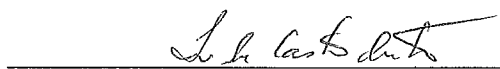
UM ESTUDO DO IMPACTO DE INFORMAÇÃO DE GRANULOSIDADE EM ESTRATÉGIAS DE ESCALONAMENTO PARA SISTEMAS PROLOG QUE EXPLORAM PARALELISMO-OU

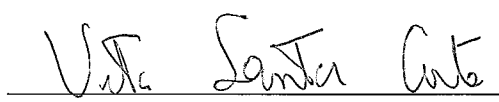
Sergio Brauna da Silva

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.

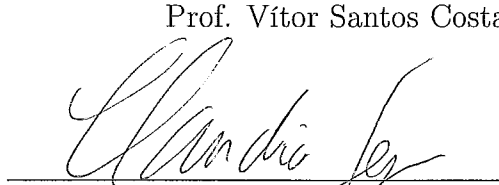
Aprovada por:



Prof. Inês de Castro Dutra, Ph.D.



Prof. Vítor Santos Costa, Ph.D.



Prof. Cláudio Fernando Resin Geyer, Dr.

RIO DE JANEIRO, RJ - BRASIL
MARÇO DE 2004

SILVA, SERGIO BRAUNA DA

Um estudo do impacto de informação de granulosidade em estratégias de escalonamento para sistemas Prolog que exploram Paralelismo-OU [Rio de Janeiro] 2004

X, 75 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2004)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1 - Paralelismo na programação em lógica

2 - Análise de granulosidade

3 - Políticas de escalonamento

I. COPPE/UFRJ II. Título (série)

Aos meus Pais Cideralina Mendes da Silva e José Brauna da Silva

Agradecimentos

A Deus, que pela sua infinita bondade e misericórdia, me concedeu saúde e inúmeras graças a fim de que conseguisse chegar ao fim de mais uma etapa. Querido Deus muito obrigado por ter colocado pessoas tão especiais em minha vida.

Em especial, à minha orientadora, Inês de Castro Dutra, pela confiança depositada no meu trabalho, pela grandeza humana e sua forma de ser, que a muito me impressiona.

Ao Professor, Herminio Zenóbio da Costa, que sempre acreditou no meu potencial. Sua vida constitui para mim um verdadeiro alicerce. É com muita alegria que me recordo de seus conselhos e orientações que me fizeram aqui chegar.

Ao Professor, Vítor Santos Costa, pessoa que muito admiro e respeito.

Ao Professor, Nelson Maculan Filho, pelo carinho demonstrado a mim ao longo desses anos aqui na COPPE.

Ao corpo administrativo da COPPE, Cláudia Helena Prata, Josefina Solange S. Santos, Maria Sueli Gonçalves, Maria Lucia Ramos de Paula, com os quais pude contar durante todo o meu curso desde o princípio.

A Adriana Mariano Carrusca, que desenvolveu a versão original do simulador utilizado nesta dissertação.

Aos perseverantes companheiros de batalha, José Afonso Lajas Sanches, Tatiana Cavalcanti Fernandes, Marluce Rodrigues Pereira, Ricardo Gonçalves Quintão. Em particular Patrícia Kayser Vargas, que com sua experiência nos passou valiosos conteúdos.

Aos amigos que diretamente ou indiretamente me ajudaram a realizar este trabalho, Thobias Salazar Trevisan, Marcelo Lobosco, Luís Otávio Rigo Júnior, Theogenes Terra Júnior, Ricardo Valença Ferreira, aos meus familiares pela compreensão e a família Zenóbio pelo incentivo.

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

UM ESTUDO DO IMPACTO DE INFORMAÇÃO DE GRANULOSIDADE EM ESTRATÉGIAS DE ESCALONAMENTO PARA SISTEMAS PROLOG QUE EXPLORAM PARALELISMO-OU

Sergio Brauna da Silva

Março/2004

Orientadora: Inês de Castro Dutra

Programa: Engenharia de Sistemas e Computação

Este trabalho tem por objetivo integrar informação de granulosidade gerada em tempo de compilação (análise estática) pela ferramenta ORCA (OR Complexity Analyzer) em um programa que simula o funcionamento paralelo de uma máquina Prolog com um número variável de processadores. Esta informação é aplicada no auxílio a decisões de escalonamento dos programas em lógica para as políticas *top-most*, *bottom-most*, menor-custo e para uma nova política chamada ORCA-Top.

Para estudar as políticas de escalonamento que já existiam no simulador, comparamos os resultados obtidos com e sem informação de granulosidade, e da nova política. Extraímos estes resultados numa situação ideal, quando nenhum custo para exploração do paralelismo é considerado e quando se leva em consideração custos arquiteturais (memória compartilhada e distribuída).

Quanto ao conjunto de benchmarks, utilizamos aplicações que possuem somente paralelismo OU.

Nossos resultados mostram que o desempenho dos programas Prolog é afetado positivamente quando introduzimos informação de granulosidade às decisões do escalonador. Mostramos que a medida *default* de complexidade pode ser muito "gulosa" ao atribuir pesos às cláusulas, o que pode levar a *speedups* menores. De forma geral, todas as aplicações estudadas melhoram seu *speedup* mínimo e máximo com a introdução de medidas de complexidade.

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

A STUDY OF THE IMPACT OF GRANULARITY INFORMATION IN SCHEDULING STRATEGIES FOR OR-PARALLEL PROLOG SYSTEMS

Sergio Brauna da Silva

March/2004

Advisor: Inês de Castro Dutra

Department: Computing and Systems Engineering

This work intends to integrate compile-time granularity information generated by the ORCA tool (static analysis) in a program that simulates the parallel execution of a Prolog machine with a varying number of processors.

This kind of information is helpful for scheduling decisions in logic programs execution, when choosing one of the following strategies: top-most, bottom-most and minor cost. That information was also used to create a new strategy that we called ORCA-Top.

In order to study the strategies already created, we compare the results obtained with and without granularity information, and the new strategy. We obtained these results under an ideal situation, when no cost for exploration of parallelism is considered and when we consider architectural costs (Shared and Distributed Memory).

The benchmarks used have only OR-Parallelism. The results showed that the performance of Prolog programs is positively affected when we add granularity information to the decisions made by the scheduler. We showed that the default complexity measure generated by ORCA may be very "greedy" when attributing weights to clauses, which can lead to smaller speedups. In general, all applications studied improve their minimum and maximum speedups when adding complexity measures.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

A exploração do paralelismo na programação em lógica é uma alternativa muito importante para obtenção de melhor desempenho. O paralelismo pode ser explorado de forma implícita ou explícita. Na primeira forma, implícita, o programador usa linguagens seqüenciais e a exploração do paralelismo fica a cargo do sistema de computação. Desta forma, o programador não se envolve com a paralelização das aplicações. Na segunda, explícita, a codificação do paralelismo é realizada pelo programador, utilizando para isso uma linguagem com construções especiais. Na programação em lógica existem algumas fontes de exploração de paralelismo implícito: paralelismo OU em que ocorre a execução paralela das cláusulas que compõem um predicado e o paralelismo E que faz a execução paralela dos objetivos que compõem a cláusula. Neste trabalho nos concentraremos apenas na exploração de paralelismo OU.

O paralelismo OU destaca-se como a principal fonte de paralelismo pesquisada na programação em lógica. As características do paralelismo OU que motivaram seu estudo são: **generalidade**, é possível explorá-lo sem restringir o poder da linguagem de programação em lógica usada; **simplicidade**, o paralelismo OU não exige anotações especiais do programador e não torna o processo de compilação muito complexo; **similaridade ao Prolog**, é possível compatibilizar o paralelismo OU com um modelo de execução bastante próximo ao do Prolog seqüencial, permitindo aproveitar toda a tecnologia existente para este na busca de melhor desempenho em cada processador, e ajudando a preservar a semântica do Prolog; **granulosidade**, o paralelismo OU tem o potencial, pelo menos para uma boa classe de problemas em Prolog, de permitir paralelismo com elevada granulosidade; e **aplicações**, um

significativo potencial de paralelismo OU é inerente a um grande grupo de aplicações, especialmente na área de inteligência artificial.

A análise de granulosidade na programação em lógica tem sido foco de atenção em alguns trabalhos na literatura [21, 39]. Barbosa [3] define grão como uma tarefa resultante do particionamento de um programa, a qual deverá ser executada seqüencialmente num único processador. Também define granulosidade como o tamanho do grão, isto é, o esforço computacional necessário para o processamento do grão (complexidade do grão). Dessa forma, análise de granulosidade consiste na análise do tamanho dos grãos, ou seja, a determinação da complexidade adequada para os módulos que deverão ser processados seqüencialmente num único processador. Esta análise consiste basicamente numa refinada identificação dos grãos, objetivando a máxima eficiência na exploração do paralelismo.

Uma das aplicações para a análise de granulosidade é o auxílio ao escalonamento de tarefas. Podendo ser realizada em tempo de compilação (análise estática), tempo de execução (análise dinâmica) ou ambos (análise combinada).

Segundo Geyer *et al.* [8] execução paralela ou distribuída significa que diferentes partes de um programa irão ser divididas em várias tarefas e essas tarefas associadas a processadores livres. O escalonador de tarefas é a entidade responsável por decidir o que, quando e para quem exportar tarefas. Todas as políticas de escalonamento devem distribuir tarefas para os recursos computacionais disponíveis de modo a melhorar uma ou mais medidas de desempenho. O escalonador é responsável pela utilização eficiente do processador. Logo, escalonamento é um aspecto fundamental em qualquer sistema paralelo e, conseqüentemente, é um importante campo de pesquisa em programação em lógica.

Em sistemas paralelos de programação em lógica um escalonador-OU tem a função de escolher as melhores alternativas que podem ser exploradas, de modo a utilizar os recursos da melhor forma.

Programas que apresentam uma razoável quantidade de paralelismo podem levar a um bom desempenho nos sistemas paralelos. Entretanto, na prática, existem vários casos que acabam contribuindo para uma piora do desempenho, mesmo quando a aplicação tem paralelismo suficiente para atender os processadores disponíveis, devido ao comportamento indevido das políticas de escalonamento.

Neste trabalho investigamos o impacto da utilização de informação de granulosidade no desempenho da execução paralela de programas Prolog que

têm paralelismo OU. Utilizamos para isto o simulador escrito por Carrusca [9], e modificamos o simulador para incorporar as informações de granulosidade fornecidas por um sistema denominado ORCA (OR Complexity Analyzer). Fizemos também algumas modificações no ORCA para produzir uma outra medida de complexidade diferente da medida *default* produzida pelo sistema, que atribui menor peso aos ramos da árvore de execução. ORCA determina os grãos existentes num programa em lógica e possibilita a obtenção de informações relacionadas com estes grãos. Essas informações serão utilizadas como auxílio a decisões de escalonamento.

1.2 Descrição do trabalho

O objetivo geral do nosso trabalho é a integração de informação de granulosidade gerada em tempo de compilação (análise estática) pela ferramenta ORCA em um programa que simula o funcionamento da máquina Prolog em uma máquina monoprocessada, programa este que simula a existência de mais de um processador.

Pretendemos com esta simulação avaliar diferentes estratégias de escalonamento quanto à quantidade de paralelismo alcançado e verificar o desempenho de cada uma na presença de custos de execução paralela associados a diferentes arquiteturas (memória compartilhada e distribuída), com ou sem a análise de granulosidade geradas pelo ORCA.

No simulador avaliamos políticas de escalonamento que já existiam, comparando os resultados com e sem informação de granulosidade. Avaliamos também uma estratégia que utiliza apenas informação de granulosidade e dá preferência a tarefas *top-most*.

1.3 Contribuições do autor

As principais contribuições deste trabalho são a inclusão de informação de granulosidade em tempo de compilação gerada pela ferramenta ORCA (OR Complexity Analyzer) no simulador, modificação do sistema ORCA para procurar outra medidas diferentes da *default* produzida pelo sistema e o estudo de estratégias de escalonamento na presença de informação de granulosidade.

1.4 Estrutura do texto

Esta tese está estruturada em seis capítulos, segundo descreveremos a seguir. O capítulo 2 apresenta conceitos básicos sobre programação em lógica, paralelismo e a linguagem Prolog, além de discutir sobre análise de granulosidade na programação em lógica, estratégias de escalonamento e sistemas que exploram o paralelismo OU em ambientes de memória distribuída e compartilhada.

No capítulo 3 descrevemos o funcionamento da ferramenta ORCA (OR Complexity Analyzer), que realiza análise estática de complexidade OU para programas em lógica, e suas medidas de complexidade: por resolução, por aridade e por unificação.

No capítulo 4 descrevemos as estruturas básicas do simulador: vetor de estados dos processadores, a fila de tarefas, ambiente de execução, relógio, atraso (*timer*) e nossas modificações para a inclusão das informações de granulosidade. Descrevemos como foi feita a inclusão das informações de granulosidade nas políticas de escalonamento e a forma como os custos podem ser aplicados a qualquer uma das cinco políticas de escalonamento utilizando ou não análise de complexidade geradas pelo ORCA.

No capítulo 5 descrevemos as 4 aplicações utilizadas no nosso trabalho e que se beneficiam do paralelismo OU, descrevemos nossa metodologia para realização dos experimentos, apresentamos e analisamos os resultados obtidos com as estratégias de escalonamento.

Finalmente, no capítulo 6, são apresentadas as conclusões e perspectivas de trabalhos futuros.

Capítulo 2

Paralelismo na Programação em Lógica e Análise de Granulosidade

2.1 Programação em lógica

A expressão "programa em lógica" é creditada a Robert Kowalski [28] (1974) e designa o uso da lógica como linguagem de programação de computadores.

A programação em lógica vem sendo aplicada para o desenvolvimento de muitas aplicações tais como processamento de linguagem natural, prova de teoremas, banco de dados, sistemas especialistas e processamento simbólico em geral.

Segundo Kowalski [29] uma das principais idéias da programação em lógica é de que um algoritmo é descrito por dois elementos disjuntos: a lógica e o controle. O componente lógico corresponde à definição do que deve ser resolvido, enquanto que o componente de controle estabelece como a solução pode ser obtida. Assim, o programador precisa apenas descrever o componente lógico de um algoritmo, deixando o controle da execução para ser exercido pelo sistema de programação em lógica utilizado. Desta forma, Programação em Lógica pode ser expressa por: Algoritmo = Lógica + Controle.

O paradigma fundamental da programação em lógica é o da programação declarativa, em oposição a programação procedimental típica das linguagens convencionais.

Um programa em lógica é a representação de determinado problema ou situação expressa através de um conjunto finito de tipo especial de sentenças lógicas denominadas cláusulas. Essas cláusulas podem ser dos tipos: fatos (assertivas), regras (procedimentos) ou consultas. A seguir são apresentados esses tipo, bem como conceitos básicos relacionados.

Fatos. Um fato denota uma verdade incondicional, é uma declaração simples que expressa relacionamento entre objetos. Quanto a forma sintática, um fato pode ser considerado como a cláusula sem um corpo. Um exemplo simples é:

```
estuda(carlos,computação).
```

Neste exemplo, utilizamos a sintaxe de Prolog (sintaxe de Edinburgh [42]), a linguagem representante de programação em lógica, onde constantes são iniciadas com letras minúsculas e variáveis são iniciadas com letras maiúsculas.

Este fato estabelece uma relação entre *Carlos* e *computação*, significando que *Carlos* estuda *computação*. A relação, estuda, é chamada de predicado. Os nomes *carlos* e *computação* são chamados de átomos. Nomes de predicado e átomos são iniciados por letras minúsculas.

Regras. As regras definem as condições que devem ser satisfeitas para que uma certa declaração seja considerada verdadeira. Enquanto um fato é sempre verdadeiro, as regras especificam algo que pode ser verdadeiro se algumas condições forem satisfeitas. Todas as regras possuem cabeça e corpo separadas pelo símbolo ":-". O corpo de uma cláusula, isto é, o lado direito do símbolo ":-", possui literais denominados objetivos (*goals*). Esses objetivos podem ser tanto predicados pré-definidos quanto predicados definidos pelo usuário. Veja o exemplo:

```
candidato(P,Y,Z):- estuda(P,Y),desempenho(P,Z).
```

Uma das formas de interpretação é a visão declarativa, que interpreta regras como axiomas lógicos. Para o exemplo dado, a interpretação seria: "Para todo P, Y e Z; P é candidato se P estuda Y e P obtém desempenho Z".

Para que determinado objetivo seja avaliado, é necessário que ocorra uma unificação. A unificação é o mecanismo de substituição de variáveis que faz duas expressões idênticas. Se a unificação sucede, os objetivos (*goals*) do corpo da cláusula são executados seqüencialmente da esquerda para a direita. Se a unificação falha em algum ponto, o sistema retorna para a última alternativa selecionada desfazendo (desreferencia) todas as ligações (*bindings*) feitas até então para aquela unificação, e uma nova alternativa é selecionada. Este mecanismo é denominado *backtracking* e consiste na tentativa de procurar uma outra alternativa que satisfaça o objetivo corrente. Dois objetivos unificam se: (1) eles são idênticos, ou (2) uma ou mais substituições aplicadas a dois termos os tornam iguais, resultando em uma instanciação mais geral.

Os objetivos *candidato(P,Y,Z)* e *candidato(X,computação,excelente)* unificam. Uma instanciação que torna os dois idênticos é: P é instanciada com X, Y é instanciada com *computação* e Z é instanciada com *excelente*.

Consultas. É um tipo especial de regra que não possui cabeça e, é sempre avaliada. As consultas são meios pelos quais é possível obter informação de um programa em lógica. Um exemplo de consulta:

```
candidato(X,computação, excelente).
```

Esta consulta é realizada através do uso de variáveis lógicas. O significado dessa consulta é: "Existe algum candidato X que estuda *computação* e obtém desempenho *excelente* ?". No exemplo, X é uma variável lógica. As variáveis podem ser classificadas quanto:

- à localização durante a execução (locais ou globais): As locais não aparecem em termos compostos; As globais aparecem sempre em termos compostos. Exemplo: p(X,f(X,a),Y). onde X é global, Y é local.

- ao tempo de vida (temporárias ou permanentes): As temporárias somente aparecem na cabeça e/ou no primeiro literal do corpo da cláusula; As permanentes podem aparecer na cabeça e a partir do segundo literal da cláusula. Exemplo: d((U*V),X,((DU*V)+(U*DV))):- d(U,X,DU),d(V,X,DV). onde V, X e DV são permanentes, e U e DU são temporárias.

- à criação (condicional ou incondicional): Condicional é criada e não instanciada antes de um ponto de escolha. Podem receber valores diferentes dependendo das cláusulas alternativas do ponto de escolha; Incondicional já vem instanciada quando o ponto de escolha é criado. Portanto, não podem mudar de valor.

A interpretação procedural das cláusulas de Horn formam a base da semântica operacional da programação em lógica. Cada predicado é identificado por um nome (a cabeça da cláusula) **p** e número **n** de argumentos (aridade), sendo normalmente referenciado como predicado **p/n**. Cada objetivo do corpo de uma cláusula pode ser considerado como uma chamada de procedimento. A execução de um programa Prolog começa com a colocação da consulta como resolvente inicial e prossegue pela transformação da resolvente corrente em uma nova resolvente, até a obtenção do

resultado final. Em um determinado passo da execução de um programa Prolog, o conjunto de objetivos que devem ser executados são chamados de *resolvente pendentes.*

Quando uma resolvente falha, entra em funcionamento o mecanismo de retrocesso (*backtracking*), o que leva a descartar parte do processamento e encontrar a próxima alternativa ainda não avaliada.

2.2 A Linguagem Prolog

Prolog é a linguagem representativa de programação em lógica. Ela é uma linguagem declarativa implementada em 1973 em Marseille como resultado de pesquisas nas áreas de processamento de linguagem natural e prova de teoremas.

Um sistema Prolog aceita os fatos e regras como um conjunto de axiomas e a consulta do usuário como o teorema a ser provado. Assim a tarefa do sistema é demonstrar que o teorema pode ser provado com base nos axiomas representados pelo conjunto das cláusulas que constituem o programa Prolog. O mecanismo de execução Prolog é busca em profundidade, com seleção de objetivos da esquerda para direita, em uma árvore de busca, utilizando quando necessário o *backtracking.*

A figura 2.1, apresenta um programa que consiste de três predicados: estuda/2 com três cláusulas tipo fato, desempenho/2 com três cláusulas tipo fato e candidato/3 com uma cláusula tipo regra. A cláusula "candidato(X,computação,excelente)" é uma possível consulta para este programa. Esta consulta pergunta quem são os candidatos X que são excelente em computação. A resposta do sistema Prolog consiste em um conjunto de objetos que satisfazem as condições estabelecidas pela consulta.

Programas Prolog interagem com o ambiente externo usando efeitos colaterais [26] para escrita/leitura em terminal ou em uma base de dados. Prolog também possui procedimentos internos (*builtins*) que permitem modificar a execução do programa (*assert* para adicionar cláusulas e *retract* para remover cláusulas).

Prolog tem um operador de corte assimétrico, o *cut.* Quando uma tarefa executa um *cut*, todos os ramos à direita do ramo onde está o *cut* são removidos da árvore. Extensões do Prolog (*Full Prolog* [27]) possuem um operador de corte simétrico, o *commit.* Quando uma tarefa executa um *commit*, todos os outros ramos são removidos da árvore. A utilização de desses operadores torna o programa mais

rápido, uma vez que ramos que não contribuem para determinada solução não necessitarão ser armazenados e o programa ocupará menos espaço de memória, uma vez que alguns pontos de retrocesso (*backtracking*) não precisarão ser armazenados.

A presença de operadores de corte em um programa Prolog introduz o problema de trabalho especulativo, o qual ocorre somente quando uma das muitas soluções para o problema é necessária. Um escalonador de trabalho especulativo deve avaliar a importância de diferentes tarefas antes de escolher uma para execução.

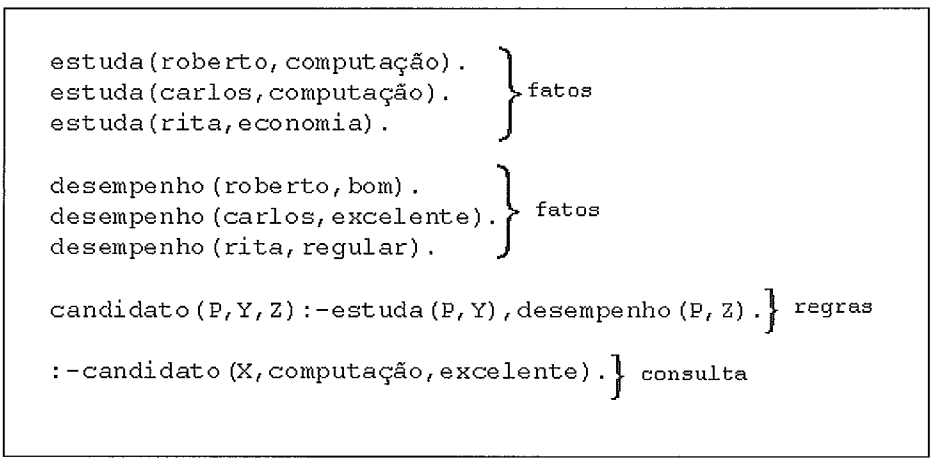


Figura 2.1: Um exemplo de programa Prolog

As primeiras implementações de Prolog eram interpretadas e, portanto, muito ineficientes. O primeiro passo para tornar o Prolog mais eficiente foi a construção do primeiro compilador Prolog, por David Warren. A WAM (Warren Abstract Machine) [1] é uma máquina abstrata formada por um conjunto de instruções, memória que inclui as pilhas de execução e área de código e um emulador.

O compilador é capaz de tornar execuções Prolog eficientes, mas mesmo assim tem se buscado alternativas para melhorar os sistemas Prolog. Uma das maneiras de se melhorar a eficiência de sistemas seqüenciais Prolog é a utilização do processamento paralelo.

2.3 Paralelismo

O processamento paralelo é uma opção para aumentar o desempenho de sistemas de computação. Sistemas de computação paralelos podem ser de arquitetura com memória distribuída, que são constituídos de vários processadores com sua própria memória local e interligados por uma rede local ou com memória compartilhada,

que consiste de vários processadores interligados por uma memória centralizada. Há outras classificações para arquiteturas paralelas, mas nosso foco neste trabalho está em arquiteturas de memória compartilhada e memória distribuída.

O termo paralelismo é usado para indicar que vários eventos estão ocorrendo ao mesmo tempo na computação. O objetivo é melhorar o desempenho dos programas, com relação a execução seqüencial. Porém nem sempre a paralelização de aplicações seqüenciais leva a exploração máxima dos recursos computacionais existentes. O principal motivo é o balanceamento de carga, ou seja, distribuição de carga de trabalho entre os recursos. Em Programação em Lógica, em particular, a computação pode ser irregular e as cargas mais difíceis de serem distribuídas.

O Prolog possui várias características que facilitam a exploração do paralelismo. Dentre elas destaca-se o não determinismo na escolha de caminhos que levem a solução, o que permite que cláusulas e objetivos possam ser executados, sob certas condições, de forma não seqüencial.

Segundo Geyer *et al.* [8] a maioria dos sistemas paralelos conhecidos são baseados em um modelo de programação imperativa e o paralelismo é explorado explicitamente, tornando a tarefa de programar mais difícil. Contrastando com a linguagem imperativa, a programação lógica/declarativa apresenta um modelo de programação de alto nível onde o programador precisa apenas se preocupar com o que resolver e não como resolver o problema.

Do ponto de vista do usuário existem duas abordagens para exploração de paralelismo [3, 8, 16, 18] que são:

- **Paralelismo explícito**: codificação do paralelismo pelo programador. Este tipo de paralelização permite que se diminua a complexidade dos compiladores, pois elimina a necessidade da detecção automática do paralelismo em tempo de compilação.

- **Paralelismo implícito**: detecção automática do paralelismo pelo ambiente de execução. Nesse caso a linguagem de programação não contém mecanismos e primitivas para paralelização dos programas. Nesta abordagem, o programador usa linguagens seqüenciais e a exploração do paralelismo fica a cargo do sistema de computação. Desta forma, o programador não se envolve com a paralelização das aplicações. Além disso, o paralelismo implícito permite a portabilidade dos programas, eliminando a necessidade da alteração

do código fonte em função da arquitetura seqüencial ou paralela desejada.

As duas principais fontes de paralelismo implícito existentes em programação lógica [8] são:

- **Paralelismo OU**: explora o paralelismo entre cláusulas de um mesmo predicado.
- **Paralelismo E**: explora o paralelismo existente no corpo de cada cláusula, ou seja, execução paralela dos diferentes objetivos que compõem uma cláusula.

Existem vários sistemas que exploram somente o paralelismo-E [24, 31], somente o paralelismo OU [5, 6] ou a combinação de ambos [13, 36], paralelismo OU e paralelismo E.

Devido a sua natureza declarativa, programas em lógica não seguem uma seqüência definida de operações, o que possibilita a execução de diferentes operações em qualquer ordem sem afetar a semântica do programa, exceto quando operações que possam afetar a semântica seqüencial são encontradas durante a execução: *cut*, *commit*, efeitos colaterais e suspensão. Dessa forma, pode-se explorar paralelismo implícito de programas em lógica. Isto significa que um programa em lógica escrito para rodar num sistema seqüencial pode rodar em um sistema paralelo sem modificações e ainda assim obter bom desempenho.

Entende-se por *suspensão voluntária* a capacidade do processador abandonar uma tarefa especulativa e mover-se para uma tarefa menos especulativa à esquerda na árvore de execução.

Segundo Geyer *et al.* [8] os dois principais problemas com a implementação do paralelismo OU, está no gerenciamento eficiente de múltiplos *bindings* da mesma variável correspondentes aos diferentes ramos da árvore de execução e o desenvolvimento eficiente de escalonadores. Um sistema que permite várias cláusulas serem processadas em paralelo precisa de um mecanismo para evitar conflitos de *bindings*. Em Prolog seqüencial este problema não ocorre porque cada ramo é executado um de cada vez e os *bindings* condicionais são anotados numa estrutura especial chamada trilha para serem desfeitos mais tarde no *backtracking*. No sistema paralelo, cada processador precisa da sua própria lista de variáveis condicionais, de modo que o processamento de cada ramo da árvore de execução possa ser feito separadamente e simultaneamente com outros processadores. Várias soluções têm

sido propostas para lidar com esse problema. Estas soluções se baseiam em um dos dois mecanismos básicos: cópia de pilhas ou compartilhamento de pilhas.

Na técnica **cópia de pilhas**, cada processador mantém uma cópia completa das pilhas na sua área de trabalho. Em busca de tarefa, um processador disponível copia a porção de pilha com o estado da computação anterior ao nó OU (*choice-point*), que contém a alternativa não explorada. Desta forma podem produzir múltiplos *bindings* para uma variável. O *overhead* da cópia pode ser reduzido através de cópia incremental das pilhas. Sistemas que usam essa técnica são: **Muse** [2] e **OPERA** [22].

Na técnica **compartilhamento de pilhas**, os processadores compartilham as partes das pilhas que têm em comum. Além disso, estruturas de dados especiais são utilizadas para armazenar as diversas ligações condicionais às variáveis das partes compartilhadas das pilhas. Um sistema que usa essa técnica é o **Aurora** [32].

2.4 Árvore de execução

O programa da figura 2.1 é simples e não possui operadores de corte (*cut* ou *commit*) ou efeitos colaterais.

Na figura 2.2 apresentamos, através de uma representação gráfica, a árvore de execução e a execução da consulta candidato(X,computação,excelente).

2.4.1 Execução seqüencial

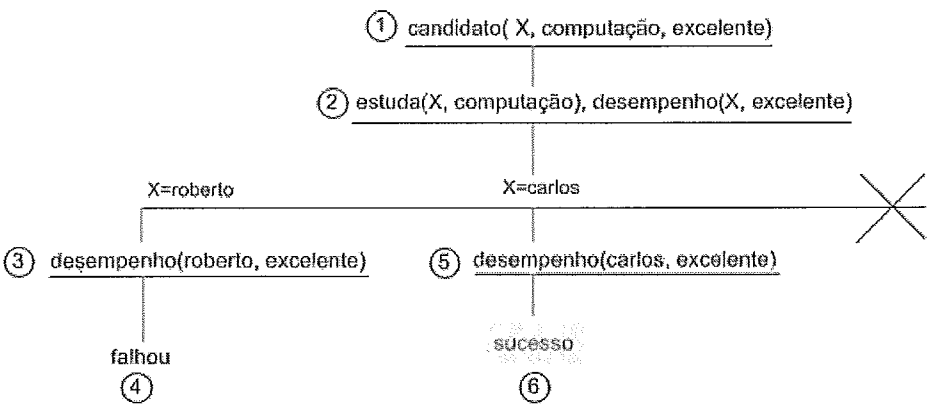


Figura 2.2: Árvore de busca seqüencial

De início a consulta é feita: "candidato(X,computação,excelente)". Este objetivo unifica com a cabeça do predicado candidato/3. Depois da unificação

da cabeça (a variável Y com "computação", e a variável Z com "excelente" e a variável X com P) passa-se a execução do corpo do procedimento. O Prolog tenta encontrar uma solução para os objetivos "estuda(X,computação)" e "desempenho(X,excelente)". Existem três alternativas para o predicado estuda/2.

Como a execução de Prolog procura alternativas no conjunto de cláusulas de acordo com a ordem textual do programa, X é instanciada primeiro com o valor roberto. Porém, não existe solução para desempenho(roberto,excelente).

Um nó na árvore de execução com mais de uma alternativa é chamado de ponto de escolha (*Choice Point).* Prolog, então, desfaz o *binding* X=roberto e procura a próxima alternativa para estuda(X,computação). Desta vez X é instanciado com carlos. Como existe uma cláusula (fato) que diz que carlos tem desempenho excelente, a computação sucede com a resposta X=carlos.

Embora existam outras ramos a serem explorados na árvore, não é necessário mais explorar porque a resposta já foi encontrada com valor "carlos" no nó 6. Se o usuário desejar outra solução, o sistema retrocede novamente e tenta a próxima solução com X=rita, que falha pois não existe fato desempenho(rita,excelente).

2.4.2 Execução paralela

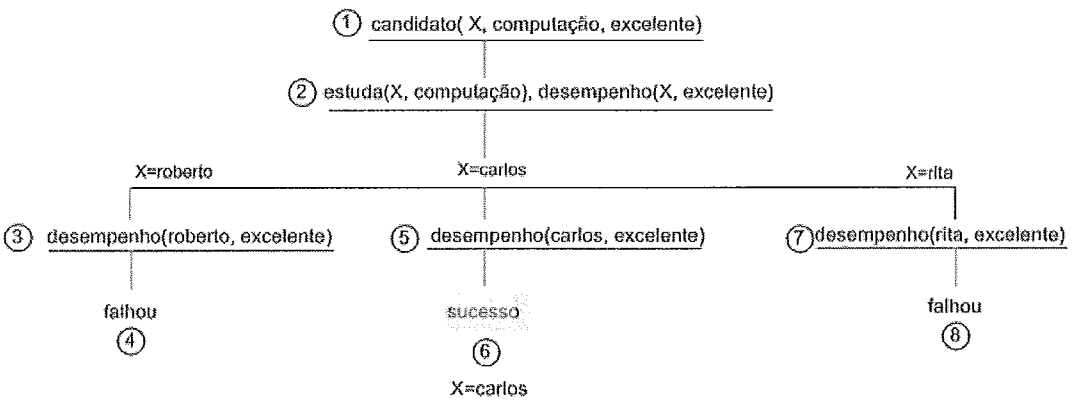


Figura 2.3: Árvore de busca paralela

A figura 2.3 mostra que os ramos da árvore de execução podem ser explorados em paralelo; o nó 2 possui um potencial de paralelismo, que é a exploração em paralelo das várias alternativas para estuda(X,computação). Este tipo de paralelismo é conhecido como paralelismo OU.

Vamos assumir que existem 3 processadores: p1, p2 e p3. p1 assume a consulta, dando início a execução do programa. Os outros processadores (p2 e p3) como

não conseguem encontrar nenhum trabalho para executar ficam em estado livre. Depois de algum tempo p1 cria o nó 2. Este nó tem três alternativas. p1 pode continuar (como na execução seqüencial) com a alternativa mais a esquerda (*left-most*). p2 e p3 podem pegar uma alternativa cada. Digamos que p2 pegou a segunda alternativa e p3 pegou a terceira (última alternativa). Os processadores p1 e p3 falham imediatamente, enquanto p2 consegue terminar com sucesso e produzir a solução X=carlos. Note que nem todos os nós precisam ser visitados para se chegar a uma solução.

2.5 Análise de granulosidade na programação em lógica

Segundo Barbosa [3], grão é definido como uma tarefa resultante do particionamento de um programa, a qual deverá ser executada seqüencialmente num único processador. Granulosidade é definida como o tamanho do grão, ou seja, o esforço computacional necessário para o processamento do grão (complexidade do grão). A análise de granulosidade é definida como a determinação da complexidade adequada para os módulos que deverão ser executados seqüencialmente num único processador, ou seja, análise do tamanho do grão.

Alguns trabalhos na literatura [14, 17, 25, 30, 39] propõem métodos para análise de granulosidade em programação em lógica. Alguns deles propõem análise de granulosidade em tempo de compilação (análise estática), outros propõem análise em tempo de execução (análise dinâmica) enquanto outros propõem de forma combinada (parcialmente em tempo de compilação e parcialmente na execução). A forma combinada, tem por objetivo trazer um balanceamento entre as análises estática e dinâmica, visando baixo custo adicional na execução e maior precisão nas decisões de escalonamento.

Conforme Barbosa [3] a execução eficiente de um programa paralelo depende do dimensionamento adequado dos grãos. Esta é a tarefa da análise de granulosidade.

O GRANLOG (Granularity Analyzer for Logic Programming) [3, 4] é um modelo para análise automática de granulosidade na programação em lógica. O modelo realiza uma análise estática, gerando informações que podem ser utilizadas dinamicamente no processo de escalonamento. O modelo propõe uma análise tipo combinada, ou seja, a realização da análise de granulosidade em tempo de compilação

e execução. Alguns trabalhos na literatura mostram indícios de que análise de granulosidade pode ajudar na tarefa de escalonamento [20, 37, 41].

2.6 Estratégias de escalonamento

A execução de programas Prolog em sistemas paralelos de programação em lógica tem uma característica muito irregular, visto que a árvore de execução Prolog é gerada dinamicamente. As estratégias de escalonamento para atribuição de tarefas a processadores variam principalmente na ordem em que as tarefas são escolhidas na árvore de procura. Algumas delas, as mais recentes, lidam com trabalho especulativo. As estratégias mais utilizadas são *top-most*, *bottom-most* e *left-most*.

Na estratégia *top-most* dá-se preferência às tarefas que estão na parte mais próxima da raiz (assemelha-se a uma busca em largura). Emprega-se a heurística de que quanto mais próximo da raiz estiver o nó a ser explorado maior a sua granulosidade. Esta tem como objetivo minimizar a freqüência de troca de contexto (*task-switching*) e reduzir a interação entre os processadores. Segundo Gupta [23] a desvantagem desse tipo de abordagem é que tarefas pequenas geram um grande número de acessos ao escalonador, levando a um *overhead* significante.

Na estratégia *bottom-most* dá-se preferência às tarefas que estão em níveis mais alto da árvore de execução (implementa uma busca em profundidade). Esta técnica tem por objetivo maximizar a localidade de referências à memória, diminuindo, assim, o custo de instalação e de-instalação de *bindings* na troca de contexto. Essa característica é fundamental porque quanto mais longe da tarefa o processador estiver, maior o *overhead* de compartilhamento do mesmo.

A estratégia *left-most* consiste em liberar trabalho a partir do nó mais à esquerda na árvore de execução, o que requer manter a ordenação da esquerda para a direita dos ramos na árvore. É usada para manter compatibilidade com a execução seqüencial e gerar menos trabalho especulativo na presença de operadores de corte.

Exploração de paralelismo OU em programação em lógica é um tópico bastante estudado e vários sistemas foram desenvolvidos. Dentre os mais populares, podemos citar OPERA e PLoSys para arquitetura de memória distribuída ; Muse, Aurora e YapOr [35] para ambiente de memória compartilhada.

2.7 Sistemas de paralelismo OU

OPERA [22] foi implementado em um **Supernode**, um multiprocessador com memória distribuída baseado em Transputers [7]. Cada trabalhador no sistema OPERA é um TWAM (Transputer Warren Abstract Machine), executando a sua cópia local do programa Prolog em questão. Nenhum paralelismo é disparado pelos trabalhadores ativos. Os trabalhadores ociosos, pedem tarefa aos ativos, que dispõem de alternativas ainda não exploradas nos seus nós OU (*Choice-points*). Ao receber a tarefa nova para processar o trabalhador inativo torna-se ativo.

Em busca da máxima granulosidade, o escalonador Opera seleciona o trabalho disponível mais alto na árvore, segundo a heurística de, quanto mais alto estiver a alternativa selecionada (mais próxima da raiz), maior a granulosidade do trabalho que esta contém.

PLoSys é um sistema que explora de forma implícita o paralelismo OU na programação em lógica e que foi desenvolvido para ambientes de arquitetura paralela com memória distribuída. Escolhe tarefas mais acima na árvore de busca utilizando a estratégia *top-most dispatching*. O escalonador do PLoSys é centralizado, mas já existe uma proposta de escalonamento distribuído para o PLoSys [10, 11]. No PLoSys, a carga de trabalho é medida pelo número de pontos de escolha não explorados em cada processador. O processador pode assumir três estados: *idle*, *quiet*, *overload*. Estes estados são definidos conforme o número de pontos de escolha criados e ainda não explorados.

No estado *idle* o processador não tem qualquer tarefa Prolog para executar. Este então pede trabalho para um processador *overloaded*. Após a exportação, o processador *idle* torna-se *quiet*, já que possui trabalho para executar.

No estado *quiet* o processador está ativo mas não tem trabalho suficiente para dividir (exportação de tarefa) com um *idle*.

Quando o processador está com carga de trabalho acima do limite, é colocado em estado *overload*, podendo então dividir (exportação de tarefa) com um processador *idle*.

No modelo centralizado, quando o processador termina a fase de importação e exportação de tarefas, os processadores enviam ao escalonador a nova carga de trabalho.

Diferente de sistemas implementados para arquiteturas de memória distribuída,

escalonadores de sistemas de memória compartilhada não precisam lidar com o problema de comunicação. Em vez disso, o maior problema a ser resolvido pelo escalonador OU é controlar a freqüência de acessos à fila de tarefas do sistema, que não tem tempo constante para essa operação. O maior problema é manter localidade de dados.

O sistema **Muse** [2] foi o primeiro a implementar a idéia de escalonar em *bottom-most*. Podemos destacar algumas de suas características:

- árvore de busca é dividida em privada e pública (compartilhada);
- trata de vários tipos de aplicações inclusive trabalho especulativo.
- O processador pode assumir dois modos: **(a)** *modo escalonador*, quando entra na parte compartilhada da árvore de procura ou quando executa efeitos colaterais; **(b)** *modo máquina*, quando sai do modo escalonador. No *modo máquina*, o processador trabalha exatamente como em uma máquina seqüencial Prolog em nós privados.
- processadores somente procuram por trabalho mais a esquerda (*leftmost*) quando ocorre uma suspensão voluntária. Não ocorre suspensão voluntária se existem processadores em estado ocioso. Se um processador ocioso está procurando trabalho, a estratégia é a seguinte: **(a)** Procura trabalho disponível mais próximo dentro do seu ramo; **(b)** Se não existir trabalho disponível, o processador tenta selecionar um processador sobrecarregado; **(c)** Se não encontrar trabalho, o processador ocioso situa-se em uma posição apropriada no ramo para que possa pegar trabalho mais tarde.

Um dos problemas do escalonador Muse é que a suspensão voluntária também ocorre em trabalho obrigatório (isto é, não especulativo). Muse utiliza cópia de pilhas.

Aurora [32] é um sistema de programação lógica que explora somente paralelismo OU. Foi desenvolvido para arquitetura de memória compartilhada. Nesse sistema, assim como no Muse, a árvore de busca é dividida em privada e pública. Cada processador busca primeiro trabalho na parte privada, se não houver mais trabalho privado, ele tentará a busca na parte pública da árvore. Através de uma interface utilizada no sistema, Aurora pode utilizar diversos escalonadores. Pelo menos quatro já foram implementados com bons resultados.

O primeiro deles, o escalonador de **Argonne** propõe uma estratégia de busca por tarefas mais próximas da raiz, seguindo a heurística já mencionada que quanto mais alto está o trabalho na árvore de execução maior é sua granulosidade. O objetivo é minimizar o custo de movimentação na árvore de busca, e minimizar o tamanho da região pública para reduzir a interação entre processadores. Este primeiro escalonador não tratava da execução de *cuts* ou efeitos colaterais.

O escalonador **Manchester** também propõe uma estratégia de busca por tarefas mais próximas da raiz, a estratégia *Top-most*. Foi desenvolvido para solucionar os pontos fracos do escalonador Argonne. Neste escalonador, assim como no Muse e Argonne, a árvore de busca é dividida em privada e pública. Essas informações diminuem o deslocamento desnecessário, e o custo total de movimentação na árvore de execução. Uma outra diferença é a simetria entre a liberação e requisição de trabalho. O escalonador Manchester está um passo a frente do Argonne na questão de lidar com operadores de corte e efeitos colaterais.

No escalonador **Bristol**, os processadores que estão trabalhando, eventualmente e voluntariamente, suspendem a execução para procurar por trabalho menos especulativo gerado mais à esquerda na árvore de execução. Processadores ociosos procuram por trabalho dos processadores mais sobrecarregados e obtêm trabalho dos nós de nível mais alto da árvore, de modo a manter localidade. Desta forma, o escalonador Bristol utiliza estratégia *bottom-most*. O escalonador usa contadores de operadores de corte para verificar se o trabalho é ou não especulativo. Este escalonador dá preferência a trabalho obrigatório e está sempre procurando por melhor trabalho, isto é, menos especulativo, se estiver fazendo trabalho especulativo.

Várias estratégias foram implementadas no escalonador Bristol [5, 6]: **(a)** *Start High*, forma um nó público de cada vez; **(b)** *Start Rich*, processador ocioso escolhe o ramo mais rico e todos os nós são feitos públicos de uma só vez; **(c)** *Start Left*, sempre escolhe a tarefa mais a esquerda, todos os nós do ramo mais a esquerda são feitos públicos; **(d)** *Start Random*, todos os nós de um ramo qualquer (*random*) tornam-se públicos - em oposição ao ramo mais rico (*richest*).

Em (a) aplica-se a estratégia *top-most*. Em (b), (c) e (d) é aplicado o escalonamento *bottom-most*.

Em [6], são tratadas duas subestratégias. Se o processador considerar seu trabalho como mandatório, ele completará a execução (*finish work*); se ele considerar o trabalho como especulativo, periodicamente ocorrerá uma suspensão voluntária a

procura de uma nova tarefa (*seek better work*) mandatória ou menos especulativa.

O escalonador **Dharma** foi implementado com dois objetivos: controlar melhor a execução de trabalho especulativo em relação ao escalonador Bristol e diminuir os acessos compartilhados aos nós da árvore de execução. Para isto o escalonador Dharma assume que toda a árvore é especulativa. A estratégia de escalonamento busca concentrar os processadores na região mais profunda e mais à esquerda na árvore de busca (implementação *bottom-most*), e permite suspensão voluntária. Portanto tenta minimizar o trabalho especulativo dos processadores. O escalonador Dharma é implementado segundo o algoritmo *branch-level dispatching*. Para reduzir o número de acessos aos nós compartilhados da árvore, o Dharma utiliza as estruturas de dados *tip* e *active chain*. Um *tip* representa um ramo da árvore de busca. A *active chain* é formada por um conjunto de *tips*, isto é, uma lista duplamente encadeada que permite rapidamente a localização de trabalho mais a esquerda (ou menos especulativo) e evitar ter que percorrer os nós compartilhados da árvore de execução para procurar trabalho, como é feito no escalonador Bristol. Quando um novo ramo é criado na árvore, um novo *tip* é criado na *active chain*, e quando um ramo é excluído, o *tip* correspondente é removido da *active chain*.

2.8 Resumo

Neste capítulo apresentamos os conceitos básicos relativos à programação em lógica, descrevendo a linguagem Prolog e a motivação para paralelização de execução de programas em lógica. Discutimos sobre formas de paralelização e nos concentramos em exploração de paralelismo OU. Para aumentar a eficiência na execução paralela dos programas em lógica, alguns sistemas têm se beneficiado do uso da análise de granulosidade em suas execuções. Por último destacamos a importância do componente escalonador e suas estratégias e o escalonamento de sistemas OU paralelos de programação em lógica com memória distribuída, e com memória compartilhada. No capítulo seguinte descrevemos o modelo ORCA, utilizado em nosso trabalho, que realiza análise estática de complexidade OU para programas em lógica.

Capítulo 3

ORCA

O ORCA (OR Complexity Analyzer) [12, 40] é um sistema que realiza análise estática de complexidade OU para programas em lógica, através da análise dos códigos fontes Prolog. Estas informações são aplicadas no auxílio a decisões de escalonamento dos programas em lógica. Esta característica mantém o ORCA num alto nível de abstração, permitindo portabilidade e flexibilidade para suas anotações.

O ORCA possui como entrada o programa em lógica e como saída o programa em lógica acrescido das informações de complexidade OU (programa anotado). O programa anotado é composto do programa em lógica acrescido de *anotações de granulosidade.* As anotações de granulosidade armazenam as informações resultantes da análise de granulosidade.

O modelo ORCA está organizado em dois módulos, como ilustrado na figura 3.1.

O primeiro módulo lê o texto do programa em lógica e cria uma tabela contendo o nome de todos os procedimentos e suas respectivas cláusulas, bem como uma lista de chamadas realizadas por cada uma destas. Esta tabela contém um espaço reservado para os valores de complexidade para cada procedimento e para cada cláusula.

O segunda módulo realiza o cálculo de complexidade de cada procedimento e de cada cláusula, gerando então uma complexidade para cada procedimento. Também ocorre nessa segunda parte a geração de saída, ou seja, o texto do programa fonte com suas respectivas anotações de complexidade OU.

O que diferencia um procedimento dos demais na linguagem Prolog é o seu nome junto com sua aridade. Portanto, os procedimentos com mesmo nome e com aridade diferentes são procedimentos distintos. Desse modo, o nome de cada procedimento e cláusula é feito utilizando-se o nome do procedimento adicionado a aridade. Para separar essas informações é usado o caracter "/".

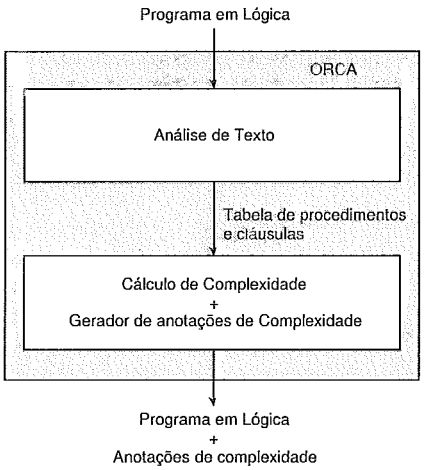


Figura 3.1: Estrutura do modelo ORCA

ORCA utiliza diferentes medidas para calcular o tamanho dos grãos. A primeira medida, resolução, conta o tamanho de uma cláusula através da contagem do número de resoluções simples que o programa executaria. Uma segunda medida é a aridade, que conta o número de argumentos que o predicado possui. Uma terceira alternativa de medida de complexidade são as unificações, que consiste em tomar dois termos e tentar torná-los idênticos. Para cada unificação bem sucedida dos dois termos acrescenta-se uma unidade ao valor da granulosidade para aquele predicado.

ORCA 2.0 tem como medida de complexidade *default* resoluções. Modificamos a versão 2.0 para incluir uma nova medida de complexidade, por aridade. A medida de complexidade por unificação ainda não foi implementada.

O funcionamento do sistema ORCA é simples. Para utilizá-lo basta chamar o programa na linha de comando do sistema operacional. Caso não especificada a medida de complexidade, o ORCA assumirá o cálculo de complexidade utilizando resolução por *default*. Se o arquivo de saída não for especificado, o ORCA realizará todos os cálculos necessários e escreverá as informações de complexidade na tela (saída padrão). Caso contrário o ORCA irá gerar um arquivo com o programa fonte acrescido de suas anotações de complexidade. As medidas de complexidade utilizadas pelo ORCA são definidas pelas opções: -r (para resolução), -a (para aridade) e -u (para unificação). Formato do comando: ORCA [medida-complexidade] arquivo-fonte [arquivo-destino]

As informações geradas pelo ORCA são: um fato cujo nome são os caracteres `'__c_'` acrescido do nome do procedimento e sua aridade. Como parâmetros têm-se: o valor da complexidade do procedimento, números de cláusulas do procedimento,

uma lista com a complexidade de cada cláusula deste procedimento, a complexidade da chamada recursiva e uma lista dos procedimentos mutuamente recursivos a ele. No exemplo de saída retirado da figura 3.3:

$'\_\_c\_hanoi/5'(46, 2, [1, 45], 16, [\ ])$, é a informação gerada pelo ORCA onde

$\_\_c\_$ = caracteres do ORCA;

hanoi = nome do procedimento;

5 = aridade do procedimento;

46 = complexidade total do procedimento;

2 = número de cláusulas do procedimento;

1 = complexidade da primeira cláusula;

45 = complexidade da segunda cláusula;

16 = complexidade da chamada recursiva;

[ ] = lista de procedimentos mutuamente recursivos;

Os cálculos de complexidade (por resolução, por aridade e por unificação), descritos a seguir, são baseados no programa *Hanoi*, figura 3.2. As anotações após o símbolo % são comentários de linha de Prolog e serão usadas na explicação de cálculo da complexidade.

```
hanoi(1,A,B,C,[mv(A,C)]).          % Ca1- Complexidade fato/procedimento hanoi
hanoi(N,A,B,C,M):-                 % Ca2- Complexidade regra/procedimento hanoi
        N > 1,
        N1 is N-1,                 % Ca2builtins - Complexidade do builtins
        hanoi(N1,A,C,B,M1),        % Ca2rec1- Complexidade recursiva 1
        hanoi(N1,B,A,C,M2),        % Ca2rec2- Complexidade recursiva 2
        append(M1,[mv(A,C)],T),
        append(T,M2,M).

append([],L,L).                    % Cb1- Complexidade fato/procedimento append
append([H|L],L1,[H|R]):-           % Cb2- Complexidade regra/procedimento append
        append(L,L1,R).            % Cb2rec- Complexidade recursiva/
                                   % Cb2meta - Complexidade da meta
```

Figura 3.2: Programa fonte hanoi

A complexidade por resolução, conforme mostra a figura 3.3, é calculada da seguinte forma:

1. A complexidade da primeira cláusula do segundo procedimento (C_{b1}) é 1 (uma resolução para a solução), por esta ser um fato. C_{b1}=1;

2. A complexidade de uma regra é a soma da complexidade dos procedimentos chamados no seu corpo, acrescidos de 1 (resolução da cabeça da regra). A segunda cláusula do segundo procedimento possui apenas uma meta, que é uma chamada recursiva. Assim soma-se o valor da complexidade da primeira cláusula (C_{b1}=1), mais o valor da complexidade da cabeça (C_{b2cab}=1), mais a complexidade da sua chamada recursiva(C_{b2meta}=1). A complexidade de uma chamada recursiva possui valor 1 por definição. Deste modo obtém-se a complexidade da chamada recursiva (C_{b2rec}=3). Para achar agora a complexidade da cláusula soma-se o valor da chamada recursiva (C_{b2rec}=3) ao valor da complexidade da cabeça (C_{b2cab}=1), assim obtém-se a complexidade. C_{b2}=4;

3. Para achar a complexidade do segundo procedimento (C_b), soma-se a complexidade das suas cláusulas ($C_{b1}+C_{b2}$=5);

4. A complexidade da primeira cláusula do primeiro procedimento (C_{a1}) é 1 (uma resolução para a solução), por esta ser um fato. C_{a1}=1;

5. Para calcular a complexidade da segunda cláusula do primeiro procedimento (C_{a2}), soma-se a complexidade da cabeça, com a complexidade do seu corpo. Observe que no corpo existem duas chamadas recursivas (C_{a2rec1} e C_{a2rec2}). Para obter a complexidade das chamadas recursivas, soma-se a complexidade da primeira cláusula (C_{a1}=1), com a complexidade da cabeça da segunda cláusula (C_{a2cab}=1), mais a complexidade dos built-ins ($C_{a2built\text{-}ins}$=2), mais a complexidade das duas chamadas para o procedimento *append* (5+5) e mais a complexidade das duas chamadas recursivas (C_{a2rec1}=1 e C_{a2rec2}=1). Deste modo, somando-se acha-se o valor da chamada recursiva (C_{a2rec}=16). Com isto achou-se a complexidade da chamada recursiva, e para achar a complexidade geral da cláusula, soma-se a complexidade da cabeça da cláusula (C_{a2cab}=1), mais a complexidade dos built-ins(2), mais a complexidade das duas chamadas recursivas (16+16), e por fim mais a complexidade das duas chamadas para o procedimento *append* (5+5). C_{a2}=45 ;

6. Finalmente, para conseguir-se a complexidade geral do segundo procedimento, soma-se a complexidade de suas cláusulas. ($C_{a1}+C_{a2}$=46).

```
'__c_hanoi/5'(46,2,[1,45],16,[ ]).
'__c_append/3'(5,2,[1,4],3,[ ]).
```

Figura 3.3: Exemplo do uso de resoluções como medida de complexidade

A complexidade por aridade, conforme mostra a figura 3.4, é calculada da seguinte forma:

1. A complexidade da primeira cláusula do segundo procedimento (C_{b1}) é o número de argumentos (aridade) da cabeça. C_{b1}=3;

2. A complexidade da segunda cláusula do segundo procedimento (C_{b2}) é o número de argumentos da cabeça mais a complexidade de sua meta. Esta cláusula possui apenas uma meta, que é uma chamada recursiva. Assim soma-se o valor da complexidade da primeira cláusula (C_{b1}=3), mais o valor da complexidade da cabeça (C_{b2cab}=3), mais a complexidade (aridade) da sua chamada recursiva (C_{b2meta}=3). Deste modo obtém-se a complexidade da chamada recursiva (C_{b2rec}=9). Para achar agora a complexidade da cláusula soma-se o valor da chamada recursiva (C_{b2rec}=9) ao valor da complexidade (aridade) da cabeça (C_{b2cab}=3). Assim obtém-se a complexidade C_{b2}=12;

3. Para achar a complexidade do segundo procedimento (C_b), soma-se a complexidade das suas cláusulas (C_{b1}+C_{b2}=15);

4. A complexidade da primeira cláusula do primeiro procedimento (C_{a1}) é o número de argumentos (aridade) da cabeça. C_{a1}=5 ;

5. A complexidade da segunda cláusula do primeiro procedimento (C_{a2}) é obtida, somando-se a complexidade da cabeça com a complexidade do seu corpo. Observe que no seu corpo existem duas chamadas recursivas (C_{a2rec1} e C_{a2rec2}). Para obter a complexidade das chamadas recursivas, soma-se a complexidade da primeira cláusula (C_{a1}=5), mais a complexidade da cabeça da segunda cláusula (C_{a2cab}=5), mais a complexidade das duas chamadas para o procedimento *append* (15+15) e mais a complexidade das duas chamadas recursivas (C_{a2rec1}=5 e C_{a2rec2}=5). Deste modo, somando-se acha-se o valor da chamada recursiva (C_{a2rec}=50). Com isto achou-se a complexidade da chamada recursiva, e para achar a complexidade geral da cláusula, soma-se a complexidade da cabeça da cláusula (C_{a2cab}=5), mais a complexidade das

suas chamadas recursivas (50+50), e por fim mais a complexidade das duas chamadas para o procedimento *append* (15+15). C_{a2}=135 ;

6. Finalmente, para conseguir-se a complexidade geral do segundo procedimento, soma-se a complexidade de suas cláusulas. (C_{a1}+C_{a2}=140).

```
'__c_hanoi/5'(140,2,[5,135],50,[ ]).
'__c_append/3'(15,2,[3,12],9,[ ]).
```

Figura 3.4: Exemplo do uso de aridade como medida de complexidade

Finalmente, a complexidade por unificação, conforme mostra a figura 3.5, é calculada da seguinte forma:

1. A complexidade da primeira cláusula do segundo procedimento é 4. Acha-se este valor somando-se o número de unificações necessárias para os argumentos que são (3), mais a unificação da cabeça (1). Portanto C_{b1}=4 ;

2. Para se obter o valor de complexidade da segunda cláusula do segundo procedimento (C_{b2}) soma-se o número de unificações contidas na cabeça mais a soma das unificações de suas metas. Nesta cláusula tem-se apenas uma chamada, que é uma chamada recursiva. Assim soma-se o valor da complexidade da primeira cláusula (C_{b1}=4), mais as unificações da cabeça (C_{b2cab}=6), e as unificações da chamada recursiva (C_{b2meta}=4). Deste modo obtém-se a complexidade da chamada recursiva (C_{b2rec}=14). Para achar agora a complexidade da cláusula soma-se o valor da chamada recursiva (C_{b2rec}=14) ao valor da complexidade da cabeça (C_{b2cab}=6), C_{b2}=20;

3. Para achar a complexidade do segundo procedimento (C_b), soma-se a complexidade das suas cláusulas (C_{b1}+C_{b2}=24);

4. A complexidade da primeira cláusula do primeiro procedimento é 8. Acha-se este valor somando-se o número de unificações necessárias para seus argumentos que são (7), mais a unificação da cabeça (1). Portanto C_{b1}=8;

5. A complexidade da segunda cláusula do primeiro procedimento (C_{a2}) é obtida somando-se a complexidade (unificações) da cabeça mais as unificações necessárias para a sua meta. Observe que no seu corpo existem duas chamadas

recursivas (C_{a2rec1} e C_{a2rec2}). Para obter a complexidade das chamadas recursivas, soma-se a complexidade da primeira cláusula (C_{a1}=8), mais a complexidade da cabeça da segunda cláusula (C_{a2cab}=6), mais a complexidade dos built-ins ($C_{a2built\text{-}ins}$=2), mais a complexidade das duas chamadas para o procedimento *append* (24+24) e mais a complexidade das duas chamadas recursivas (C_{a2rec1}=6 e C_{a2rec2}=6). Deste modo, somando-se acha-se o valor da chamada recursiva (C_{a2rec}=76). Com isto achou-se a complexidade da chamada recursiva. Para achar a complexidade geral da cláusula, soma-se a complexidade da cabeça da cláusula (C_{a2cab}=6), mais a complexidade dos built-ins (2), mais a complexidade das suas chamadas recursivas (76+76), e por fim a complexidade das duas chamadas para o procedimento *append* (24+24). C_{a2}=208;

6. Finalmente, para conseguir-se a complexidade geral do segundo procedimento, soma-se a complexidade de suas cláusulas. (C_{a1}+C_{a2}=216).

```
'__c_hanoi/6'(214,2,[8,206],76,[ ]).
'__c_append/4'(24,2,[4,20],14,[ ]).
```

Figura 3.5: Exemplo do uso de unificação como medida de complexidade

No nosso trabalho, adotamos a medidas de resolução e aridade que, atribuem pesos altos a cada cláusula (Figura 3.6). Utilizamos também uma medida alternativa, baseada no algoritmo de Tick [39], que dá peso 1 a cada chamada recursiva.

```
'__c_hanoi/5'(12,2,[1,11],1,[ ]).
'__c_append/3'(3,2,[1,2],1,[ ]).
```

Figura 3.6: Exemplo do uso de Tick como medida de complexidade

3.1 Resumo

Este capítulo apresentou o modelo ORCA, cuja função é gerar informações de granulosidade em tempo de compilação. O ORCA utiliza diferentes medidas para calcular o tamanho dos grãos: resolução, aridade e unificação. Utilizamos duas das medidas: resolução e aridade, e modificamos o ORCA para produzir uma medida alternativa baseada no algoritmo de Tick, que utiliza uma forma de cálculo mais relaxada, onde cada chamada recursiva recebe peso 1. No capítulo seguinte apresentamos o ambiente de simulação utilizado neste trabalho e nossas modificações para incorporar a informação de granulosidade.

Capítulo 4

Simulador de Execução de programas Prolog com Análise de Complexidade

Neste capítulo descrevemos o simulador implementado por Carrusca [9] e nossas modificações. Este simula a execução seqüencial e paralela de programas Prolog. Optamos por realizar este trabalho com base no simulador porque resultados podem ser comparados com a versão anterior e porque um ambiente de simulação nos traz maior flexibilidade e controle dos experimentos.

O simulador computa todas as soluções de um programa, porém não são tratadas as execuções de *cuts*, *commits* e nem de efeitos colaterais. Neste simulador, modificamos a execução paralela para dar suporte a informação de granulosidade.

O simulador de execução paralela conta com cinco estruturas de dados principais: um vetor de estados dos processadores, pilha de ambientes de execução, uma fila de tarefas a serem executadas, o relógio lógico e o atraso. Introduzimos a estrutura ORCA na nossa versão.

A primeira estrutura, estados dos processadores, serve para informar o estado dos mesmos, isto é, se ele está livre, ocupado ou esperando. Esse último estado é utilizado somente na simulação das arquiteturas de memória compartilhada e de memória distribuída, ou seja, simulação com custos arquiteturais.

A segunda estrutura, pilha de execução, também apresenta uma posição para cada processador, sendo, cada uma delas um registro. A descrição dos campos do registro é a seguinte:

- pt_varre_bd: aponta para o objetivo corrente, equivalente ao contador de programa da WAM.

- pt_aux_bd: aponta para a cláusula que unificou com o objetivo apontado por pt_varre_bd.
- pt_amb_ant: aponta para o ambiente pai do objetivo apontado por pt_varre_bd.
- pt_ini_pilha: Aponta para o início da pilha de execução. Cada registro desta pilha é referente a um ponto de escolha.
- pt_fim_pilha: aponta para o último registro da pilha de execução.
- lista_idade: aponta para uma lista que contém a idade das variáveis.

A terceira estrutura, fila de tarefas, é uma lista encadeada. O processador corrente toma para si a primeira alternativa, e insere as demais numa lista intermediária, de tarefas irmãs, para depois então serem encadeadas na fila de tarefas, de acordo com a política de escalonamento em questão. A fim de que sejam tomadas pelos processadores livres, até mesmo o processador que a originou pode escaloná-la. Elas são apenas marcadas como true (tomada) ou false (não tomada). Uma tarefa corresponde a cada alternativa encontrada para o objetivo em questão, sendo esta a fonte de paralelismo OU. A descrição dos campos do registro é a seguinte:

- pilha: ponto até onde deve ser feita a cópia na pilha
- código: pt_aux_bd, ou seja, próxima alternativa ainda não explorada.
- id: identificador único da tarefa, cada vez que uma tarefa é criada este id e incrementado.
- tomado: tipo booleano, indica se a tarefa já foi processada. Ao varrer a fila em busca de tarefas a serem executadas, aquelas que apresentarem campo tomado igual a True não são consideradas. Este campo serve a dois propósitos: (1) não precisa freqüentemente atualizar a fila de tarefas e (2) manter o pai de tarefas na fila para manutenção da ordenação da esquerda para a direita na política *leftmost*.
- origem: número do processador que gerou a tarefa. Este campo é necessário para a implementação da política *leftmost* e para manutenção da localidade de dados.

- destino: número do processador que tomou a tarefa.
- ini_pilha: aponta para o primeiro ponto de escolha na pilha de ambientes de execução.
- fim_pilha: aponta para o último ponto de escolha na pilha de ambientes de execução.
- id_visual: informação necessária para gerar o trace da execução paralela. Indica para a ferramenta de visualização de que nó os ramos devem partir.
- num_ramo_visual: também informação necessária para o trace. Indica para a ferramenta de visualização a que ramo pertence a tarefa.
- complexidade: valor da complexidade da tarefa. Campo acrescentado em nosso trabalho.
- prox: aponta para o próxima tarefa da fila.

A quarta estrutura, o relógio, é um vetor com cada posição referente a um processador, que é incrementado cada vez que se realiza uma redução, isto é, no momento em que o processador toma para si a primeira alternativa e depois no escalonamento, quando a tarefa assume o estado tomado igual a true. Assim o tempo total de execução paralela é o relógio do processador mais lento, ou seja, o último a terminar as tarefas.

A quinta estrutura, atraso, se aplica apenas à simulação de ambientes de memória compartilhada e de memória distribuída, isto é, com custo arquiteturais.

Cada vez que um processador vai tomar uma tarefa, pode sofrer um atraso. Este processador, então, inicializa um relógio de atraso e "aguarda" até conseguir tomar a tarefa.

Uma sexta estrutura, estrutura ORCA, incluída neste trabalho é formada por nove campos que serão descritos a seguir. Caso o programa Prolog utilizado como entrada do sistema contenha as informações de complexidade geradas pelo ORCA, as mesmas serão lidas primeiro, linha por linha, com o objetivo de gerar outras listas encadeadas, alocadas exatamente na ordem em que aparecem no programa Prolog. Cada elemento da lista é uma estrutura de dados contendo os campos mostrados na figura 4.1

- marca_orca: é uma marca da presença de análise de complexidade no programa Prolog, geradas pelo ORCA, aparece sempre com os caracteres $\_\_c\_$.
- nome_proced: nome do procedimento.
- aridade: aridade do procedimento, indica o número de argumentos do termo. No caso de variáveis e átomos este campo possuirá o valor 0.
- complex_proced: valor da complexidade total do procedimento.
- num_clausulas: número de cláusulas do procedimento.
- lista_complex: lista com a complexidade de cada cláusula do procedimento.
- ch_recursiva: complexidade da chamada recursiva, no caso de haver recursividade.
- lista_mutua: lista de procedimentos mutuamente recursivos ao procedimento.
- prox: aponta para o próximo elemento da lista.

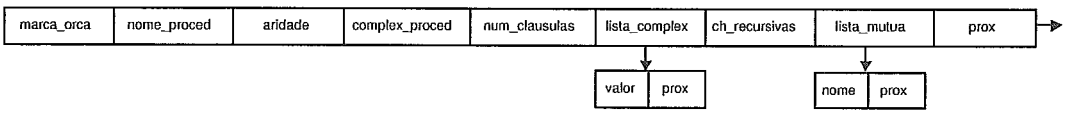


Figura 4.1: Estrutura ORCA

Nas seções seguintes, descreveremos o inclusão das informações de granulosidade geradas pela ferramenta ORCA nas etapas de análise sintática e escalonamento.

4.1 O analisador sintático

Esta seção explica o funcionamento do analisador sintático, responsável pelo armazenamento em memória do programa Prolog, na forma de lista encadeada. Tal programa deve estar armazenado na forma de um arquivo texto que será lido linha por linha, até atingir o final do arquivo(*fim). Cada caractere é guardado num vetor, que será utilizado como entrada para a fase posterior. O algoritmo básico é mostrado na figura 4.2.

O próximo passo a ser executado é a análise sintática do banco de dados(totalmente representado agora pelo vetor de caracteres), com o objetivo de

```
Abre o arquivo para leitura
j = 1
Inicializa vetor (vet_linha) com @
Lê registro
Se ativa_orca=0 //Sim - usar info. granulosidade
   então
      Se ver_orca = '__c_'
            então
               Lê informações de complexidade
            senão
               exibir mensagem 'Este arquivo não possui análise de complexidade'
      Fim se
   senão
      Enquanto registro=' ' ou (registro[2]='_' e registro[3]='_' e registro[4]='C'
               e registro[5]='_') faça
         Lê registro
      Fim enquanto
Fim se
Repita
   Lê registro
   Enquanto i <= comprimento do registro + 1 faça
      vet_linha[j] = registro[i]
      inc(i)
      inc(j)
   Fim enquanto
até o fim do arquivo
Fecha arquivo
```

Figura 4.2: Algoritmo: Leitura do arquivo de entrada e identificação das informações de complexidade

gerar uma lista encadeada de regras e fatos, dispostos exatamente na ordem em que aparecem no programa Prolog. Cada elemento da lista é uma estrutura de dados contendo os campos mostrados na figura 4.3

- tipo: indica se a cadeia formada possui um dos seguintes tipos: variável, átomo, inteiro, estrutura ou lista.
- nome: contém, por exemplo, o nome da variável em questão.
- aridade: indica o número de argumentos do termo. No caso de variáveis e átomos este campo possuirá o valor 0.
- built-in: pode possuir os valores true/false, indicando se o termo se trata de um dos predicados internos de Prolog, tais como var, nonvar.
- lista-arg: é um ponteiro para a lista de argumentos do predicado. Recebe o valor nil caso a aridade do termo seja igual a 0.

- lista-obj: tem por função inicial apontar para a lista de objetivos da cláusula. Assim, por exemplo, uma regra é um tipo átomo ou estrutura em que o campo lista-obj é diferente de nil e um fato se enquadra no mesmo caso, mas com o campo lista-obj igual a nil.

- prox: aponta para o próximo elemento da lista.

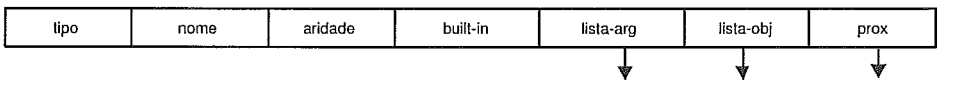


Figura 4.3: Representação interna dos elementos que constituem o banco de dados

A figura 4.4 apresenta as informações de complexidade extraídas de uma das aplicações e carregadas na estrutura ORCA, Figura 4.11.

```
'__c_main1/0'(134,2,[133,1],0,[]).
'__c_main2/0'(133,1,[133],0,[]).
'__c_main3/0'(133,1,[133],0,[]).
'__c_main4/0'(133,1,[133],0,[]).
'__c_mu_top1/0'(132,1,[132],0,[]).
'__c_mu_top2/0'(132,1,[132],0,[]).
'__c_mu_top3/0'(132,1,[132],0,[]).
'__c_rules/2'(61,4,[26,16,12,7],0,[]).
'__c_rule1/2'(11,1,[11],0,[]).
'__c_rule2/2'(6,1,[6],0,[]).
'__c_rule3/2'(25,2,[11,14],13,[]).
'__c_rule4/2'(15,2,[6,9],8,[]).
'__c_theorem/2'(131,2,[1,130],66,[]).
'__c_app/3'(5,2,[1,4],3,[]).
```

Figura 4.4: Exemplo de Informações de granulosidade

A etapa de análise sintática do banco de dados constitui a primeira fase na implementação. O armazenamento do programa em estruturas eficazes tem importância fundamental para as etapas seguintes.

4.2 Informação de granulosidade em Top-Most

A figura 4.5 apresenta o algoritmo de como as tarefas irmãs são inseridas na fila de tarefas (*task queue*). Para isso, torna-se necessário verificar o nível. Como a política adotada é *top-most*, a ordenação é feita de modo crescente. Assim, é feita uma comparação entre os níveis das tarefas que já existem na fila de tarefas (*task queue*) e aquelas que serão inseridas. Quando a opção análise de granulosidade for

igual a sim, além da comparação feita pelo campo nível (primeiro passo), é feita uma comparação com o grau de granulosidade das tarefas (desempate). Desta forma dá-se prioridade às tarefas de maior grau de granulosidade. As tarefas são escalonadas, marcadas como *true*, na ordem que se encontram na fila de tarefas.

```
pt_aux_tarefa = pt_ini_tarefa
pt_ant_aux_tarefa = nil
Se pt_aux_tarefa <> nil então
   Se (ordem = 'crescente') então
      Se ativa_orca=0 então //Sim - usar info. granulosidade
         Enquanto(pt_aux_tarefa <> nil)and(pt_ini_lista_alt.fim_pilha.nivel >
                  pt_aux_tarefa.fim_pilha.nivel) faça
            pt_ant_aux_tarefa = pt_aux_tarefa
            pt_aux_tarefa = pt_aux_tarefa.prox
         Fim enquanto
         Enquanto(pt_aux_tarefa <> nil) e
                  (pt_ini_lista_alt.fim_pilha.nivel = pt_aux_tarefa.fim_pilha.nivel) e
                  (pt_ini_lista_alt.complexidade <= pt_aux_tarefa.complexidade) faça
            pt_ant_aux_tarefa = pt_aux_tarefa
            pt_aux_tarefa = pt_aux_tarefa.prox
         Fim enquanto
         senão
         Enquanto(pt_aux_tarefa <> nil) e (pt_ini_lista_alt.fim_pilha.nivel >=
                  pt_aux_tarefa.fim_pilha.nivel) faça
            pt_ant_aux_tarefa = pt_aux_tarefa
            pt_aux_tarefa = pt_aux_tarefa.prox
         Fim enquanto
      Fim se
   Fim se
Fim se
```

Figura 4.5: Algoritmo: informações de granulosidade em Top-Most

4.3 Informação de granulosidade em Bottom-Most

A figura 4.6 apresenta o algoritmo de como as tarefas irmãs são inseridas na fila de tarefas (*task queue*). O procedimento é análogo a política *top-most*, só que neste caso a ordenação é feita de modo decrescente. Aqui também as tarefas são escalonadas, e marcadas como *true*, na ordem que se encontram na fila de tarefas.

```
Se (ordem = 'decrescente') então
   Se ativa_orca=0 então //Sim - usar info. granulosidade
      Enquanto(pt_aux_tarefa <> nil) e (pt_ini_lista_alt.fim_pilha.nivel <
               pt_aux_tarefa.fim_pilha.nivel) faça  {
         pt_ant_aux_tarefa = pt_aux_tarefa
         pt_aux_tarefa = pt_aux_tarefa.prox }
      Enquanto(pt_aux_tarefa <> nil) e (pt_ini_lista_alt.fim_pilha.nivel = pt_aux_tarefa.fim_pilha.nivel)
               e (pt_ini_lista_alt.complexidade <= pt_aux_tarefa.complexidade) faça {
         pt_ant_aux_tarefa = pt_aux_tarefa
         pt_aux_tarefa = pt_aux_tarefa.prox }
      senão
      Enquanto(pt_aux_tarefa<>nil) e (pt_ini_lista_alt.fim_pilha.nivel <= pt_aux_tarefa.fim_pilha.nivel) faça {
         pt_ant_aux_tarefa = pt_aux_tarefa
         pt_aux_tarefa = pt_aux_tarefa.prox }
   Fim se
Fim se
```

Figura 4.6: Algoritmo: informações de granulosidade em Bottom-Most

4.4 Informação de granulosidade em Menor-Custo

Na política menor-custo as tarefas irmãs são inseridas a medida que vão sendo criadas, isto é, não é necessária a busca do local exato para fazer a inserção.

Diferente de *top-most* e *bottom-most*, na política menor-custo a fila de tarefas não é previamente ordenada. A figura 4.7 descreve o algoritmo de como as tarefas são escalonadas, ou seja, alocação de tarefas aos processadores livres, para que sejam executadas. Neste caso, no momento do escalonamento, são analisadas todas as tarefas com o propósito de selecionar aquela que tem menor custo. Se a opção análise de granulosidade for igual a sim, além de selecionar a tarefa de menor custo, verifica-se sua granulosidade. Desta forma dentro do grupo de menor custo, marca-se a tarefa com maior grau de granulosidade. Uma tarefa de menor custo é aquela em que o processador precisa atravessar menos pontos de escolha.

Como discutido em Carrusca [9], da forma como é implementada no simulador há um *overhead* adicional de se buscar a tarefa de menor custo, porém numa implementação real, este *overhead* pode ser minimizado se utilizarmos a organização dos ramos da árvore de execução como proposta por Sindaha [38].

```
Se opção análise de granulosidade = não
   então
         Se valor_custo < menor_custo então
            menor_custo := valor_custo;
            pt_marca := pt_tarefa;
          Fim se
   senão
         Se valor_custo < menor_custo
            então
                  menor_custo := valor_custo;
                  pt_marca := pt_tarefa;
            senão
                  Se valor_custo=menor_custo então
                     Se pt_marca.complexidade < pt_tarefa.complexidade então
                        menor_custo := valor_custo;
                        pt_marca := pt_tarefa;
                  Fim se
         Fim se
Fim se
```

Figura 4.7: Algoritmo: informações de granulosidade em Menor-custo

4.5 ORCA-Top

A figura 4.8 apresenta o algoritmo de como as tarefas irmãs são inseridas na fila de tarefas. De modo que é feita uma comparação ente os graus de complexidade das tarefas que já existem na fila de tarefas e aquelas que serão inseridas. Caso

tenham o mesmo grau de complexidade, é feita uma comparação pelo campo nível, ordem crescente. Uma vez que a fila de tarefas é previamente ordenada, as tarefas são escalonadas na ordem em que se encontram na fila de tarefas.

A lista intermediária, de tarefas irmãs, utilizada nas políticas *top-most* com informação de granulosidade, *bottom-most* com informação de granulosidade e orca-Top são ordenadas pelo grau de complexidade, antes de sua inserção na lista principal. *Leftmost* não foi implementada com informações de granulosidade por razões óbvias.

```
Se (ordem = 'orca_decrescente') então
   Enquanto (pt_aux_tarefa <> nil)and(pt_ini_lista_alt.complexidade < pt_aux_tarefa.complexidade faça
      pt_ant_aux_tarefa = pt_aux_tarefa
      pt_aux_tarefa = pt_aux_tarefa.prox
   Fim enquanto

   Enquanto(pt_aux_tarefa <> nil)and (pt_ini_lista_alt.complexidade = pt_aux_tarefa.complexidade) e
      (pt_ini_lista_alt.fim_pilha.nivel >= pt_aux_tarefa.fim_pilha.nivel) faça
      pt_ant_aux_tarefa = pt_aux_tarefa
      pt_aux_tarefa = pt_aux_tarefa.prox
   Fim enquanto
Fim Se
```

Figura 4.8: Algoritmo: ORCA-Top

4.6 Ambiente de simulação

Com o objetivo de avaliar o impacto das informações de granulosidade e as políticas de escalonamento utilizamos um ambiente simulado onde todos os experimentos são controlados. Além do controle, um ambiente de simulação também permite maior flexibilidade para realização dos experimentos, tais como, implementação de diferente estratégias de escalonamento. O programa simula o funcionamento da máquina Prolog em uma máquina monoprocessada, porém simula a existência de mais de um processador.

O simulador utiliza os seguintes parâmetros de entrada: o programa Prolog a ser rodado, a consulta, o número de processadores, a política de escalonamento, utilização ou não das informações de complexidade geradas pelo ORCA, e tipo de modelagem de custo. Ao final da execução o usuário poderá ver os resultados através de gráficos e trace de execução. Ele simula cinco diferentes estratégias: *top-most*, *bottom-most*, *left-most*, menor-custo e uma política denominada ORCA-Top que foi adicionada. A Figura 4.9 mostra a tela de entrada do simulador paralelo com os parâmetros de entrada.

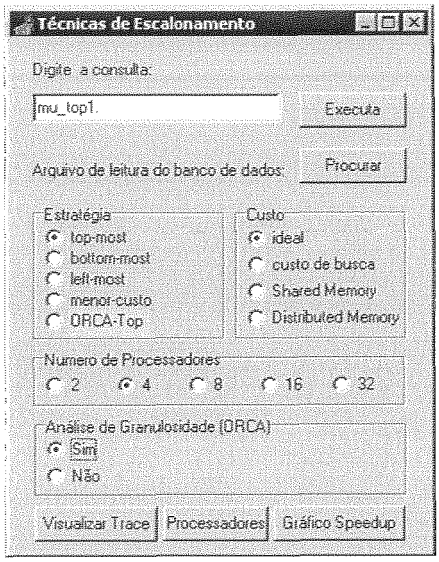


Figura 4.9: Tela de entrada do simulador paralelo

A figura 4.10, mostra a organização básica do ambiente de simulação com nossas modificações já incluídas.

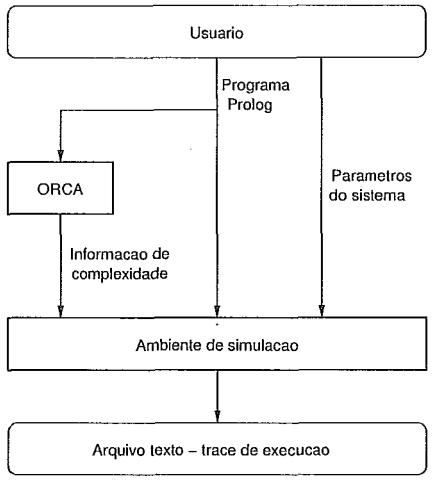


Figura 4.10: Arquitetura do simulador

4.6.1 Descrição do funcionamento

- **Armazenamento em memória do programa Prolog,** na forma de lista encadeada. Como descrito anteriormente, tal programa deve estar armazenado na forma de um arquivo texto que será lido linha por linha, até atingir o final do arquivo *(*fim)*. Cada caracter é guardado num vetor. Se o arquivo texto contiver as informações de complexidade geradas pelo ORCA,

essas informações serão armazenadas em outra estrutura.

- **Análise sintática do vetor de caracteres**, formando uma lista encadeada de regras e fatos, dispostos exatamente na ordem que aparecem no programa Prolog. Esse processo é feito pelo analisador sintático.

- **Uma consulta é feita ao programa Prolog**. O algoritmo varre esta base de dados à procura de uma cláusula que unifique com a mesma, isto é, que apresente o mesmo nome e aridade. Quando esta é encontrada, cria-se um ponto de escolha. Caso a unificação seja bem sucedida, inicia-se a execução dos objetivos do corpo. Se falhar, o programa Prolog continua a ser percorrido a partir da cláusula seguinte à que falhou.

- **Caso haja mais de uma cláusula**, o processador toma para si a primeira alternativa e armazena as demais na fila de tarefas, de tal forma que cada uma delas representa uma nova tarefa. Uma tarefa corresponde a cada alternativa encontrada para o objetivo em execução.

- **Em seguida, o simulador faz o escalonamento**, ou seja, a atribuição de tarefas aos processadores livres (que pode ser o mesmo que criou a tarefa), de acordo com a política de escalonamento em questão. Desta forma, até que se esgote o número de processadores, a fila de tarefas é percorrida em busca de tarefas que ainda não foram tomadas. Ao encontrar uma tarefa nestas condições, verifica-se se o processador em questão está disponível. Caso o processador esteja disponível, a tarefa é a ele atribuída.

- **Gerar gráficos e arquivo de trace**, que posteriormente serão utilizados para depuração e pela ferramenta de visualização VisAll [19].

4.6.2 Modelagem dos custos

Custos podem ser aplicados a qualquer uma das cinco políticas de escalonamento utilizando ou não análise de complexidade gerada pelo ORCA. Neste caso, o relógio

do processador sofrerá um incremento no momento do escalonamento igual ao custo de encontrar a tarefa, isto é, a penalidade sofrida pelo processador para caminhar na árvore de execução até chegar à nova tarefa que lhe foi atribuída. Este custo é proporcional ao número de pontos de escolha que devem ser percorridos até que o processador se localize na posição correta na árvore. Portanto, para um processador que está executando trabalho pela primeira vez, este valor será igual ao número de pontos de escolha da tarefa sendo atribuída, visto que é necessário percorrer todos os nós até atingir a nova tarefa. Se o processador já executou algum trabalho, então o custo será igual ao somatório do número de pontos de escolha que devem ser percorridos na tarefa antiga e na nova até chegar a um nó comum. Adicionado ao custo para se posicionar na árvore de execução, consideramos custos referentes à simulação de duas diferentes arquiteturas: **memória compartilhada** (SM) e **memória distribuída** (DM). Portanto, em nossa simulação é possível optar por uma das seguintes possibilidades:

- **ideal.** Não são computados quaisquer custos. Neste caso, obtém-se o máximo de paralelismo que uma aplicação poderia conseguir para um determinado número de processadores.

- **custo de busca.** Nesse caso, é computada a distância que cada processador precisa percorrer para atingir a tarefa selecionada da fila de tarefas, ou seja, o custo é igual ao número de pontos de escolha que devem ser percorridos até que o processador se localize na posição correta na árvore. Adicionado a este custo, consideramos ambiente de memória compartilhada(SM) e de memória distribuída(DM).

- **SM.** a simulação de um ambiente de memória compartilhada é feita criando-se um *timer* que é inicializado com o valor do custo de procurar tarefa na árvore. Desta forma no momento do escalonamento, o processador é colocado em um estado intermediário: **esperando**. O processador assume o estado de **ocupado** quando o *timer* fica igual a zero, dando prosseguimento à sua execução. Este *timer* é decrementado a cada resolução, isto é, a cada decremento do relógio global.

- **DM.** se a tarefa selecionada tiver sido gerada pelo próprio processador, o custo é semelhante o anterior. Caso o processador venha a receber uma tarefa que

foi criada por outro, então o *timer* é inicializado não apenas com o custo de procurar a tarefa na árvore, mas também com o **custo de comunicação via rede**, inerente a sistemas de memória distribuída. O *timer* é inicializado com o valor da expressão abaixo:

$$num\_pt\_escolha2 + \frac{(num\_pt\_escolha1 \times tam\_pt\_escolha)}{tam\_msg} \times latencia$$

Nesta fórmula, *num_pt_escolha2* corresponde aos nós locais que o processador que está pegando a tarefa precisa atravessar, *num_pt_escolha1* corresponde ao número de nós que esse mesmo processador precisa atravessar para apanhar a tarefa do vizinho, *tam_pt_escolha* (igual a 120 bytes, tamanho médio de ponto de escolha nas implementações mais conhecidas) representa o número de bytes de cada ponto de escolha, *tam_msg* (igual a 4 bytes, tamanho típico de uma mensagem que transita numa rede de comunicação) representa o tamanho da mensagem na rede e *latencia* (igual a 0.5 redução) representa o atraso para enviar uma mensagem através da rede. O número 0.5 reduções foi escolhido da seguinte forma: mediu-se o tempo de execução de um sistema paralelo real, para um processador. Contou-se o número de reduções efetuado durante o tempo de execução. Dividiu-se o tempo total de execução pelo número de reduções. Desta forma, obtivemos o tempo gasto para cada redução. Em seguida estimamos o tempo médio para uma mensagem atravessar numa rede de comunicação atual. Comparamos e obtivemos que a latência é equivalente a 0.5 redução.

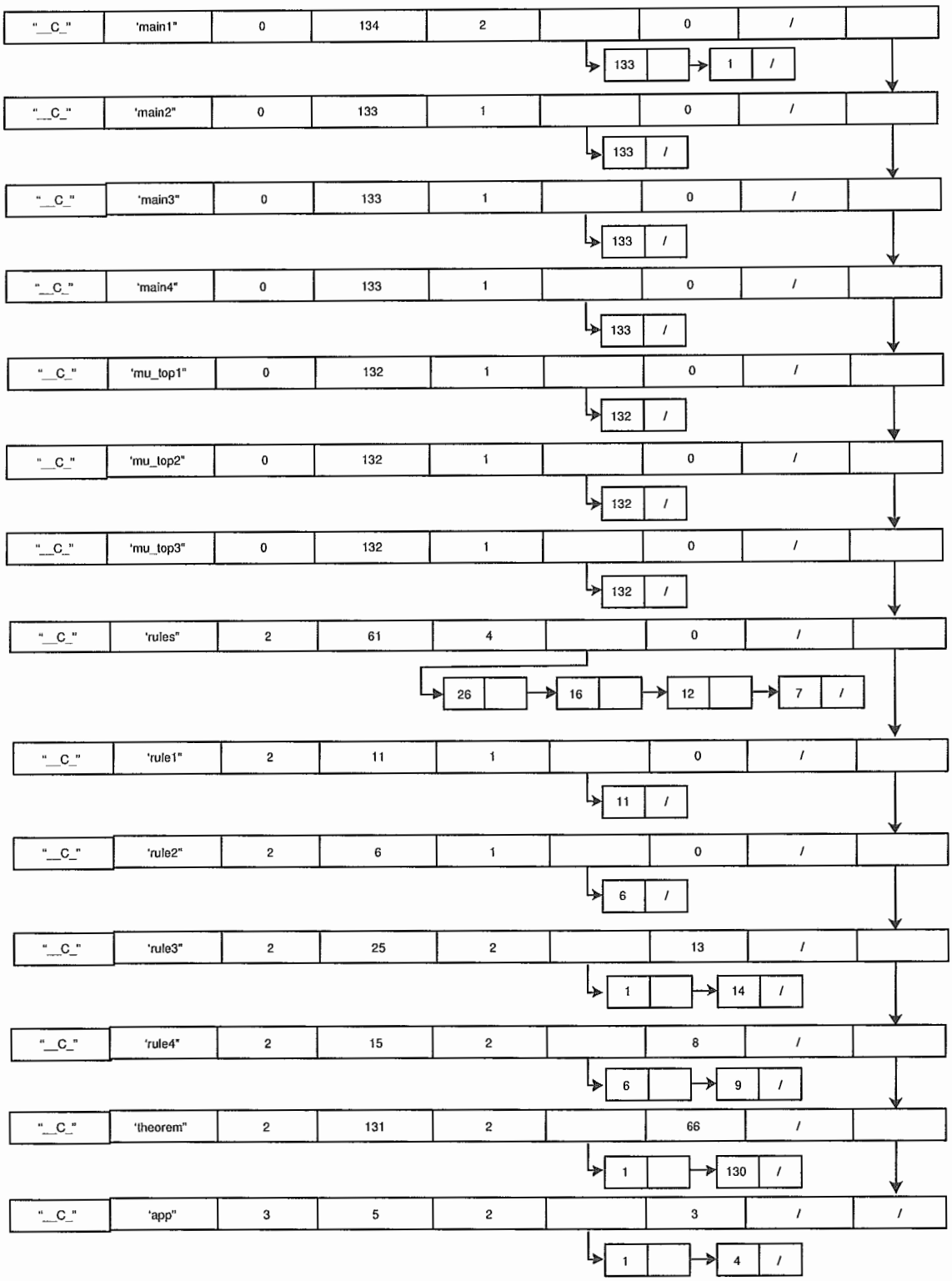


Figura 4.11: Exemplo de representação da estrutura ORCA

4.7 Resumo

Neste capítulo descrevemos as cinco estruturas de dados principais do simulador: o vetor de estados dos processadores, o ambiente de execução, a fila de tarefas, o relógio e o atraso, e nossas modificações: a inclusão de um campo complexidade na estrutura de dados fila de tarefas e a criação de uma nova estrutura para gerência da informação de granulosidade. No código relacionado ao escalonamento, foram apresentados os algoritmos com a inclusão de informação de granulosidade nas políticas *top-most*, *bottom-most* e menor-custo e a criação de uma nova política chamada ORCA-Top. Apresentamos também o ambiente de simulação com os parâmetros de entrada utilizados pelo simulador e explicamos a forma como os custos podem ser aplicados a qualquer uma das cinco políticas de escalonamento utilizando ou não análise de complexidade geradas pelo ORCA.

No capítulo seguinte explicaremos os experimentos que foram realizados e faremos uma análise detalhada dos resultados obtidos em nossas simulações.

Capítulo 5

Resultados

5.1 O conjunto de benchmark

Nosso benchmark é composto por 4 programas que se beneficiam do paralelismo OU, ou seja, predominantemente exploram o paralelismo OU. Estes programas são os mesmos utilizados por Carrusca [9] em seus experimentos.

O primeiro programa, `mutest`, consiste de uma prova de teorema, que resolve o problema descrito em [34]. O problema é produzir a string "mi", a partir de uma dada string, através de regras que podem ser aplicadas em qualquer ordem. O teorema a ser aprovado é que `muiiu` produz `mi`. A consulta para esse programa é `mu_top1`. Este programa contém somente paralelismo OU.

O segundo programa, `family`, consiste de um banco de dados com relações de família e ocupação profissional. A consulta utilizada para este programa expande a árvore de execução em 180 ramos que percorrem o banco de dados procurando todas as informações de família armazenadas. Os ramos têm profundidade baixa.

O terceiro programa `queens` [33] consiste em colocar N rainhas em um tabuleiro de xadrez NxN tal que as rainhas não se ataquem na mesma linha, coluna ou diagonal. A consulta utilizada para este programa é `queens(6,M)`, que procura a alocação de rainhas em um tabuleiro 6 x 6. Devido a limitação de memória do software que utilizamos para rodar o simulador não pudemos rodar consultas para tabuleiros maiores. Porém, o tabuleiro 6 x 6 gera uma uma quantidade razoável de paralelismo.

A quarta aplicação é um exemplo de sistema de linguagem natural e banco de dados escrito na Universidade de Edimburgo por Pereira e Warren [15], `chat`. A versão de *chat* usada no nosso sistema opera sobre o domínio de geografia. O problema utilizado faz uma consulta ao banco de dados geográfico. O `chat`

contém somente paralelismo OU. A consulta utilizada para este programa gera uma quantidade razoável de paralelismo.

5.2 Metodologia

Utilizamos uma máquina PC, Pentium III, de 450Mhz com memória RAM de 128MBytes e Sistema Operacional Windows 2000. Nossos dois simuladores foram implementados utilizando o Delphi 6. O modelo de execução é determinístico, ou seja, durante a execução é utilizado um relógio lógico. O tempo total de execução paralela é contabilizado através do relógio do processador mais lento, ou seja, o último a terminar tarefas. Os experimentos foram rodados simulando máquinas com 1, 2, 4, 8, 16 e 32 processadores.

Os tempos de execução conseguidos para um processador não consideram nenhum tipo de custo paralelo ou da análise de complexidade gerada pelo ORCA. Todos os *speedups* apresentados nas tabelas são relativos a estes tempos seqüenciais.

As tabelas mostradas neste capítulo apresentam os speedups das aplicações para todas as políticas de escalonamento simuladas com e sem informações de granulosidade. A coluna 1 mostra o número de processadores, a coluna 2 indica se a simulação utiliza informação de complexidade para as políticas *Top-Most* (TM), *Bottom-Most* (BM), *Left-Most*(LM) e Menor Custo(MC). A última coluna, ORCA-Top, apresenta speedups para a simulação cujo escalonamento se baseia apenas na informação de granulosidade. A tabela 5.1 apresenta as características das quatro aplicações que serão utilizadas em nossos experimentos.

Aplicação	**Características**
`mutest`	árvore binária e profunda
`family`	espaço de busca largo e raso
`queens`	árvore irregular com ramos de profundidade alta
`chat`	espaço de busca relativamente estreito e de média profundidade

Tabela 5.1: Características das aplicações

5.3 Análise dos resultados

Nossos resultados estão organizados da seguinte forma. Na seção 5.3.1 apresentamos e discutimos os resultados para todos os programas, com e sem informação de granulosidade, para 1, 2, 4, 8, 16 e 32 processadores, numa

situação ideal. Nesta situação não consideramos nenhum custo para exploração do paralelismo. Os resultados desta seção indicam a quantidade potencial de paralelismo existente em cada aplicação de acordo com as políticas de escalonamento. Nas próximas seções apresentamos e discutimos os resultados quando levamos em consideração custos associados a *overheads* arquiteturais e quando utilizamos outras medidas alternativas de granulosidade. Na seção 5.3.2 apresentamos resultados com a simulação do atraso natural existente em arquiteturas baseadas em memória centralizada. Na seção 5.3.3 apresentamos resultados com o atraso existente em arquiteturas baseadas em memória distribuída. Todos os resultados destas seções são produzidos com a versão de ORCA que utiliza medida de complexidade baseada em resolução. Na última seção deste capítulo apresentamos e discutimos resultados obtidos utilizando uma outra medida alternativa de granulosidade produzida pelo ORCA. A tabela 5.2 mostra os tempos de execução seqüencial para as aplicações `mutest`, `family`, `queens` e `chat`. Estes tempos correspondem à execução da simulação paralela usando um processador.

Aplicação	`mutest`	`family`	`queens`	`chat`
Tempo seq.	6508	284	4019	7177

Tabela 5.2: Tempo sequencial das aplicações

5.3.1 Simulação ideal

As tabelas 5.3, 5.4, 5.5 e 5.6 mostram, respectivamente, os desempenhos das aplicações `mutest`, `family`, `queens` e `chat`. Tempos são medidos através do relógio global do processador mais lento. Os *speedups* são mostrados entre parênteses.

Observando apenas os resultados sem informação de granulosidade, a melhor política varia de aplicação para aplicação e com o número de processadores. Por exemplo, `mutest` (Tabela 5.3), para número pequeno de processadores (até 8), tem melhor desempenho com a estratégia *top-most*, enquanto para 16 e 32 processadores, a melhor política é a de menor-custo. Esta variação é interessante levando-se em consideração que as aplicações foram executadas pelo mesmo simulador sem nenhum custo adicional embutido no escalonador. Ressalta que estratégias diferentes podem causar um impacto positivo ou negativo no desempenho, mesmo em condições ideais de execução. As diferenças entre o menor e o maior speedups aumentam linearmente a medida que aumentamos o número de processadores para quase todas as aplicações,

ou seja, quanto mais processadores adicionamos, maior é o impacto de uma estratégia sobre o escalonamento da aplicação. Isto já havia sido constatado no trabalho de Carrusca [9].

A aplicação `queens` (Tabela 5.5) se beneficia da estratégia *left-most* com 2 processadores. A partir de 4 processadores, a melhor estratégia passa a ser *top-most*. Como esta aplicação possui uma árvore irregular com ramos de profundidade alta, *top-most* passa a ser uma política mais adequada, pois favorece a exploração dos ramos mais altos primeiro, ganhando assim em tempo de execução por processador. Esta situação é melhor ilustrada quando comparamos o balanceamento de carga, por exemplo, para 32 processadores, de *top-most* (Figura 5.1(a)) com a pior política, *left-most* (Figura 5.1(b)). Neste caso, o processador *p2*, na estratégia *left-most*, ficou com a maior parte das tarefas (total de 45 tarefas) em contraste com o processador mais lento de *top-most* que executou um total de 32 tarefas. Como a política *left-most* dá preferência a trabalho mais à esquerda na árvore de execução, os processadores não tiveram a chance de começar mais cedo a trabalhar em um dos ramos críticos (mais profundos) do espaço de busca de `queens`.

As figuras 5.2(a), 5.2(b), 5.2(c) e 5.2(d) mostram os speedups para nossos benchmarks até 32 processadores com análise de granulosidade, para as cinco políticas estudadas.

A aplicação `chat` (Tabela 5.6) é a que sofre menor variação geral em tempo de execução com uma diferença percentual máxima de 4,5%, para 32 processadores, entre a pior política, menor custo, e a melhor política, *bottom-most* ou *left-most*. `chat` começa se beneficiando da estratégia *top-most* até 8 processadores. Para 16 e 32 processadores, a melhor política passa a ser *bottom-most*. *Top-most* não traz vantagens para números maiores de processadores porque o espaço de busca desta aplicação não é largo o suficiente para manter muitos pontos de escolha (para satisfazer o número de processadores) na parte mais alta da árvore.

Das aplicações estudadas, `queens`, que possui uma árvore de execução irregular, é a mais afetada pelas estratégias empregadas nas decisões de escalonamento. A diferença chega a ser de 19,8%, para 32 processadores, entre a pior política, *left-most*, e a melhor política, *top-most*.

mutest						
	Complex.	TM	BM	LM	MC	ORCA-Top
2	sem	3256(2.00)	3255(2.00)	3273(1.99)	3291(1.98)	-
	com	3256(2.00)	3257(2.00)	-	3287(1.98)	3257(2.00)
4	sem	1633(3.99)	1634(3.98)	1634(3.98)	1652(3.94)	-
	com	1633(3.99)	1637(3.98)	-	1651(3.94)	1632(3.99)
8	sem	821(7.93)	847(7.68)	839(7.76)	824(7.90)	-
	com	821(7.93)	820(7.94)	-	821(7.93)	820(7.94)
16	sem	417(15.61)	424(15.35)	442(14.72)	416(15.64)	-
	com	417(15.61)	422(15.42)	-	417(15.61)	444(14.66)
32	sem	221(29.45)	228(28.54)	224(29.05)	218(29.85)	-
	com	220(29.58)	230(28.30)	-	221(29.45)	234(27.81)

Tabela 5.3: Speedups de mutest, ideal

family						
	Complex.	TM	BM	LM	MC	ORCA-Top
2	sem	147(1.93)	145(1.96)	145(1.96)	149(1.91)	-
	com	145(1.96)	145(1.96)	-	148(1.92)	145(1.96)
4	sem	74(3.84)	73(3.89)	73(3.89)	74(3.84)	-
	com	75(3.79)	76(3.74)	-	75(3.79)	75(3.79)
8	sem	38(7.47)	39(7.28)	39(7.28)	40(7.10)	-
	com	39(7.28)	37(7.68)	-	38(7.47)	39(7.28)
16	sem	20(14.20)	21(13.52)	21(13.52)	21(13.52)	-
	com	21(13.52)	24(11.83)	-	21(13.52)	21(13.52)
32	sem	12(23.67)	12(23.67)	12(23.67)	12(23.67)	-
	com	12(23.67)	12(23.67)	-	12(23.67)	12(23.67)

Tabela 5.4: Speedups de family, ideal

queens						
	Complex.	TM	BM	LM	MC	ORCA-Top
2	sem	2020(1.99)	2016(1.99)	2012(2.00)	2016(1.99)	-
	com	2020(1.99)	2016(1.99)	-	2016(1.99)	2020(1.99)
4	sem	1021(3.94)	1033(3.89)	1045(3.85)	1020(3.94)	-
	com	1021(3.94)	1033(3.89)	-	1020(3.94)	1021(3.94)
8	sem	517(7.77)	532(7.55)	553(7.27)	529(7.60)	-
	com	517(7.77)	532(7.55)	-	535(7.51)	517(7.77)
16	sem	277(14.51)	293(13.72)	311(12.92)	282(14.25)	-
	com	277(14.51)	293(13.72)	-	282(14.25)	275(14.61)
32	sem	157(25.60)	169(23.78)	196(20.51)	181(22.20)	-
	com	157(25.60)	169(23.78)	-	154(26.10)	156(25.76)

Tabela 5.5: Speedups de queens, ideal

chat						
	Complex.	TM	BM	LM	MC	ORCA-Top
2	sem	3590(2.00)	3590(2.00)	3590(2.00)	3631(1.98)	-
	com	3590(2.00)	3590(2.00)	-	3628(1.98)	3590(2.00)
4	sem	1798(3.99)	1798(3.99)	1798(3.99)	1813(3.96)	-
	com	1798(3.99)	1798(3.99)	-	1824(3.93)	1799(3.99)
8	sem	902(7.96)	903(7.95)	903(7.95)	907(7.91)	-
	com	902(7.96)	903(7.95)	-	909(7.90)	909(7.90)
16	sem	458(15.67)	455(15.77)	457(15.70)	464(15.47)	-
	com	464(15.47)	455(15.77)	-	465(15.43)	471(15.24)
32	sem	236(30.41)	234(30.67)	234(30.67)	245(29.29)	-
	com	240(29.90)	233(30.80)	-	238(30.16)	241(29.78)

Tabela 5.6: Speedups de chat, ideal

A aplicação `family` (Tabela 5.4) apresenta uma pequena variação de desempenho com a mudança de estratégia para números pequenos de processadores. Para 16 e 32 processadores, nenhuma política afeta o desempenho desta aplicação de forma

significativa. Isto se deve ao fato do espaço de busca desta aplicação ser largo e raso.

Com a introdução da informação de granulosidade em cada política, o desempenho continua variando dependendo da aplicação e do número de processadores. Porém os resultados são mais estáveis com uma variação percentual máxima de 12,5%, para a aplicação `family` (Tabela 5.4), com 16 processadores, onde a melhor política é *top-most*, menor custo ou ORCA-Top, e a pior é *bottom-most*.

Entretanto, o fato mais relevante e importante resultante da inclusão de informação de granulosidade nas decisões de escalonamento é que os tempos de execução mínimos e máximos, isto é, produzidos pela estratégia que beneficia menos a aplicação e pela estratégia que beneficia mais a aplicação, respectivamente, melhoraram para algumas aplicações, com o aumento do número de processadores. `Queens` (Tabela 5.5) que, para 32 processadores, tinha speedup mínimo de 20,51, com a política *left-most*, passou a ter speedup mínimo de 23,78 com a política *bottom-most* (uma variação significativa de 13,7%). O mesmo ocorreu para a aplicação `chat` (Tabela 5.6), que, para 32 processadores, tinha speedup mínimo de 29,29, com a política de menor custo, e passou a ter speedup mínimo de 29,78, com a política ORCA-Top (uma variação discreta de 1,6%). Ainda na aplicação `queens`, o speedup máximo, para 32 processadores, subiu de 25,60, com a política *top-most*, para 26,10, com a política de menor-custo, uma diferença de aproximadamente 2%. `Chat` teve speedup máximo, para 32 processadores, aumentado de 30,67, com as políticas *left-most* ou *bottom-most*, para 30,80, com a política *bottom-most*.

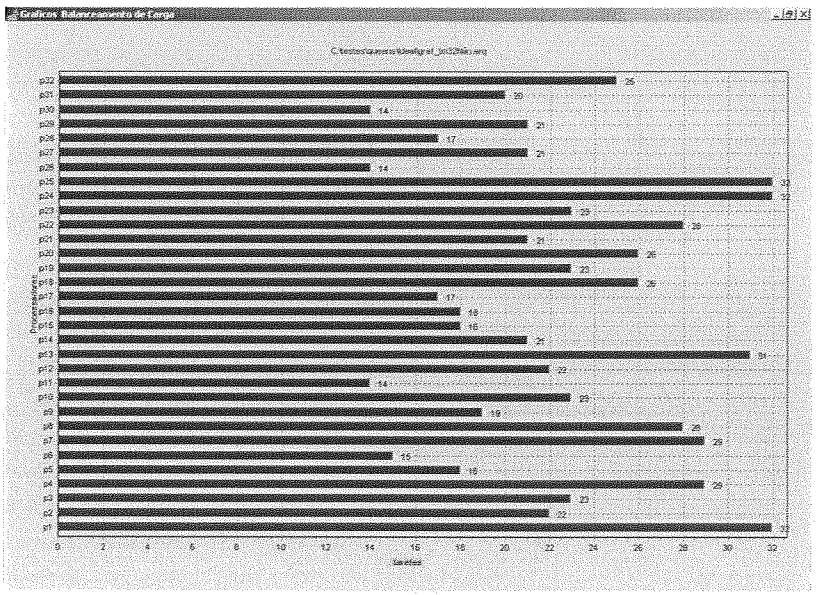
No geral, a introdução de informação de granulosidade aumentou os speedups originais das estratégias para as aplicações `queens` e `chat`. `Family` (Tabela 5.4), para 8 processadores, teve melhora mínima de 2,5% e máxima de 2,7%. `mutest` também teve melhora para 8 processadores, com um aumento de 3,2% no speedup mínimo.

A maior contribuição da informação de granulosidade foi uma melhora de quase 15% no desempenho da aplicação `queens` (Tabela 5.5), para 32 processadores, na política de menor custo, cujo speedup subiu de 22,20 para 26,10. Em geral, a informação de granulosidade permitiu a melhora de desempenho quando combinada com alguma estratégia. Este é um resultado interessante que indica que não devemos apenas concentrar as decisões de escalonamento na ordenação pura de granulosidade das tarefas, mas também levar em consideração o custo de movimentação na árvore de execução. De fato, se observarmos as políticas que se beneficiaram mais da

informação de granulosidade, podemos concluir que a política de menor custo, que era, em geral, a melhor estratégia, é a que mais se beneficia, em geral, das informações de granulosidade. `Queens` se beneficia da política menor-custo com informação de granulosidade para 2, 4 e 32 processadores, enquanto, a versão sem granulosidade se beneficiava da política de menor custo apenas com 4 processadores. `Family` se beneficiava da política de menor custo apenas para 32 processadores. Com a informação de granulosidade, a política de menor custo passa a ser a melhor para 4, 16 e 32 processadores. `Chat` não se beneficiava da política de menor custo, e continua sem se beneficiar desta mesma política com a introdução de granulosidade. `Mutest` é a única aplicação que se beneficiava da política de menor-custo para 16 e 32 processadores e com a informação de granulosidade passa a se beneficiar apenas com 16 processadores.

A aplicação `mutest` foi a única que teve uma queda de desempenho para 32 processadores com a inclusão de informação de granulosidade. Isto se deve à forma como a informação de granulosidade é calculada para chamadas recursivas. Como `mutest` tem uma chamada recursiva e o procedimento que contém esta chamada tem vários outros objetivos a serem invocados, a análise de complexidade atribui um peso demasiado alto a esta cláusula fazendo com que a computação fique concentrada neste procedimento, sem chance de computar outras partes da árvore de igual ou maior importância. Na seção 5.3.4 veremos que uma medida de granulosidade que atribui um peso menor a cláusulas com chamadas recursivas produz resultados melhores.

(a) Queens, ideal, TM,32,sem



(b) Queens, ideal, LM,32,sem

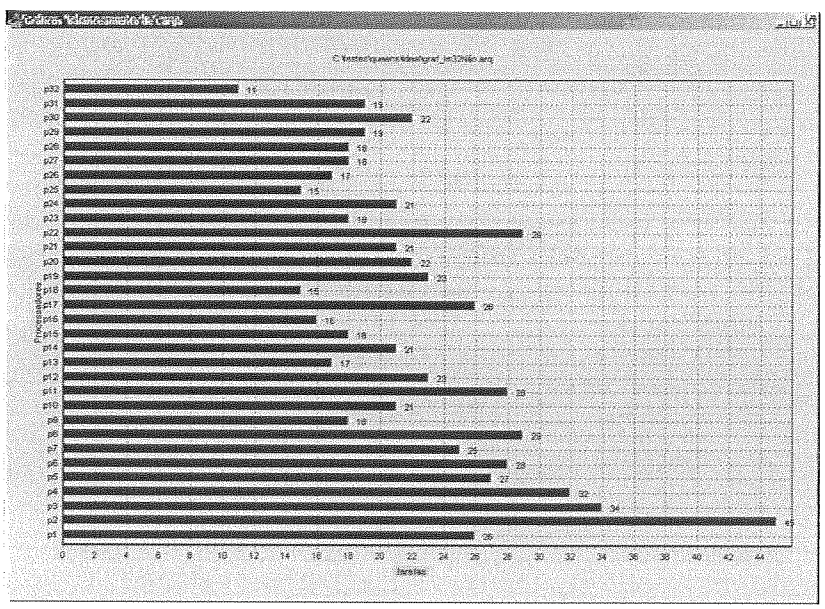
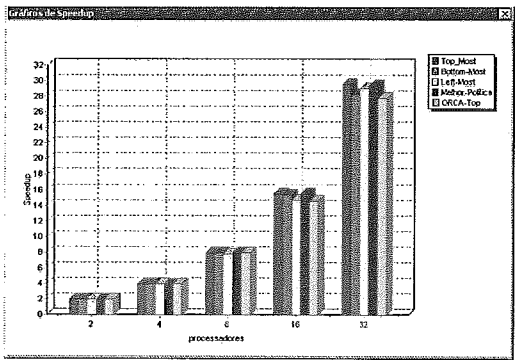
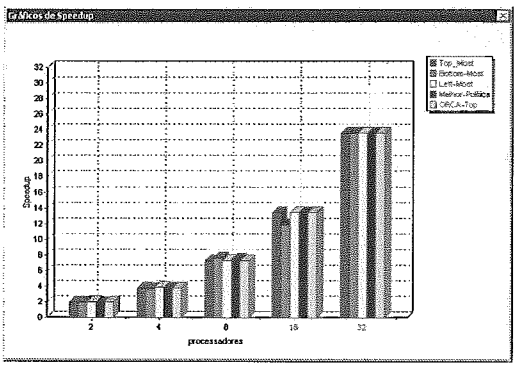


Figura 5.1: Balanceamento de carga, queens, 32 procs, sem informação de granulosidade, TM e LM.

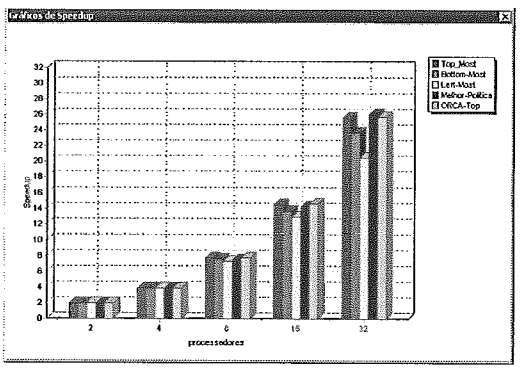
(a) Mutest, mu_top1.



(b) Family, main.



(c) Queens, queens(6,M).



(d) Chat, q5_demo(A).

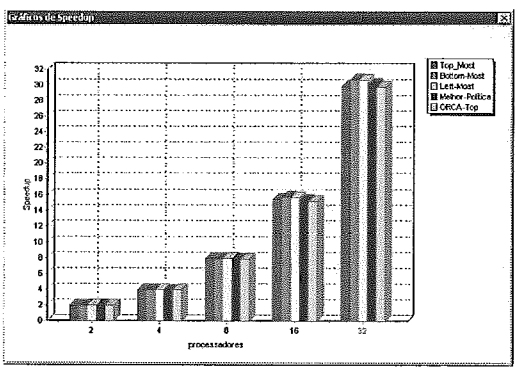


Figura 5.2: Speedups na simulação ideal

5.3.2 Simulação com custo de memória compartilhada

As tabelas 5.7, 5.8, 5.9 e 5.10 mostram respectivamente os speedups das aplicações `mutest`, `family`, `queens` e `chat` com 2, 4, 8, 16 e 32 processadores, com ou sem informação de granulosidade para todas as políticas de escalonamento.

Observando os resultados sem informação de granulosidade, notamos mais uma vez que o desempenho das aplicações varia conforme a variação do número de processadores e com a estratégia. Porém, no caso desta simulação com *overhead* de memória compartilhada, todas as aplicações se beneficiam da estratégia menor-custo. Isto confirma os dados já apresentados por Carrusca [9]. Somente `family` (Tabela 5.8), para 2 processadores, tem desempenho melhor com as estratégias *left-most* ou *bottom-most*. `Family` também é a que sofre menor variação em tempo de execução com uma diferença percentual máxima de 9,7% para 2 processadores, entre a pior política, *top-most*, e a melhor política, *left-most* ou *bottom-most*.

A aplicação `mutest` (Tabela 5.7) é a que sofre maior variação em tempo de execução com uma diferença percentual máxima de 82%; para 4 processadores, entre a pior política, *top-most*, e a melhor política, menor-custo.

Com a inclusão da informação de granulosidade em cada política, todas as aplicações continuam se beneficiando da estratégia menor-custo. Apenas a aplicação `family` (Tabela 5.8), com 2 processadores, apresenta *bottom-most* ou *left-most* como melhor política.

A introdução de informação de granulosidade aumentou os *speedups* originais das estratégias para a aplicação `mutest`, para quase todos os números de processadores, com destaque para 16 processadores, que teve melhora do *speedup* mínimo de 2,8% e do *speedup* máximo de 20%. A aplicação `chat` teve uma queda de desempenho do *speedup* máximo para todos os números de processadores, enquanto melhorou o *speedup* mínimo a partir de 8 processadores. Desta vez `chat` foi a aplicação que sofreu com o tipo de informação de granulosidade obtida que atribui pesos muito altos às cláusulas. Na seção 5.3.4 veremos que medidas alternativas podem produzir melhor desempenho.

A maior contribuição da informação de granulosidade foi um aumento de 20% no speedup da aplicação `mutest` (Tabela 5.7), para 16 processadores, na política de menor-custo, cujo speedup subiu de 4,48 para 5,61.

As figuras 5.3(a), 5.3(b), 5.3(c) e 5.3(d) mostram os speedups para nossos

benchmarks até 32 processadores com análise de granulosidade, para as cinco políticas estudadas.

mutest						
	Complex.	TM	BM	LM	MC	ORCA-Top
2	sem	32976(0.20)	9603(0.68)	10264(0.63)	5912(1.10)	-
	com	35330(0.18)	10212(0.64)	-	5401(1.20)	12526(0.52)
4	sem	17876(0.36)	8663(0.75)	8477(0.77)	3234(2.01)	-
	com	17763(0.37)	8816(0.74)	-	2960(2.20)	8732(0.75)
8	sem	9068(0.72)	5723(1.14)	5791(1.12)	1950(3.34)	-
	com	9008(0.72)	5802(1.12)	-	1786(3.64)	5407(1.20)
16	sem	4710(1.38)	3577(1.82)	3509(1.85)	1452(4.48)	-
	com	4598(1.42)	3592(1.81)	-	1161(5.61)	3424(1.90)
32	sem	2431(2.68)	2098(3.10)	2322(2.80)	1272(5.12)	-
	com	2391(2.72)	2155(3.02)	-	1143(5.69)	2220(2.93)

Tabela 5.7: Speedups de mutest, memória compartilhada

family						
	Complex.	TM	BM	LM	MC	ORCA-Top
2	sem	279(1.02)	251(1.13)	252(1.13)	256(1.11)	-
	com	276(1.03)	254(1.12)	-	256(1.11)	276(1.03)
4	sem	145(1.96)	135(2.10)	142(2.00)	126(2.25)	-
	com	144(1.97)	132(2.15)	-	130(2.18)	144(1.97)
8	sem	73(3.89)	72(3.94)	74(3.84)	65(4.37)	-
	com	73(3.89)	75(3.79)	-	65(4.37)	73(3.89)
16	sem	37(7.68)	39(7.28)	39(7.28)	31(9.16)	-
	com	39(7.28)	39(7.28)	-	31(9.16)	39(7.28)
32	sem	19(14.95)	19(14.95)	19(14.95)	15(18.93)	-
	com	19(14.95)	19(14.95)	-	15(18.93)	19(14.95)

Tabela 5.8: Speedups de family, memória compartilhada

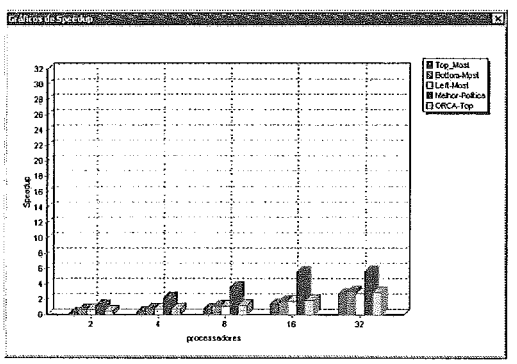
queens						
	Complex.	TM	BM	LM	MC	ORCA-Top
2	sem	7890(0.51)	4822(0.83)	4552(0.88)	2679(1.50)	-
	com	7890(0.51)	4822(0.83)	-	2671(1.50)	7779(0.52)
4	sem	3943(1.02)	2954(1.36)	3125(1.29)	1400(2.87)	-
	com	3943(1.02)	2954(1.36)	-	1396(2.88)	3893(1.03)
8	sem	1998(2.01)	1751(2.30)	1837(2.19)	766(5.25)	-
	com	1998(2.01)	1751(2.30)	-	773(5.20)	1995(2.01)
16	sem	1015(3.96)	985(4.08)	1059(3.80)	513(7.83)	-
	com	1015(3.96)	985(4.08)	-	513(7.83)	990(4.06)
32	sem	574(7.00)	696(5.77)	662(6.07)	357(11.26)	-
	com	574(7.00)	696(5.77)	-	357(11.26)	559(7.19)

Tabela 5.9: Speedups de queens, memória compartilhada

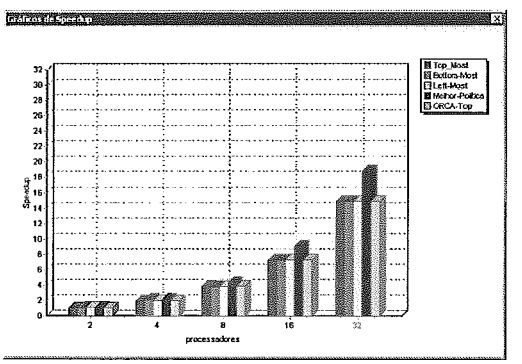
chat						
	Complex.	TM	BM	LM	MC	ORCA-Top
2	sem	28062(0.26)	11405(0.63)	11306(0.63)	6315(1.14)	-
	com	32060(0.22)	11549(0.62)	-	6640(1.08)	11930(0.60)
4	sem	16433(0.44)	8987(0.80)	8934(0.80)	3243(2.21)	-
	com	16368(0.44)	8614(0.83)	-	3393(2.12)	9951(0.72)
8	sem	8302(0.86)	5795(1.24)	5659(1.27)	1729(4.15)	-
	com	8255(0.87)	5639(1.27)	-	1769(4.06)	6130(1.17)
16	sem	4204(1.71)	3389(2.12)	3433(2.09)	990(7.25)	-
	com	4190(1.71)	3387(2.12)	-	994(7.22)	3623(1.98)
32	sem	2133(3.36)	1901(3.78)	1944(3.69)	599(11.98)	-
	com	2116(3.39)	1934(3.71)	-	607(11.82)	2027(3.54)

Tabela 5.10: Speedups de chat, memória compartilhada

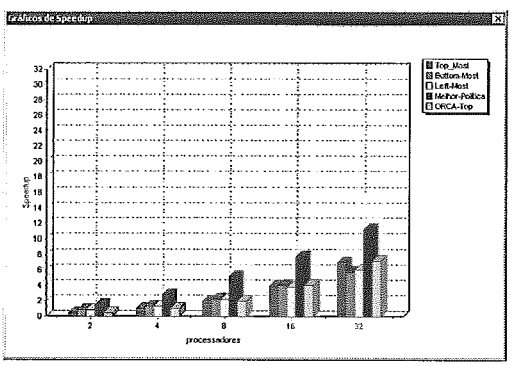
(a) Mutest, mu_top1.



(b) Family, main.



(c) Queens, queens(6,M).



(d) Chat, q5_demo(A).

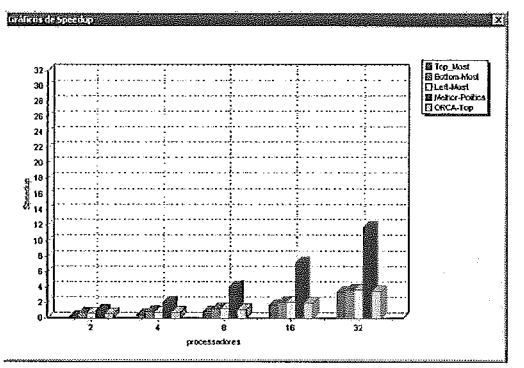


Figura 5.3: Speedups na simulação memória compartilhada

5.3.3 Simulação com custo de memória distribuída

As tabelas 5.11, 5.12, 5.13 e 5.14 mostram os speedups para todas as aplicações com 2, 4, 8, 16 e 32 processadores, com ou sem análise de complexidade para todas as políticas de escalonamento.

Os resultados sem análise de granulosidade apresentam *slowdown* para quase todas as aplicações e estratégias. Observamos mais uma vez que a estratégia menor-custo beneficia todas as aplicações, confirmando os resultados de Carrusca. `Chat` é a que sofre maior variação em tempo de execução, com uma diferença percentual máxima de 95,8%, para 4 processadores, entre a pior política, *top-most*, e a melhor política, menor-custo. `Queens`, para 32 processadores, é a que sofre menor variação em tempo de execução com uma diferença percentual máxima de 36,7%, entre a pior política, *bottom-most* e a melhor política, menor-custo.

No contexto de inclusão de informação de granulosidade, as aplicações `family` e `mutest` aumentaram seu desempenho original, porém `mutest` continuou com *slowdown* em relação a execução sequencial. `Mutest` (Tabela 5.11) para 8 processadores, teve um aumento de 26,8% no desempenho máximo. `Family` (Tabela 5.12) apresenta para 16 processadores, uma melhora mínima de 1,6% e máxima de 6,4%.

Notamos que houve uma contribuição considerável de 22,6% no *speedup* da aplicação `family` (Tabela 5.12), para 8 processadores, na política menor-custo, cujo *speedup* subiu de 1,64 para 2,12.

`Mutest` (Tabela 5.11) é a que sofre maior variação em tempo de execução, com uma diferença percentual máxima de 96%, para 2 processadores entre a pior política, *top-most*, e a melhor política, menor-custo.

A aplicação `chat` é a única que teve queda de desempenho para todos os números de processadores com a inclusão de informação de granulosidade. Com destaque para 16 processadores na política menor-custo, cujo *speedup* diminuiu de 2,77 para 0,81 (*slowdown*), uma diminuição de 70,8%, e para 8 processadores, também para política menor-custo, cujo *speedup* diminuiu de 1,88 para 0,56 (*slowdown*), uma diminuição de 70,2%.

mutest						
	Complex.	TM	BM	LM	MC	ORCA-Top
2	sem	247686(0.03)	18176(0.36)	16201(0.40)	12785(0.51)	-
	com	279220(0.02)	18176(0.36)	-	13000(0.50)	29659(0.22)
4	sem	187421(0.03)	23331(0.28)	23600(0.28)	14130(0.46)	-
	com	201139(0.03)	21124(0.31)	-	13113(0.50)	38332(0.17)
8	sem	111963(0.06)	22835(0.29)	30435(0.21)	16051(0.41)	-
	com	114979(0.06)	28966(0.22)	-	11648(0.56)	27312(0.24)
16	sem	60552(0.11)	22739(0.29)	28923(0.23)	15100(0.43)	-
	com	57994(0.11)	21932(0.30)	-	12253(0.53)	23745(0.27)
32	sem	31310(0.21)	19272(0.34)	21173(0.31)	13388(0.49)	-
	com	28675(0.23)	19177(0.34)	-	13307(0.49)	20901(0.31)

Tabela 5.11: Speedups de mutest, memória distribuída

family						
	Complex.	TM	BM	LM	MC	ORCA-Top
2	sem	879(0.32)	456(0.62)	456(0.62)	272(1.04)	-
	com	759(0.37)	417(0.68)	-	290(0.98)	759(0.37)
4	sem	795(0.36)	435(0.65)	405(0.70)	187(1.52)	-
	com	539(0.53)	400(0.71)	-	227(1.25)	539(0.53)
8	sem	364(0.78)	348(0.82)	302(0.94)	173(1.64)	-
	com	392(0.72)	296(0.96)	-	134(2.12)	392(0.72)
16	sem	232(1.22)	193(1.47)	194(1.46)	108(2.63)	-
	com	229(1.24)	190(1.49)	-	101(2.81)	229(1.24)
32	sem	131(2.17)	131(2.17)	131(2.17)	81(3.51)	-
	com	131(2.17)	131(2.17)	-	81(3.51)	131(2.17)

Tabela 5.12: Speedups de family, memória distribuída

queens						
	Complex.	TM	BM	LM	MC	ORCA-Top
2	sem	50480(0.08)	7862(0.51)	8674(0.46)	6766(0.59)	-
	com	50480(0.08)	7862(0.51)	-	6741(0.60)	49514(0.08)
4	sem	36322(0.11)	12460(0.32)	11041(0.36)	6226(0.65)	-
	com	36322(0.11)	12460(0.32)	-	6628(0.61)	35466(0.11)
8	sem	20164(0.20)	11076(0.36)	9540(0.42)	4703(0.85)	-
	com	20164(0.20)	11076(0.36)	-	4677(0.86)	19938(0.20)
16	sem	9681(0.42)	8466(0.47)	7987(0.50)	4447(0.90)	-
	com	9681(0.42)	8466(0.47)	-	4177(0.96)	10077(0.40)
32	sem	5601(0.72)	6495(0.62)	6313(0.64)	4106(0.98)	-
	com	5601(0.72)	6495(0.62)	-	4106(0.98)	5092(0.79)

Tabela 5.13: Speedups de queens, memória distribuída

chat						
	Complex.	TM	BM	LM	MC	ORCA-Top
2	sem	119837(0.06)	15836(0.45)	14900(0.48)	9661(0.74)	-
	com	203493(0.04)	17101(0.42)	-	24443(0.29)	39367(0.18)
4	sem	143194(0.05)	23562(0.30)	19445(0.37)	6100(1.18)	-
	com	178727(0.04)	23243(0.31)	-	18569(0.39)	42947(0.17)
8	sem	88798(0.08)	27335(0.26)	22316(0.32)	3826(1.88)	-
	com	99335(0.07)	23740(0.30)	-	12755(0.56)	37278(0.19)
16	sem	50475(0.14)	19923(0.36)	20897(0.34)	2589(2.77)	-
	com	52715(0.14)	19625(0.37)	-	8817(0.81)	25145(0.29)
32	sem	28031(0.26)	14285(0.50)	14641(0.49)	4389(1.64)	-
	com	26629(0.27)	13746(0.52)	-	5091(1.41)	18977(0.38)

Tabela 5.14: Speedups de chat, memória distribuída

5.3.4 Outras medidas de complexidade

A medida de complexidade *default* utilizada pelo ORCA apresenta uma desvantagem que é a atribuição de pesos muito altos a chamadas recursivas de cláusulas dos programas Prolog. Esta atribuição de pesos muito altos pode fazer com que a exploração de ramos mais importantes do espaço de busca seja postergada, atrasando a execução total do programa.

Os quatro programas que utilizamos têm chamadas recursivas em alguma cláusula, porém os programas que mais sofrem com a medida *default* do ORCA são `mutest` e `chat`. Estes são os que possuem chamadas de maior peso como mostra a tabela 5.15.

Nas seções anteriores, vimos que `mutest` tem piora de desempenho na simulação ideal para 32 processadores (Tabela 5.3). Fazendo uma simulação ideal onde a informação de granulosidade atribui peso 1 às chamadas recursivas (assim como no trabalho de Tick [39]), o speedup máximo, para 32 processadores, sobe de 29,58 para 29,99, maior do que o melhor resultado sem nenhuma informação de granulosidade que era 29,85.

Vimos também que a aplicação `chat`, na simulação de memória compartilhada teve piora de desempenho com a inclusão de informação de granulosidade *default*. Utilizando a medida que atribui peso 1 às chamadas recursivas, melhoramos o speedup de `chat`, para 32 processadores, de 11,82 para 12,04 com a política menor custo. Este resultado é melhor do que o *speedup* para 32 processadores, sem informação de granulosidade, que era de 11,98.

programa	resolução	tick
`mutest`	rule3/25, theorem/131 , rule4/15, app/5	rule3/9, theorem/35 , rule4/6, app/3
`family`	membro/5 , renda/7	membro/3 , renda/4
`queens`	queens/39, not_attack/11 , sel/5, range/9	queens/13, not_attack/6 , sel/3, range/5
`chat`	contains/753	contains/377

Tabela 5.15: Peso dos procedimentos

Estes resultados indicam claramente que a informação *default* produzida pelo ORCA não é a mais adequada quando tratamos de chamadas recursivas. Uma medida mais simples e que atribui pesos menores às cláusulas recursivas pode ser uma alternativa mais interessante.

5.3.5 Resumo

Na simulação ideal a informação de granulosidade baseada em resolução melhora os speedups mínimos e máximos em duas aplicações consideradas reais, `queens` e `chat`, e não piora speedups máximos e mínimos de `family`. `mutest` piora alguns speedups, mas a piora máxima é de 2,5%, para 32 processadores, no speedup mínimo, produzido com a política que beneficia menos a aplicação. É importante notar que a informação de granulosidade beneficiou aplicações irregulares cuja quantidade de trabalho é difícil de ser obtida em tempo de execução sem auxílio desta informação.

Na simulação com custos arquiteturais a presença de informação de granulosidade baseada em resolução beneficia quase todas as aplicações, produzindo um ganho significativo para `mutest` quando combinamos a informação de granulosidade com a política de menor custo. `chat` é a única aplicação que sofre um impacto negativo com os custos arquiteturais, porém alcança uma melhora significativa para números altos de processadores quando utilizamos informação de granulosidade baseada em medidas que atribuem menor peso às cláusulas do programa. O mesmo ocorre com `mutest` na simulação ideal quando adotamos medidas que atribuem menor peso às cláusulas.

Capítulo 6

Conclusões e Trabalhos Futuros

Neste capítulo da dissertação serão expostas as conclusões e contribuições do autor assim como as perspectivas de trabalhos futuros.

6.1 Conclusões e contribuições

Este trabalho teve como objetivo avaliar o impacto da introdução de informação de granulosidade em políticas de escalonamento para sistemas de programação em lógica que exploram paralelismo OU. Para isto utilizamos um simulador de execução paralela de programas Prolog implementado por Carrusca [9] e a ferramenta de análise de granulosidade ORCA, implementada por Salazar [40]. Modificamos o simulador para dar suporte à informação de granulosidade produzida pelo ORCA e modificamos o ORCA para fornecer outra medida de complexidade, onde o peso de uma chamada recursiva numa cláusula é atribuído o valor 1.

Para que as informações de granulosidade fossem incorporadas ao simulador, criou-se uma nova estrutura de dados, chamada de ORCA. Essas informações foram então utilizadas pelas políticas *top-most*, *bottom-most* e menor-custo, além da uma nova política ORCA-Top, que ordena tarefas levando em consideração apenas o peso de cada uma delas gerado pelo ORCA.

Nossos resultados mostraram que a combinação das informações de granulosidade geradas em tempo de compilação, com as políticas de escalonamento até então sugeridas, produziram resultados positivos para quase todas as aplicações. O desempenho mínimo e máximo de quase todas as aplicações aumentou tanto para a simulação ideal quanto para a simulação de memória compartilhada, e até mesmo para a simulação de memória distribuída que havia apresentado *slowdown*.

Os dois tipos de medida de granulosidade utilizados neste trabalho não são

ideais, porém produziram resultados significativamente positivos para aplicações com diferentes características. Uma das vantagens de se utilizar tais medidas mesmo não sendo ideais é que a geração da informação pode ser efetuada durante a compilação sem maiores *overheads*.

Uma outra alternativa que gere informações mais precisas nos moldes do CASLOG [30] poderia ser investigada, embora sua complexidade tanto para análise quanto para manutenção das informações seja mais alta.

Como trabalhos futuros pretendemos investigar outras medidas de complexidade, avaliar seus custos e impacto no desempenho das aplicações. Pretendemos também simular outras aplicações e dar suporte à execução de operadores de corte para melhor entender o comportamento das políticas de escalonamento nas várias árvores de execução.

Apêndice A

Códigos fontes de alguns Benchmarks

```
%*************************************************************************
% mutest código com informações de granulosidade por Resolução - tem recursão
%*************************************************************************

'__c_main1/0'(134,2,[133,1],0,[]).
'__c_main2/0'(133,1,[133],0,[]).
'__c_main3/0'(133,1,[133],0,[]).
'__c_main4/0'(133,1,[133],0,[]).
'__c_mu_top1/0'(132,1,[132],0,[]).
'__c_mu_top2/0'(132,1,[132],0,[]).
'__c_mu_top3/0'(132,1,[132],0,[]).
'__c_rules/2'(61,4,[26,16,12,7],0,[]).
'__c_rule1/2'(11,1,[11],0,[]).
'__c_rule2/2'(6,1,[6],0,[]).
'__c_rule3/2'(25,2,[11,14],13,[]).
'__c_rule4/2'(15,2,[6,9],8,[]).
'__c_theorem/2'(131,2,[1,130],66,[]).
'__c_app/3'(5,2,[1,4],3,[]).

main1 :- mu_top,fail.
main1.

main2 :- mu_top1,fail.
main2.

main3 :- mu_top2,fail.
main3.

main4 :- mu_top3,fail.
main4.

mu_top :- theorem(5,[m,u,i,i,u]).
mu_top1 :- theorem(6,[m,i,i,i,i,i]).
mu_top2 :- theorem(4,[m,u,u,i]).
mu_top3 :- theorem(18,[m,i,i,i,i,i,m,u,u,u,u,u,m,i,i,i,i,i]).

rules(S,R) :- rule3(S,R).
rules(S,R) :- rule4(S,R).
rules(S,R) :- rule1(S,R).
rules(S,R) :- rule2(S,R).
```

```
rule1(S,R) :-
    app(X,[i],S),
    app(X,[i,u],R).

rule2([m|T],[m|R]) :- app(T,T,R).

rule3(R,T) :-
    app([i,i,i],S,R),
    app([u],S,T).
rule3([H|T],[H|R]) :- rule3(T,R).

rule4([H|L],T) :- app([u,u],T,[H|L]).
rule4([H|T],[H|R]) :- rule4(T,R).

theorem(_,[m,i]).
theorem(Depth,[H|L]) :-
    Depth > 0,
    D is Depth-1,
    theorem(D,S),
    rules(S,[H|L]).

app([],X,X).
app([A|B],X,[A|B1]) :-
    app(B,X,B1).
*fim
```

%**
%**family** código com informações de granulosidade por Resolução - tem recursão
%**

```
'__c_main/0'(324,32,[13,2,13,13,9,2,13,2,13,13,13,2,13,2,
        13,13,13,13,13,2,13,13,13,13,13,2,13,13,13,13,13,2],0,[]).
'__c_pai/1'(2,1,[2],0,[]).
'__c_mae/1'(2,1,[2],0,[]).
'__c_filho/1'(7,1,[7],0,[]).
'__c_membro/2'(5,2,[1,4],3,[]).
'__c_existe/1'(12,1,[12],0,[]).
'__c_nasceu/2'(1,1,[1],0,[]).
'__c_salario/1'(2,1,[2],0,[]).
'__c_salario/2'(1,1,[1],0,[]).
'__c_trabalho/1'(2,1,[2],0,[]).
'__c_trabalho/2'(1,1,[1],0,[]).
'__c_renda/2'(7,2,[1,6],4,[]).
'__c_familia/2'(4,1,[4],0,[]).

main:- existe(pessoa(andre,X,Y,Z)).
main:- existe(pessoa(X,Y,data(07,11,96),Z)).
main:- existe(pessoa(X,Y,Z,trab(prefeito,W))).
main:- existe(pessoa(X,Y,Z,trab(W,2500))).
main:- familia(X,Y,Z), renda([X,Y|Z],R).
main:- familia(X,Y,Z).
main:- existe(pessoa(erly,X,Y,Z)).
main:- existe(pessoa(X,Y,data(10,04,79),Z)).
main:- existe(pessoa(X,Y,Z,trab(secretario,W))).
main:- existe(pessoa(X,Y,Z,trab(W,4000))).
main:- existe(pessoa(valmir,X,Y,Z)).
main:- existe(pessoa(X,Y,data(01,09,49),Z)).
```

```
main:- existe(pessoa(marilete,X,Y,Z)).
main:- existe(pessoa(X,Y,data(07,08,73),Z)).
main:- existe(pessoa(X,Y,Z,trab(secretaria,W))).
main:- existe(pessoa(X,Y,Z,trab(W,1500))).
main:- existe(pessoa(X,Y,Z,trab(professora,W))).
main:- existe(pessoa(X,Y,Z,trab(analista,W))).
main:- existe(pessoa(viviane,X,Y,Z)).
main:- existe(pessoa(X,Y,data(15,10,54),Z)).
main:- existe(pessoa(X,Y,Z,trab(comerciante,W))).
main:- existe(pessoa(X,Y,Z,trab(W,300))).
main:- existe(pessoa(creusa,X,Y,Z)).
main:- existe(pessoa(wagner,X,Y,Z)).
main:- existe(pessoa(waldeci,X,Y,Z)).
main:- existe(pessoa(X,Y,data(17,02,79),Z)).
main:- existe(pessoa(X,Y,Z,trab(medico,W))).
main:- existe(pessoa(X,Y,Z,trab(W,800))).
main:- existe(pessoa(hugo,X,Y,Z)).
main:- existe(pessoa(X,Y,Z,trab(W,2300))).
main:- existe(pessoa(georgina,X,Y,Z)).
main:- existe(pessoa(X,Y,data(17,05,52),Z)).

pai(X):-
    familia(X, _, _).

mae(X):-
    familia(_, X, _).

filho(X):-
    familia(_, _, Filhos),
    membro(X, Filhos).

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

existe(Pessoa):-
        pai(Pessoa);
        mae(Pessoa);
        filho(Pessoa).

nasceu(pessoa(_, _, Data, _), Data).

salario(pessoa(_, _, _, trab(_, S)), S).
salario(pessoa(_, _, _,nt), 0).

trabalho(pessoa(_, _, _, trab(T,_)),T).
trabalho(pessoa(_, _, _,nt),'nao trabalha').

renda([],0).
renda([Pessoa|Lista],Total):-salario(Pessoa,Salario),
                renda(Lista,Soma),
                Total is Soma+Salario.

familia(pessoa(ari, oliveira, data(17, 05, 65), trab(medico, 1500)),
        pessoa(ana, oliveira, data(06, 11, 68), trab(professora, 300)),
        [pessoa(ada, oliveira, data(18, 02, 91), nt)]).
```

```
*fim
```

%**
%**queens** código com informações de granulosidade por Resolução - tem recursão
%**

```
'__c_queens/2'(49,1,[49],0,[]).
'__c_queens/3'(39,2,[1,38],20,[]).
'__c_not_attack/2'(12,1,[12],0,[]).
'__c_not_attack/3'(11,2,[1,10],6,[]).
'__c_sel/3'(5,2,[1,4],3,[]).
'__c_range/3'(9,2,[1,8],5,[]).

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

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

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

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

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

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

%**
% **chat** informações de granulosidade por Resolução
%**

```
'__c_q5_demo/1'(8378,1,[8378],0,[]).
'__c_african/1'(1102,1,[1102],0,[]).
'__c_american/1'(1102,1,[1102],0,[]).
'__c_area/1'(1,1,[1],0,[]).
'__c_asian/1'(1102,1,[1102],0,[]).
'__c_capital/1'(74,1,[74],0,[]).
'__c_circle_of_latitude/1'(5,5,[1,1,1,1,1],0,[]).
```

```
'__c_city/1'(56,1,[56],0,[]).
'__c_continent/1'(2,2,[1,1],0,[]).
'__c_country/1'(73,1,[73],0,[]).
'__c_european/1'(1102,1,[1102],0,[]).
'__c_latitude/1'(1,1,[1],0,[]).
'__c_longitude/1'(1,1,[1],0,[]).
'__c_ocean/1'(5,5,[1,1,1,1,1],0,[]).
'__c_place/1'(98,1,[98],0,[]).
'__c_place1/1'(97,4,[3,11,9,74],0,[]).
'__c_population/1'(2,2,[1,1],0,[]).
'__c_region/1'(10,1,[10],0,[]).
'__c_river/1'(24,1,[24],0,[]).
'__c_river/2'(23,23,[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],0,[]).
'__c_seamass/1'(8,2,[6,2],0,[]).
'__c_area/2'(73,1,[73],0,[]).
'__c_capital/2'(73,1,[73],0,[]).
'__c_drains/2'(25,1,[25],0,[]).
'__c_eastof/2'(151,1,[151],0,[]).
'__c_exceeds/2'(4,2,[1,3],0,[]).
'__c_first/2'(1,1,[1],0,[]).
'__c_flows/2'(30,1,[30],0,[]).
'__c_in/2'(1101,3,[756,172,173],0,[]).
'__c_in0/2'(170,4,[10,56,73,31],0,[]).
'__c_in1/2'(2,2,[1,1],0,[]).
'__c_in_continent/2'(9,9,[1,1,1,1,1,1,1,1,1],0,[]).
'__c_last/2'(5,2,[1,4],3,[]).
'__c_longitude/2'(73,1,[73],0,[]).
'__c_northof/2'(7,1,[7],0,[]).
'__c_population/2'(129,2,[56,73],0,[]).
'__c_rises/2'(29,1,[29],0,[]).
'__c_southof/2'(7,1,[7],0,[]).
'__c_westof/2'(151,1,[151],0,[]).
'__c_flows/3'(29,1,[29],0,[]).
'__c_links/3'(5,2,[1,4],3,[]).
'__c_ratio/4'(4,4,[1,1,1,1],0,[]).
'__c_city/3'(55,55,[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
              1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],0,[]).
'__c_country/10'(72,72,[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
              1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
              1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],0,[]).
'__c_contains/2'(753,2,[188,565],377,[]).
'__c_contains0/2'(187,187,[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],0,[]).
'__c_borders/2'(492,492,[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
```

```
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1],0,[]).
```

%***
% **mutest** informações de granulosidade por Aridade
%***

```
'__c_main1/0'(344,2,[344,0],0,[]).
'__c_main2/0'(344,1,[344],0,[]).
'__c_main3/0'(344,1,[344],0,[]).
'__c_main4/0'(344,1,[344],0,[]).
'__c_mu_top1/0'(344,1,[344],0,[]).
'__c_mu_top2/0'(344,1,[344],0,[]).
'__c_mu_top3/0'(344,1,[344],0,[]).
'__c_rules/2'(167,4,[72,42,34,19],0,[]).
'__c_rule1/2'(32,1,[32],0,[]).
'__c_rule2/2'(17,1,[17],0,[]).
'__c_rule3/2'(70,2,[32,38],36,[]).
'__c_rule4/2'(40,2,[17,23],21,[]).
'__c_theorem/2'(344,2,[2,342],173,[]).
'__c_app/3'(15,2,[3,12],9,[]).
```

%***
% **family** informações de granulosidade por Aridade
%***

```
'__c_main/0'(563,32,[23,2,23,23,17,3,23,2,23,23,23,2,23,2,23,23,23,23,23,2,
             23,23,23,23,23,2,23,23,23,23,23,2],0,[]).
'__c_pai/1'(4,1,[4],0,[]).
'__c_mae/1'(4,1,[4],0,[]).
'__c_filho/1'(14,1,[14],0,[]).
'__c_membro/2'(10,2,[2,8],6,[]).
'__c_existe/1'(23,1,[23],0,[]).
'__c_nasceu/2'(2,1,[2],0,[]).
'__c_salario/1'(1,1,[1],0,[]).
'__c_salario/2'(2,1,[2],0,[]).
'__c_trabalho/1'(1,1,[1],0,[]).
'__c_trabalho/2'(2,1,[2],0,[]).
'__c_renda/2'(14,2,[2,12],8,[]).
'__c_familia/2'(10,1,[10],0,[]).
```

```
%*************************************************************************
% queens informações de granulosidade por Aridade
%*************************************************************************

'__c_queens/2'(96,1,[96],0,[]).
'__c_queens/3'(79,2,[3,76],41,[]).
'__c_not_attack/2'(17,1,[17],0,[]).
'__c_not_attack/3'(15,2,[3,12],9,[]).
'__c_sel/3'(15,2,[3,12],9,[]).
'__c_range/3'(15,2,[3,12],9,[]).



%*************************************************************************
% chat informações de granulosidade por Aridade
%*************************************************************************

'__c_q5_demo/1'(28172,1,[28172],0,[]).
'__c_african/1'(3476,1,[3476],0,[]).
'__c_american/1'(3476,1,[3476],0,[]).
'__c_area/1'(1,1,[1],0,[]).
'__c_asian/1'(3476,1,[3476],0,[]).
'__c_capital/1'(723,1,[723],0,[]).
'__c_circle_of_latitude/1'(5,5,[1,1,1,1,1],0,[]).
'__c_city/1'(166,1,[166],0,[]).
'__c_continent/1'(2,2,[1,1],0,[]).
'__c_country/1'(721,1,[721],0,[]).
'__c_european/1'(3476,1,[3476],0,[]).
'__c_latitude/1'(1,1,[1],0,[]).
'__c_longitude/1'(1,1,[1],0,[]).
'__c_ocean/1'(5,5,[1,1,1,1,1],0,[]).
'__c_place/1'(755,1,[755],0,[]).
'__c_place1/1'(754,4,[3,20,9,722],0,[]).
'__c_population/1'(2,2,[1,1],0,[]).
'__c_region/1'(19,1,[19],0,[]).
'__c_river/1'(47,1,[47],0,[]).
'__c_river/2'(46,23,[2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2],0,[]).
'__c_seamass/1'(8,2,[6,2],0,[]).
'__c_area/2'(722,1,[722],0,[]).
'__c_capital/2'(722,1,[722],0,[]).
'__c_drains/2'(50,1,[50],0,[]).
'__c_eastof/2'(1454,1,[1454],0,[]).
'__c_exceeds/2'(8,2,[2,6],0,[]).
'__c_first/2'(2,1,[2],0,[]).
'__c_flows/2'(66,1,[66],0,[]).
'__c_in/2'(3475,3,[1510,982,983],0,[]).
'__c_in0/2'(977,4,[20,167,722,68],0,[]).
'__c_in1/2'(2,2,[2,0],0,[]).
'__c_in_continent/2'(18,9,[2,2,2,2,2,2,2,2,2],0,[]).
'__c_last/2'(10,2,[2,8],6,[]).
'__c_longitude/2'(722,1,[722],0,[]).
'__c_northof/2'(14,1,[14],0,[]).
'__c_population/2'(889,2,[167,722],0,[]).
'__c_rises/2'(58,1,[58],0,[]).
'__c_southof/2'(14,1,[14],0,[]).
'__c_westof/2'(1454,1,[1454],0,[]).
```

```
'__c_flows/3'(64,1,[64],0,[]).
'__c_links/3'(15,2,[3,12],9,[]).
'__c_ratio/4'(16,4,[4,4,4,4],0,[]).
'__c_city/3'(165,55,[3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,
             3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3],0,[]).
'__c_country/10'(720,72,[10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,
                 10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,
                 10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,
                 10,10,10,10,10,10,10,10,10,10,10,10],0,[]).
'__c_contains/2'(1506,2,[376,1130],754,[]).
'__c_contains0/2'(374,187,[2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2
                  ,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
                  2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
                  2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
                  2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
                  2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
                  2,2,2,2,2],0,[]).
'__c_borders/2'(984,492,[2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
                  2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
                  2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
                  2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
                  2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
                  2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
                  2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
                  2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
                  2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
                  2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
                  2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
                  2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
                  2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
                  2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
                  2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
                  2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2],0,[]).
```

%***

% **mutest** informações de granulosidade por Tick

%***

```
'__c_main1/0'(38,2,[37,1],0,[]).
'__c_main2/0'(37,1,[37],0,[]).
'__c_main3/0'(37,1,[37],0,[]).
'__c_main4/0'(37,1,[37],0,[]).
'__c_mu_top1/0'(36,1,[36],0,[]).
'__c_mu_top2/0'(36,1,[36],0,[]).
'__c_mu_top3/0'(36,1,[36],0,[]).
'__c_rules/2'(30,4,[10,7,8,5],0,[]).
'__c_rule1/2'(7,1,[7],0,[]).
'__c_rule2/2'(4,1,[4],0,[]).
'__c_rule3/2'(9,2,[7,2],1,[]).
'__c_rule4/2'(6,2,[4,2],1,[]).
'__c_theorem/2'(35,2,[1,34],1,[]).
'__c_app/3'(3,2,[1,2],1,[]).
```

```
%*************************************************************************
% family informações de granulosidade por Tick
%*************************************************************************

'__c_main/0'(275,32,[11,2,11,11,6,2,11,2,11,11,11,2,11,2,11,11,11,11,11,2,
            11,11,11,11,11,11,2,11,11,11,11,11,11,2],0,[]).
'__c_pai/1'(2,1,[2],0,[]).
'__c_mae/1'(2,1,[2],0,[]).
'__c_filho/1'(5,1,[5],0,[]).
'__c_membro/2'(3,2,[1,2],1,[]).
'__c_existe/1'(10,1,[10],0,[]).
'__c_nasceu/2'(1,1,[1],0,[]).
'__c_salario/1'(2,1,[2],0,[]).
'__c_salario/2'(1,1,[1],0,[]).
'__c_trabalho/1'(2,1,[2],0,[]).
'__c_trabalho/2'(1,1,[1],0,[]).
'__c_renda/2'(4,2,[1,3],1,[]).
'__c_familia/2'(4,1,[4],0,[]).



%*************************************************************************
% queens informações de granulosidade por Tick
%*************************************************************************

'__c_queens/2'(19,1,[19],0,[]).
'__c_queens/3'(13,2,[1,12],1,[]).
'__c_not_attack/2'(7,1,[7],0,[]).
'__c_not_attack/3'(6,2,[1,5],1,[]).
'__c_sel/3'(3,2,[1,2],1,[]).
'__c_range/3'(5,2,[1,4],1,[]).



%*************************************************************************
% chat informações de granulosidade por Tick
%*************************************************************************

'__c_q5_demo/1'(6098,1,[6098],0,[]).
'__c_african/1'(722,1,[722],0,[]).
'__c_american/1'(722,1,[722],0,[]).
'__c_area/1'(1,1,[1],0,[]).
'__c_asian/1'(722,1,[722],0,[]).
'__c_capital/1'(74,1,[74],0,[]).
'__c_circle_of_latitude/1'(5,5,[1,1,1,1,1],0,[]).
'__c_city/1'(56,1,[56],0,[]).
'__c_continent/1'(2,2,[1,1],0,[]).
'__c_country/1'(73,1,[73],0,[]).
'__c_european/1'(722,1,[722],0,[]).
'__c_latitude/1'(1,1,[1],0,[]).
'__c_longitude/1'(1,1,[1],0,[]).
'__c_ocean/1'(5,5,[1,1,1,1,1],0,[]).
'__c_place/1'(98,1,[98],0,[]).
'__c_place1/1'(97,4,[3,11,9,74],0,[]).
'__c_population/1'(2,2,[1,1],0,[]).
'__c_region/1'(10,1,[10],0,[]).
'__c_river/1'(24,1,[24],0,[]).
```

```
'__c_river/2'(23,23,[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],0,[]).
'__c_seamass/1'(8,2,[6,2],0,[]).
'__c_area/2'(73,1,[73],0,[]).
'__c_capital/2'(73,1,[73],0,[]).
'__c_drains/2'(25,1,[25],0,[]).
'__c_eastof/2'(151,1,[151],0,[]).
'__c_exceeds/2'(4,2,[1,3],0,[]).
'__c_first/2'(1,1,[1],0,[]).
'__c_flows/2'(28,1,[28],0,[]).
'__c_in/2'(721,3,[380,170,171],0,[]).
'__c_in0/2'(168,4,[10,56,73,29],0,[]).
'__c_in1/2'(0,2,[1,-1],0,[]).
'__c_in_continent/2'(9,9,[1,1,1,1,1,1,1,1,1],0,[]).
'__c_last/2'(3,2,[1,2],1,[]).
'__c_longitude/2'(73,1,[73],0,[]).
'__c_northof/2'(7,1,[7],0,[]).
'__c_population/2'(129,2,[56,73],0,[]).
'__c_rises/2'(27,1,[27],0,[]).
'__c_southof/2'(7,1,[7],0,[]).
'__c_westof/2'(151,1,[151],0,[]).
'__c_flows/3'(27,1,[27],0,[]).
'__c_links/3'(3,2,[1,2],1,[]).
'__c_ratio/4'(4,4,[1,1,1,1],0,[]).
'__c_city/3'(55,55,[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                    1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],0,[]).
'__c_country/10'(72,72,[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                        1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                        1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],0,[]).
'__c_contains/2'(377,2,[188,189],1,[]).
'__c_contains0/2'(187,187,[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                           1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                           1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                           1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                           1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                           1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                           1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],0,[]).
'__c_borders/2'(492,492,[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                         1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                         1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                         1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                         1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                         1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                         1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                         1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                         1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                         1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                         1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                         1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                         1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                         1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                         1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                         1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                         1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                         1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],0,[]).
```

Referências Bibliográficas

[1] Hassan Ait-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, 1991.

[2] Khayri A. M. Ali and Roland Karlsson. The Muse approach to OR-Parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, 1990.

[3] J. L. V. Barbosa. GRANLOG: Um Modelo para Análise Automática de Granulosidade na Programação em Lógica. Master's thesis, Universidade Federal do Rio Grande do Sul, Instituto de Informática, Curso de Pós-Graduação em Ciência da Computação, 1996.

[4] J. L. V. Barbosa, P. Kayser, C. F. R. Geyer, and I. C. Dutra. GRANLOG: An Integrated Granularity Analysis Model for Parallel Logic Programming. In *Workshop on Parallelism and Implementation Technology for (Constraint) Logic Languages*, July 2000.

[5] Anthony Beaumont. *Scheduling in Or-Parallel Prolog Systems.* PhD thesis, University of Bristol, Department of Computer Science, 1993.

[6] Anthony Beaumont, S. Muthu Raman, and Péter Szeredi. Scheduling Or-Parallelism in Aurora with the Bristol Scheduler. Technical Report TR-90-04, University of Bristol, Computer Science Department, March 1990.

[7] J. Briat, M. Favre, C. Geyer, and J. Chassin de Kergommeaux. Opera: Or-parallel prolog system on supernode. In P. Kacsuk and M. Wise, editors, *Implementations of Distributed Prolog*, pages 45–64. Wiley & Sons, New York, 1992.

[8] P. K. Vargas C. F. R. Geyer and I. C. Dutra. Parallelism in logic programming. In *Intl. School on Advanced Techniques for Parallel Computation with Applications*, page 35, Natal, RN, Brasil, Sep-Oct 1999.

[9] Adriana Marino Carrusca. Análise de políticas de escalonamento para sistemas prolog que exploram paralelismo ou. Dissertação de mestrado, COPPE - Engenharia de Sistemas e Computação - Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, Brasil, Março 2000.

[10] Ana Paula Bluhm Centeno. Penelope – Um Modelo de Escalonador Hierárquico para o Sistemas Plosys. Master's thesis, Universidade Federal do Rio Grande do Sul, Instituto de Informática, Curso de Pós-Graduação em Ciência da Computação, 1999.

[11] Ana Paula Bluhm Centeno and C. Geyer. Penelope – Um Modelo de Escalonador Hierárquico para o Sistemas Plosys. In *X Simpósio Brasileiro de Arquitetura de Computadores, SBAC-PAD*, Setembro 1998.

[12] Luciano Roth Coelho. Instalação e aprimoramento do protótipo granlog na universidade católica de pelotas. Dissertação de graduação, Universidade Católica de Pelotas, UCPEL, 1997.

[13] Cristiano André da Costa. Um Estudo das Propostas que Integram o Paralelismo E/OU na Programação em Lógica. Technical Report TI n. 493, CPGCC/UFRGS, Universidade Federal do Rio Grande do Sul, Instituto de Informática, Curso de Pós-Graduação em Ciência da Computação, Jan 1996.

[14] Soumya K. Debray. A Remark on Tick's Algorithm for Compile-Time Granularity Analysis. *Logic Programming Newsletter*, 3(1):9–10, 1989.

[15] I. C. Dutra. *Distributing And- and Or-Work in the Andorra-I Parallel Logic Programming System.* PhD thesis, University of Bristol, Department of Computer Science, February 1995. available at http://www.cos.ufrj.br/~ines.

[16] I. C. Dutra and A. M. Carrusca. Or-parallel scheduling strategies revisited. In *Anais do XIII Simposio Brasileiro de Arquitetura de Computadores, SBAC'00*, pages 263–268, Outubro 2000.

[17] M. J. Fernandez, M. Carro, and M. V. Hermenegildo. IDRA (IDeal Resource Allocation): A Tool for Computing Ideal Speedups. In *ICLP'94 Pre-Conference Workshop on Parallel and Data-Parallel Execution of Logic Languages*, Facultad de Informática, Universidad Politécnica de Madrid, June 1994.

[18] Débora Nice Ferrari, Patrícia Kayser Vargas, Cláudio F. R. Geyer, and Jorge L. V. Barbosa. Modelo de integração plosys-granlog: Aplicação da análise de granulosidade na exploração do paralelismo ou. In *XXV Latinamerican Conference of Informatics CLEI- Panel 99*, volume 2, pages 911–922, Assuncion, Paraguay, ago. 30-set. 3 1999.

[19] N. Fonseca, V. S. Costa, and I. C. Dutra. Visall: A new tool to visualise parallel execution of logic programs. In *Proceedings of the Post-ILPS'97 Workshop on Parallelism and Implementation of (Constraint) Logic Programming Languages*, Port Jefferson, NY, USA, October 1997.

[20] Pedro López García and Manuel Hermenegildo. A Technique for Dynamic Term Size Computation via Program Transformation. Research Report, Facultad de Informática, Universidad Politécnica de Madrid, TR CLIP 8/93.1(94), March 1994.

[21] Pedro López Garcia, Manuel Hermenegildo, and S. K. Debray. Towards granularity based control of parallelism in logic programs. In *International symposium on parallel computation*, Linz, Austria, September 1994.

[22] C. GEYER, A. YAMIN, and O. WERNER. O projeto opera: Um modelo e/ou para prolog. In *Anais do Seminário Integrado de Software e Hardware da SBC*, Rio de Janeiro, 1992. SBC.

[23] Gopal Gupta and Bharat Jayaraman. Analysis of Or-parallel execution models. *ACM Transactions on Programming Languages and Systems*, 15(4):659–680, September 1993.

[24] Manuel Hermenegildo. An Abstract Machine for Restricted And-Parallel Execution of Logic Programs. In Ehud Shapiro, editor, *ICLP86*, pages 25–39. Springer-Verlag, 1986.

[25] I. C. Dutra and V. Santos Costa and J. L. V. Barbosa and C. F. R. Geyer. Using Compile-Time Granularity Information to Support Dynamic Work Distribution in Parallel Logic Programming Systems. Internal Project Report, June 1998.

[26] L. V. Kalé, D. A. Padua, and D. C. Sehr. OR-Parallel execution of Prolog with Side Effects. *The Journal of Supercomputing*, 1988.

[27] R. Karlsson. *A High Performance OR-parallel Prolog System.* PhD thesis, The Royal Institute of Technology, Stockholm, March 1992.

[28] R. Kowalski. *Logic for Problem Solving.* Elsevier North-Holland, 1979.

[29] Robert Kowalski. Algorithm = logic + control. *Commun. ACM*, 22(7):424–436, 1979.

[30] Nai-Wei Lin and Saumya K. Debray. Cost Analysis of Logic Programs. Technical Report, Department of Computer Science, The University of Arizona, August 1992.

[31] Y-J. Lin and V. Kumar. An Execution Model for Exploiting And-Parallelism in Logic Programs. *New Generation Computing*, 5(4):393–425, 1988.

[32] Ewing Lusk, David H. D. Warren, Seif Haridi, et al. The Aurora Or-parallel Prolog System. *New Generation Computing*, 7(2,3):243–271, 1990.

[33] M. R. Pereira. Paralelização de algoritmos de consistência de arcos em um cluster de pc's. Dissertação de mestrado, COPPE - Engenharia de Sistemas e Computação - Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, Brasil, Agosto 2001.

[34] Douglas R. Hofstadter. *Gödel, Escher, Bach: an eternal golden braid.* Harmondsworth, Penguin, 1980.

[35] Ricardo Rocha, Fernando M. A. Silva, and Vitor Santos Costa. Yapor: an or-parallel prolog system based on environment copying. In *Portuguese Conference on Artificial Intelligence*, pages 178–192, 1999.

[36] Luís Fernando Castro, Vítor Santos Costa, Cláudio F. R. Geyer, Fernando Silva, Patrícia Kayser Vargas, and Manuel E. Correia. Daos — scalable and-or parallelism. In *Europar'99*, Toulose, France, September 1999.

[37] Kish Shen, Vítor Santos Costa, and Andy King. A New Metric for Controlling Granularity for Parallel Execution. In *Joint International Conference and Symposium on Logic Programming*, Manchester, UK, June 1998.

[38] Raéd Yousef Sindaha. *Branch-Level Scheduling in Aurora: The Dharma Scheduler.* PhD thesis, University of Bristol, Department of Computer Science, In preparation, 1993.

[39] Evan Tick. Compile time granularity analysis for parallel logic programming systems. In *Proceedings of the 1988 International Conference Fifth Generation Computer Systems*, ICOT, 1988.

[40] Thobias Salazar Trevisan and Jorge L. V. Barbosa. Orca 2.0 or complexity analyzer. Tese de graduação, Universidade Católica de Pelotas, UCPEL, Dezembro 1998.

[41] Patrícia Kayser Vargas, Jorge L. V. Barbosa, Débora Nice Ferrai, Cláudio F. R. Geyer, and Jacques Chassin. Distributed OR scheduling with granularity information. In *12th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2000)*, pages 253– 260, São Pedro - SP, October 24-27 2000.

[42] David H. D. Warren. *Applied Logic—Its Use and Implementation as a Programming Tool.* PhD thesis, Edinburgh University, 1977. Available as Technical Note 290, SRI International.