GRANLOG: Um Modelo para Análise Automática de Granulosidade na Programação em Lógica

Jorge Luis Victória Barbosa [1]
Cláudio Fernando Resin Geyer [2]

Universidade Federal do Rio Grande do Sul -UFRGS
Instituto de Informática
Pós-Graduação em Ciência da Computação - CPGCC
Caixa Postal 15064 - CEP 91591-970
Porto Alegre, RS, Brasil

RESUMO

Este texto descreve um estudo sobre análise automática de granulosidade na programação em lógica. A análise de granulosidade determina o tamanho dos grãos, ou seja, a complexidade dos módulos que serão executados seqüencialmente num único processador. A análise de granulosidade é vital para exploração eficiente do paralelismo. Inicialmente, são apresentados os princípios básicos que motivam e orientam o desenvolvimento deste trabalho. Logo após, é apresentado um modelo para geração de informações de granulosidade na programação em lógica, denominado GRANLOG. Finalmente, são descritas duas aplicações para o modelo proposto, ou seja, auxílio a decisões de escalonamento e simulação da execução de programas.

ABSTRACT

This text describes a study on automatic granularity analysis considering logic programming. The granularity analysis determines the grains size, namely, complexity of modules that will be sequentially executed in a single processor. The grains size determination is vital for efficient exploitation of paralelism. First of all, the basic principles which motivate and orientate this work are presented. After that, it is shown a model to the generation of granularity information in logic programming, called GRANLOG. Finally, the text describes two applications for the proposed model, namely, helping in scheduling decisions and program execution simulation.

[1]Professor na Universidade Católica de Pelotas (UCPel/RS), Tecnólogo em Processamento de Dados (UCPel/RS, 1989), Engenheiro Eletricista (UCPel/RS, 1990), Especialista em Engenharia de Software (UCPel/RS, 1992), Mestrando do CPGCC/UFRGS. E-mail: barbosa@atlas.ucpel.tche.br.

[2]Professor na Universidade Federal do Rio Grande do Sul (UFRGS/RS), Engenheiro Mecânico (UFRGS/RS, 1978), Mestre em Ciência da Computação (UFRGS/RS, 1986), Doutor em Informática (Université Joseph Fourier, Grenoble, França). E-mail: geyer@inf.ufrgs.br.

1 INTRODUÇÃO

A exploração do paralelismo implícito existente na programação em lógica é considerada uma alternativa para simplificação da programação de máquinas paralelas e para aumento do desempenho dos programas em lógica. Desta forma, a integração da programação em lógica e sistemas paralelos tornou-se nos últimos anos um centro de atenções da comunidade científica. Na medida em que as pesquisas nesta área apresentam resultados concretos e estimulantes, aumentam os esforços para exploração adequada do paralelismo nos programas em lógica.

Dentre os problemas que devem ser solucionados para exploração eficiente do paralelismo, encontra-se a análise de granulosidade ([KRU88], [MCC89], [SAR89]). A análise de granulosidade determina o tamanho dos grãos, ou seja, a complexidade dos módulos que serão executados seqüencialmente num único processador. Recentemente, a análise de granulosidade na programação em lógica tem recebido atenção especial por parte da comunidade científica ([KIN90], [TIC90], [ZHO92], [DEB93], [DEB94], [GAR94]).

Neste texto é proposto um modelo para análise automática de granulosidade na programação em lógica, denominado GRANLOG (GRanularity ANalyser for LOGic programming). Este modelo determina os grãos existentes num programa em lógica e possibilita a obtenção de informações relacionadas com estes grãos, tais como, complexidade do grão (granulosidade) e custo para transmissão de suas entradas e saídas. O principal objetivo do GRANLOG é a geração de informações de granulosidade, através da exploração do paralelismo existente num programa em lógica. Estas informações poderão ser utilizadas em aplicações tais como auxílio a decisões de escalonamento e simulação da execução de programas.

O texto está organizado da seguinte forma. Na seção 2 são apresentados os princípios básicos da proposta. A seção 3 descreve a organização do GRANLOG. A seção 4 apresenta o módulo de análise global. A seção 5 aborda especificamente o módulo de análise de grãos. A seção 6 descreve o módulo de análise de complexidade. Na seção 7 são apresentadas duas aplicações para o GRANLOG. Na seção 8 estão as conclusões deste trabalho.

2 PRINCÍPIOS BÁSICOS

Esta seção apresenta os princípios básicos que motivam e orientam o desenvolvimento do GRANLOG.

2.1 INDEPENDÊNCIA DO SISTEMA PARALELO

A grande variedade de sistemas paralelos (arquitetura paralela e plataforma para execução) atualmente disponíveis faz com que o desenvolvimento, a portabilidade e a manutenção de programas paralelos tornem-se tarefas difíceis e onerosas. Dentre as diversas características que variam entre sistemas paralelos destacam-se: tipo de memória (compartilhada ou distribuída), número de processadores, tipo de processadores (homogêneos ou heterogêneos), tipo de escalonamento (centralizado ou distribuído), velocidade dos canais de comunicação e existência de processadores de entrada/saída. Estas características influenciam diretamente na forma de exploração do paralelismo.

PRINCÍPIO 1: Independência do sistema paralelo. As informações de granulosidade, obtidas através da análise realizada pelo GRANLOG, são completamente independentes do sistema paralelo onde será executado o programa. Desta forma, obtem-se flexibilidade e portabilidade para estas informações.

2.2 EXPLORAÇÃO AUTOMÁTICA DO PARALELISMO

Existem basicamente duas abordagens para o problema de particionamento de um programa em tarefas a serem executadas paralelamente: detecção do paralelismo pelo programador (paralelismo explícito) e detecção automática do paralelismo (paralelismo implícito). No primeiro método o programador determina através da linguagem de programação quais são as tarefas paralelas. No segundo método as tarefas paralelas são determinadas automaticamente pelo sistema. Ambos os métodos possuem méritos e devem continuar sendo pesquisados. No entanto, na programação em lógica, a exploração automática do paralelismo é estimulada, devido ao paralelismo implícito existente na avaliação das expressões lógicas. Além disso, a programação em lógica permite uma clara distinção entre a semântica e o controle da linguagem, proporcionando uma abordagem distinta entre o que o programa deve resolver, e como serão obtidas as soluções. Estas características possibilitam a programação de computadores paralelos em nível de dificuldade semelhante à da programação seqüencial, e ainda facilitam a migração de programas entre máquinas seqüenciais e paralelas.

PRINCÍPIO 2: Detecção automática do paralelismo (paralelismo implícito). O GRANLOG realiza a exploração automática do paralelismo nos programas em lógica. Desta forma, o programador não participa da paralelização dos programas e a linguagem de programação mantém-se inalterada. Este príncipio permite o aproveitamento de programas em lógica já existentes, além de liberar o programador do encargo de gerenciar explicitamente o paralelismo do problema.

2.3 MÁXIMA EXPLORAÇÃO DO PARALELISMO

Basicamente, pode-se classificar as fontes de paralelismo na programação em lógica como: paralelismo OU e paralelismo E. O paralelismo OU explora a execução paralela das cláusulas componentes de um predicado, aproveitando o paralelismo implícito do não-determinismo inerente à programação em lógica. O paralelismo E explora a execução paralela dos literais componentes de uma cláusula. Neste tipo de paralelismo, deve-se considerar a possibilidade de conflitos de instanciação entre variáveis compartilhadas pelos literais a serem explorados em paralelo. Qualquer modelo que pretenda explorar ao máximo o paralelismo existente na programação em lógica deverá analisar tanto o paralelismo OU quanto o paralelismo E ([KAL87]).

PRINCÍPIO 3: Máxima exploração do paralelismo existente nos programas em lógica. A análise realizada pelo GRANLOG gera informações de granulosidade relacionadas com o paralelismo OU e com o paralelismo E, visando desta forma, a máxima exploração do paralelismo existente nos programas em lógica.

3 ORGANIZAÇÃO DO MODELO

O GRANLOG é composto de três módulos: Analisador Global (AGL), Analisador de Grãos (AGR) e Analisador de Complexidade (AC). A figura 3.1 apresenta a organização básica do modelo.

O módulo AGL realiza uma análise global do programa, visando determinar os modos, os tipos e as medidas de tamanho dos argumentos de cada procedimento (cláusula ou predicado). O modo de um argumento determina a direcionalidade de sua instanciação (*entrada*, *saída* ou *entrada/saída*). O tipo de um argumento determina o padrão de sua instanciação (lista, inteiro, etc). A medida de tamanho de um argumento estabelece qual a medida deve ser utilizada para determinar o tamanho de um argumento. Este tamanho influencia de forma direta na complexidade de um procedimento. O módulo AGL determina ainda as dependências de dados entre os literais de cada cláusula. A análise destas dependências determina a ordem obrigatória para execução dos literais, estabelecendo o fluxo de dados na execução da cláusula. Através de análises, pode-se inferir os modos ([MEL85], [DEB89]), os tipos e medidas ([MIS84], [YAR87]) e as dependências de dados ([CHA85a], [CHA85b], [DEB89]) para um programa em lógica.

O módulo AGR utiliza as informações de dependências geradas pelo AGL para determinar quais são os grãos existentes no corpo de uma cláusula. O módulo AGR realiza ainda a análise de entradas e saídas, ou seja, determina quais são as entradas e as saídas de cada um dos grãos existentes no programa.

O módulo AC avalia a complexidade dos grãos (granulosidade) detectados pelo Analisador de Grãos. O Analisador de Complexidade determina ainda a relação entre o tamanho das entradas e saídas de cada grão (avaliação dos custos de comunicação).

Figura 3.1 - Organização básica do GRANLOG

4 ANALISADOR GLOBAL (AGL)

O AGL infere os modos, os tipos e as medidas de tamanho dos argumentos de cada procedimento de um programa em lógica. O AGL infere ainda as dependências de dados entre os literais de cada cláusula do programa. A subseção 4.1 apresenta a análise de modos, a subseção 4.2 descreve a análise de tipos, a subseção 4.3 apresenta a análise de medidas e a subseção 4.4 descreve a análise de dependências.

4.1 ANÁLISE DE MODOS

O GRANLOG utiliza quatro modos de argumentos, ou seja: entrada, saída, entrada/saída e indefinido. Um argumento é considerado entrada se nunca sofre nenhuma instanciação durante a execução do procedimento, ou seja, o argumento é consumido pelo procedimento. Normalmente, um argumento de entrada está fechado antes da chamada, ou seja, não possui variáveis livres. Um argumento de saída sempre será uma variável livre antes da chamada de um procedimento, ou seja, este argumento será produzido pelo procedimento. Um argumento será entrada/saída se estiver parcialmente instanciado antes da chamada (entrada) e sofrer instanciações adicionais durante a execução do procedimento (saída). Uma parte deste argumento é consumida pelo procedimento e outra é produzida (instanciações adicionais). Finalmente, o modo do argumento pode ser indefinido. Os modos dos argumentos são abreviados pelos caracteres *i* (*input*), *o* (*output*), *io* (*input/output*) e *?* (indefinido). O predicado apresentado na figura 4.1 será utilizado para exemplificar a análise realizada pelo GRANLOG.

```
q(A,B,C)        :- b(B),f(A,B,C).
q([H|L],K,Z)    :- a(L),b(K),c([H|L],Y),d(H,K,Y),e(L,Y,Z).
```

Figura 4.1 - Predicado utilizado para exemplificar o GRANLOG

A análise de modos do predicado apresentado na figura 4.1, poderia resultar nas informações apresentadas na figura 4.2, onde cada argumento é substituído pela indicação do seu modo.

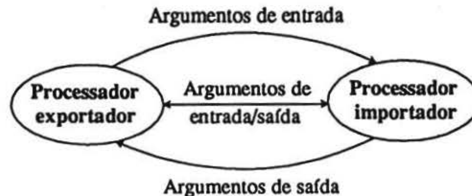
```
q(i,i,o) :- b(i), f(i,i,o).
q(i,i,o) :- a(i), b(i), c(i,o), d(i,i,i),e(i,i,o).
```

Figura 4.2 - Informações geradas pela análise de modos

Do ponto de vista do paralelismo, os modos determinam o fluxo de dados entre os grãos durante a execução paralela de um programa em lógica. Analisando-se os modos pode-se identificar as entradas e as saídas de cada um dos grãos do programa. Se um procedimento for executado em determinado processador, seus argumentos de entrada e entrada/saída deverão estar disponíveis no início da execução. Em arquiteturas de memória distribuída esta exigência implica o envio de mensagens entre processadores. O processador origem (exportador de trabalho) deverá enviar os argumentos de entrada e entrada/saída para o processador destino (importador de trabalho).

Os resultados da execução de um procedimento devem ficar disponíveis no processador que executou a chamada para o procedimento (exportador). No caso de arquiteturas com memória distribuída, esta situação exige o envio de uma mensagem do processador importador para o processador exportador contendo os resultados, ou seja,

os argumentos de saída e de entrada/saída alterados. Portanto, os argumentos dos procedimentos podem ser considerados como canais de comunicação unidirecionais (entrada e saída) ou bidirecionais (entrada/saída) entre os processadores durante a execução paralela de programas em lógica. A figura 4.3 demonstra as possíveis configurações de argumentos como canais de comunicação entre processadores.



4.3 - Argumentos como canais de comunicação

Os modos são utilizados no GRANLOG durante a análise de dependências e para avaliação dos custos de comunicação na execução paralela de tarefas.

4.2 ANÁLISE DE TIPOS

O GRANLOG utiliza os seguintes tipos para os argumentos dos procedimentos de um programa em lógica: inteiro (*int*), lista (*list*), estrutura (*struct*), ponto-flutuante (*float*), átomo (*atom*), variável (*var*), indefinido (*?*) e entrada/saída (*io*). A determinação de tipos possui como principal objetivo permitir a previsão dos custos de comunicação.

Em arquiteturas de memória distribuída a execução de tarefas em paralelo depende da troca de mensagens entre processadores. Estas mensagens representam um dos principais custos da paralelização de um programa. O custo para transmissão de uma mensagem é função do seu tamanho. Portanto, a previsão do tamanho da mensagem permitirá a previsão do custo. Nos programas em lógica, as mensagens entre processadores possuem como principal conteúdo as entradas e saídas dos grãos. Desta forma, conhecendo-se o tipo e o tamanho dos argumentos, pode-se prever o tamanho das mensagens e consequentemente o custo de comunicação envolvido na execução de um grão. Os tipos podem ser inferidos durante a compilação. O tamanho dos argumentos de entrada e saída será obtido durante a execução.

A seguir é apresentada a notação de tipos utilizada no GRANLOG:

- Inteiro: **int**

Indica que o argumento é um inteiro. Não possui parâmetros. Por exemplo, o número 100 será representado simplesmente como *int*.

- Ponto-flutuante: **float**

Indica que o argumento é um número de ponto-flutuante. Não possui parâmetros. Por exemplo, o número 24.4 será representado simplesmente como *float*.

- Átomo: **atom(*TAMANHO*)**

Indica que o argumento é um átomo. O parâmetro *TAMANHO* contém o número de caracteres do identificador do átomo. O símbolo *?* será utilizado no caso de indefinição de *TAMANHO*. Por exemplo, o átomo *mae* seria representado como *atom(3)*.

- Variável: **var**

Indica que o argumento é uma variável livre. Neste caso não são utilizados parâmetros.

- Lista: **list(*TAMANHO,TIPOS*)**

Indica que o argumento é uma lista. O parâmetro *TAMANHO* contém o número de elementos da lista e o parâmetro *TIPOS* é uma lista contendo em ordem o tipo de cada um dos elementos. Se o número de elementos de *TIPOS* for menor que o *TAMANHO*, então o conteúdo de *TIPOS* determina a proporção dos tipos dos elementos da lista representada. O símbolo *?* será utilizado no caso de indefinição dos parâmetros. Por exemplo, a lista [12,abc,124.56] seria representada como *list(3,[int,atom(3),float])*.

- Estrutura: **struct(*FUNCTOR,ELEMENTOS,TIPOS*)**

Indica que o argumento é uma estrutura. O parâmetro *FUNCTOR* contém o tamanho do functor. O parâmetro *ELEMENTOS* determina o número de argumentos da estrutura. O parâmetro *TIPOS* é uma lista contendo o tipo de cada um dos argumentos da estrutura. Se o número de elementos de *TIPOS* for menor do que *ELEMENTOS*, então o conteúdo de *TIPOS* determina a proporção dos tipos dos argumentos da estrutura representada. O símbolo *?* será utilizado em caso de indefinição dos parâmetros. Por exemplo, a estrutura teste(100,12.3) seria representada como *struct(5,2,[int,float])*.

- Indefinido: **?**

Indica que o tipo do argumento não pode ser determinado.

- Entrada/saída: **io(*ENTRADA,SAÍDA*)**

Indica que o argumento possui duas notações de tipos, uma para a entrada e outra para a saída. O tipo *io* é utilizado em argumentos de entrada/saída para indicar o estado de instanciação antes da chamada (entrada) e após a chamada (saída). O parâmetro *ENTRADA* descreve a instanciação antes da chamada. O parâmetro *SAÍDA* indica o estado de instanciação após a chamada. O símbolo *?* será utilizado em caso de indefinição dos parâmetros. Um exemplo deste tipo poderia ser *io(list(3,[var]),list(3,[int]))*.

A análise de tipos do predicado da figura 4.1, poderia resultar nas informações apresentadas na figura 4.4, onde cada argumento foi substituído pela notação que descreve seu tipo.

```
q(list(?,[int]),int,list(?,[int])) :-
                b(int),
                f(list(?,[int]),int,list(?,[int])).
q(list(?,[int]),int,list(?,[int])) :-
                a(list(?,[int])),
                b(int), c(list(?,[int]),list(?,[int])),
                d(int,int,list(?,[int])),
                e(list(?,[int]),list(?,[int]),list(?,[int])).
```

Figura 4.4 - Informações geradas pela análise de tipos

4.3 ANÁLISE DE MEDIDAS

O GRANLOG utiliza as mesmas medidas de tamanho de argumentos apresentadas em [DEB93] e [LIN93]. Estas medidas são as seguintes: valor de um número inteiro (*int*), tamanho de lista (*length*), número de constantes de um termo (*size*), profundidade de uma estrutura (*depth*) e irrelevante para a análise de complexidade (*void*). A determinação de medidas possui como principal objetivo permitir a previsão do tamanho dos grãos (granulosidade) através de uma análise de complexidade.

A granulosidade normalmente depende do tamanho dos argumentos de entrada do grão. Estes argumentos somente serão conhecidos em tempo de execução. No entanto, durante a compilação é possível inferir a medida de tamanho dos argumentos. Conhecendo-se esta medida, pode-se gerar expressões que possibilitem o cálculo da granulosidade em tempo de execução. Estas expressões possuem como incógnitas o tamanho dos argumentos de entrada. Este tamanho depende da medida a ser utilizada e poderá ser determinado em tempo de execução. Portanto, a determinação das medidas de tamanho dos argumentos de entrada é indispensável para a geração das expressões de granulosidade em tempo de compilação e para solução destas expressões durante a execução.

A análise de medidas do predicado da figura 4.1, poderia resultar nas informações apresentadas na figura 4.5, onde cada argumento foi substituído pela notação que descreve sua medida de tamanho.

```
q(length,int,length) :- b(int), f(length,int,length).
q(length,int,length) :- a(length), b(int), c(length,length),
                        d(void,int,void), e(length,length,length).
```

Figura 4.5 - Informações geradas pela análise de medidas

4.4 ANÁLISE DE DEPENDÊNCIAS

A análise de dependências determina as dependências de dados existentes entre os literais de cada cláusula do programa. Estas dependências estabelecem a ordem obrigatória de execução dos literais. A análise de dependências possui como principal objetivo gerar informações de dependências para determinação dos grãos existentes no programa em lógica. Esta determinação de grãos é realizada pelo módulo de análise de grãos.

No GRANLOG é realizada uma análise de dependências para cada cláusula do programa. Desta análise, resulta um grafo que representa as dependências entre os literais da cláusula. A análise é realizada por um algoritmo que considera a organização da cláusula (variáveis e posição dos literais) e os modos dos argumentos (análise de modos). Este algoritmo não será apresentado por falta de espaço. No entanto, pode-se encontrar em [LIN93] um algoritmo semelhante. Além disso, existem propostas ([CHA85a], [CHA85b], [DEB89]) que realizam análises mais completas do que a apresentada em [LIN93] e a utilizada no GRANLOG.

A análise de dependências do predicado da figura 4.1 poderia resultar nos grafos apresentados na figura 4.6, onde cada grafo representa as dependências de uma cláusula. Os nodos **I** (*input*) e **O** (*output*) representam a cabeça da cláusula, demonstrando suas dependências.

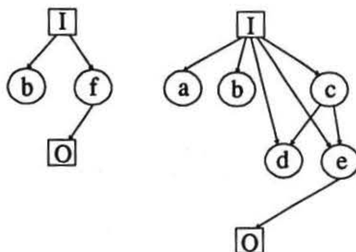


Figura 4.6 - Grafos de dependências gerados pela análise de dependências

5 ANALISADOR DE GRÃOS (AGR)

O módulo AGR utiliza as informações inferidas pelo AGL para gerar a anotação de grãos, onde estão explicitados os grãos existentes no programa acompanhados de suas características. As informações de dependências são utilizadas para determinar os grãos e suas entradas e saídas. Além disso, o AGR determina os tipos e as medidas das entradas e saídas de cada grão. A figura 5.1 apresenta a anotação de grãos gerada pelo AGR para o predicado da figura 4.1

```
:- grain_clause(q/3,g1,[i,i,o],[length,int,length],[list(?,[int]),int,list(?,[int])]).
:- grain_goal(q/3,g1_1,[i],[int],[int]).
:- grain_goal(q/3,g1_2,[i,i,o],[length,int,length],[list(?,[int]),int,list(?,[int])]).
q(A,B,C) :- b(B) & f(A,B,C).

:- grain_clause(q/3,g2,[i,i,o],[length,int,length],[list(?,[int]),int,list(?,[int])]).
:- grain_goal(q/3,g2_1,[i],[length],[list(?,[int])]).
:- grain_goal(q/3,g2_2,[i],[int],[int]).
:- grain_goals(q/3,g2_3,[[H|L],K,Z],[i,i,o],[length,int,length],
          [list(?,[int]),int,list(?,[int])]).
:- grain_goal(q/3,g2_3_1,[i,o],[length],[list(?,[int])]).
:- grain_goal(q/3,g2_3_2,[i,i,i],[void,int,void].[int,int,list(?,[int])]).
:- grain_goal(q/3,g2_3_3,[i,i,o],[length],[list(?,[int])]).
q([H|L],K,Z):-a(L) & b(K) & (c([H|L],Y) , (d(H,K,Y) & e(L,Y,Z))).
```

Figura 5.1 - Anotação de grãos gerada pelo módulo AGR

Na anotação apresentada na figura 5.1, o símbolo & delimita no corpo das cláusulas as tarefas que podem ser executadas em paralelo (paralelismo E independente [HER89]). Visando compatibilizar o GRANLOG com o sistema proposto em [DEB93] e [LIN93], considera-se que os argumentos de saída de um procedimento sempre retornam fechados, ou seja, os procedimentos produzem argumentos de saída completamente instanciados (procedimentos com modos definidos). Esta consideração facilita tanto a análise de complexidade quanto a análise de granulosidade, mas restringe suas aplicações.

As anotações *grain* descrevem as características de cada um dos grãos existentes numa cláusula. Existem três tipos de anotação *grain*, ou seja: *grain_clause*, *grain_goal* e *grain_goals*. A anotação *grain_clause* descreve as características de um grão-cláusula, ou seja, uma cláusula considerada como grão. A anotação *grain_goal* descreve as características de um grão-meta, ou seja, uma meta considerada como grão. A anotação

grain_goals descreve as características de um grão-metas, ou seja, um conjunto de metas consideradas como um único grão.

As anotações *grain_clause* e *grain_goal* possuem os mesmos parâmetros, ou seja, o procedimento ao qual pertencem (no exemplo, *q/3*), o identificador de posição (por exemplo, *g1_1*) e três listas contendo os modos, as medidas e os tipos das entradas e saídas dos grãos. A anotação *grain_goals* possui um parâmetro adicional, contendo o nome das entradas e saídas do grão (terceiro parâmetro do grão-metas *g2_3* na figura 5.1). Este parâmetro é necessário em razão da inexistência destas informações na codificação da cláusula. O módulo Analisador de Grãos analisa os grão-metas e identifica suas entradas e saídas.

O Analisador de Grãos explora o paralelismo entre cláusulas de um predicado (paralelismo OU) e o paralelismo existente no corpo de cada cláusula (paralelismo E). O paralelismo OU é explorado através da anotação *grain_clause* para cada cláusula do predicado (no exemplo, os grãos-cláusula *g1* e *g2*, representam a exploração do paralelismo OU). A exploração do paralelismo existente no corpo de um cláusula é mais difícil e depende da análise de dependências.

A maioria dos sistemas que explora o paralelismo existente no corpo de uma cláusula é baseado na paralelização individual de metas, ou seja, utiliza como grãos apenas metas isoladas. No entanto, a máxima exploração do paralelismo, existente no corpo de uma cláusula (princípio 3), somente poderá ser realizada se os grãos não ficarem limitados a metas isoladas.

Através da análise de dependências pode-se determinar quais são as partes do corpo de uma cláusula que poderão ser executadas em paralelo, definindo-se assim vários níveis de grãos. Desta forma, os grãos não serão necessariamente metas isoladas, mas sim partes da cláusula, que poderão coincidentemente ser uma meta. No exemplo apresentado neste texto, o grão-metas *g2_3* é composto por três metas (*c*, *d* e *e*). Todos os demais grãos da figura 5.1 são grãos-meta, com exceção de *g1* e *g2* (grãos-cláusula). Deve-se ressaltar, que o grão-metas *g2_3* possui internamente outros três grãos (*g2_3_1*, *g2_3_2* e *g2_3_3*) os quais são grãos-meta (*c*, *d* e *e*).

6 ANALISADOR DE COMPLEXIDADE (AC)

Denomina-se granulosidade o tamanho de um grão, ou seja, a complexidade de sua execução. A avaliação da granulosidade pode ser realizada em tempo de compilação, tempo de execução ou em ambos ([SAR89]). Sendo realizada em tempo de compilação, esta avaliação introduz pouco *overhead* na execução. No entanto, a complexidade de um grão normalmente depende do tamanho de suas entradas, o que não pode ser previsto pelo compilador. Portanto, a avaliação em tempo de compilação não permite uma análise apurada da granulosidade. A avaliação em tempo de execução é mais precisa, introduzindo no entanto um considerável *overhead* na execução do programa. Recentes abordagens da avaliação de granulosidade na programação em lógica ([KIN90], [ZHO92], [DEB93], [DEB94], [GAR94]) indicam que a integração da análise em tempo de compilação e execução produz resultados mais promissores.

O módulo AC realiza uma avaliação da complexidade dos grãos identificados pelo Analisador de Grãos. A avaliação de granulosidade gera, para cada grão, uma expressão que permite o cálculo de sua complexidade em função do tamanho de suas entradas. Desta forma, o GRANLOG concilia a análise em tempo de compilação e execução. Em tempo de compilação são geradas expressões que poderão ser solucionadas durante a

execução, de acordo com a exploração do paralelismo. Assim, mantém-se os benefícios introduzidos pela solução em tempo de compilação (baixo *overhead* na execução) e pela solução em tempo de execução (precisão no cálculo da granulosidade). O módulo Analisador de Complexidade possui como base o sistema proposto em [DEB93] e [LIN93].

Além das expressões de granulosidade, o módulo AC gera expressões para determinação do tamanho das saídas de um grão em função do tamanho de suas entradas. Estas expressões possibilitam o cálculo do custo para transmissão dos resultados do grão (saídas). Desta forma, conhecidas as entradas de um grão, pode-se determinar todas as informações necessárias para tratamento do paralelismo, ou seja, complexidade do grão (granulosidade) e custo para transmissão de suas entradas e saídas. Continuando o exemplo apresentado na seção anterior, o módulo Analisador de Complexidade gera a anotação apresentada na figura 6.1.

```
:- granularity(e1,3*$1+5*$2+3).
:- granularity(e2,3*$1+1).
:- granularity(e3,3*$1+2*$2+1).
:- granularity(e4,6*$1+4*$2).
:- granularity(e5,$1).
:- granularity(e6,5*$1+$2-1).
:- granularity(e7,3*$1).
:- granularity(e8,$2).
:- granularity(e9,$1+$2).

:- out_size(r1,[$1,$2,$1]).
:- out_size(r2,[$1,$2,3*$1-2]).
:- out_size(r3,[$1,$1]).
:- out_size(r4,[$1,$2,2*$1+$2]).

:- grain_clause(q/3,g1,[i,i,o],[length,int,length],[list(?,[int]),int,list(?,[int])],e1,r1).
:- grain_goal(q/3,g1_1,[i],[int],[int],e2,_).
:- grain_goal(q/3,g1_2,[i,i,o],[length,int,length],[list(?,[int]),int,list(?,[int])],e3,r1).
q(A,B,C) :- b(B) & f(A,B,C).

:- grain_clause(q/3,g2,[i,i,o],[length,int,length],[list(?,[int]),int,list(?,[int])],e4,r2).
:- grain_goal(q/3,g2_1,[i],[length],[list(?,[int])],e5,_).
:- grain_goal(q/3,g2_2,[i],[int],[int],e2,_).
:- grain_goals(q/3,g2_3,[[H|L],K,Z],[i,i,o],[length,int,length],
                [list(?,[int]),int,list(?,[int])],e6,r2).
:- grain_goal(q/3,g2_3_1,[i,o],[length],[list(?,[int])],e7,r3).
:- grain_goal(q/3,g2_3_2,[i,i,i],[void,int,void],[int,int,list(?,[int])],e8,_).
:- grain_goal(q/3,g2_3_3,[i,i,o],[length],[list(?,[int])],e9,r4).
q([H|L],K,Z):-a(L) & b(K) & (c([H|L],Y) , (d(H,K,Y) & e(L,Y,Z))).
```

Figura 6.1 - Anotação de granulosidade gerada pelo GRANLOG

A anotação *granularity* contém a expressão para o cálculo da complexidade de um grão. Os dois parâmetros desta anotação consistem de um nome para a expressão (*e1*, por exemplo) e da própria expressão (*3*\$1+5*\$2+3*, por exemplo). A anotação *out_size* contém a relação entre os tamanhos das entradas e saídas de um grão. Os dois parâmetros desta anotação consistem de um nome para a relação (*r4*, por exemplo) e da própria relação (*[\$1,\$2,2*\$1+\$2]*, por exemplo).

A notação das expressões utiliza operadores matemáticos (*,/,+,-), funções matemáticas (por exemplo, *fact*, *exp* e *log*) e o símbolo *$* significando o tamanho de determinado argumento do grão (*$1* por exemplo, seria o tamanho do primeiro argumento do grão). Na anotação *out_size*, a relação entre o tamanho dos argumentos de um grão é descrita pela notação do segundo parâmetro. Por exemplo, a notação *[$1,$2,2*$1+$2]* significa que o grão possui três argumentos, onde o tamanho do terceiro argumento (saída) é calculado em função do tamanho do primeiro (*$1*) e do segundo argumento (*$2*) através da expressão *2*$1+$2*.

Conforme constata-se na figura 6.1, foram adicionados pelo módulo AC, dois novos parâmetros nas anotações *grain*. O primeiro novo parâmetro (penúltimo das anotações *grain*) consiste do nome da expressão que calcula a complexidade do grão (por exemplo, *e1*). Este parâmetro liga as anotações *grain* com a anotação *granularity*. O segundo novo parâmetro (último das anotações *grain*) consiste do nome da relação que determina o tamanho das saídas do grão, em função do tamanho de suas entradas (por exemplo, *r1*). Este parâmetro liga as anotações *grain* com a anotação *out_size*.

Algumas anotações *granularity* e *out_size* são utilizadas por mais de um grão, o que torna-se evidente pela utilização da mesma ligação (por exemplo, os grãos *g1_1* e *g2_2* utilizam a mesma expressão de granulosidade, os grãos *g1* e *g1_2* e os grãos *g2* e *g2_3* utilizam a mesma relação de tamanho). Além disso, alguns grãos, não contém saídas e portanto não possuem uma anotação *out_size* (situação simbolizada pela colocação de "_" no último parâmetro das anotações *grain*).

Algumas vezes o módulo Analisador de Complexidade poderá inferir que um grão possui uma complexidade constante. Quando isso acontecer, não será utilizada uma anotação *granularity*, sendo anotado diretamente a constante no penúltimo parâmetro das anotações *grain*.

7 APLICAÇÕES

Nesta seção são apresentadas duas aplicações para as informações geradas pelo GRANLOG. A subseção 7.1 apresenta a aplicação no auxílio a decisões de escalonamento. A subseção 7.2 descreve a aplicação na simulação da execução de programas.

7.1 AUXÍLIO A DECISÕES DE ESCALONAMENTO

Utilizando a anotação fornecida pelo GRANLOG, um compilador paralelizador poderá gerar, além do código objeto, informações de granulosidade que auxiliem o escalonamento das tarefas paralelas durante a execução de um programa. O próprio código objeto poderá ser adaptado para conter as decisões de escalonamento, utilizando como base para tomada de decisões as informações de granulosidade.

As informações geradas pelo GRANLOG estão limitadas pela análise estática, ou seja, não são fornecidas informações que dependam do sistema paralelo a ser utilizado. Esta característica mantém o GRANLOG num alto nível de abstração, permitindo portabilidade e flexibilidade para suas anotações (princípio 1). Por exemplo, o número de processadores e o custo de comunicação dependem da arquitetura paralela a ser utilizada. Portanto, estas informações somente poderão ser fornecidas para o escalonador pelo próprio sistema paralelo. A figura 7.1 apresenta uma possível configuração para aplicação das informações geradas pelo GRANLOG no auxílio a decisões de escalonamento.

Pode-se identificar na figura 7.1 três níveis de abstração. Em alto nível de abstração (nível de anotação) encontra-se o GRANLOG, fornecendo informações de paralelismo completamente independentes do sistema paralelo. No nível médio de abstração (nível de compilação) encontra-se o compilador paralelizador, onde poderão ser consideradas algumas informações específicas do sistema paralelo, tal como o tipo de memória (compartilhada ou distribuída). Desta forma, o compilador poderá complementar as informações de granulosidade geradas pelo GRANLOG, adicionando por exemplo, expressões que permitam o cálculo do custo de comunicação dos grãos, baseado no tamanho de suas entradas e saídas. A compilação deverá ser dirigida para um sistema paralelo específico. No entanto, mantém-se ainda neste nível de abstração alguma flexibilidade, por exemplo, quanto ao número de processadores disponíveis e seus tipos. No baixo nível de abstração (nível de execução) encontra-se a arquitetura paralela e o sistema de escalonamento. Neste nível o conhecimento do sistema paralelo deverá ser completo, ou seja, deverão ser conhecidas informações tais como o número de processadores disponíveis e seus tipos.

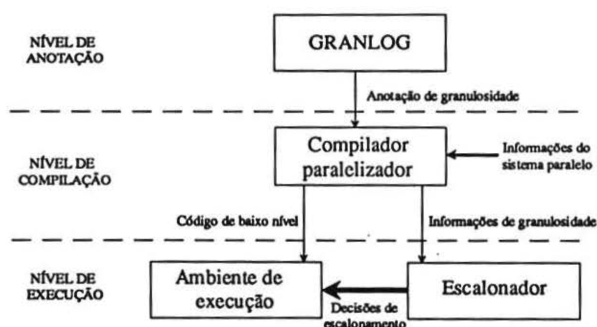


Figura 7.1 - Aplicação do GRANLOG no auxílio ao escalonamento

Atualmente, está sendo desenvolvida a integração OPERA-GRANLOG. O OPERA ([WER94]) é um ambiente para execução paralela de programas em lógica (nível de execução). Este ambiente utilizará as informações fornecidas pelo GRANLOG para aumentar a eficiência na execução dos programas.

7.2 SIMULAÇÃO DA EXECUÇÃO DE PROGRAMAS

Utilizando as informações geradas pelo GRANLOG pode-se simular a execução de programas em lógica. Dentre as possíveis simulações destacam-se:

- **Simulação da complexidade de programas em lógica.** Nesta simulação, pode-se analisar de forma genérica as características de um programa, tais como: complexidade de execução (metas, cláusulas, predicados e programa), tamanho dos argumentos e dinâmica de execução (iteração das recursões);
- **Simulação da paralelização de programas em lógica.** Nesta simulação podem ser previstas as decisões de escalonamento e os custos para execução paralela de um programa. As características do sistema paralelo podem ser alteradas conforme a simulação a ser realizada, permitindo assim a modelagem de várias arquiteturas paralelas e distribuídas.

A figura 7.2 apresenta uma possível configuração para aplicação do GRANLOG na construção de um ambiente para simulação da execução de programas em lógica. Na figura 7.2 o GRANLOG fornece informações de granulosidade para um ambiente, onde será simulada a execução de programas em lógica. Este ambiente poderá fornecer várias opções de simulação e um suporte gráfico para visualização da execução. Além disso, o ambiente proporcionará uma visualização gráfica da dinâmica de execução do programa em lógica, permitindo ao usuário analisar em grafos as alterações da complexidade dos grãos em função das iterações das recursões.

O ambiente possibilitará ao usuário configurar o sistema paralelo a ser utilizado na simulação, determinando o número de processadores e seus tipos. O usuário fornecerá ainda informações do tipo de memória (compartilhada ou distribuída), velocidade dos canais de comunicação e existência de processadores de entrada/saída. Desta forma, pode-se simular a execução de programas em vários sistemas paralelos configurados de diversas maneiras.

Figura 7.2 - Aplicação na simulação de programas em lógica

8 CONCLUSÕES

Este trabalho apresenta um estudo sobre a análise automática de granulosidade na programação em lógica. Foram descritos um modelo para geração de informações de granulosidade de programas em lógica (GRANLOG) e duas de suas possíveis aplicações. Através da descrição destas aplicações, demonstrou-se a importância das informações geradas pelo GRANLOG.

As contribuições deste trabalho são as seguintes. Primeiro, foram estabelecidos os princípios de um modelo para análise de granulosidade, que permitisse a exploração adequada do paralelismo na programação em lógica. Segundo, foi apresentado um modelo para análise automática de granulosidade na programação em lógica. Terceiro, foi descrita a aplicação das informações geradas pelo GRANLOG no auxílio a decisões de escalonamento. Quarto, foi proposta a aplicação do GRANLOG na simulação da execução de programas em lógica.

Futuros trabalhos poderão explorar novas capacidades e aplicações para o modelo proposto. O GRANLOG deverá acompanhar as evoluções do sistema apresentado em [DEB93] e [LIN93]. Uma evolução prevista para este sistema é a análise de procedimentos que possuam múltiplos modos e argumentos de saída abertos, ou seja, argumentos que não tenham necessariamente as variáveis completamente instanciadas. Além disso, podem ser criadas novas anotações, como por exemplo, anotações para representação das dependências de dados entre os literais das cláusulas . Dentre as novas aplicações do GRANLOG a serem pesquisadas, destaca-se a reorganização do código fonte de programas em lógica, visando aprimorar a exploração do paralelismo.

REFERÊNCIAS BIBLIOGRÁFICAS

[CHA85a] CHANG, Jung-Herng; DESPAIN, Alvin M. e DEGROOT, Doug. **AND-Parallelism of Logic Programs Based on A Static Data Dependency Analysis.** Digest of Papers of COMPCON 85, IEEE, February 1985.

[CHA85b] CHANG, Jung-Herng e DESPAIN, Alvin M. **Semi-Intelligent Backtracking of Prolog Based on Static Data Dependency Analysis.** Proceedings of the Symposium on Logic Programming, IEEE, p.10-21, 1985.

[DEB89] DEBRAY, Saumya K. **Static Inference of Modes and Data Dependencies in Logic Programs.** ACM Transactions on Programming Languages and Systems, v.11, n.3, p.418-450, July 1989.

[DEB93] DEBRAY, Saumya K. e LIN, Nai-Wei. **Cost Analysis of Logic Programs.** ACM Transactions on Programming Languages and Systems, v.15, n.5, p.826-875, November 1993.

[DEB94] DEBRAY, Saumya K.; GARCÍA, Pedro L.; HERMENEGILDO, Manuel e LIN, Nai-Wei. **Estimating the Computacional Cost of Logic Programs.** Proceedings of the International Static Analysis Symposium, Namur, Belgium, Springer-Verlag Lecture Notes in Computer Science, v.864, September 1994.

[GAR94] GARCÍA, Pedro L.; HERMENEGILDO, Manuel e DEBRAY, Saumya K. **Towards Granularity Based Control of Parallelism in Logic Programs.** Proceedings of the International Symposium on Parallel Computation, Linz, Austria, September 1994.

[HER89] HERMENEGILDO, M. e ROSSI, F. **On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs.** Proceedings of the 1989 North American Conference on Logic Programs, MIT Press, pp. 369-390, October 1989.

[KAL87] KALÉ, L. V. **Completeness and Full Parallelism of Parallel Logic Programming Schemes.** Proceedings of the Fourth IEEE Symposium on Logic Programming, IEEE, p.125-133, 1987.

[KIN90] KING, A. e SOPER, P. **Granularity Control for Concurrent Logic Programs.** Proceedings of the International Symposium on Computer and Information Sciences, Nevsehir, Turkey, October 1990.

[KRU88] KRUATRACHUE, Boontee e LEWIS, Ted. **Grain Size Determination for Parallel Processing.** IEEE Software, p.23-32, January 1988.

[LIN93] LIN, Nai-Wei. **Automatic Complexity Analysis of Logic Programs.** University of Arizona, 1993. (Ph.D. Thesis).

[MCC89] MCCREARY, Carolyn e GILL, Helen. **Automatic Determination of Grain Size for Efficient Parallel Processing.** Communications of the ACM, v.32, n.9, p.1073-1078, September 1989.

[MEL85] MELLISH, C.S. **Some Global Otimizations for a Prolog Compiler.** Journal of Logic Programming, April 1985.

[MIS84] MISHRA, P. **Toward a Theory of Types in Prolog.** Proceedings of the First IEEE Symposium on Logic Programming, IEEE, p. 289-298, 1984.

[SAR89] SARKAR,V. **Partitioning and Scheduling Parallel Programs for Multiprocessors.** MIT Press, 1989.

[TIC90] TICK, Evan. **Compile-Time Granularity Analysis of Parallel Logic Programming Languages.** New Generation Computing, v.7, n.2-3, p.325-337, 1990.

[WER94] WERNER, Otilia; YAMIN, Adenauer C.; BARBOSA, Jorge L. V. e GEYER, Cláudio F. R. **OPERA Project: an Approach Towards Parallelism Exploitation on Logic Programming.** Proceedings of the Tenth Logic Programming Workshop, Zurich, Institut für Informatik der Univesität Zürich, September 1994.

[YAR87] YARDENI, E. e SHAPIRO, E. **A Type System for Logic Programs.** Concurrent Prolog: Collected Papers, MIT Press, v.2, p.211-244, 1987.

[ZHO92] ZHONG, X.; TICK, E.; DUVVURU, S.; HANSEN, L.; SASTRY A.V.S. e SUNDARARAJAN, R. **Towards an Efficient Compile-Time Granularity Analysis Algorithm.** Proceedings of the Fifth Generation Computer Systems, Tokyo, Japan, p.809-816, 1992.