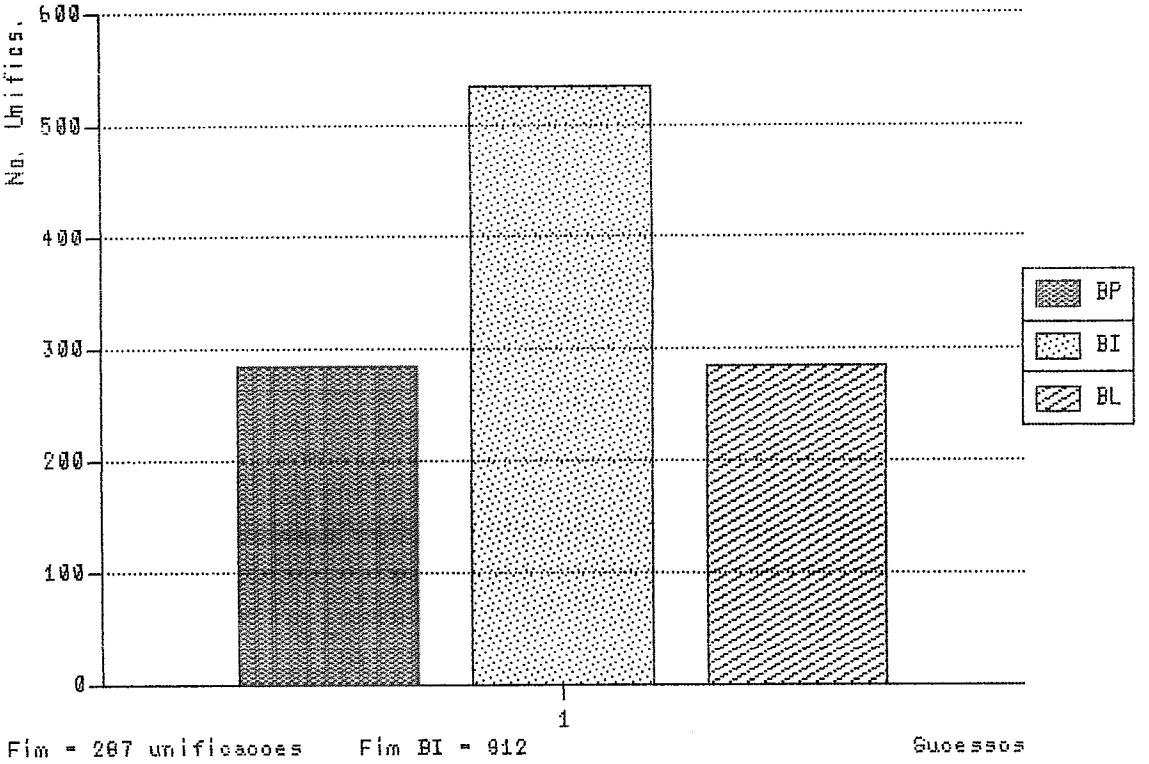
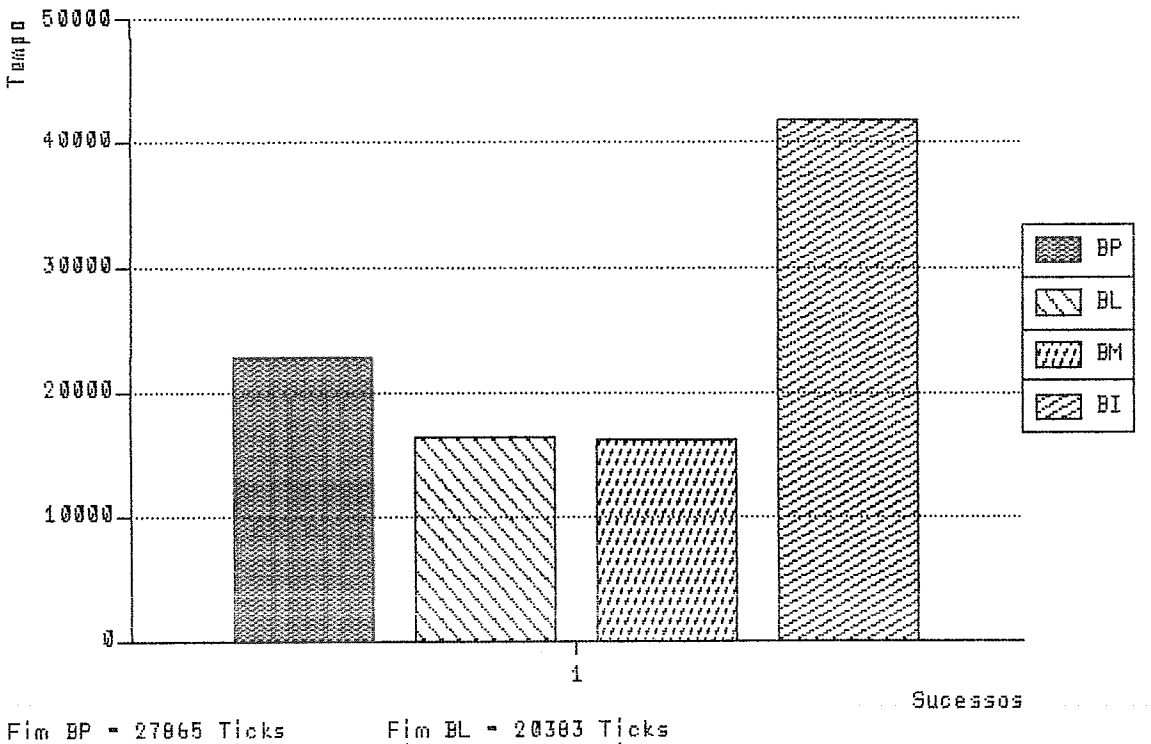


Grafico V.8 - Resultados de Execucao de quicksort (2a. fase)



Fim BP = 27865 Ticks
Fim BL = 20383 Ticks
Fim BM = 20567 Ticks
Fim BI = 74024 Ticks

	Sucesso	Busca em Prof.	Busca em Larg.	Busca Mista	Busca Padrao
Tempo	1 - -	22713	16349	16177	41806
Numero de Unifics	1 - -	284	285	287	614
Tamanho Memoria Objs	1 - -	2	1	1	6
Tamanho de Obj	1 - -	7	7	7	12
Numero de Falhas	1 - -	4	5	5	4
Numero de Comunics	1 - -	4262	4352	4352	11177
Numero de Bytes	1 - -	47506	49259	49259	124604
Acessos Memoria Dinamica	1 - -	97	132	128	218
Acessos Memoria Estatica	1 - -	90	96	92	207

Tabela V.4: Avaliação de Desempenho com quicksort (vários WP)

Programas Paralelisados

Como exemplos de programas paralelisados, isto é, programas seqüenciais modificados para execução em paralelo, apresentaremos o de interseção de listas e o de produto escalar de matrizes.

O programa para interseção de listas é o seguinte:

```
? inter(U, [a, b, c, d], [f, a, d, c]).

inter(H, L, [H|X]) :- membro(H, L).
inter(R1, L, [X|R2]) :- inter(R1, L, R2).

membro(X, [X|Y]).
membro(X, [Z|Y]) :- dif(X, Z), membro(X, Y).

dif(X, Y) :- gt(X, Y).
dif(X, Y) :- lt(X, Y).
```

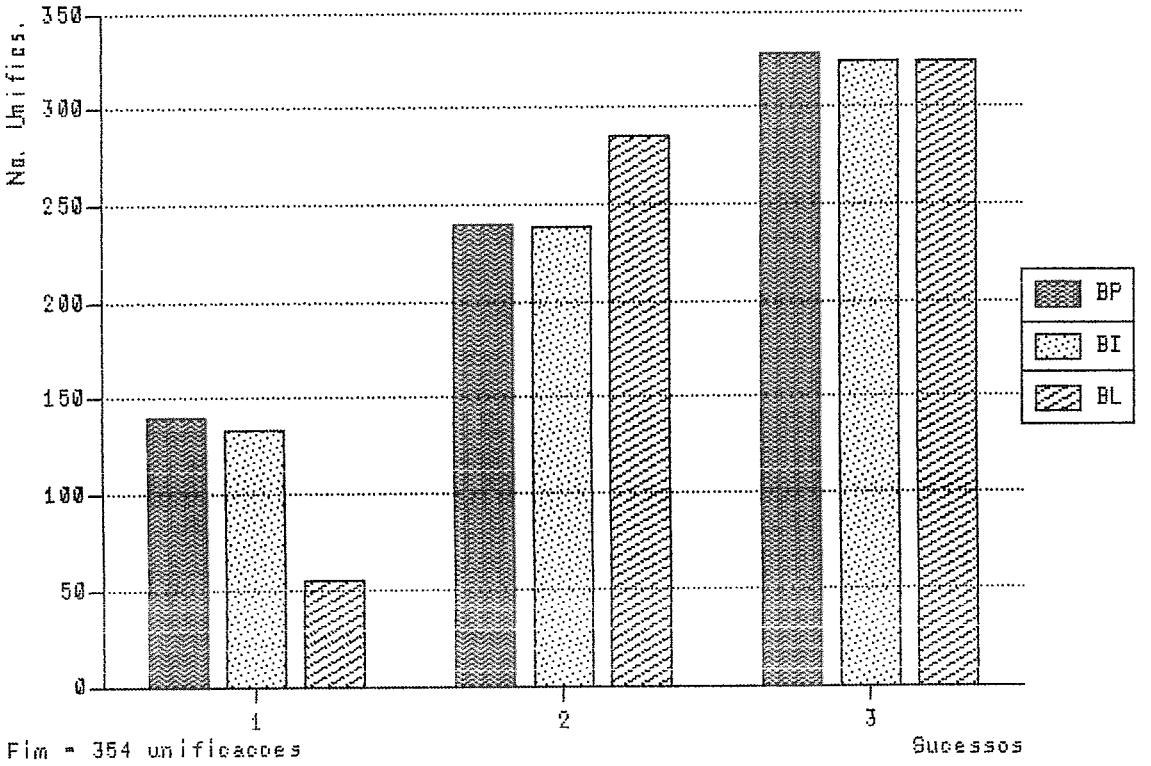
Na avaliação das heurísticas de Gol-Log, os resultados deste programa (gráfico V.9) dizem o seguinte: o primeiro sucesso foi encontrado com menor número de unificações pela busca em largura, no entanto, a partir do segundo sucesso, as outras duas estratégias praticamente se igualam a primeira.

Vale notar que a busca "inteligente" no segundo sucesso já tinha encontrado 11 soluções, bem mais do que as outras buscas, mostrando que, para este programa, as heurísticas foram bem sucedidas na procura por soluções.

A segunda fase de avaliações (gráfico V.10) apresentou a busca em largura e a busca "inteligente" executando da mesma forma até o primeiro sucesso (a busca padrão levou mais tempo), com a busca mista um pouco atrás (a busca de Prolog fica muito atrás, neste caso). Já na segunda solução, a busca mista se torna mais eficiente que as outras.

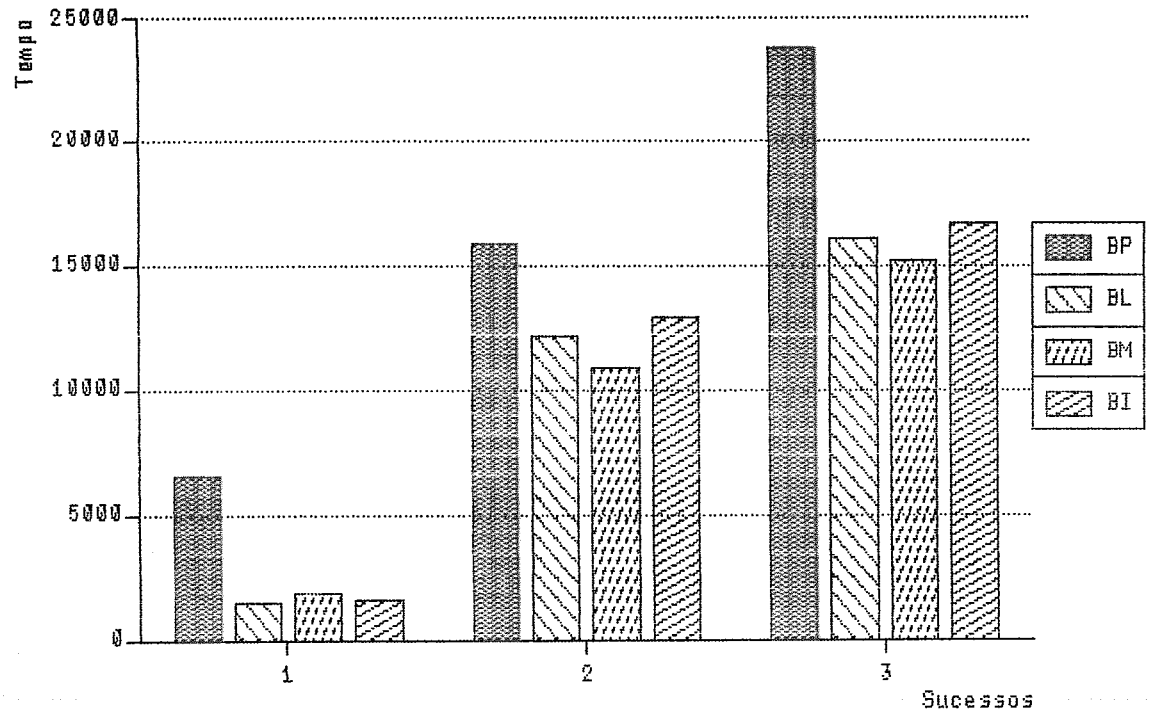
Todas as estratégias de controle utilizadas apresentam potencial semelhante para contenção, em arquiteturas de memória compartilhada, para este programa (tabela V.5).

Grafico V.9 - Resultados de Execucao de inter (1a. fase)



Fim = 354 unificacoes

Grafico V.10 - Resultados de Execucao de inter (2a. fase)



Fim BP = 20920 Ticks
Fim BL = 20370 Ticks
Fim BM = 20201 Ticks
Fim BI = 21263 Ticks

	Sucesso	Busca em Prof.	Busca em Larg.	Busca Mista	Busca Padrao
Tempo	1	6575	1458	1874	1586
	2	15846	12181	10812	12855
	3	23771	16015	15188	16647
Numero de Unifics	1	140	47	88	47
	2	239	297	263	300
	3	328	323	305	320
Tamanho Memoria Objs	1	6	2	4	2
	2	6	5	5	7
	3	6	5	5	7
Tamanho de Obj	1	2	2	2	2
	2	2	2	2	2
	3	2	2	2	2
Numero de Falhas	1	5	0	0	0
	2	7	11	9	11
	3	12	13	12	12
Numero de Comunics	1	1059	326	494	326
	2	1881	2443	2094	2437
	3	2591	2566	2417	2538
Numero de Bytes	1	11698	4347	6180	4347
	2	20566	27187	23432	27121
	3	18224	28513	26908	28210
Acessos Memoria Dinamica	1	62	19	27	19
	2	107	141	117	141
	3	148	149	139	147
Acessos Memoria Estatica	1	49	10	13	10
	2	85	112	89	113
	3	119	122	107	120

Tabela V.5: Avaliação de Desempenho com inter (vários WP)

O programa de produto escalar de matrizes é o seguinte:

```
? row_mul([[1,2,3,4],[1,2,3,4]],[[1,2,3,4],[1,2,3,4]],X).

row_mul([L1|R1], [L2|R2], V) :- list_mul(L1, L2, V).
row_mul([X|R1], [Y|R2], V) :- row_mul(R1, R2, V).

list_mul([ ], [ ], 0).
list_mul([X1|R1], [X2|R2], P) :- list_mul(R1, R2, P1), is(X3, mult(X1, X2)), is(P, add(P1,
X3)).
```

Este é um programa simplificado para cálculo do produto escalar de vetores representados como listas.

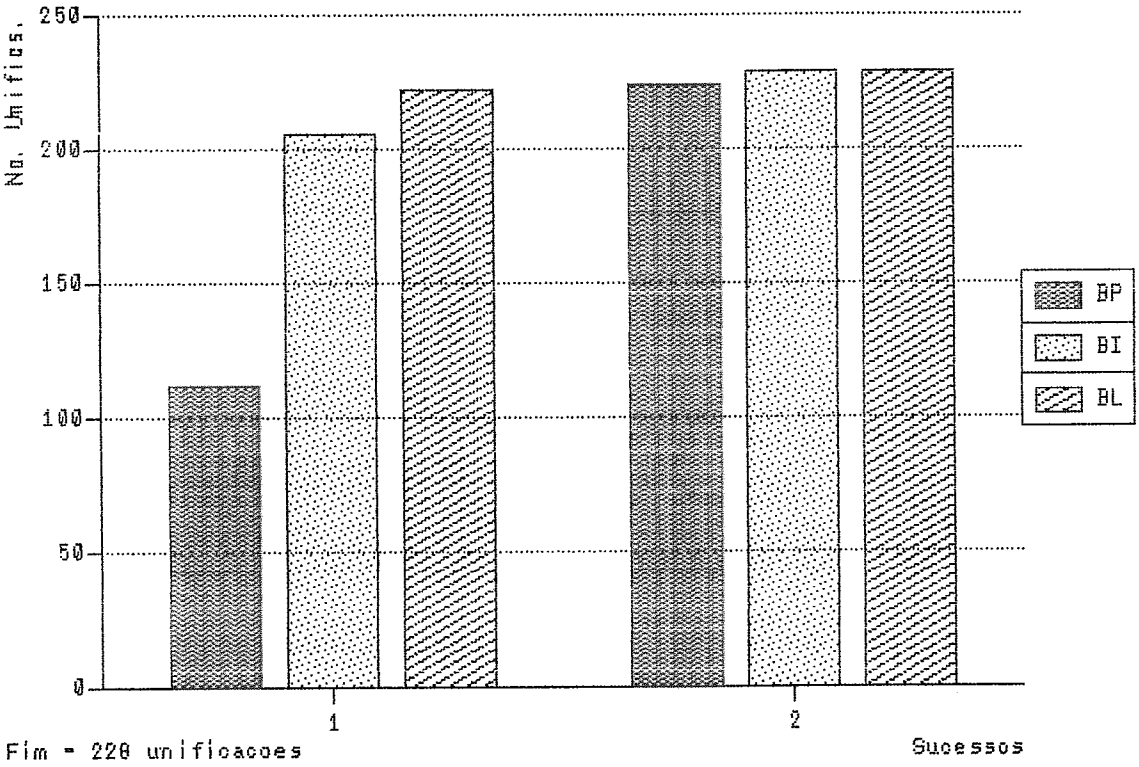
O primeiro sucesso é encontrado com menor número de unificações pela busca em profundidade, seguida pela "inteligente" e depois pela em largura (gráfico V.11). No entanto, no segundo sucesso todas se aproximam.

Em matéria de tempos de execução (gráfico V.12), a busca padrão só perde no primeiro sucesso para a estratégia de Prolog (até este momento o Prolog só tinha encontrado esta solução, enquanto a busca "inteligente" já tinha encontrado outra), terminando o processamento bem antes de todas as outras buscas.

Também para este programa, o potencial para contenção de todas as estratégias de busca é aproximadamente igual (tabela V.6).

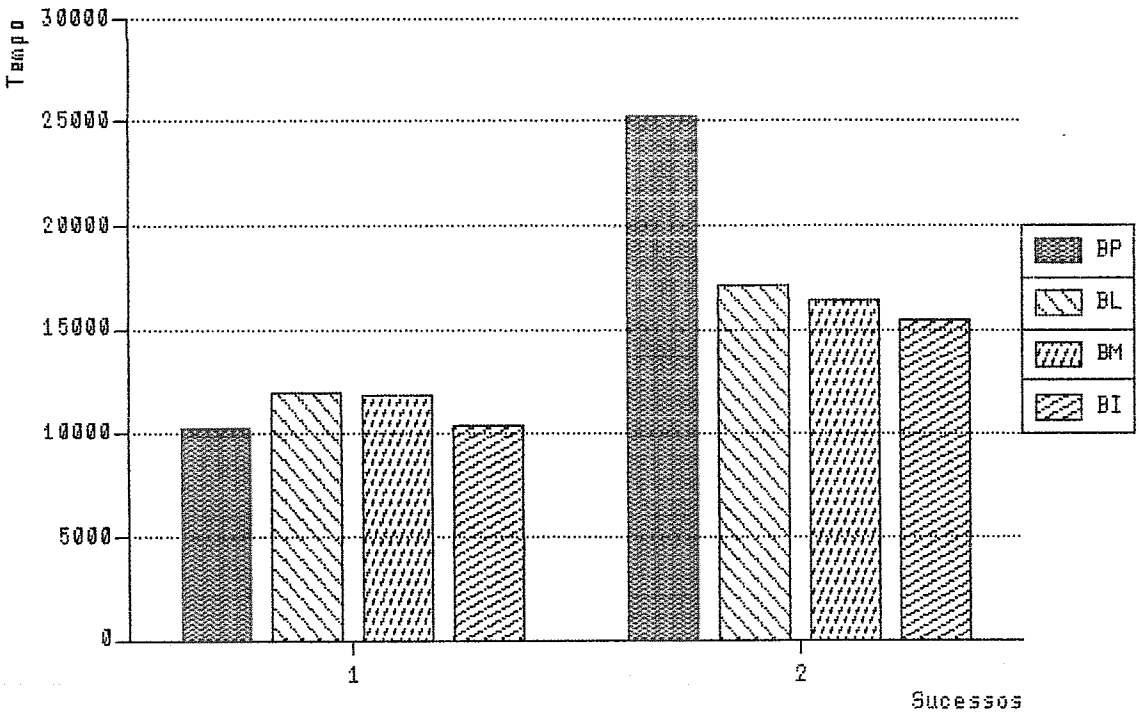
Quanto as outras medidas (tabela V.6), é importante ressaltar o que pode ter dado esta vantagem a busca padrão: o número de comunicações e, conseqüentemente o número de bytes transferidos.

Grafico V.11 - Resultados de Execucao de row-mul (1a. fase)



Fim - 220 unificacoes

Grafico V.12 - Resultados de Execucao de row-mul (2a. fase)



Fim BP - 20752 Ticks
Fim BL - 20296 Ticks
Fim BM - 19604 Ticks
Fim BI - 18645 Ticks

	Sucesso	Busca em Prof.	Busca em Larg.	Busca Mista	Busca Padrao
Tempo	1	10282	11980	11803	10355
	2	25291	17046	16336	15368
Numero de Unifics	1	112	219	225	222
	2	224	228	228	228
Tamanho Memoria Objs	1	2	3	2	2
	2	2	3	2	2
Tamanho de Obj	1	9	9	9	6
	2	9	9	9	6
Numero de Falhas	1	0	1	1	1
	2	0	1	1	1
Numero de Comunics	1	1978	3770	3928	2978
	2	3935	3939	3935	3067
Numero de Bytes	1	22407	43082	44857	34581
	2	43999	44985	44937	35597
Acessos Memoria Dinamica	1	43	82	94	89
	2	88	94	94	97
Acessos Memoria Estatica	1	32	63	65	64
	2	65	70	66	70

Tabela V.6: Avaliação de Desempenho com row-mul (vários WP)

Programas de Busca com uma Solução

Nesta classe de programas apresentaremos o *sublist* e o *permsort.*

O programa que verifica se uma lista está contida em outra lista é o seguinte:

```
? sublist([a, b, c], [d, b, e, a, b, c, g]).

sublist(T,U) :- concat(H, T, V), concat(V, W, U).

concat ([ ], L, L).
concat ([X|Y], [X|L], Z) :- concat (Y, L, Z).
```

Para este programa é necessário especificar que queremos apenas um sucesso, caso contrário serão fornecidos 4 sucessos. A nossa análise é feita para os 4 sucessos, pois, desta forma, estará incluído o primeiro sucesso e ainda teremos mais informações.

Neste programa a busca "inteligente" se comporta melhor que as outras. Em seguida, aparece a busca em profundidade (gráfico V.13), e depois a busca em largura, em relação ao número de unificações para os primeiros sucessos.

A busca mista consegue chegar mais rapidamente ao primeiro dos 4 sucessos do programa (gráfico V.14). A partir do segundo sucesso, a busca em largura se torna aproximadamente tão eficiente quanto a busca mista, com a busca "inteligente" mais atrás, apesar de apresentar medidas (a não ser o tempo) em relativo equilíbrio com as outras duas.

A terceira fase de avaliação apresenta a busca em profundidade de Prolog com o melhor resultado (tabela V.7). As outras estratégias têm resultados semelhantes, um pouco piores.

Grafico V.13 - Resultados de Execucao de sublist (1a. fase)

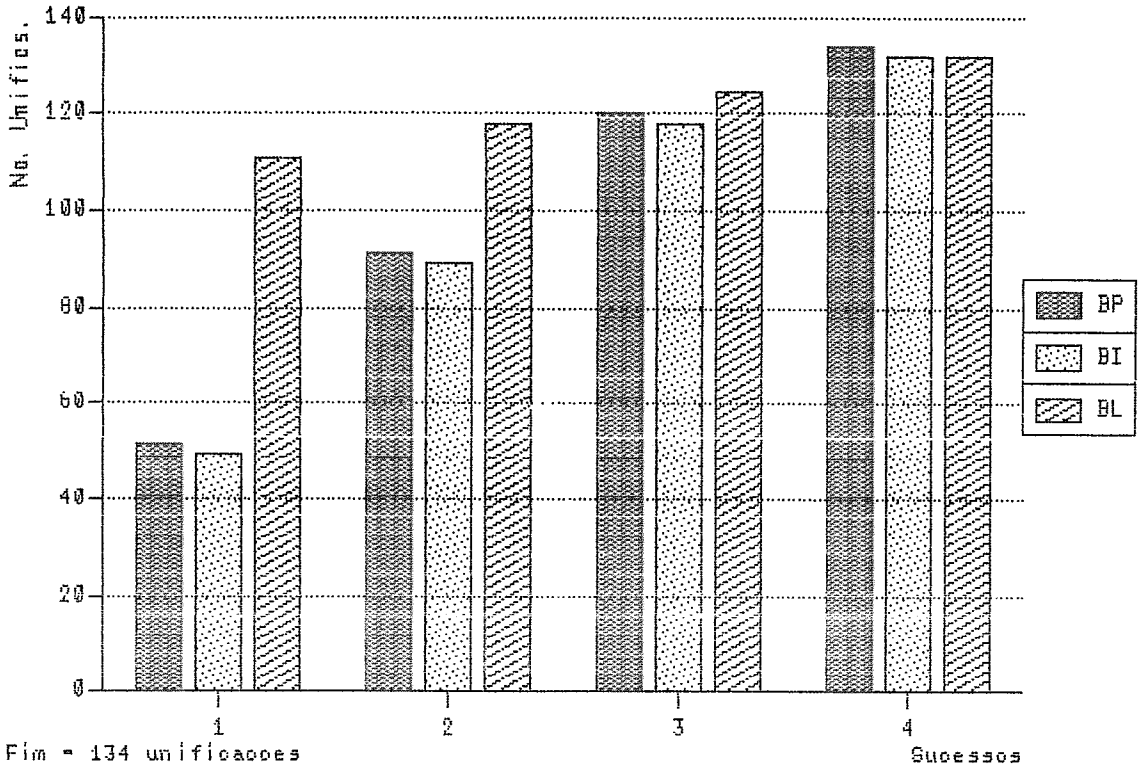
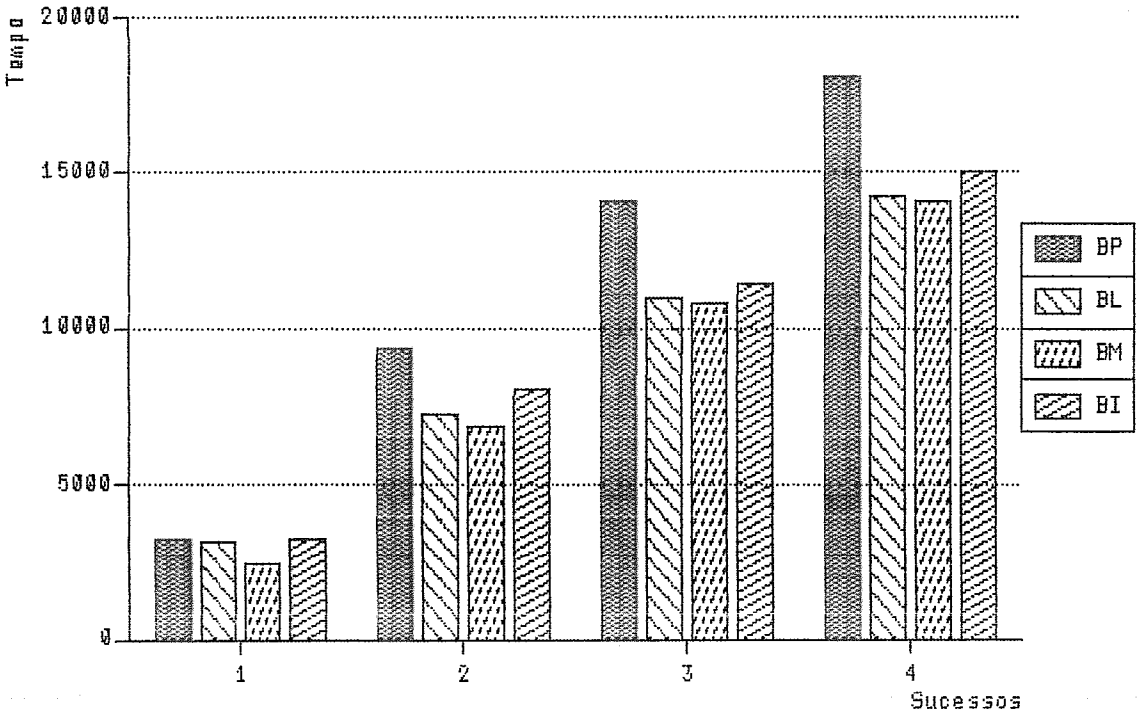


Grafico V.14 - Resultados de Execucao de sublist (2a. fase)



Fim BP - 21610 Ticks
Fim BL - 17945 Ticks
Fim BM - 17764 Ticks
Fim BI - 18677 Ticks

	Sucesso	Busca em Prof.	Busca em Larg.	Busca Mista	Busca Padrao
Tempo	1	3208	3164	2410	3207
	2	9351	7223	6865	8067
	4	18070	14242	14027	15025
Numero de Unifics	1	52	94	73	94
	2	91	99	90	111
	4	134	127	132	132
Tamanho Memoria Objs	1	2	4	3	3
	2	2	4	3	3
	4	2	4	3	3
Tamanho de Obj	1	2	2	2	2
	2	2	2	2	2
	4	2	2	2	2
Numero de Falhas	1	0	0	0	0
	2	0	0	0	0
	4	0	0	0	0
Numero de Comunics	1	429	845	698	815
	2	704	848	824	946
	4	1009	1011	1011	1015
Numero de Bytes	1	4938	9958	8358	9629
	2	7966	9990	9736	11072
	4	11330	11782	11782	11830
Acessos Memoria Dinamica	1	19	40	32	38
	2	31	40	39	45
	4	45	48	49	50
Acessos Memoria Estatica	1	15	27	21	26
	2	27	28	27	33
	4	42	38	39	42

Tabela V.7: Avaliação de Desempenho com sublist (vários WP)

O programa *permsort* é o seguinte:

```
? sort([2,1,3],X).

sort(X,Y) :- permute(X,Y), sorted(Y).

sorted([X]).
sorted([X,Y|Z]) :- le(X,Y), sorted([Y|Z]).

permute([ ],[ ]).
permute([X|Y],[U|V]) :- delete(U,[X|Y],Z), permute(Z,V).

delete(X,[X|Y],Y).
delete(X,[Y|Z],[Y|W]) :- delete(X,Z,W).
```

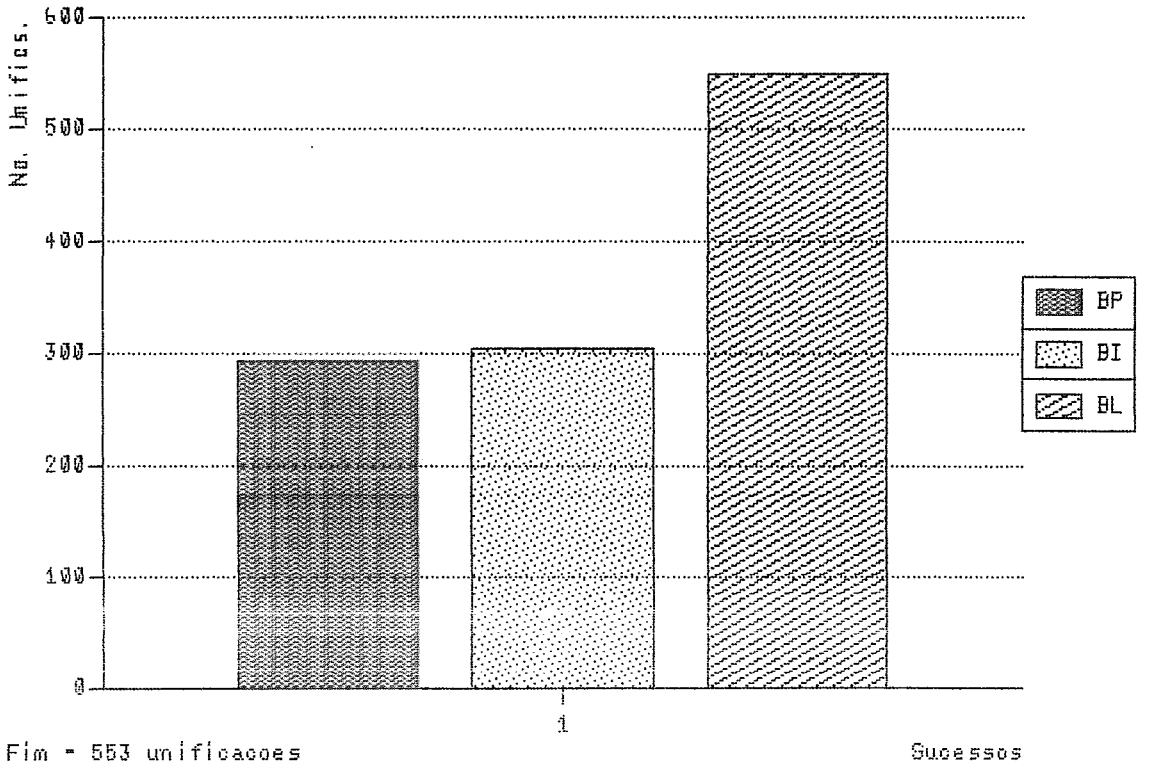
Este programa ordena a lista de elementos fornecida na consulta, através da geração de permutações de tais elementos.

Em relação ao número de unificações realizadas, a busca de Prolog tem melhor resultado que a busca "inteligente" e muito melhor que a busca em largura (gráfico V.15).

A análise de tempo de execução com mais de um WP (gráfico V.16) mostra a busca padrão com o melhor resultado, um pouco melhor que a busca mista e muito melhor que as outras duas estratégias.

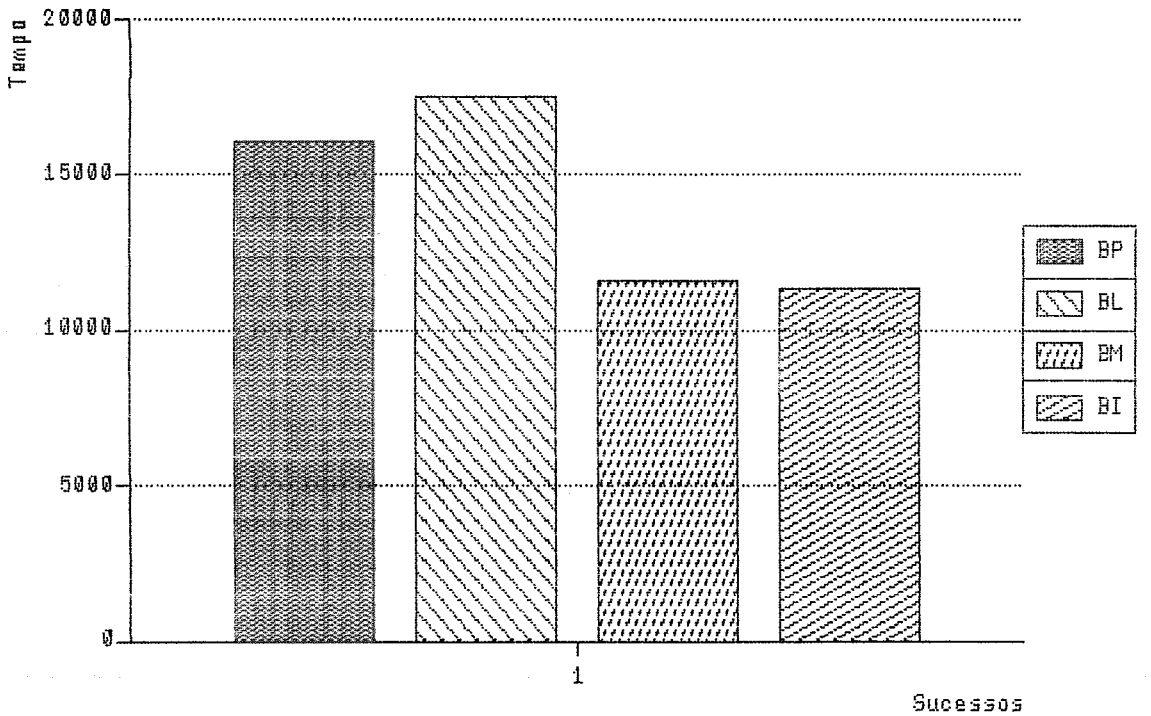
O potencial para contenção em arquiteturas com memória compartilhada é menor, para este programa, para a busca padrão, seguida de perto pela busca em largura e a busca mista (tabela V.8). A busca em profundidade apresenta o pior resultado.

Grafico V.15 - Resultados de Execucao de permsort (1a. fase)



Fim - 553 unificacoes

Grafico V.1b - Resultados de Execucao de permsort (2a. fase)



Fim BP - 36811 Ticks
Fim BL - 21064 Ticks
Fim BM - 20630 Ticks
Fim BI - 23124 Ticks

	Sucesso	Busca em Prof.	Busca em Larg.	Busca Mista	Busca Padrao
Tempo	1 - -	16082	17445	11569	11349
Numero de Unifics	1 - -	293	544	400	349
Tamanho Memoria Objs	1 - -	4	9	7	9
Tamanho de Obj	1 - -	3	3	3	3
Numero de Falhas	1 - -	5	13	9	4
Numero de Comunics	1 - -	2778	5231	3889	3407
Numero de Bytes	1 - -	30930	58638	43808	38574
Acessos Memoria Dinamica	1 - -	99	185	142	118
Acessos Memoria Estatica	1 - -	88	167	122	99

Tabela V.8: Avaliação de Desempenho com permsort (vários WP)

Programas de Busca com mais de uma Solução

Os programas apresentados para esta classe serão o de densidades populacionais e o de referências bibliográficas.

O programa que analisa densidades populacionais é o seguinte:

```
query(X,Y,Z,W,R,T) :- info(X,Y,Z), info(W,R,T), aux1(Y,Z,R,T).

aux1(Y,Z,R,T) :- is(W, mult(R,2)), gt(Y,W), is (V, mult(Z,2)), lt(V, T).

info(C,P,A) :- pop(C,P), area(C,A).

pop(china,8250).
pop(india,5863).
pop(ussr,2521).
pop(usa,2119).

area(china,3380).
area(india,1139).
area(ussr,8708).
area(usa,3609).

? query(X,Y,Z,W,R,T).
```

Este programa é capaz de gerar pares de países, observando-se as suas densidades populacionais.

Em relação ao número de unificações realizadas por cada estratégia de controle analisada (gráfico V.17), fica claro que, para este programa, a busca em profundidade de Prolog tem o menor custo para encontrar os sucessos. Em seguida, aparece a busca "inteligente" e, mais atrás, a busca em largura.

Na segunda fase de avaliação (gráfico V.18), um resultado interessante, a busca em profundidade de Prolog obteve o último sucesso antes de todas as estratégias paralelas. O fim da execução ocorreu mais rapidamente, no entanto, para a busca mista.

A partir da tabela V.9 podemos observar que até o primeiro sucesso, a busca em

profundidade apresenta um menor potencial para contenção. Já no segundo sucesso, todas as estratégias de busca passam a se equivaler.

Grafico V.17 - Resultados de Execucao de dens (1a. fase)

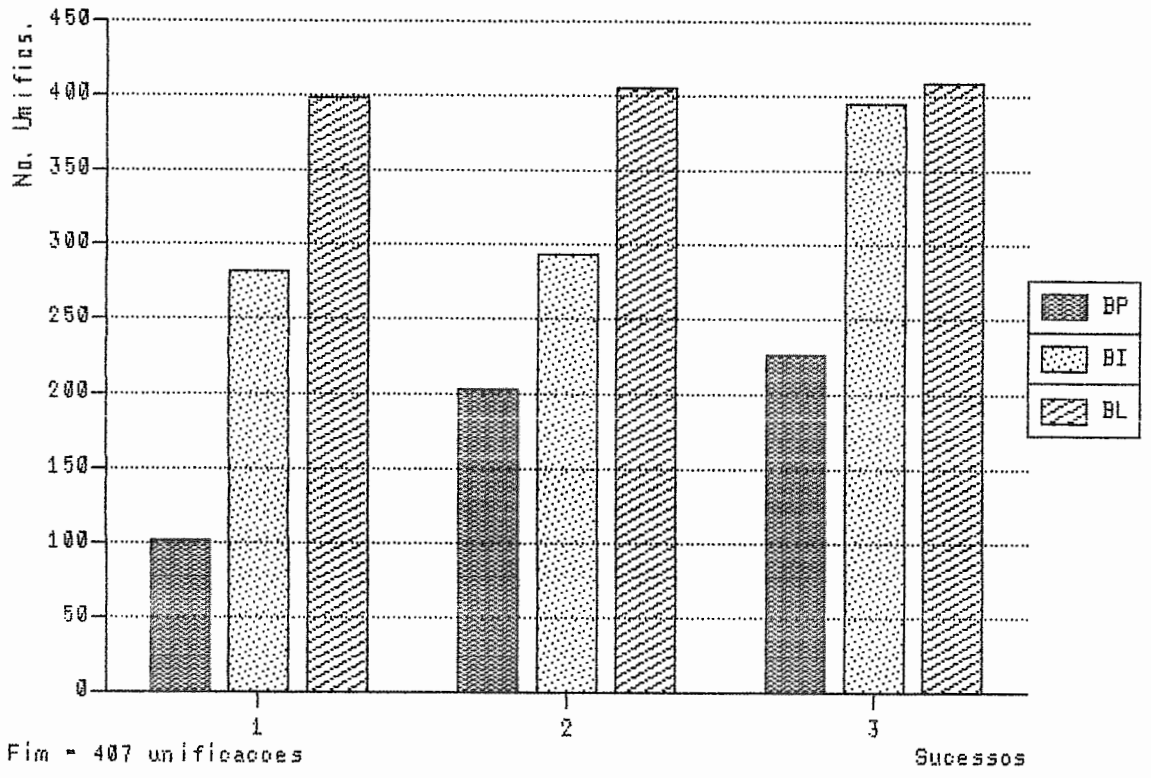
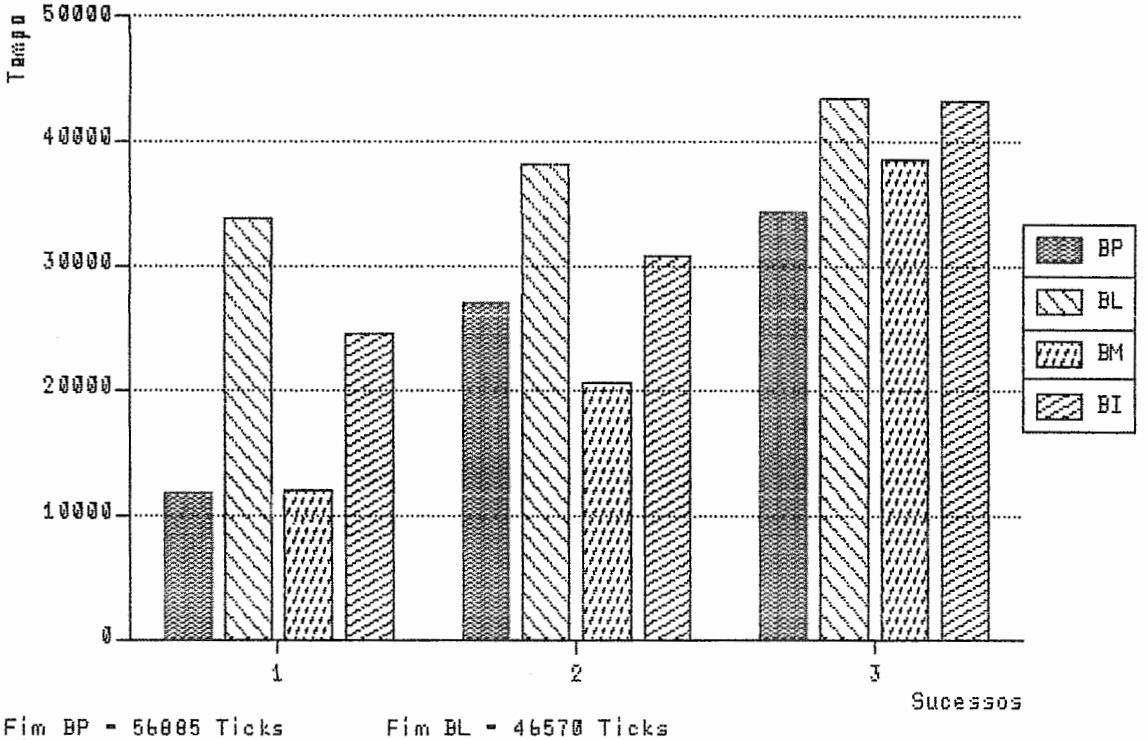


Grafico V.18 - Resultados de Execucao de dens (2a. fase)



Fim BP - 56085 Ticks
Fim BL - 46570 Ticks
Fim BM - 41021 Ticks
Fim BI - 47277 Ticks

	Sucesso	Busca em Prof.	Busca em Larg.	Busca Mista	Busca Padrao
Tempo	1	11669	33640	12003	24520
	2	26931	38142	20431	30635
	3	34326	43246	38500	43036
Numero de Unifics	1	102	398	164	297
	2	204	401	227	312
	3	227	407	407	398
Tamanho Memoria Objs	1	7	16	10	10
	2	7	16	10	10
	3	7	16	10	10
Tamanho de Obj	1	4	4	4	4
	2	4	4	4	4
	3	4	4	4	4
Numero de Falhas	1	2	12	4	8
	2	5	12	6	9
	3	5	13	13	11
Numero de Comunics	1	2254	8918	3646	6482
	2	4557	8945	4975	6913
	3	5126	9012	9012	8830
Numero de Bytes	1	25947	102196	42544	74684
	2	52045	102499	57556	79547
	3	58467	103257	103257	101205
Acessos Memoria Dinamica	1	64	263	109	191
	2	132	263	147	204
	3	149	267	271	263
Acessos Memoria Estatica	1	56	242	95	179
	2	120	243	136	189
	3	135	248	247	242

Tabela V.9: Avaliação de Desempenho com dens (vários WP)

O outro programa para esta classe é o seguinte:

```
paper (P, D, I) :- date (P, D), author (P, A), loc (A, I, D).
paper (P, D, I) :- tr (P, I), date (P, D).
paper (xform, 1978, uci).

author (fp, backus).
author (df, arvind).
author (eft, kling).
author (pro, pereira).
author (sem, vanemdem).
author (db, warren).
author (sasl, turner).
author (xform, standish).

date (fp, 1978).
date (df, 1978).
date (eft, 1978).
date (pro, 1978).
date (sem, 1976).
date (db, 1981).
date (sasl, 1979).

title (db, efficient_processing_of_interactive).
title (df, an_asynchronous_programming_language).
title (eft, value_conflicts_and_social_choice).
title (fp, can_programming_be_liberated).
title (pro, dec_ten_prolog_user_manual).
title (sasl, a_new_implementation_technique).
title (sem, the_semantics_of_predicate_logic).
title (xform, irvine_program_transformation_catalog).
```

```
loc (arvind, mit, 1980).
loc (backus, ibm, 1978).
loc (kling, uci, 1978).
loc (pereira, lisbon, 1978).
loc (vanemdem, waterloo, 1980).
loc (turner, kent, 1981).
loc (warren, edinburgh, 1977).
loc (warren, sri, 1982).

journal (fp, cacm).
journal (sasl, spe).
journal (kling, cacm).
journal (sem, jacm).

tr (db, edinburgh).
tr (df, uci).

? paper (X, Y, Z).
```

Este programa mostra a pesquisa sobre um banco de dados de autores e referências.

Este, assim como o *quicksort* é um exemplo onde as heurísticas não vão bem, é um exemplo onde o comportamento das heurísticas é excelente, reduzindo bastante a computação necessária para encontrar as soluções do programa.

Podemos observar isto pelos resultados obtidos na execução do programa com as diversas estratégias de controle (gráfico V.19). Todos os sucessos, exceto o primeiro, foram encontrados pela busca "inteligente" com número bem menor de unificações que as outras formas de busca.

Na parte de tempo de execução (gráfico V.20), pudemos constatar um problema que já nos parecia óbvio: o processamento das heurísticas, principalmente de H2, é muito custoso. Isto fica claro, quando observamos que, apesar de a estratégia "inteligente" ter chegado ao último sucesso com um número muito menor de unificações, o seu tempo de processamento ainda assim é um dos maiores.

Em matéria de potencial para contenção, podemos dizer, baseados na tabela V.10, que a busca em profundidade tem o melhor resultado de execução para este programa. Em seguida, aparecem a busca em largura e a busca padrão.

Grafico V.19 - Resultados de Execucao de paper (1a. fase)

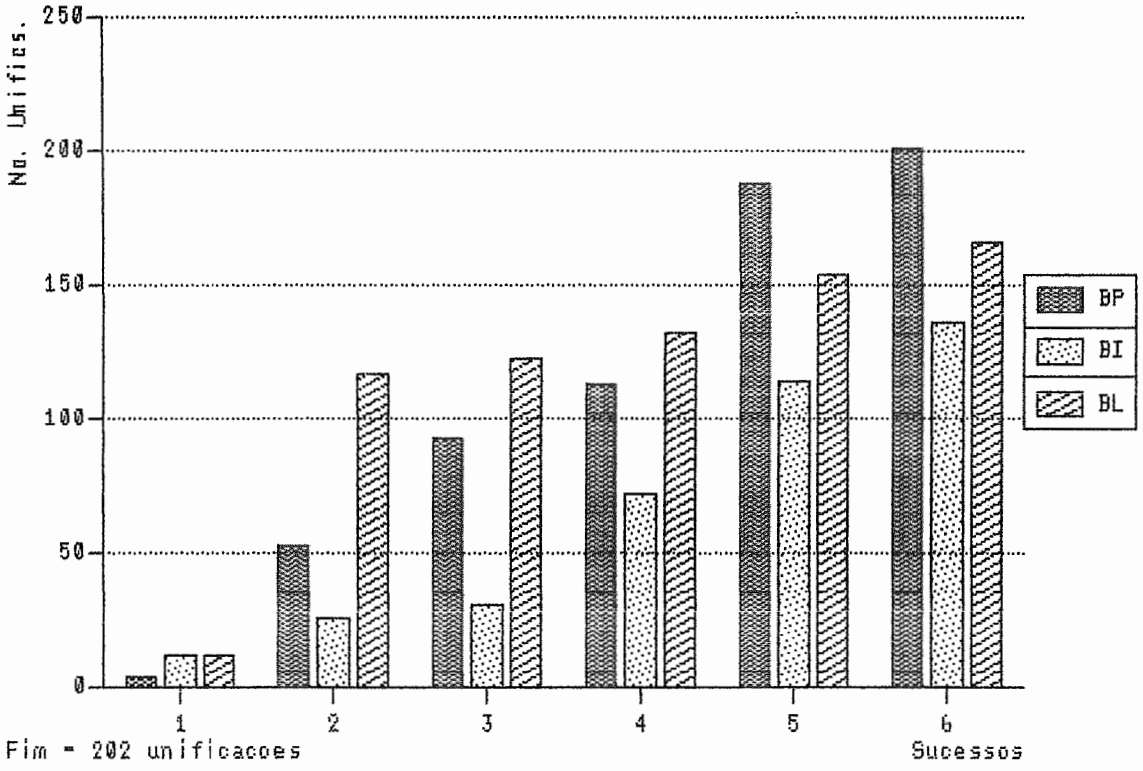
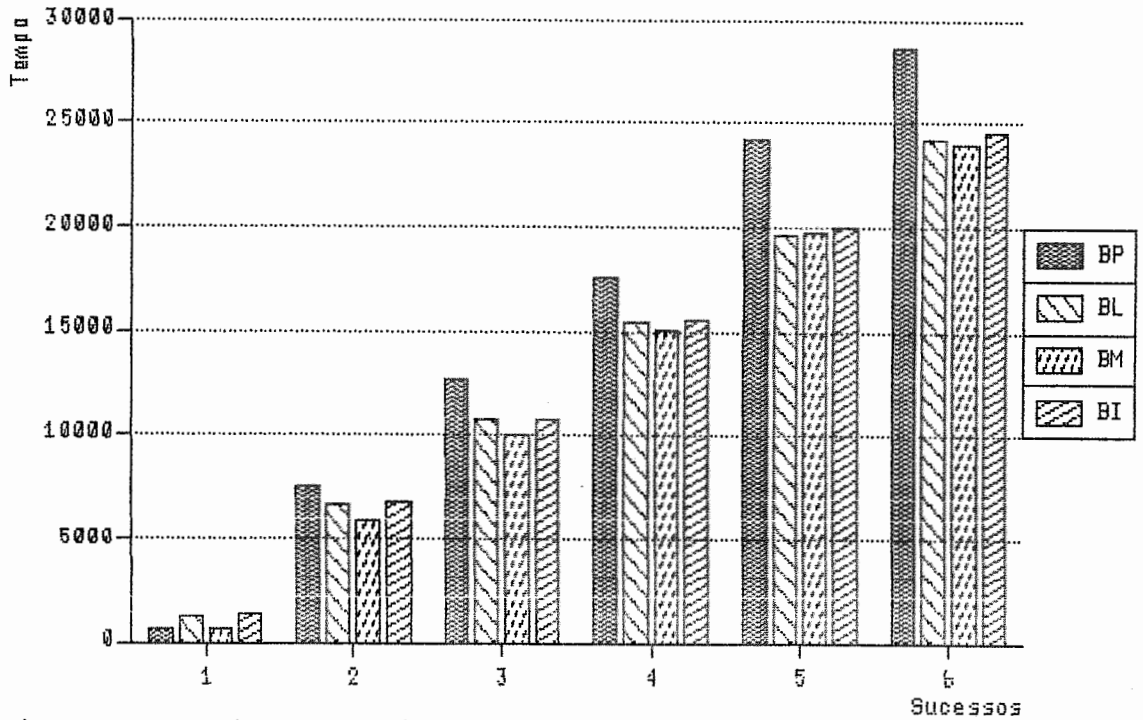


Grafico V.20 - Resultados de Execucao de paper (2a. fase)



Fim BP - 31092 Ticks
Fim BL - 27844 Ticks
Fim BM - 27803 Ticks
Fim BI - 28929 Ticks

	Sucesso	Busca em Prof.	Busca em Larg.	Busca Mista	Busca Padrao
Tempo	1	708	1228	702	1329
	3	12644	10793	10013	10784
	6	28628	24153	23986	24521
Numero de Unifics	1	4	18	4	15
	3	93	69	40	46
	6	201	165	179	127
Tamanho Memoria Objs	1	1	2	1	1
	3	8	6	3	5
	6	8	7	5	6
Tamanho de Obj	1	1	3	1	3
	3	3	3	3	3
	6	3	3	3	3
Numero de Falhas	1	0	0	0	0
	3	1	0	0	0
	6	4	1	3	1
Numero de Comunics	1	36	233	42	210
	3	651	643	431	495
	6	1157	1107	1077	907
Numero de Bytes	1	1718	6568	4432	6311
	3	8625	11161	8780	9498
	6	14305	16385	16044	14126
Acessos Memoria Dinamica	1	3	13	5	12
	3	29	31	22	24
	6	57	55	54	44
Acessos Memoria Estatica	1	1	5	1	4
	3	62	41	19	22
	6	154	123	133	87

Tabela V.10: Avaliação de Desempenho com paper (vários WP)

V.5 - Comentários dos Resultados

De acordo com a análise dos resultados obtidos, até aqui apresentada, pudemos verificar que:

- certas classes de aplicações apresentam pouca ou nenhuma possibilidade de exploração de paralelismo OR, o que faz com que um sistema utilizando paralelismo AND, ou mesmo uma implementação seqüencial tradicional, sejam mais indicados;
- o desempenho do modelo Gol-Log fica muito prejudicado pela utilização, como na nossa implementação, de arquiteturas com memória distribuída;
- como já mencionado, o processamento das heurísticas da estratégia de controle é bastante custoso; e
- nem sempre é interessante utilizar a estratégia de controle "inteligente" de Gol-Log. Somente para problemas onde exista boa possibilidade de exploração de paralelismo OR, exista um espaço de busca grande a ser explorado e as características do programa sejam adequadas. De qualquer forma, o próprio Gol-Log fornece outras estratégias para quando a padrão não for ideal.

Quanto a avaliação de Gol-Log em relação a outros modelos paralelos de programação lógica, não foi possível realizar uma comparação, já que os resultados finais do Simulador de Modelos de Execução Paralela de Programas Lógicos, da COPPE Sistemas/UFRJ, ainda não se encontram disponíveis.

Em relação aos resultados apresentados e analisados, esperamos ter sido imparciais, tendo evitado escolher programas simplesmente pela sua maior possibilidade de eficiência quando executados na nossa implementação. Nosso critério para escolha de programas para o 'benchmark' visou considerar apenas as classes de programas e a proximidade destes programas com aplicações reais. É claro, no entanto, que tais programas de forma alguma são representativos das aplicações reais, principalmente devido ao seu tamanho reduzido.

Apenas como curiosidade, um exemplo de programa que poderia ter sido incluído,

simplesmente para mostrar a aplicabilidade de Gol-Log (busca "inteligente" e paralelismo OR) é o seguinte:

```
? prog (L, E, P).

prog (L, E, P) :- p1 (L, mestre), p2 (ricardo, E, P).
prog (L, E, P) :- p1 (L, mestre), p2 (marcia, E, P).
prog (L, E, P) :- p1 (L, mestre), p2 (ines, E, P).
prog (L, E, P) :- p1 (L, mestre), p2 (ruy, E, P).
prog (L, ny, P) :- p1 (L, doutor), p2 (roberto, ny, P).

p1 (bra, mestre) :- p3 (ufrj), p4 (manha), p5 (marco).
p1 (bra, mestre) :- p3 (uerj), p4 (manha), p5 (marco).
p1 (bra, mestre) :- p3 (uerj), p4 (manha), p5 (agosto).
p1 (bra, mestre) :- p3 (puc), p4 (manha), p5 (marco).
p1 (bra, mestre) :- p3 (puc), p4 (manha), p5 (agosto).
p1 (ext, doutor).

p2 (roberto, ny, usa).

p3 (ufrj).
p3 (uerj).
p3 (puc).

p4 (manha).
p4 (tarde).
p4 (noite).
```

Na verdade este programa pode ser considerado tão "bom" quanto os outros, no entanto, foi feito "artesanalmente".

No apêndice podem ser encontradas tabelas resumindo todos os dados obtidos no 'benchmark'. A primeira tabela lista resultados para o primeiro sucesso de cada programa. A segunda tabela traz resultados para o sucesso médio e a terceira para o útimo sucesso.

CAPÍTULO VI

Conclusões

Nesta tese, foi proposto, descrito, implementado e avaliado um novo modelo de execução paralela de programas lógicos, denominado Gol-Log. A dissertação procurou concentrar-se nos pontos mais relevantes do modelo, tais como as soluções que ele apresenta para problemas comuns de modelos paralelos e as inovações trazidas por Gol-Log.

As principais características do modelo Gol-Log são:

- se baseia na exploração de paralelismo OR;
- utiliza uma busca "inteligente" (padrão), que visa reduzir a computação e chegar mais rapidamente às soluções dos programas;
- existe a possibilidade de aplicar outras estratégias de controle quando a padrão não for a ideal; e
- o modelo é orientado à arquiteturas paralelas que apresentem memória compartilhada.

Na implementação que fizemos, alguns pontos são, também, importantes:

- utilizamos uma arquitetura paralela com memória distribuída, o que, apesar de modificar a proposta original, levantou questões interessantes sobre as formas de implementar modelos do tipo de Gol-Log; e
- geramos informações relevantes, não só para a avaliação de Gol-Log, como também para o estudo da própria "estrutura" dos programas executados.

A avaliação realizada procurou ser a mais imparcial possível, tendo alcançado seus objetivos fundamentais: avaliar as heurísticas propostas para Gol-Log e analisar o seu desempenho dentro das limitações e escopo do 'benchmark' utilizado.

Como próximos caminhos a seguir, destacamos:

- comparar o desempenho de Gol-Log com outros modelos paralelos;
- estudar a possibilidade de utilização de paralelismo AND;
- otimizar o processamento das heurísticas, principalmente de H2;
- estudar a possibilidade de criar uma forma de modificar a estratégia de controle dinamicamente;
- definir a função de 'hash' para o controle dos ramos infinitos, e implementar este recurso; e
- implementar Gol-Log numa arquitetura adequada.

Os resultados obtidos permitem considerar Gol-Log um bom sistema, o qual pode, se implementado numa arquitetura adequada, apresentar bons resultados práticos.

Referências Bibliográficas

BIANCHINI (1989) Bianchini, R. *"Execução Paralela de Programas Lógicos"*. Projeto Final de Graduação, UFRJ, Dez. 1989.

BIANCHINI (1988) Bianchini, R., Dutra, Ines de C., Eizirik, L. M. R. & Amorim, C. L. de. "Em Direção a uma Estação Prolog de Alto Desempenho - Implementação da Máquina Virtual". *V Simpósio Brasileiro de Inteligência Artificial.* Natal, RN, Out. 1988.

BRUYNOOGHE (1982) Bruynooghe, M. "The Memory Management of Prolog Implementations". *A.P.I.C. Studies in Data Programming No. 16, Logic Programming.* K. L. Clark and S.-A. Tarnlund, Academic Press.

CASANOVA (1987) Casanova, M. A., Giorno, F. A. C. & Furtado, A. L. *"Programação em Lógica e Linguagem Prolog"*. Edgard Blucher, 1987.

CHANG (1985a) Chang, J.-H. & Despain, A. M. "Semi-intelligent Backtracking of Prolog Based on Static Dependency Analysis". *Proceedings of the 1985 International Symposium on Logic Programming.* July 1985.

CHANG (1985b) Chang, J.-H., Despain, A. M. & DeGroot, D. "AND- parallelism of Logic Programs Based on Static Data Dependency Analysis". *COMPCON Spring 85*, IEEE, Feb. 1985.

CIEPIELEWSKI (1984) Ciepielewski, A. & Haridi, S. "Control Activities in the OR-parallel Token Machine". *Proceedings of the IEEE International Symposium on Logic Programming.* IEEE Computer Society Press, Feb. 1984.

CIEPIELEWSKI (1986) Ciepielewski, A., Hausman, B. & Haridi, S. "Initial Evaluation of a Virtual Machine for OR-parallel Execution of Logic Programs". *Fifth Generation Computer Architectures.* Ed. J. V. Woods, North-Holland. IFIP, 1986.

CLOCKSIN (1981) Clocksin, W. F. and Mellish, C. S. *"Programming in Prolog"*. Springer-Verlag, 1981.

CONERY (1987) Conery, John S. *"Parallel Execution of Logic Programs"*. Kluwer Academic Publishers, 1987.

DEGROOT (1984) DeGroot, D. "Restricted AND Parallelism". *Proceedings of the International Conference on Fifth Generation Computer Systems.* Amsterdam: Elsevier/North-Holland, Nov. 1984.

DUTRA (1988a) Dutra, Ines de C. *"Implementação de uma Máquina Virtual Prolog - Tradução e Execução de Programas".* Tese de Mestrado COPPE/UFRJ, Nov. 1988.

DUTRA (1988b) Dutra, Ines de C., Bianchini, R., Eizirik, L. M. R. & Amorim, C. L. de. "Em Direção a uma Estação Prolog de Alto Desempenho - Tradução de Programas". *V Simpósio Brasileiro de Inteligência Artificial.* Natal, RN, Out. 1988.

DUTRA (1989) Dutra, I. C., Fuentes, M. G. S., Bianchini, R. & Mendes, S. B. T. "Um Modelo AND/OR Paralelo: Uma Visão Crítica". *IX Congresso da Sociedade Brasileira de Computação.* Uberlândia, MG, Jul. 1989.

GREGORY (1987) Gregory, S. *"Parallel Logic Programming in Parlog - The Language and its Implementation".* Addison-Wesley Publishing Company, 1987.

KACSUK (1990) Kacsuk, P. *"Execution Models of Prolog for Parallel Computers".* Research Monographs in Parallel and Distributed Computing. Pitman Publishing, London, 1990.

KASIF (1983) Kasif, S., Kohli, M. & Minker, J. "PRISM: a Parallel Inference System for Problem Solving". *Proceedings of the Logic Programming Workshop 83.* Lisbon: Universidade Nova Lisboa, June 1983.

LEBLANC (1988) LeBlanc, T. J., Scott, M. L. & Brown, C. "Large- Scale Parallel Programming: Experience with the BBN Butterfly Parallel Processor". *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming.* New Haven, CT, July 1988.

LIPOVSKI (1985) Lipovski, G. J. & Hermenegildo, M. V. "B-Log: A Branch and Bound Methodology for the Parallel Execution of Logic Programs". *Proceedings of the 1985 International Conference on Parallel Processing.* Aug. 1985.

MOTO-OKA (1984) Moto-oka, T., Tanaka, H., Aida, H., Hirata, K. & Maruyama, T. "The Architecture of a Parallel Inference Engine - PIE". *Proceedings of the*

International Conference on Fifth Generation Computer Systems. Amsterdam: Elsevier/-North- Holland, Nov. 1984.

SOHMA (1986) Sohma, Y., Satoh, K., Kumon, K., Masuzawa, H. & Itashiki, A. "A New Parallel Inference Mechanism Based on Sequential Processing". *Fifth Generation Computer Architectures.* Ed. J. V. Woods, North-Holland. IFIP, 1986.

WARREN (1977) Warren, D. H. D. "Implementing Prolog - Compiling Predicate Logic Programs". *D. A. I. Research Reports 39 & 40.* University of Edinburgh, 1977.

WARREN (1983a) Warren, D. H. D. "An Abstract Prolog Instruction Set". *Technical Note 309, Artificial Intelligence Center, Computer Science and Technology Division, SRI International,* Oct. 1983.

WARREN (1983b) Warren, D. H. D. "Runtime Environment for a Prolog Compiler". *SUNY at Stony Brook, Department of Computer Science, Technical Report no.83/02.*

WISE (1986) Wise, M. J. *"Prolog Multiprocessors".* Prentice-Hall, 1986.

WOO (1986) Woo, N. S. & Choe, K.-M. "Selecting the Backtrack Literal in the AND Process of the AND/OR Process Model". *Proceedings of the 1986 Symposium on Logic Programming.* 1986.

APÊNDICE

Resumo do 'Benchmark'

Neste apêndice apresentamos um resumo dos resultados obtidos para os dez programas listados no texto da tese.

As três primeiras tabelas apresentadas trazem os resultados de execução de Gol-Log, utilizando-se apenas um WP e o PM. As três tabelas seguintes mostram os resultados usando-se três WPs, além do PM. Cada grupo de tabelas traz uma tabela para o primeiro sucesso, uma para o sucesso médio e uma para o último sucesso.

	Tempo	NU	TMO	TO	NF	NC	NB	MD	ME
1 BP	983	12	1	1	0	84	1072	4	2
BL	655	4	1	1	0	49	689	3	1
BI	721	4	1	1	0	49	689	3	1
2 BP	5107	106	5	2	0	780	8855	31	26
BL	30713	590	22	2	4	4424	49179	165	142
BI	20435	379	16	2	0	2850	31758	108	91
3 BP	30180	136	2	14	0	7680	88564	124	91
BL	29442	139	2	14	1	7859	90630	126	94
BI	29975	136	2	14	0	7680	88564	124	92
4 BP	22713	284	2	7	4	4262	47506	97	90
BL	22093	285	2	7	5	4346	48437	99	96
BI	50892	535	5	12	4	9435	104687	177	181
5 BP	6575	140	6	2	5	1059	11698	62	49
BL	2512	55	4	2	0	355	4119	18	13
BI	6408	133	3	2	5	1030	11385	61	50
6 BP	10282	112	2	9	0	3913	43999	85	65
BL	19779	222	3	9	1	3848	43260	82	66
BI	14851	205	3	6	1	2498	28415	70	57
7 BP	3208	52	2	2	0	429	4938	19	15
BL	6090	111	4	2	0	910	10247	39	33
BI	3124	50	2	2	0	429	4938	19	15
8 BP	16082	293	4	3	5	2778	30930	99	88
BL	30972	547	9	3	14	5227	57964	181	169
BI	18370	305	9	3	2	2964	33066	98	84
9 BP	11669	102	7	4	2	2254	25947	64	56
BL	43884	398	16	4	12	8844	100545	255	242
BI	29786	281	7	4	8	5968	68063	172	168
1 BP	708	4	1	1	0	36	1718	3	1
0 BL	1080	12	2	3	0	105	2490	5	3
BI	1173	12	2	3	0	105	2490	5	3

1. concat
2. permute
3. fatorial
4. quicksort
5. inter
6. row-mul
7. sublist
8. permsort
9. dens
10. paper

NU - no. unifics.
TMO - tam. mem. obj.
TO - tam. de obj.
NF - no. falhas
NC - no. comunics.
NB - no. bytes
MD - mem. dinâmica
ME - mem. estática

Tabela A.1: Resumo do 'Benchmark' (primeiro sucesso, 1 WP)

	Tempo	NU	TMO	TO	NF	NC	NB	MD	ME
1 BP	9866	36	1	1	0	268	3076	10	6
BL	9457	28	1	1	0	325	3695	12	10
BI	9692	28	1	1	0	233	2693	9	7
2 BP	88956	875	5	2	17	6533	72461	250	234
BL	123350	1417	28	2	25	10606	117549	402	382
BI	118793	1218	26	2	12	9118	101057	345	319
3 BP									
BL									
BI									
4 BP									
BL									
BI									
5 BP	15846	239	6	2	7	1881	20566	107	85
BL	18388	285	6	2	10	2264	24720	126	105
BI	16393	238	3	2	9	1916	20945	110	91
6 BP									
BL									
BI									
7 BP	9351	91	2	2	0	704	7966	31	27
BL	10446	118	4	2	0	943	10608	41	36
BI	9575	89	2	2	0	704	7966	31	27
8 BP									
BL									
BI									
9 BP	26931	204	7	4	5	4557	52045	132	120
BL	48762	404	16	4	13	8943	101665	259	246
BI	35808	293	7	4	8	6343	72297	183	176
10 BP	12644	93	8	3	1	651	8625	29	62
BL	14372	122	9	3	0	930	11757	43	88
BI	10494	31	3	3	0	260	4208	13	17

1. concat
2. permute
3. fatorial
4. quicksort
5. inter
6. row-mul
7. sublist
8. permsort
9. dens
10. paper

NU - no. unifics.
TMO - tam. mem. obj.
TO - tam. de obj.
NF - no. falhas
NC - no. comunics.
NB - no. bytes
MD - mem. dinâmica
ME - mem. estática

Tabela A.2: Resumo do 'Benchmark' (sucesso médio, 1 WP)

	Tempo	NU	TMO	TO	NF	NC	NB	MD	ME
1 BP	18179	56	1	1	0	417	4697	15	10
BL	17621	52	2	1	0	417	4697	15	13
BI	18094	52	1	1	0	417	4697	15	13
2 BP	190135	1759	5	2	37	13145	145573	502	474
BL	191044	1775	28	2	40	13283	147106	508	505
BI	203619	1757	26	2	37	13145	145573	502	496
3 BP									
BL									
BI									
4 BP									
BL									
BI									
5 BP	23771	328	6	2	12	2591	28224	148	119
BL	23725	323	6	2	12	2595	28276	147	122
BI	24129	324	3	2	13	2596	28279	149	124
6 BP	25291	224	2	9	0	3913	43999	85	65
BL	24700	228	3	9	1	3933	44219	87	70
BI	22356	228	3	6	1	3061	34831	87	70
7 BP	18070	134	2	2	0	1009	11330	45	42
BL	17493	132	4	2	0	1009	11330	45	42
BI	18132	132	2	2	0	1009	11330	45	42
8 BP									
BL									
BI									
9 BP	34326	227	7	4	5	5126	58467	149	135
BL	53919	407	16	4	13	9006	102375	261	248
BI	51719	395	7	4	11	8646	98316	251	240
10 BP	28628	201	8	3	4	1157	14305	57	154
BL	27377	166	9	3	1	1079	13424	51	125
BI	26761	136	7	3	1	905	11460	42	98

1. concat
2. permute
3. fatorial
4. quicksort
5. inter
6. row-mul
7. sublist
8. permsort
9. dens
10. paper

NU - no. unifics.
TMO - tam. mem. obj.
TO - tam. de obj.
NF - no. falhas
NC - no. comunics.
NB - no. bytes
MD - mem. dinâmica
ME - mem. estática

Tabela A.3: Resumo do 'Benchmark' (último sucesso, 1 WP)

	Tempo	NU	TMO	TO	NF	NC	NB	MD	ME
1 BP	983	12	1	1	0	84	1072	4	2
BL	688	4	1	1	0	55	1077	5	1
BM	950	12	1	1	0	124	1831	8	2
BI	794	4	1	1	0	55	1077	5	1
2 BP	5107	106	5	2	0	780	8855	31	26
BL	12846	508	19	2	4	3898	43817	150	124
BM	6138	266	9	2	2	2019	23005	80	64
BI	10202	379	16	2	0	2900	32763	113	91
3 BP	30180	136	2	14	0	7680	88564	124	91
BL	25912	139	1	14	1	7865	91096	168	94
BM	24798	139	1	14	1	7865	91096	169	93
BI	27237	139	1	14	1	7865	91096	168	94
4 BP	22713	284	2	7	4	4262	47506	97	90
BL	16349	285	1	7	5	4352	49259	132	96
BM	16177	287	1	7	5	4352	49259	128	93
BI	41806	614	6	12	4	11177	124604	218	207
5 BP	6575	140	6	2	5	1059	11698	62	49
BL	1458	47	2	2	0	326	4347	19	10
BM	1874	88	4	2	0	494	6180	27	13
BI	1586	47	2	2	0	326	4347	19	10
6 BP	10282	112	2	9	0	3913	43999	85	65
BL	11980	219	3	9	1	3770	43082	82	63
BM	11803	225	2	9	1	3928	44857	94	65
BI	10355	222	2	6	1	2978	34581	89	64
7 BP	3208	52	2	2	0	429	4938	19	15
BL	3164	94	4	2	0	845	9958	40	27
BM	2410	73	3	2	0	698	8358	32	21
BI	3207	94	3	2	0	815	9629	38	26
8 BP	16082	293	4	3	5	2778	30930	99	88
BL	17445	544	9	3	13	5231	58638	185	167
BM	11569	400	7	3	9	3889	43808	142	122
BI	11349	349	9	3	4	3407	38574	118	99
9 BP	11669	102	7	4	2	2254	25947	64	56
BL	33640	398	16	4	12	8918	102196	263	242
BM	12003	164	10	4	4	3646	42544	109	95
BI	24520	297	10	4	8	6482	74684	191	179
1 BP	708	4	1	1	0	36	1718	3	1
0 BL	1228	18	2	3	0	233	6568	13	5
BM	702	4	1	1	0	42	4432	5	1
BI	1329	15	1	3	0	210	6311	12	4

1. concat
2. permute
3. fatorial
4. quicksort
5. inter
6. row-mul
7. sublist
8. permsort
9. dens
10. paper

NU - no. unifics.
TMO - tam. mem. obj.
TO - tam. de obj.
NF - no. falhas
NC - no. comunics.

NB - no. bytes
MD - mem. dinâmica
ME - mem. estática

Tabela A.4: Resumo do 'Benchmark' (primeiro sucesso, vários WPs)

		Tempo	NU	TMO	TO	NF	NC	NB	MD	ME
1	BP	9866	36	1	1	0	268	3076	10	6
	BL	9478	28	1	1	0	239	3081	13	7
	BM	9462	36	1	1	0	308	3835	16	6
	BI	9874	28	1	1	0	239	3081	13	7
2	BP	88956	875	5	2	17	6533	72461	250	234
	BL	81592	1433	26	2	25	10771	119828	411	385
	BM	64101	949	11	2	21	7181	80095	279	258
	BI	84322	1199	25	2	13	9027	100500	345	315
3	BP									
	BL									
	BM									
	BI									
4	BP									
	BL									
	BM									
	BI									
5	BP	15846	239	6	2	7	1881	20566	107	85
	BL	12181	297	5	2	11	2443	27187	141	112
	BM	10812	263	5	2	9	2094	23432	117	89
	BI	12855	300	7	2	11	2437	27121	141	113
6	BP									
	BL									
	BM									
	BI									
7	BP	9351	91	2	2	0	704	7966	31	27
	BL	7223	99	4	2	0	848	9990	40	28
	BM	6865	90	3	2	0	824	9736	39	27
	BI	8067	111	3	2	0	946	11072	45	33
8	BP									
	BL									
	BM									
	BI									
9	BP	26931	204	7	4	5	4557	52045	132	120
	BL	38142	401	16	4	12	8945	102499	263	243
	BM	20431	227	10	4	6	4975	57556	147	136
	BI	30635	312	10	4	9	6913	79547	204	189
10	BP	12644	93	8	3	1	651	8625	29	62
	BL	10793	69	6	3	0	643	11161	31	41
	BM	10013	40	3	3	0	431	8780	22	19
	BI	10784	46	5	3	0	495	9498	24	22

1. concat
2. permute
3. fatorial
4. quicksort
5. inter
6. row-mul
7. sublist
8. permsort
9. dens
10. paper

NU - no. unifics.
TMO - tam. mem. obj.
TO - tam. de obj.
NF - no. falhas
NC - no. comunics.
NB - no. bytes
MD - mem. dinâmica
ME - mem. estática

Tabela A.5: Resumo do 'Benchmark' (sucesso médio, vários WPs)

		Tempo	NU	TMO	TO	NF	NC	NB	MD	ME
1	BP	18179	56	1	1	0	417	4697	15	10
	BL	17694	52	1	1	0	423	5085	21	13
	BM	17276	56	1	1	0	421	5061	20	10
	BI	18051	52	1	1	0	423	5085	21	13
2	BP	190135	1759	5	2	37	13145	145573	502	474
	BL	139387	1767	26	2	40	13242	147101	512	503
	BM	130385	1765	11	2	38	13193	146554	507	477
	BI	151974	1749	25	2	37	13148	146055	506	494
3	BP									
	BL									
	BM									
	BI									
4	BP									
	BL									
	BM									
	BI									
5	BP	23771	328	6	2	12	2591	28224	148	119
	BL	16015	323	5	2	13	2566	28513	149	122
	BM	15188	305	5	2	12	2417	26908	139	107
	BI	16647	320	7	2	12	2538	28210	147	120
6	BP	25291	224	2	9	0	3913	43999	85	65
	BL	17046	228	3	9	1	3939	44985	94	70
	BM	16336	228	2	9	1	3935	44937	94	66
	BI	15368	228	2	6	1	3067	35597	97	70
7	BP	18070	134	2	2	0	1009	11330	45	42
	BL	14242	127	4	2	0	1011	11782	48	38
	BM	14027	132	3	2	0	1011	11782	49	39
	BI	15025	132	3	2	0	1015	11830	50	42
8	BP									
	BL									
	BM									
	BI									
9	BP	34326	227	7	4	5	5126	58467	149	135
	BL	43246	407	16	4	13	9012	103257	267	248
	BM	38500	407	10	4	13	9012	103257	271	247
	BI	43036	398	10	4	11	8830	101205	263	242
10	BP	28628	201	8	3	4	1157	14305	57	154
	BL	24153	165	7	3	1	1107	16385	55	123
	BM	23986	179	5	3	3	1062	15880	54	126
	BI	24521	127	6	3	1	907	14126	44	87

1. concat
2. permute
3. fatorial
4. quicksort
5. inter
6. row-mul
7. sublist
8. permsort
9. dens
10. paper

NU - no. unifics.
TMO - tam. mem. obj.
TO - tam. de obj.
NF - no. falhas
NC - no. comunics.
NB - no. bytes
MD - mem. dinâmica
ME - mem. estática

Tabela A.6: Resumo do 'Benchmark' (último sucesso, vários WPs)