Uma Proposta de um Escalonador para *Gamma*

Felipe Maia Galvão França[1], Juarez Assumpção Muylaert Filho[2], Gabriel Antoine Louis Paillard[3]
[1] Programa de Engenharia de Sistemas e Computação, Universidade Federal do Rio de Janeiro
Caixa Postal 68511, Cep: 21945-970, Rio de Janeiro, RJ, Brasil
{felipe@cos.ufrj.br}
[2] Instituto Politécnico, Universidade Estadual do Rio de Janeiro
Caixa Postal 97282, Cep: 28610-970, Nova Friburgo, RJ, Brasil
{jamf@iprj.uerj.br}
[3] Centro de Ciências Formais e Tecnologia, Universidade Tiradentes
Av. Murilo Dantas, 300, Farolândia, Cep: 49032-490, Aracaju, SE, Brasil
{paillard@cos.ufrj.br}

Resumo—
Esse artigo apresenta um novo modelo de escalonamento distribuído de tarefas para *Gamma*, onde as tarefas são reações definidas em Gamma. Gamma é um formalismo para programação paralela baseado na reescrita não determinística de multiconjuntos. O novo modelo de escalonamento que nós propomos traz algumas vantagens quando comparado a outras implementações de Gamma, em termos da quantidade de concorrência oferecida pelo controle distribuído. Nosso modelo permite que múltiplas instâncias da mesma reação sobre diferentes elementos do multiconjunto em questão coexistam de uma forma natural.
Palavras-chave— **Gamma, Multiconjunto, Escalonamento**

Abstract—
This paper presents a new jobs distributed scheduling scheme for *Gamma* where jobs are actions defined in Gamma. Gamma is a parallel programming formalism based on the nondeterministic rewriting of multisets. The new scheduling model we propose brings some advantages when compared to anothers implementations of Gamma, in terms of the amount of "concurrency" offered by the distributed control. Our scheme allows that multiple instances of a same reaction over different elements of the target multiset coexist in a natural way.
Keywords— **Gamma, Multiconjunto, Escalonamento**

I. Introdução

Programar paralelamente pode não ser uma tarefa fácil, especialmente se a linguagem de programação a ser utilizada não dispõe de recursos que permitam separar o problema a ser solucionado, da sua implementação propriamente dita. A linguagem Gamma foi criada como um meio de se abstrair apenas sobre o problema a ser solucionado, deixando de lado preceitos pertencentes, por exemplo, ao paradigma imperativo, que podem tornar a tarefa de criar programas paralelos mais árdua. Gamma foi criado em 1986 por Jean-Pierre Banâtre e Daniel Le Métayer, como um formalismo para a especificação de programas, baseado na reescrita paralela de multiconjuntos, facilitando as provas de corretude e derivações de programas [BAN 86].

A principal característica de um programa em Gamma é a interação livre entre os elementos do multiconjunto, que torna o modelo de execução não determinístico, pois não existem restrições e todos os elementos de dados podem reagir livremente, levando a linguagem a ser naturalmente paralela. O modelo Gamma, especifica uma espécie de comportamento caótico, pois a proposta do formalismo Gamma, é justamente possuir o menor número de restrições. No entanto, no momento da implementação de um compilador para Gamma, podemos nos deparar com uma série de dificuldades, no que diz respeito a fornecer todas as características propostas no formalismo Gamma, para a linguagem resultante.

O trabalho aqui proposto, consiste na apresentação de um modelo de escalonamento para Gamma. Nossa motivação principal, veio do fato de termos constatado que nas implementações presentes de Gamma, os escalonadores são modelos simples que não abordam interações entre várias reações. Esse problema que abordamos, pode trazer uma melhoria na eficiência da execução de programas em Gamma, já que levamos em conta a existência de reações n-árias, estas podendo estar presentes em qualquer número, atuando sobre o mesmo multiconjunto.

Trataremos dos três problemas mais relevantes nesse aspecto que são: selecionar os elementos do multiconjunto que serão testados nas devidas condições de reações; como distribuir esses testes entre os elementos de processamento e por fim, como distribuir os processos no sistema que executarão os programas em Gamma.

O paradigma de reescrita de multiconjuntos, incluindo exemplos, será abordado na próxima seção. Na seção III apresentaremos duas implementações de Gamma. A proposta do escalonador para Gamma e todos seus aspectos será mostrada na seção IV. A seção V apresentará nossas conclusões e trabalhos futuros.

II. Gamma

A computação seqüencial foi usada como base no projeto da maioria das linguagens de programação num passado

recente [BAN 96]. Isto aconteceu por duas razões principais:

- Modelos seqüencias de execução forneciam uma boa abstração dos algoritmos, definindo um programa como uma receita para obter o resultado desejado;
- As implementações eram realizadas em máquinas contendo apenas um processador.

Entretanto as limitações da computação seqüencial se tornaram cada vez mais óbvias, pois o tamanho e a complexidade dos softwares cresceram de maneira considerável, além do grande progresso na área de hardware. No entanto, o que se deseja é um desempenho cada vez maior das aplicações, que pode ser obtido através da computação paralela ou distribuída.

A seqüencialidade não pode mais ser considerada como o paradigma no desenvolvimento de programas, mas como uma das possíveis formas de cooperação entre entidades individuais. O formalismo Gamma foi proposto há quinze anos para descrever a computação como a evolução global de valores atômicos interagindo livremente [BAN 86]. Gamma pode ser introduzido através da metáfora da reação química, onde o único modelo de dados é o multiconjunto, que pode ser visto como uma solução química, no qual os itens de dados são os reagentes desta solução.

Um programa simples consiste de um multiconjunto com uma reação básica, que pode ser comparada a um procedimento ou função, que aceita como argumentos os elementos do multiconjunto:

$v_1,v_2,..,v_n \rightarrow$ *Ação* $\Leftarrow$ *Condição de Reação*

onde $v_1,v_2,..,v_n$ são associados a elementos do multiconjunto, a condição de reação é uma expressao booleana, possivelmente abrangendo $v_1,v_2,..,v_n$, a qual sendo verdadeira possibilita que a *Ação* seja executada, especificando os possíveis valores que serão acrescidos ao multiconjunto.

A execução transcorre substituindo elementos no multiconjunto satisfazendo a condição de reação pelo produto da ação. O fim do programa é atingido quando um estado estável é alcançado, ou seja, quando nenhuma reação pode ser realizada.

Logo abaixo mostramos um exemplo de um programa Gamma, usando o operador Γ (mecanismo de execução de programas escritos em Gamma), que retorna o maior elemento de um multiconjunto não vazio de números inteiros:

$$MaxConj(S) = \Gamma((R,A))(S) \text{ onde}$$
$$R(x,y) = x \leq y$$
$$A(x,y) = y$$

A função R especifica a propriedade que deverá ser satisfeita pelos dois elementos selecionados (anteriormente definida como condição de reação). Esses elementos serão substituídos no multiconjunto pelo resultado da aplicação da função A (anteriormente definida como ação).

Então, constatamos que temos um algoritmo abstrato que estabelece a seguinte meta:

"Enquanto existir pelo menos dois elementos no multiconjunto, selecione os mesmos, compare-os e remova o que possuir o menor valor."

A possibilidade de se livrar da seqüencialidade em Gamma traz duas importantes conseqüências [BAN 96]:

- A primeira é o fato de dar um alto nível para a linguagem permitindo ao programador descrever programas de uma maneira abstrata. Isso torna Gamma desejável como linguagem intermediária no processo de derivação de programas. Programas em Gamma são mais fáceis de ter sua corretude provada em relação à determinada especificação e podem ser refinados, dando origem a programas mais eficientes;
- Como Gamma não usa seqüencialidade como paradigma, e também pelo fato de poder selecionar qualquer elemento do multiconjunto, a linguagem leva naturalmente, à construção de programas paralelos.

Voltando ao exemplo anterior, o máximo de um conjunto pode ser obtido através de comparações em qualquer ordem. Nada é dito sobre a ordem da avaliação das comparações. Se alguns pares de elementos disjuntos satisfazem à propriedade R, as comparações e as substituições podem ser realizadas em paralelo.

Vamos agora dar uma apresentação formal de Gamma. O modelo de dados em Gamma (*General Abstract Model for Multiset mAnipulation*) é o multiconjunto, semelhante a um conjunto, exceto que ele pode conter múltiplas ocorrências do mesmo elemento. A vantagem de usar um multiconjunto é a possibilidade de descrever dados simples ou compostos sem nenhuma forma de hierarquia entre os componentes (visão topológica do mesmo). Isto não é o caso para estruturas de dados definidas recursivamente, como listas, que impõem um ordenamento no exame de seus elementos. A estrutura de controle associada aos multiconjuntos é o operador Γ, sua definição formal pode ser vista da seguinte forma:

$\Gamma((R_1,A_1),..,(R_m,A_m))(M) =$
if $\forall i \in [1,m], \forall x_1,..,x_n \in M, \neg R_i(x_1,..,x_n)$
then M
else seja $x_1,..,x_n \in M$ *tal que* $R_i(x_1,..,x_n)$*, então:*
$\Gamma((R_1,A_1),..,(R_m,A_m))\ ((M-(x_1,..,x_n)) + A_i(x_1,..,x_n))$

(R_i,A_i) são pares de funções sem variáveis livres, cujas definições portanto, não envolvem variáveis globais. O efeito de uma reação (R_i,A_i) sobre um multiconjunto M é substituir em M um subconjunto de elementos $(x_1,..,x_n)$, tal que $R_i(x_1,..,x_n)$ seja verdadeiro, pelos elementos de $A_i(x_1,..,x_n)$. Se nenhum elemento de M satisfaz a qualquer reação ($\forall i \in [1,m], \forall x_1,..,x_n \in M, \neg R_i(x_1,..,x_n)$), então o resultado é o próprio M. Caso contrário, o resultado é obtido realizando uma reação $((M-(x_1,..,x_n)) + A_i(x_1,..,x_n))$ e repetindo o mesmo processo até que não possa mais ocorrer nenhuma reação. A definição acima implica em que se uma ou mais condições de reações podem ser verificadas para alguns subconjuntos ao mesmo tempo, a escolha que é feita

entre elas não é determinística, inclusive podendo ser realizadas em paralelo. A importância da propriedade de localidade não pode ser subestimada, se a condição de reação é satisfeita por alguns subconjuntos disjuntos, as reações podem ser realizadas independentemente e simultaneamente. Essa propriedade é a razão básica pela qual os programas em Gamma exibem um paralelismo em potencial [PAI 99].

Mas vale ressaltar, que para testar as condições de reações sobre os elementos do multiconjunto, algumas regras devem ser seguidas no momento da implementação de um compilador para Gamma, visando sua eficácia. Podemos exemplificar, através da exclusão mútua entre variáveis, pois se não houver esse controle, uma determinada ação de uma reação pode modificar uma variável, enquanto outra reação está utilizando a mesma, provocando assim uma condição de corrida.

No nosso modelo de escalonamento, o próprio teste das condições de reação sobre os elementos do multiconjunto é realizado de forma paralela, proporcionando assim, uma possível melhora na eficiência, tanto na compilação como na execução de programas em Gamma.

Outros aspectos interessantes na implementação de Gamma podem ser constatados nas operações básicas sobre os multiconjuntos, entre elas podemos destacar a seleção de um elemento de maneira não determinística do multiconjunto, mas esta operação é apenas uma metáfora usada para estar de acordo com o formalismo proposto e não deve ser visto como um modelo de sua implementação real. Outro fato, consiste na associação das variáveis das reações com aquelas do multiconjunto, para isso é preciso implementar um algoritmo como o UMG (Unificador Mais Geral), utilizando um conjunto de discórdia [CAS 87, PAI 99].

A. Construção de programas em Gamma

O modelo Gamma de computação traz uma abordagem diferente para projetar programas. Um programa não consiste mais em uma seqüência de instruções modificando um estado, ou em uma função aplicada aos seus argumentos, mas em um transformador de multiconjuntos operando seus dados de uma só vez. O desenvolvimento de programas em Gamma implica na escolha de uma representação de dados e na escolha do tipo de transformação aplicada aos dados [BAN 93].

O único modelo de dados fornecido por Gamma é o multiconjunto. Elementos do multiconjunto podem ser valores simples ou compostos, mas não existe estruturas de dados recursivas em Gamma. Então, a primeira etapa a ser realizada quando projetamos um programa Gamma, é achar uma representação de dados como um multiconjunto de itens. Se o programa tem que operar sobre valores básicos, tais como inteiros, uma decomposição desejável dos valores tem que ser achada. Dados complexos como seqüências, árvores ou grafos podem ser representados por multiconjuntos de uma maneira simples. Por exemplo, os componentes de um multiconjunto representando uma árvore são os nós e níveis da árvore associados a informações; os componentes de uma seqüência, são pares (índice, valor). É uma propriedade de Gamma ver toda a estrutura de dados por inteira Todos os componentes da estrutura de dados são diretamente acessíveis, independentemente da sua posição na estrutura. Se a posição na estrutura é relevante para a reação, ela tem que ser expressa através da condição de reação *R(x,y) = leftson(x,y)*, por exemplo, num programa que manipula árvores. Podemos dizer que Gamma tem uma visão topológica dos tipos de dados. Isso contrasta com as tradicionais visões recursivas dos tipos de dados, onde temos que percorrer a estrutura para acessar a um componente particular [BAN93]. Será mostrado agora um exemplo onde o objetivo é produzir os números primos menores ou iguais que um dado número natural em Gamma:

$$Prime_numbers(N) = \Gamma((R,A))\ (\{2,..,N\})\ where$$
$$R(x,y) = multiple(x,y)$$
$$A(x,y) = y$$

Comparada com soluções imperativas ou funcionais, o programa em Gamma é bem mais conciso e fácil de entender [PAI 99]. A razão é que muitos detalhes computacionais não são especificados. Uma solução imperativa seria mais complexa, porque ela daria uma descrição precisa da ordem de execução (expressa, por exemplo, através do uso do operador " ; ") e do gerenciamento de memória, através do uso de uma coleção de variáveis. A solução funcional seria também de baixo nível, pois teríamos que precisar a construção e decomposição da lista, por exemplo, de forma explícita. O programa Gamma captura de uma forma natural, a estratégia básica para solucionar o problema, que consiste na eliminação de elementos múltiplos do multiconjunto *{2,..,N}*.

Como conseqüência da ausência de compromisso com uma ordem particular de execução, um programa em Gamma pode ser implementado naturalmente de forma paralela.

III. Implementações de Gamma

Descreveremos agora um exemplo de uma implementação SIMD do modelo Gamma, que foi realizada por Christian Creveuil, e que considerou apenas condições de reações binárias [CRE 92].

A avaliação de um programa em Gamma envolve dois tipos de tarefas:

1. A busca de elementos satisfazendo a condição de reação;
2. A aplicação da ação nesses elementos.

Aplicação da ação, como dito anteriormente, é uma operação local, podendo ser realizada independentemente do restante do multiconjunto. Logo, o problema principal foi

projetar um algoritmo apta xaminar todos os pares de elementos.

Foi considerado que o número de processadores era igual ao úmero ed lementos no multiconjunto. Sejam $\{x_1,..,x_n\}$ os elementos do multiconjunto, onde o elemento k está armazenado no k-ésimo processador. O algoritmo usado nesta implementação é baseado no *odd-even transposition sort* [KNU 72], e está descrito logo abaixo:

- Nos passos ímpares, os elementos x_k e x_{k+1}, onde k é ímpar, são examinados, então a permuta de dados é realizada para cada par;
- Nos passos pares, os elementos x_k e $x_{k+1,}$ onde k é par, são examinados e novamente a troca de dados acontece.

Portanto, ele leva n passos para realizar todas as trocas necessárias, tendo complexidade *O(N)*, e assim sendo, verificar todos os elementos do multiconjunto. Um exemplo com *n* igual a seis é dado logo abaixo:

1	x_1	↔	x_2		x_3	↔	x_4		x_5	↔	x_6
2	x_2		x_1	↔	x_4		x_3	↔	x_6		x_5
3	x_3	↔	x_4		x_1	↔	x_6		x_3	↔	x_5
4	x_4		x_2	↔	x_6		x_1	↔	x_5		x_3
5	x_4	↔	x_6		x_2	↔	x_5		x_1	↔	x_3
6	x_5		x_4	↔	x_5		x_2	↔	x_3		x_1
	x_6		x_5		x_4		x_3		x_2		x_1

Essa implementação foi realizada numa Connection Machine. A condição de terminação espera que *n* passos sem nenhuma reação ocorram, para ser aplicada. Podemos observar o fato dessa implementação ter sido realizada apenas para testar reações binárias, as quais sempre produzem nessa implementação, dois elementos para cada ação. A implementação não abordou programas em Gamma que possuam mais de uma reação. É nesse aspecto, que nosso modelo de escalonamento se distingue dos demais, pois nele podemos considerar programas com várias reações.

Uma segunda implementação, também SIMD, realizada por Juarez Muylaert e Simon Gay para um sistema distribuído [PAI 99], considerou reações n-árias.

A fase onde são testadas as condições de reações, foi feita enviando uma cópia do multiconjunto para cada elemento de processamento, no caso uma reação, essa então realiza os testes necessários sobre a condição de reação juntamente com os elementos do multiconjunto. Esse esquema pode ser visualizado na Figura 1.

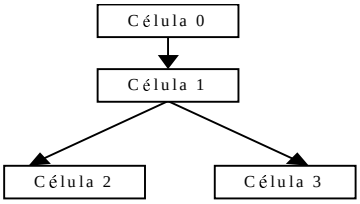

Fig.1 Esquema de escalonamento.

A Célula 0 envia para as células que executarão uma reação, uma cópia do multiconjunto, a Célula 1 controla a execução dos demais processos, inclusive o término do programa que ocorre quando nenhuma reação de condição é verificada entre as reações. O esquema de escalonamento adotado foi bem simples, distribuir para cada elemento de processamento, uma cópia do multiconjunto, mas sempre tendo que devolver essas cópias para a Célula 0, e requisitando novamente, se necessário uma nova cópia do multiconjunto. Essa foi a maneira encontrada, para que os valores do multiconjunto não fossem acessados simultaneamente por reações diferentes, ou seja, para garantir a exclusão mútua das variáveis do multiconjunto.

Nessa implementação, também outras estratégias de controle foram adotadas, que fogem um pouco do formalismo Gamma, como explicitar quando deve ocorrer paralelismo entre reações (expresso através do comando " | ") e quando uma reação deve esperar pelo término de uma ou mais reações (expresso através do comando " ; "). Cada comando desse é atribuído a uma célula, que controlará as reações ligadas a mesma. Supondo que um determinado programa executasse P1 | P2; P3, ou seja, primeiro seriam executados em paralelo, as reações P1 e P2, para somente depois ser iniciada a execução de P3, teríamos o esquema mostrado na Figura 2, onde a célula 4 e 5 representam as estratégias de controle mencionadas.

Essa implementação foi realizada para um sistema distribuído, utilizando uma interface de passagem de mensagens (MPI) [GDB 96].

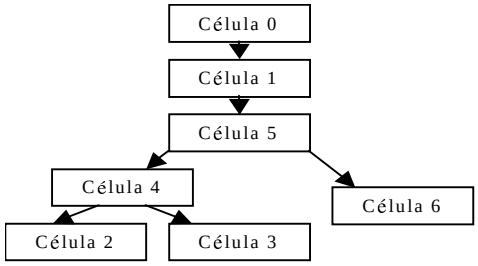

Fig.2 Células de Gamma.

Fica constatado, que na prática ao se realizar uma implementação de Gamma, algumas restrições de controle têm que ser acrescentadas no compilador. Podemos citar um exemplo, onde uma linguagem de escalonamento para Gamma foi criada, justamente para evitar a descaracterização dos princípios formais de Gamma, ou seja, nessa proposta de escalonador, existiam duas linguagens, uma para definir o programa Gamma, e outra para especificar o escalonamento [CHA 96]. Essa proposta difere da nossa, pois fazemos a distribuição dos processos em tempo de compilação e execução, tornando o processo de escalonamento mais seguro, e deixando a linguagem mais transparente para o usuário, e também podemos citar novamente, o fato do nosso modelo trabalhar com programas em Gamma, que

podem ter mais de uma reação. Outros esquemas foram verificados, com outras arquiteturas, como MIMD [BAN 88, KUC 93].

IV. UM MODELO DE ESCALONAMENTO PARA GAMMA

Desenvolvemos um modelo de escalonamento para a linguagem Gamma, que tem como principal função tentar otimizar a distribuição dos processos entre os diversos elementos de computação, para tornar a execução de programas mais eficientes, em relação aos modelos de escalonamento existentes, possivelmente reduzindo seus tempos de execução.

O escalonador beneficiará principalmente programas em Gamma que tiverem mais de uma reação. Também é analisado para cada reação, todos os subconjuntos do multiconjunto que produzem uma ação, ou seja todas as combinações são testadas.

A sua execução se dará em tempo de compilação e execução, pois novos elementos no multiconjunto podem ser adicionados ou retirados durante a execução, e no caso da adição de elementos, possivelmente teremos que criar novos processos, mas isso pode ser realizado com bibliotecas de passagem de mensagens como MPI ou PVM (no caso de uma implementação distribuída), que permitem a criação de processos em tempo de execução.

O algoritmo que realiza o escalonamento foi baseado no mapeamento ótimo de sistemas com restrições sobre a vizinhança [FRA 95]. Esse mapeamento é um algoritmo baseado na técnica *SER(Scheduling by edge reversal)* [BAR 93] e *MCC(Minimum Clique Covering)*, onde recursos compartilhados são representados por um grafo orientado finito. *SER* é inicializado a partir de uma orientação acíclica sobre o grafo G, isso significa que teremos pelo menos um nó *sink*[1]. A cada passo do algoritmo as arestas são revertidas, assim sendo teremos novos *sink's* a cada passo. Vale ressaltar, que achar uma orientação inicial acíclica que maximize a concorrência sobre um grafo genêrico, pertence à classe de problemas NP-completos [BAR 89]. Então, dado um grafo G e seu complemento G', para cada orientação acíclica sobre G', existe um conjunto independente de *sink's*, o que necessariamente equivale a um *clique*[2] em G.

Como o número de tais orientações acíclicas é finito, eventualmente um conjunto de orientações se repetirá. Assim teremos um período de tamanho *p* de orientações. É provado que dentro de um período cada nó opera exatamente *m* vezes. Esta simples dinâmica garante que não ocorrerá *deadlock* ou *starvation*, pois cada orientação acíclica possui pelo menos um *sink*. A concorrência de um período é definida pelo coeficiente *m/p*. Fica claro, que quanto maior esse coeficiente, menos vazio ficará cada nó (processadores) durante o período. Também fica claro que em qualquer sistema conectado $1/n \leq m/p \leq 1/2$, $n = |N|$ [BAR 93]. Isso implica que se tivermos um sistema onde, alocamos um processador por processo, pelo menos a metade deles estará vazia em qualquer momento. Para otimizarmos o uso dos processadores, alocamos para cada, um *clique*, pois sabemos que os vértices pertencentes a um *clique*, não podem operar simultaneamente, pois dividem recursos compartilhados [FRA 95].

O programa que usaremos para ilustrar os passos do algoritmo é (utilizando a sintaxe adotada na implementação do Juarez Muylaert e Simon Gay) [PAI 99]:

(P1 | P2); P3 {[2,1],[5,1],[3,1],[5,1],[7,1],[2,1],[5,2], [8,2], [9,2], [6,2]}
where
P1 = replace [x,1],[y,1] by [x,1] if x≥ y
P2 = replace [x,2],[y,2] by [x,2] if x ≥ y
P3 = replace [z,1],[u,2] by z + u if true

O programa acima tem no seu multiconjunto uma série de tuplas, que representam valores com seus respectivos índices. O objetivo do programa é achar o maior elemento de cada índice e retornar a soma dos mesmos. Podemos notar que foi utilizado a restrição " ; ", a qual cria um ponto de sincronismo entre P1, P2 e P3.

Os passos a serem seguidos pelo escalonador são:

1. Primeiro passo:

São criados agentes de reações (ReAgentes), um para cada reação, idealmente em processadores distribuídos, juntamente com uma cópia do multiconjunto, esse primeiro passo está ilustrado na Figura 3;

2. Segundo passo:

Cada gerente de reação gera instâncias candidatas a partir de composições dos elementos do multiconjunto. Os elementos então, se oferecem como candidatos a participar de uma instância de uma reação, no caso, uma restrição de controle aqui pode ser notada, pois cada ReAgente terá que ter acesso a todos os elementos do multiconjunto, para assim gerar todas as instâncias possíveis. No nosso exemplo, cada reação precisa de dois elementos com o mesmo índice, vale também lembrar que multiconjuntos permitem elementos repetidos. Podemos observar como está o grafo após esse passo na Figura 4;

3. Terceiro passo:

Todas as reações enviam para todas as outras reações, quais as instâncias estabelecidas, de modo a permitir a criação das arestas (ainda não orientadas) entre elas. Cada instância se tornará um novo vértice, e as arestas serão acrescidas entre os vértices que possuam elementos em comum, para garantir a exclusão mútua. Podemos ver na Figura 5, uma ilustração desse passo (note que criamos apenas restrições entre os elementos que possuíam os mesmos índices, pelo fato de ser inerente ao problema);

[1] Sink é um vértice do grafo, onde todas as arestas que estão conectadas ao mesmo, são direcionadas a seu favor.

[2] Denomina-se clique de um grafo G a um subgrafo de G que seja completo, ou seja, existe uma aresta entre cada par de vértices distintos [SZW 86].

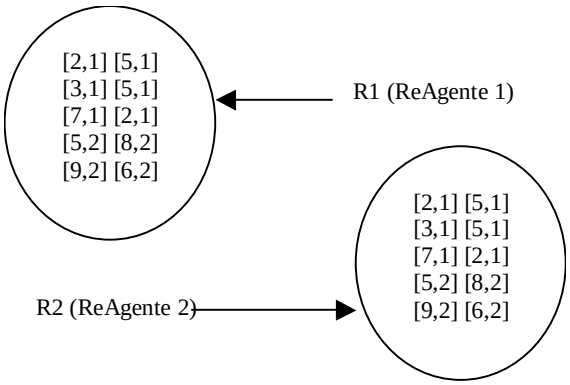

Fig. 3 Criação dos *ReAgentes.*

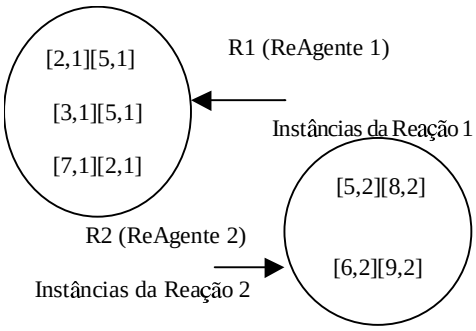

Fig. 4 Geração das instâncias.

4. Quarto passo:

É criado um grafo complementar (G')[3], que será utilizado para distribuição final dos processos. Esse passo, pode ser visto na Figura 6;

5. Quinto passo:

Sorteio de uma orientação acíclica no grafo com-plementar, que transcorre da seguinte forma: cada nó joga um dado (com seis faces) e envia o seu resultado para todos os nós com quem está ligado por arestas; as arestas são orientadas pelos ganhadores do sorteio, caso aconteça algum empate entre um ou mais vizinhos, repete-se o sorteio apenas entre estes, garantindo assim a orientação do grafo que será acíclica. Podemos observar esse passo na Figura 7, onde mostramos os valores dos dados referentes ao sorteio ao lado de cada vértice e na Figura 8, com o grafo G' já orientado;

6. Sexto passo:

Este passo é utilizado para detecção de *sink's* no grafo complementar, que correspondem a um *clique* no grafo original, estes poderão ser alocados em processadores separados (se existir um número suficiente disponível de processadores) para a reação ser realizada de forma exclusiva, garantindo assim, a integridade do multiconjunto. Após a detecção dos *sink's* no grafo complementar, retiramos estes do grafo, e procuramos por novos *sink's*, obtendo assim

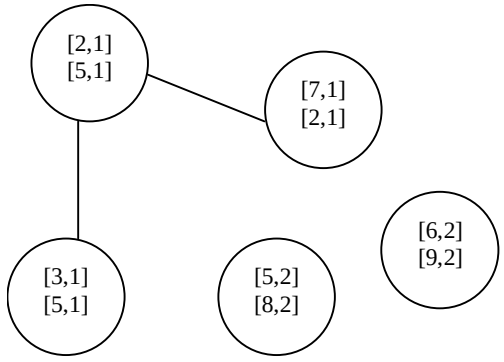

Fig. 5 Geração das restrições entre as instâncias.

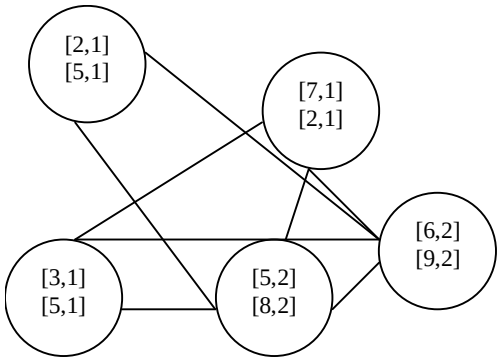

Fig. 6 Grafo complementar.

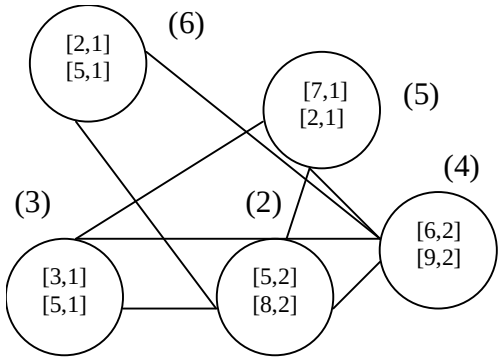

Fig. 7 Respectivos valores dos dados.

novos cliques no grafo original, ou então revertemos às arestas como foi dito anteriormente. Este passo se encerra, quando não mais encontrarmos *sink's* no grafo complementar, ou quando tivermos completado um período *p*. No caso, adotamos o método *SRE*, que a cada período reverte suas arestas, possibilitando que todos os *sink's* possam operar. Este passo está ilustrado nas Figuras 9, 10,11 e 12;

[3] Denomina-se complemento de um grafo G(V,E) ao grafo G', o qual possui o mesmo conjunto de vértices do que G e tal que para todo par de vértices distintos v, w ∈ V, tem-se que (v,w) é aresta de G' se e somente se não o for de G [SZW 86].

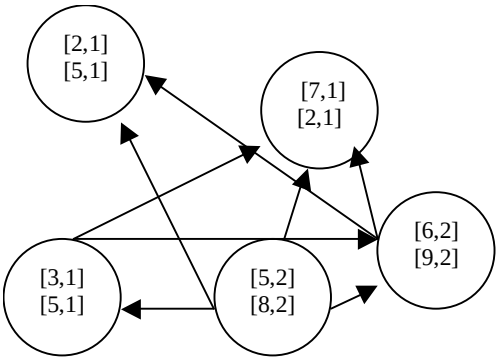

Fig. 8 Grafo complementar orientado.

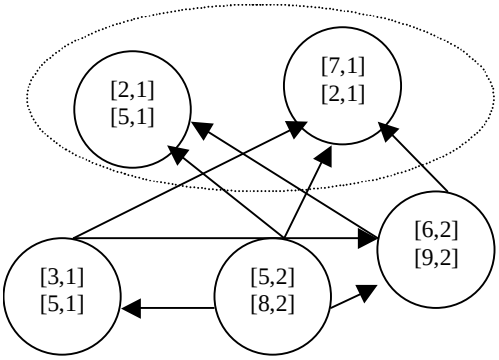

Fig. 9 Detecção dos dois primeiros *sink's*.

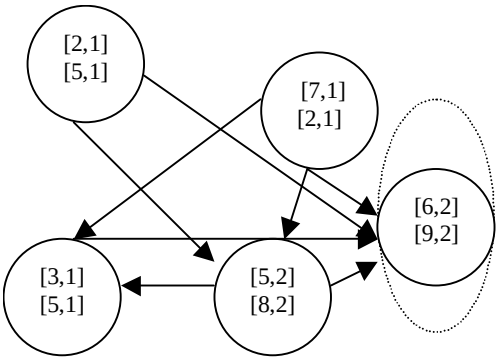

Fig. 10 Reversão de arestas e detecção de um novo *sink*.

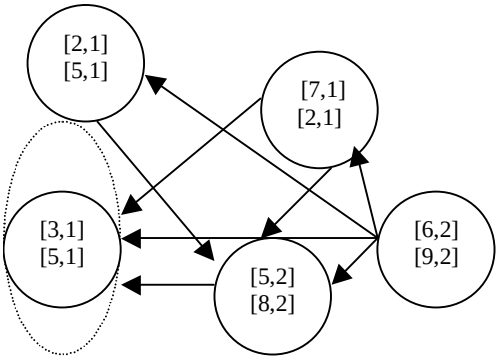

Fig. 11 Reversão de arestas e detecção de um novo *sink*.

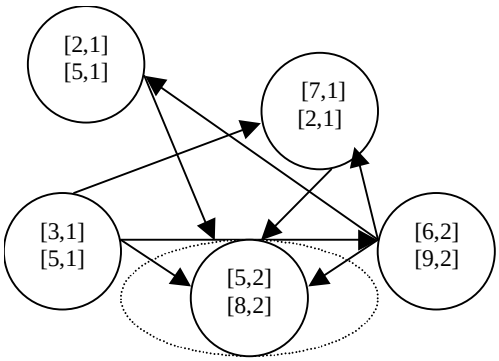

Fig. 12 Reversão de arestas e detecção do último *sink*.

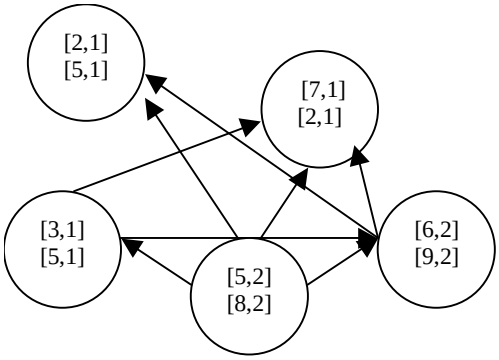

Fig. 13 Início de um novo período.

Podemos observar que o período recomeça na Figura 13, temos então que o período tem $p = 4$, e $m = 1$, ou seja, a cada quatro iterações, teremos a detecção dos mesmos *sink's*. Podemos notar ainda que, cada grupo de *sink's* detectados, equivale a um clique no grafo original, sendo este alocado a um processador. Podemos também, fazer a fase de geração de instâncias em paralelo, seguindo um esquema semelhante ao utilizado na connection machine, só que para reações n-árias [CRE 92].

A. Inclusão e retirada de elementos do Multiconjunto

Este passo consiste na retirada ou inclusão de novos elementos no multiconjunto em paralelo (concorrentemente). Na retirada, as próprias instâncias sinalizam via "falta de restrição de aresta" (elemento do multiconjunto em comum entre duas reações), a exclusão de um elemento do multiconjunto, causando uma modificação no grafo G, que pode ou não gerar um novo grafo complementar.

Na inclusão, a instância de uma determinada reação que, após a execução criou um novo elemento, sinaliza ao respectivo gerente de reação o novo elemento do multiconjunto. Esse gerente de reação pode comunicar o aparecimento desse novo elemento aos outros gerentes de reação, seletivamente, se for possível ter esse conhecimento prévio.

Então, os gerentes de reação estabelecem novas possíveis instâncias e novas arestas ao grafo que restou. Se for possível saber que um novo elemento não reage, ele pode descartar a criação de novas instâncias (arestas e instâncias). Entretanto, se o gerente de reação trata de uma propriedade que pode ser analisada antes da reação, é possível polarizar o dado que define o sorteio da orientação acíclica negativamente, caso seja verificado que determinado elemento do multiconjunto não reage face a determinada reação, isso no grafo complementar, que também precisará ser gerado novamente, ou apenas modificado.

Temos então, que após a compilação do programa, deve-se realizar a cada iteração, possíveis modificações nos grafos (G e G'), podendo ocorrer ou não um acréscimo de elementos no multiconjunto.

V. Conclusões e perspectivas

Apresentamos um novo modelo de escalonamento para linguagens baseadas no formalismo Gamma. Esse modelo tem como objetivo tornar a distribuição de processos mais eficiente. Sua motivação, foi o fato de termos constatado que para as linguagens existentes de Gamma, os modelos de escalonamento não se referem a programas com mais de uma reação.

Algumas outras considerações poderão ser feitas no momento da implementação deste modelo, como que estruturas de dados utilizar para representar o multiconjunto. Se utilizarmos uma árvore, por exemplo, poderemos trazer melhorias de desempenho no momento da busca de elementos no multiconjunto que satisfaçam as condições de reação.

Esse modelo de escalonamento, por ter uma política mais elaborada que os demais modelos apresentados anteriormente, poderá trazer alguns custos adicionais, como comunicação e sincronização. Mas esses custos, poderão ser superados pelos ganhos na eficiência, fornecidos pelo escalonador.

Como extensão do nosso trabalho, pretendemos aplicar esse modelo de escalonamento para Gamma Estruturada, que possui características adicionais, como a possibilidade de inserir restrições de controle, que podem ser muito úteis na verificação dos elementos que devem ser selecionados do multiconjunto. Essas características adicionais, poderão inclusive melhorar a eficiência do escalonador [PAI 01].

Também queremos implementar um compilador em Gamma (paralelo e distribuído), que possa operar sobre diversos multiconjuntos simultaneamente, e também extender para essa nova implementação, o nosso modelo de escalonamento.

Por fim, pretendemos realizar medidas de execução de programas, as quais poderão comprovar a eficiência do modelo de escalonamento aqui proposto.

Agradecimentos

Gostaríamos de agradecer as agências de amparo à pesquisa CAPES e CNPq, que financiaram parcialmente este trabalho.

References

[BAN 86] BANÂTRE, J.-P.; MÉTAYER, D. Le. *A New Computaional Model and its Discipline of Programming*. Rapport de Recherche INRIA, nº 566, septembre 1986.

[BAN 96] BANÂTRE, J.-P.; MÉTAYER, D. Le. *Gamma and the Chemical reaction Model: Ten Years After*. Coordination Programming Mechanisms, Models and Semantics. Imperial College Press, 1996.

[PAI 99] PAILLARD, G. A. L. *Uma Implementação Paralela e Distribuída De Gamma Estruturada*. Tese de Mestrado, PESC/COPPE, Universidade Federal do Rio de Janeiro, Setembro de 1999.

[CAS 87] CASANOVA, M. A.; GIORNO, F. A. C.; FURTADO, A. L. *Programação em Lógica e a Linguagem Prolog*. Capítulo 4, Rio de Janeiro, Brasil, Editora Edgard Blücher Ltda., 1987.

[BAN 93] BANÂTRE, J.-P.; MÉTAYER, D. Le. *Programming by multiset transformations*. Communications of the ACM, Vol. 36-1, pp. 98-111, January 1993.

[CRE 92] CREVEUIL, C. *Implementation of Gamma on the Connection Machine*. Proc. of the Workshop on Research Directions in High-level Parallel Programming Languages. Spring Verlag, LNCS 574, 1992.

[KNU 72] KNUTH, D. E. *The Art of Computer Programming*. Volume 3. Addison Wesley, 1972.

[GDB 96] GDB/RDB. *MPI Primer Developing with LAM*. The Ohio State University, 1996.

[CHA 96] CHAUDRON, M.; JONG, E. De. *In Coordination Programming : Mechanisms, Models and Semantics*. Imperial College Press, 1996.

[BAN 88] BANÂTRE, J.-P.; MÉTAYER, D. Le. *A parallel machine for multiset transformation and its programming style*. Future Generation Computer Systems, 4, pp. 133-144, 1988.

[KUC 93] KUCHEN, T.; GLADITZ K. *Parallel Implementatiopn of bags*. Proceedings FPCA'93, ACM Press, pp. 299-307, 1993.

[FRA 95] FRANÇA, F. M. G.; FARIA, L. *Optimal Mapping of Neighbourhood-Constrained Systems*. Lecture Notes in Computer Science, 980, 1995.

[BAR 93] BARBOSA, V. C. *Massively Parallel Models of Computation*. Ellis Horwood, Chichester, UK, 1993.

[BAR 89] BARBOSA, V. C.; GAFNI, E. *Concurrency in heavily loaded neighborhood-constrained systems*. ACM Tr. on Prog. Lang. and Sys., pp. 562584, vol. 11, nº. 4.

[SZW 86] SZWARCFITER, J. L. *Grafos e Algoritmos Computacionais*. 2ª Edição, Editora Campus, Rio de Janeiro, 1986.

[PAI 01] PAILLARD, G. A. L.; FRANÇA, F. M. G.; MUYLAERT FILHO, J. A. *A Distributed Implementation of Structured Gamma*. To appear in the Proc. of ICPADS 2001, KyongJu City, Korea, June 2001.