Ambiente Gráfico para o Desenvolvimento de Aplicações Distribuídas

João F.L. Schramm, Juliano Malacarne, Cláudio Geyer
{schramm, malacarn, geyer}@inf.ufrgs.br

Instituto de Informática - UFRGS
Caixa Postal 15064
91501-970 Porto Alegre - RS
Fone: (051) 316-6802

Resumo

A complexidade natural das aplicações distribuídas em relação ao sistemas isolados é um dos motivos que leva este tipo de aplicação a ser mais difícil de desenvolver, depurar e manter. A fim de facilitar a programação de aplicações paralelas, foi projetada uma ferramenta que visa a liberar o programador da tarefa se preocupar com as funções de gerenciamento e comunicação entre processos. Esta ferramenta é um ambiente de programação visual composto por um editor de grafos e um gerador de código. O editor de grafos tem a função de representar a aplicação distribuída (através de um grafo) numa estrutura acessível posteriormente ao gerador de código. Este gerará o código da aplicação a ser executado nos sistemas HetNOS (*Heterogeneous Network Operating System*) e PVM (*Parallel Virtual Machine*).

Palavras chaves: sistemas distribuídos e paralelismo, interfaces homem-computador, linguagens de programação.

1 Introdução

O desenvolvimento de programas paralelos envolve novas preocupações não existentes em programas seqüenciais. Aplicações paralelas são mais complexas pelo fato de possuírem duas dimensões [NEW93]. O programador se preocupa não somente com o fluxo seqüencial dos elementos do programa paralelo, mas também com a interação entre estes elementos seqüenciais. A visualização dos pontos de sincronismo e da comunicação entre os vários elementos de processamento torna-se essencial para o entendimento do programa. Linguagens convencionais representam implicitamente estes aspectos de sincronismo e comunicação, dificultando o entendimento do funcionamento global do sistema.

Este trabalho propõe um ambiente de programação visual [SCH96] cujo principal objetivo é simplificar a fase de projeto e implementação de programas paralelos em arquiteturas **MIMD** (*Multiple Instruction, Multiple Data*). A ferramenta gerará código executável para os *middlewares* **HetNOS** (*Heterogeneous Network Operating System*) [BAR94] e **PVM** (*Parallel Virtual Machine*). Durante a criação de uma aplicação paralela o programador só se preocupa com a estrutura do programa: dados a serem manipulados, elementos de processamento e suas interações. O nível de abstração oferecido tira do programador a responsabilidade sobre a troca explícita de mensagens e gerência de processos.

A seção 2 descreve as características essenciais em ambientes de programação de aplicações paralelas e expõe os objetivos da utilização de visualização em sistemas paralelos. Na seção 3 o modelo é descrito em detalhes. São mostradas as diversas etapas a serem executadas pelo usuário durante a criação de programas paralelos. A seção 4 expõe alguns aspectos de implementação. Encontra-se uma comparação desse modelo com outros ambientes gráficos para programação paralela na seção 5 e um exemplo na seção 6. Por fim é feita uma conclusão na qual aparece uma análise crítica do sistema e possibilidades de trabalhos futuros.

2 Ambientes de Programação Paralela

Durante o projeto de uma ferramenta para o auxílio ao desenvolvimento de aplicações paralelas, existem diversas questões a serem consideradas: **facilidade de uso**, **flexibilidade** (diz respeito ao conjunto

de aplicações que podem ser desenvolvidas com a ferramenta), **eficiência, facilidade de monitoração e depuração e portabilidade.**

Uma linguagem visual contribui bastante para o aspecto facilidade de uso. Esta linguagem deve possuir elementos visuais para a representação das unidades seqüenciais e das interações entre elas, sendo flexível o suficiente para representar o maior número de classes de aplicações paralelas possível (aplicações cliente/servidor, mestre/escravo etc), sem que o desenvolvimento destas aplicações seja uma tarefa árdua para o usuário.

2.1 Comportamento das Unidades Seqüenciais

A afirmativa de que grafo dirigido é uma representação natural de programas paralelos é encontrada em muitos autores [BRO94] [NEW93]. Os círculos representam nodos de processamento e os arcos representam a interação entre estes nodos.

No que diz respeito aos nodos de processamento, existem duas semânticas de funcionamento a considerar: processos e *tasks*. Esta distinção tem por objetivo destacar dois tipos de comportamentos dos componentes seqüenciais de programas paralelos quanto a dois aspectos: gerência de processos e troca de mensagens. Os processos podem criar processos-filhos e interagir com outros processos em qualquer ponto de execução. Por outro lado, existem as *tasks* que podem ser comparadas a processos que não podem criar processos-filhos e possuem dois pontos de interação (antes e depois do processamento seqüencial). Recebem dados, executam um código seqüencial e propagam dados para outros nodos.

2.2 Semântica dos Arcos

No caso dos arcos também existe a necessidade de explicitar-se qual a semântica a ser adotada. Pelo menos duas semânticas podem estar associadas a um arco:

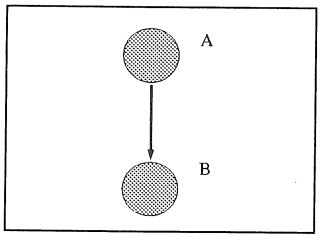


Figura 2.1 - Semântica dos arcos

- B depende de A para executar: B inicia a execução somente após A, recebendo todos os dados de A, caracterizando dependência total de dados;
- A fornece dados para B: existe uma dependência parcial de dados entre A e B. B necessita dos dados de A somente em determinado momento, especificado pelo programador através de uma linguagem declarativa de ativação sobre os dados de entrada. Por exemplo: "inicia processamento quando chegar uma informação X ou Y". X e Y poderiam ser fornecidos por nodos diferentes;

A escolha de uma ou outra semântica determinará o nível de flexibilidade e representabilidade da linguagem visual. A relação flexibilidade X facilidade de uso [BRO94] deve ser considerada de tal maneira que haja a maior flexibilidade e facilidade de uso possíveis.

3 Apresentação do Modelo

Esta seção descreve as características do trabalho proposto. O modelo utiliza tanto elementos visuais quanto textuais para a representação de um programa paralelo. A linguagem visual permite ao programador um controle sobre a troca de mensagens de uma maneira simples. Não é mais necessária também uma preocupação com a gerência de processos. As *tasks* (ou instâncias de nodos) são criadas na medida em que for necessário de acordo com o tipo de interação entre os nodos de processamento. Ao leitor interessado, uma descrição detalhada do modelo em geral pode ser encontrada em [SCH96].

3.1 Visão Geral

Um programa no trabalho proposto é um grafo dirigido em que os nodos representam *tasks* que possuem *interfaces* de entrada e saída bem definidas. A funcionalidade destes nodos é descrita em linguagem C acrescida de alguns elementos sintáticos. A interação entre os nodos é determinada pelos arcos que representam as dependências de dados entre os vários elementos de processamento.

Para cada nodo são definidas inicialmente diversas expressões de ativação. Estas expressões fazem referências às variáveis de entrada de dados e determinam qual processamento deve ser realizado num determinado instante. Após a análise das expressões de ativação e conseqüente execução de procedimentos, o nodo pode tornar dados de saída disponíveis para nodos sucessores (figura 3.1).

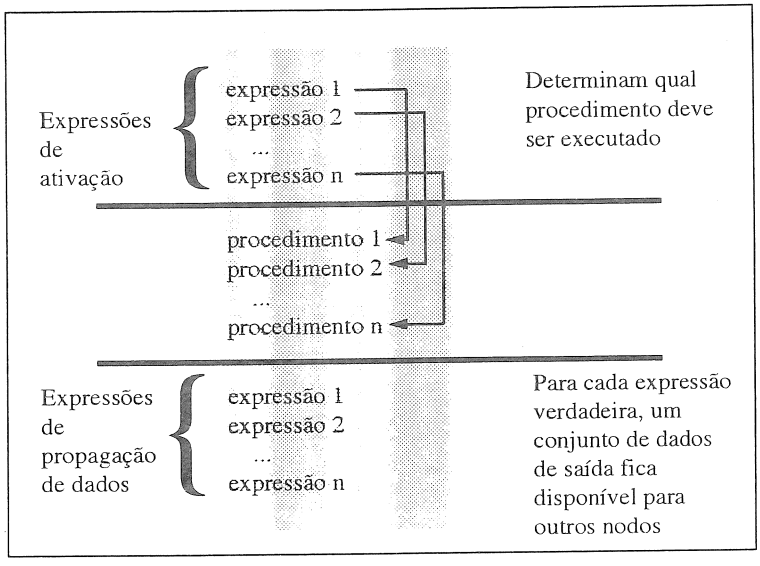


Figura 3.1 - Estrutura de Anotação de Nodo

Durante a anotação dos nodos, o programador não precisa se preocupar com a definição dos parâmetros de entrada e saída, que são realizadas no momento da criação dos arcos. O fluxo de execução de um nodo de processamento está descrito a seguir.

1. recebe dados
2. analisa expressões de ativação
3. escolhe uma entre as expressões verdadeiras
4. executa procedimento associado
5. repete procedimentos 2, 3 e 4 até que não haja mais expressões de ativação verdadeiras
6. para cada expressão de propagação de dados verdadeira, torna dados de saída correspondentes disponíveis para outros nodos

A existência do procedimento 5 é justificada pelo fato de que as expressões de ativação fazem referências não somente aos dados de entrada, mas também a variáveis locais do nodo, que podem ser alteradas pela execução de um bloco de procedimentos.

3.2 Edição do Grafo

O modelo proposto baseado em grafos dirigidos permite ao programador desenvolver a aplicação paralela ao mesmo tempo em que mantém uma visualização de três aspectos fundamentais:

- interação entre nodos;
- possibilidade de criação dinâmica de instâncias de nodos de processamento;
- particionamento de dados.

A linguagem visual é composta pelos nodos de processamento e arcos. Os nodos de processamento

são caracterizados pelo fato de permitirem ou não a criação dinâmica de instâncias de mesmo tipo, podendo ser simples (figura 3.2 (b), nodo 1) ou instanciáveis (figura 3.2 (b), nodo 2). Um nodo simples corresponde a somente uma *task* em execução. Já o nodo instanciável pode estar associado a uma ou mais *tasks* (instâncias) durante a execução de uma aplicação.

Os arcos representam o fluxo de dados entre os nodos e possuem uma lista de atributos:

- *variáveis de entrada e saída* com os respectivos tipos associados (os tipos são definidos globalmente a um grafo e podem ser do tipo simples ou matriz);
- *variável de instanciamento*: número de instâncias de um nodo-destino necessárias para o processamento da informação propagada pelo arco;
- *índice de propagação de dados*: identificador da primeira instância a receber dados.

Por exemplo, seja uma aplicação em que um nodo-mestre (nodo 1) deseja propagar uma informação X para N instâncias de um nodo-escravo (nodo 2). Cada uma das instâncias do nodo-escravo realiza um processamento e retorna um resultado Y para o nodo-mestre. A Figura 3.2 (b) mostra a representação visual deste exemplo. O número de instâncias de nodo-escravo é determinado dinamicamente pelo valor da variável de instanciamento N, que é manipulada pelo nodo-mestre. A primeira instância é numerada com i, a segunda com i+1, e assim por diante.

No início da execução de uma aplicação paralela existe somente uma instância para cada nodo do grafo. Durante a execução da aplicação, os nodos manipulam com os valores das variáveis de instanciamento e índices de propagação de dados permitindo a criação dinâmica de novas instâncias dos nodos instanciáveis.

No exemplo acima, o nodo 1 poderia inicializar o valor de N com 3 e o valor de i com 2. Neste caso, estaria indicando que, no momento em que X estivesse disponível para o nodo-destino a partir do nodo 1, haveria a necessidade da criação de três instâncias do nodo 2. Cada instância do nodo 2 recebe uma cópia de X, realiza um processamento e retorna um resultado emY.

Existem diversos tipos de interações a serem consideradas dependendo dos tipos dos nodos e tipos de variáveis associadas aos arcos. As interações são indicadas através de uma representação textual exibida no próprio arco. Visualmente o programador pode estabelecer diversos tipos de interações entre nodos de processamento (conforme figura 3.2):

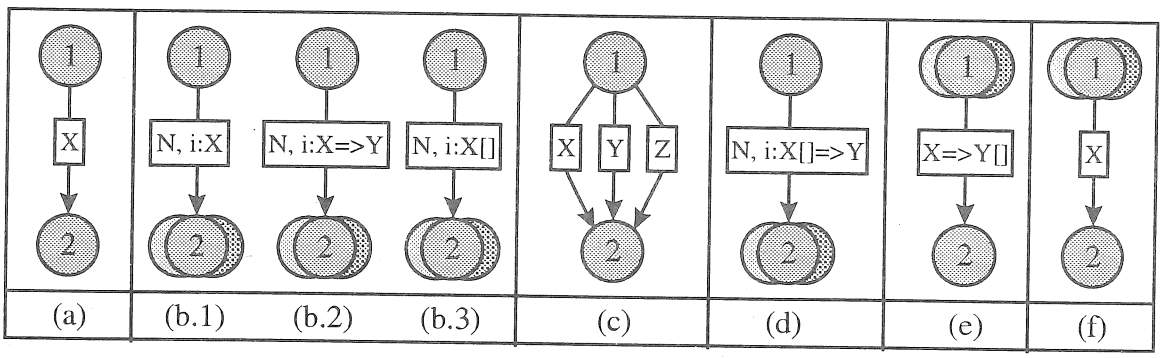


Figura 3.2 - Interações entre nodos de processamento

a) *transferência de dados ponto-a-ponto*: a variável X possui um tipo associado e pode ser manipulada tanto pelo nodo 1 quanto pelo nodo 2. X está definida como variável de saída no nodo 1 e como variável de entrada no nodo 2.

b) *multicast - propagação de uma informação para vários nodos*: comunicação entre um nodo simples e um nodo instanciável, propagando a variável X para N instâncias do nodo 2. O índice de propagação de dados i determina os valores dos identificadores das instâncias criadas dinamicamente. A expressão N, i: X, por exemplo, determina que sejam criadas N instâncias de um nodo-destino, identificadas a partir do valor i (i, i+1, ..., i+N-1). A transferência *multicast* pode ocorrer com renomeação de identificadores

(b.2) e também pode envolver variáveis do tipo *array* (b.3). Neste caso, todo o *array* é replicado em vários nodos-destinos.

c) *propagação de vários dados a partir de um nodo*: utilizada quando os dados podem não estar disponíveis todos ao mesmo tempo e devem ser enviados em momentos diferentes.

d) *particionamento de dados*: pode ocorrer quando a variável de saída associada ao arco for do tipo *array*. Durante a edição do arco, o programador indica que poderá haver particionamento. Neste caso, Y vai ser do mesmo tipo de X, mas terá uma área alocada dinamicamente. A cada interação entre o nodo 1 e 2, dependendo do valor da variável de instanciamento N, diferentes partes de X serão enviadas para diferentes instâncias do nodo 2.

e) *composição*: cada instância envia a parte que lhe foi destinada, reconstruindo o arranjo completo.

f) *propagação de vários dados de mesmo tipo*: cada instância do nodo-origem envia X e o nodo-destino realiza o processamento à medida que valores deX ficam disponíveis.

Essas construções permitem representar um bom número de aplicações, caracterizando a flexibilidade do modelo.

3.3 Edição de Nodos

```
DECLARATIONS
  <C source code>
END
STARTING_RULES
  [ init <proc_label> ; ]
  [ finish <proc_label> ; ]
  [ on receive
          <input_id> : { <C source code> } ;
           ...
    end ]
  < input_id > [ , ...] :
      [  ( <C boolean expression> ) : <proc_label>
          [ otherwise  { <C source code> } ] ; ]
      | [ < proc_label > ; ]
  ...
END
PROC <proc_label>
  <C source code>
END
ROUTING_RULES
  ( <C boolean expression> ) : { <C source code> } : <input_id> , ...;
   ...
END
```

Figura 3.3. Sintaxe da linguagem textual

O nodo de processamento é o elemento-chave de qualquer aplicação paralela. O processo de anotação inclui a definição de variáveis locais, regras de ativação, regras de propagação de dados e blocos de procedimentos:

- *variáveis locais*: são de escopo local a um determinado nodo e são definidas diretamente na linguagem C. Aparecem no blocoDECLARATIONS...END;
- *momento de ativação do nodo*: é determinado a partir de uma linguagem declarativa que manipula com os dados de entrada. Aparece no bloco STARTING_RULES...END e faz referências às variáveis de entrada definidas durante a criação dos arcos que chegam no nodo. Neste bloco

também são definidos os procedimentos de início e término de execução;

- *blocos de procedimentos*: existe um bloco de procedimentos associado a cada expressão de ativação. Estes procedimentos são descritos diretamente na linguagem C, permitindo a chamada de funções definidas em outros módulos;
- *regras de propagação de dados*: a propagação de dados é definida no bloco **ROUTING_RULES...END** e é feita após a fase de análise das expressões de ativação com execução dos blocos de procedimentos correspondentes. Quando a expressão booleana <*C boolean expression*> for verdadeira, o código <*C source code*> será executado e as variáveis de saída <$output\_id_1$> ... <$output\_id_n$> ficarão disponíveis para os nodos-destinos determinados pela topologia do grafo. A sintaxe deste bloco faz com que o programador não se preocupe com os destinos dos dados. Ele simplesmente declara que existem dados disponíveis.

Toda a informação pertencente a um nodo, representada pelos elementos sintáticos descritos acima, está organizada segundo a linguagem descrita na Figura 3.3.

4 Implementação

Uma aplicação criada a partir do ambiente proposto será executada sobre o sistema operacional de rede **HetNOS**. O **HetNOS** oferece uma série de facilidades (transparência de localidade, gerência de processos distribuída, troca de mensagens, ...) para programação paralela distribuída sobre uma rede de computadores. Os grafos e arcos serão mapeados inicialmente para processos e mensagens d**HetNOS**.

O código já foi totalmente desenvolvido, e no momento estão sendo realizados testes para remoção de erros. Em seguida, exemplos mais complexos serão desenvolvidos e executados para avaliação da funcionalidade e do desempenho.

4.1 Funcionamento do Sistema

O mapeamento entre o nível de programação visual (grafos dirigidos com anotações em linguagem C) e o nível do *middleware* **HetNOS** é composto pelas seguintes etapas:

1. análise léxico-sintática das definições de tipos: criação de um arquivo *header* em C chamado "<nome da aplicação>.h", incluído em todos os outros módulos.
2. consistência entre a definição e as referências de tipos nas anotações dos arcos: caso ocorram erros, o sistema indica em quais arcos existem referências a tipos ainda não definidos.
3. análise léxico-sintática das anotações dos nodos com geração de código;
4. geração do arquivo de make (*Makefile*);
5. compilação da aplicação paralela através do compilador **cc**.

A implementação compreende dois módulos principais: o editor de grafos e o gerador de código. O editor de grafos se encarrega da interface com o usuário, oferecendo comandos para a edição da estrutura do grafo e das anotações dos nodos. A partir desses dados, é gerado para cada nodo um arquivo com as anotações dos nodos segundo a linguagem da Figura 3.3. Construído na linguagem C e auxiliado pelas rotinas do ambiente **XView**, o editor roda em ambiente Unix com interface gráfica **XView**.

O gerador de código lê os arquivos com as informações dos nodos e gera os arquivos fontes C correspondentes. Além de gerar um arquivo fonte para cada nodo, o gerador de código cria um arquivo para o processo de controle, um arquivo header e um arquivo *Makefile*. A implementação é feita em C++, com a análise léxica e sintática da linguagem textual implementada pelas ferramentas **Lex** e **Yacc**.

4.2 Gerência de Processos

A gerência de processos é feita por um processo de controle, que dispara uma instância de cada um dos nodos do grafo. Durante a execução da aplicação, o processo de controle fica num estado de recepção de mensagens. No momento em que alguma instância de nodo faz uma chamada à função *v_terminate()* ou *v_abort()*, ocorre o envio de uma mensagem de término ao processo de controle que inicia a finalização da aplicação. A finalização pode ser abrupta (através da chamada de uma função de *kill* do **HetNOS**) ou

suave (através do envio de uma mensagem de término a cada instância criada inicialmente).

4.3 *Buffer* de Eventos

Inicialmente é necessário o entendimento de dois conceitos: variável de trabalho e variável livre. Variável livre é aquela para a qual podem ser atribuídos novos valores. Já a variável de trabalho corresponde àquela variável que tem um valor a ser processado, isto é, chegou algum valor para a variável mas nenhuma expressão de ativação tornou-se verdadeira (nenhum bloco de procedimentos foi ativado).

Para o exemplo da Figura 4.1, **X**, **Y** e **Z** são inicialmente variáveis livres. Suponha que ocorram os eventos com a seguinte seqüência de chegada de dados: (**Y**, **Z**, **Y**, **Z**, ...). O procedimento **proc2** seria executado a cada chegada de um valor para **Z**, tornando-a variável livre novamente e permitindo a atribuição de novos valores para **Z**. No caso da variável **Y**, **proc1** não é ativado e, portanto, novos valores para **Y** devem ser colocados numa fila para posterior processamento, evitando perda de valores de **Y**.

```
STARTING_RULES
  X,Y: proc1;
  Z: proc2
END
```

Figura 4.1 - Exemplo

4.4 Comportamento das Instâncias de Nodos

O programa gerado para um nodo possui o seguinte fluxo de execução:

1 - Inicialização da aplicação
2 - Executa código inicial
Enquanto não for término de execução
  3 - ObtémDados
  4 - Executa código C associado ao dado de entrada
  5 - AnalisaExpressões
  6 - Propaga dados
7 - Executa código de finalização

Cada nodo do grafo está associado a um processo **HetNOS** e, conseqüentemente, a um processo **UNIX**. Para tornar-se um processo **HetNOS**, os processos **UNIX** precisam se "logar" no **HetNOS**. Este *login* é feito a partir da função *h_login()* disponível na biblioteca do **HetNOS**. O procedimento 1 ("Inicialização da aplicação") é o responsável por este *login*.

Arcos sem nodos de origem identificam os nodos destino como nodos iniciais (main) da execução. O comportamento inicial desses nodos é ligeiramente diferente do fluxo acima apresentado: após a execução do código inicial (passo 2), o fluxo é desviado automáticamente para a propagação de dados (passo 6). Em seguida o fluxo desses nodos torna-se normal, isto é, ocorre um retorno à obtenção de dados (passo 3).

Após o *login* no **HetNOS**, ocorre a execução do código inicial especificado no bloco **STARTING_RULES...END**, ponto em que o processo entra num *loop* para obtenção e processamento dos dados de entrada.

Procedimento ObtémDados
 3.1 - Resolve pendência de eventos
 Se nenhuma pendência puder ser resolvida
  3.2 - Fica bloqueado esperando mensagem
  Se mensagem for de controle
   3.3 - Realiza procedimento de controle
  senão
   3.4 - Verifica se a variável de entrada em questão pode ser processada
   Se não puder
    3.5 - Coloca no buffer de eventos

senão
3.6 - Gerencia a criação de outras instâncias se for necessário
3.7 - Retorna dados sobre a variável de entrada correspondente
senão
3.8 - Gerencia a criação de outras instâncias se for necessário
3.9 - Retorna informações sobre a variável de entrada da lista de pendências
FimProcedimento

Inicialmente, tenta-se resolver alguma pendência de evento (passo 3.1). Se algum evento puder ser processado (houver a liberação da variável correspondente), então o evento é retirado da lista de pendências e processado como se houvesse chegado uma mensagem de outro nodo (passos 3.8 e 3.9). No caso de não haver pendências a serem processadas, o nodo entra em *receive* para receber mensagens de outros nodos (passo 3.2). Podem chegar mensagens de dados ou mensagens de controle (pedidos de término de execução). Para cada nova mensagem de dados que chega, o código gerado verifica se a variável de entrada de dados correspondente está em uso (passo 3.4). Se estiver, a mensagem é colocada no *buffer* de eventos para posterior processamento (passo 3.5). Os passos 3.6 e 3.8 estão relacionados com a criação dinâmica de instâncias de nodos, tarefa executada pela primeira instância do nodo instanciável.

```
STARTING_RULES
     on receive
          X : { count1 += 1; };
          Y : { count2 += 1; };
     end
    X, Z : (count1 > count2) : proc1;
       Y : (count1 <= count2) : proc2;
END
```

Figura 4.2. Exemplo

O procedimento 4 corresponde à edição do bloco **on receive ... end**. No exemplo da figura 4.2 aparece uma anotação de nodo com este bloco. Para cada valor atribuído a **X** que puder ser processado, o código em C correspondente será executado uma vez antes da análise das expressões de ativação.

A análise das expressões de entrada de dados (passo 5) determina as situações em que o nodo deve executar algum bloco de procedimentos. São verificadas as expressões que se tornaram verdadeiras em algum momento, entre as quais é feita uma escolha aleatória (para evitar postergação indefinida na escolha de uma determinada expressão).

Procedimento AnalisaExpressões
5.1 - Verifica quais expressões se tornaram verdadeiras
Se pelo menos uma expressão se tornou verdadeira
5.2 - Escolhe uma entre as expressões verdadeiras.
5.3 - Executa bloco de procedimentos associado à expressão escolhida
5.4 - volta para o procedimento 5.1
senão
Retorna da função
FimProcedimento

Depois que todas as expressões se tornarem falsas, o nodo propagará os dados de saída, segundo as expressões do bloco **ROUTING_RULES...END** (passo 6). Após a requisição de término da aplicação por algum nodo através da chamada da função *v_terminate()*, será executado um código de finalização (passo 7) para cada nodo.

5 Trabalhos relacionados

Existem diversos ambientes de programação visual de aplicações paralelas. Aqueles que mais se aproximam do trabalho proposto são CODE [NEW93] e HeNCE [BEG93]. HeNCE e CODE utilizam

grafos dirigidos para a representação de programas paralelos e geram programas para serem executados sobre PVM (*Parallel Virtual Machine*) [BEG91]. O modelo proposto diferencia-se de HeNCE e CODE principalmente quanto à representabilidade da linguagem e facilidade de uso.

Os modelos apresentam apresentam nodos de processamento e especificação do paralelismo dada pelo usuário. A semântica dos arcos em HeNCE indica dependência de controle, exigindo a existência de nodos de controle. Um nodo só começa a executar quando o seu predecessor terminar, determinando regras de ativação fixas. Embora facilite o trabalho do programador. as regras de ativação fixas limitam o número de aplicações que podem ser criadas. Em CODE e no modelo proposto os arcos indicam fluxo de dados, com as regras de ativação determinadas pelo programador.

Com relação à eficiência da aplicação, tanto CODE como o modelo proposto criam as instâncias dos nodos somente quando um dado for recebido e encerram os processos somente ao final da aplicação, evitando muitas vezes criação desnecessária de processos. Em HeNCE não há esta possibilidade.

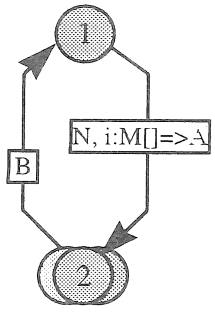
As linguagens visuais de CODE e HeNCE não diferenciam os tipos de nodos segundo a possibilidade de criação de várias instâncias. O escopo da variável de instanciamento em CODE é global ao grafo, enquanto no modelo proposto é local a um nodo, permitindo portanto que cada nodo instanciável possua o seu próprio número de instâncias. Em CODE, o programador não tem como saber o valor dos dados de entrada antes da análise das expressões de ativação. No modelo proposto isto é possível (bloco **on receive ... end**), oferecendo a possibilidade de selecionar dados de entrada de acordo com o seu conteúdo.

6 Exemplo

Apresentaremos um exemplo simples com o objetivo de dar uma visão geral da programação de uma aplicação paralela nesta ferramenta. O exemplo é o clássico cálculo do fractal. Nesta aplicação, o nodo principal inicializa os dados e envia as regiões do fractal a serem calculadas pelas instâncias. Ao final do cálculo, cada instância envia o resultado ao nodo inicial que reúne os dados recebidos.

Nodo 1

```
DECLARATIONS
  int tr = 1;
END
STARTING_RULES
  init_proc:
proc_initdata;
  B: proc_joinresults;
END
ROUTING_RULES
  (tr):{   tr=0;   N=4;
i=1; }:M;
END
```



Nodo 2

```
DECLARATIONS
  int tr = 0;
END
STARTING_RULES
  A: proc_calc;
END
ROUTING_RULES
  (tr):{ tr = 0; }:B;
END
```

No início da aplicação, o nodo 1 prepara a matriz M no procedimento **proc_initdata**. Depois, através das regras de propagação, envia-os às instâncias do nodo 2 que realizarão cálculos sobre os dados recebidos. Através da variável de instanciamento N e do índice de propagação i serão criadas 4 instâncias, identificadas a partir de 1. Os identificadores N e i são definidos no momento da criação do arco.

As variáveis tr nos dois nodos têm a função de indicar o momento em que as mensagens devem ser enviadas. No nodo 1, com tr igual a 1, a matriz M será enviada ao nodo 2. Para evitar que ela seja enviada novamente na próxima análise das regras de propagação de dados, tr recebe valor zero.

Durante a inserção do arco que indica o fluxo de M do nodo 1 para o nodo 2, é definido o tipo da interação entre os nodos. Nesse caso, definiu-se que a matriz será dividida entre as instâncias destino. No destino, há a regra de ativação que indica o procedimento a ser executado quando da chegada de A. Quando uma instância do nodo 2 receber A (A já é uma das partes da matriz M), irá executar o procedimento **proc_calc**. Este procedimento toma A e coloca o resultado em outra matriz B. No final, faz tr igual a 1 a

fim de que a regra de propagação correspondente se torne verdadeira e o resultado seja enviado ao nodo 1.

No nodo 1, há a regra de ativação que executa o procedimento **proc_joinresults** à medida que chegam as matrizes B com os resultados parciais. Este procedimento conta o número de resultados que chegam e quando o número for igual a N (recebeu de todas as instâncias) toma alguma ação (como desenhar o fractal na tela) e faz uma chamada à função v_terminate(), requisitando o fim da aplicação.

Por questões de simplicidade omitimos o código dos procedimentos referidos. Este código deve ser programado explicitamente em C pelo programador.

7 Conclusão

Este trabalho pode ser considerado um passo inicial no estudo e criação de um ambiente completo de programação visual e visualização de programas paralelos. A partir deste modelo poderá ser desenvolvido um mecanismo de depuração que utilize a mesma representação gráfica para comparar as três entidades: programa, comportamento esperado e execução [BRO94]. Outra característica importante não presente neste modelo é a existência de facilidades para a manipulação de *arrays* de dimensão variável, recurso essencial para muitos algoritmos paralelos. Há ainda a necessidade da inclusão do objeto subgrafo na linguagem visual. Desta maneira, o programador poderia organizar o programa como uma hierarquia de grafos, facilitando a representação de aplicações mais complexas (grande número de nodos).

Há ainda algumas questões pendentes na área de programação visual de aplicações paralelas. Por exemplo, ainda é incipiente a pesquisa na área de representação da execução do programa. Quais informações devem ser mostradas visualmente? Como estas informações devem ser representadas de tal maneira que se tenha uma visualização clara do que está acontecendo? As respostas a estas perguntas só poderão ser encontradas a partir de experiências com a criação de ferramentas semelhantes a este trabalho.

O objetivo principal deste trabalho é auxiliar a programação paralela explícita. Portanto, foi dada uma atenção especial ao aspecto facilidade de uso. Entretanto, não se pode esquecer da importância de outros dois aspectos: eficiência do código gerado e flexibilidade da ferramenta. O ambiente de programação deve gerar código eficiente e permitir a criação do maior número de aplicações possível.

Bibliografia

[BAR94] BARCELLOS, A.M.P. et al. Um ambiente para Programação de Aplicações Distribuídas em Redes de Workstations. In: CONGRESSO NACIONAL DE REDES DE COMPUTADORES, 12., 1994, Curitiba. **Anais...** Rio de Janeiro: SBC, 1994.

[BEG91] BEGUELIN, A. et al. **A User's Guide to PVM Parallel Virtual Machine**. Local de publicação: Oak Ridge National Laboratory, July 1991. (Technical Report ORNL/TM-11826).

[BEG93] BEGUELIN, A. et a;. **HeNCE**: A Heterogeneous Network Computing Environment. Local de publicação: University of Tennessee, 1993. (Technical Report CS-93-205).

[BRO94] BROWNE, J.C. et al. **Visual Programming and Parallel Computing**. Local de publicação: University of Tennessee, 1994. (Technical Report CS-94-229).

[CAS92] CASAVANT, T.L.; KOHL, J.A.; PAPELIS, Y.E. **Practical Use of Visualization for Parallel Systems**. Local de publicação: University of Iowa, 1992. (Technical Report Number TR-ECE-920102).

[HYD93] HYDER, S.I. et al. A Unified Model for Concurrent Debugging. In: International Conference on Parallel Processing, 1993.**Proceedings**.

[NEW93] NEWTON, P. **A Graphical Retargetable Parallel Programming Environment and Its Eficient Implementation**. Local de publicação: Univ. of Texas at Austin, 1993. (Technical Report TR93-28).

[SCH96] SCHRAMM, J.F.L. Ambiente Gráfico para o Desenvolvimento de Aplicações Distribuídas. CPGCC-UFRGS, 78pp, 1996.