OTIMIZAÇÃO DE PROGRAMAS ACTUS

Paula Marisa da Costa Panta Ferreira Maciel

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

Aprovada por:

Prof. Claudio Luís Amorim, Ph. D.
(presidente)

Prof. Edil Severiano Tavares Fernandes, Ph. D.

Prof. Leila Maria Ripoll Eizirik, D. Sc.

Prof. Adriano Joaquim de Oliveira Cruz, Ph. D.

RIO DE JANEIRO, RJ – BRASIL
AGOSTO DE 1991

MACIEL, PAULA MARISA DA COSTA PANTA FERREIRA

Otimização de Programas ACTUS [Rio de Janeiro] 1991

VI, 109 p., 29.7 cm, (COPPE/UFRJ, M. Sc., ENGENHARIA DE SISTEMAS E COMPUTAÇÃO, 1991)

TESE – Universidade Federal do Rio de Janeiro, COPPE

1 – Arquitetura de Computadores 2 – Processamento Paralelo

I. COPPE/UFRJ II. Título(Série).

Ao meu marido, Marcelo

Agradecimentos

Agradeço aos meus pais pelo incentivo que me deram durante toda a minha vida. Sem eles com certeza isto não teria passado de um sonho.

Agradeço ao meu marido que sempre me mostrou o lado bom das coisas, me estimulando nos momentos mais difíceis em que quase desisti de tudo. Sem ele isto não teria sido possível.

Agradeço ao Brasil por há 15 anos atrás ter me recebido de braços abertos dando condições, à minha família e a mim, de termos uma vida digna e com paz. Hoje ele é a minha terra.

Agradeço a todos os meus amigos verdadeiros pelas brincadeiras e pelo companheirismo em todos os momentos. Sem eles tudo teria sido muito sem graça.

Agradeço ao professor Claudio Amorim pelo interesse e pelo incentivo que sempre me deu. Sem a sua orientação os objetivos não teriam sido alcançados.

Agradeço aos demais professores com quem convivi. Sem eles não teria adquirido todos os conhecimentos que adquiri.

Uma homenagem especial a uma pessoa muito querida que sempre se orgulhou muito de mim e que eu perdi este ano, meu avô Antonio. A ele um obrigada com muito carinho.

'As vezes é preciso sonhar para não desistirmos...

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

Otimização de Programas ACTUS

Paula Marisa da Costa Panta Ferreira Maciel

Agosto de 1991

Orientador: Claudio Luís de Amorim
Programa: Engenharia de Sistemas e Computação

Este trabalho se propõe a aplicar as técnicas de otimização de linguagens seqüinciais numa linguagem vetorial: ACTUS II. Poucas foram as modificações necessárias para que as técnicas se adaptassem às características da linguagem ACTUS II e foi alcançado bastante ganho em termos, de paralelismo.

O trabalho faz parte de um projeto maior cujo objetivo é construir um compilador de ACTUS II. O compilador deverá conter o *front-end*, o *back-end* e o otimizador. A linguagem ACTUS II, as técnicas de otimização e sua aplicação na linguagem vetorial estão descritas neste trabalho.

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

Optimization of ACTUS Programs

Paula Marisa da Costa Panta Ferreira Maciel

Agosto, 1991

Thesis Supervisor: Claudio Luís de Amorim
Department: Programa de Engenharia de Sistemas e Computação

In this work we propose tha aplication of optimization techniques in sequential languages into a vector language: ACTUS II. There were few necessary changes for adjusting the techniques to the caracteristics of ACTUS II and it was reached a great amount of parallelism. The project wich the optimizer belongs to is a compiler for ACTUS II. The ACTUS II language, the optimizations techniques and their aplications in ACTUS II are discussed on this work.

Índice

Capítulo I

Introdução

São vários os motivos que levaram os pesquisadores a estudarem técnicas de deteçcão de paralelismo. A maioria dos programas é escrita em linguagens com características seqüenciais como Fortran, Pascal e outras. Com o desenvolvimento na área de arquiteturas de computadores paralelos e vetoriais passou a existir a necessidade de se expressar algum tipo de paralelismo na programação. Como esse paralelismo nao é expresso nos programas já existentes e estes não podem ser desprezados devido à carga de trabalho e informação que representam procurou-se então em desenvolver técnicas que detectassem o paralelismo e otimizassem os programas.

Já existem muitos trabalhos de otimização de código feitos para linguagens seqüenciais como Pascal e FORTRAN. Por isso achou-se mais interessante que todo o trabalho de pesquisa se voltasse para uma linguagem que já possuísse algum paralelismo embutido, como ACTUS II. Além disso, esta tese faz parte de um projeto maior cuja finalidade é construir um compilador para a linguagem ACTUS II. O compilador será formado pelo *front end*, o *back end* e o otimizador.

Ao se aplicarem técnicas de otimização numa linguagem com as características de ACTUS II, os objetivos principais são: descobrir blocos e comandos que podem ser executados em paralelo ou que podem ser vetorizados (usando-se as prerrogativas da própria linguagem) e, que neste último caso, passaram desapercebidos pelo programador. As técnicas estudadas compreendem: renomeação de variáveis, expansão de variáveis escalares, substituição para a frente de comandos, quebra de comando, reordenação de comandos, *loop blocking*, distribuição do *loop*,

redução de ciclo e *loop spreading.*

No capítulo II são mostradas as principais *dependências de dados* assim como tudo o que se relaciona com elas, inclusive a construção do *grafo de dependência.* As *técnicas de otimização* estudadas para linguagens seqüenciais são apresentadas, acompanhadas de exemplos, no capítulo III. A linguagem ACTUS II e as *técnicas de otimização* agora aplicadas a ela são vistas no capítulo IV. Neste capítulo são mostrados também os algoritmos que representam uma proposta de implementação de cada uma das técnicas. As conclusões do trabalho são apresentadas no último capítulo. Durante todo o trabalho será usado ACTUS para substituir ACTUS II.

Capítulo II

Dependências de Dados

Para que se possa fazer a reestruturação automática do programa é necessário computarem-se as dependências de dados, as quais mostram como os dados são calculados e usados durante a execução do programa.

É tarefa do compilador detectar operações vetoriais ou paralelas em *loops* seriais. Ele deve descobrir as dependências de dados dentro do *loop* e permitir a execução paralela ou vetorial sem violar essas dependências.

As informações de dependências são importantes para se verificar quando as otimizações ou transformações são legais. Uma transformação é dita legal se a execução do programa seqüencial reestruturado produzir o mesmo resultado que o programa original.

Um programa deve ser visto como uma seqüência de comandos de atribuição e de *loops* que contém ou não outros *loops*. *IN (S)* representa o conjunto de variáveis lidas pelo comando S e *OUT (S)* representa o conjunto de variáveis escritas pelo comando S.

Um comando de atribuição pode envolver variáveis escalares ou indexadas e possui a forma a seguir:

$$S_i\text{: x = E;}$$

Os *loops* têm o seguinte formato:

L_j : do I_j = p, N_j
H
end

onde p e N_j são os limites inferior e superior, respectivamente, e H é o corpo do *loop* podendo ser uma seqüência de *loops* e de comandos de atribuição. I_j assume valores i_j, p $\leq i_j \leq N_j$.

Quando H é formado por um conjunto de *loops* recebe o nome de *aninhamento.* O aninhamento é perfeito se o corpo do *loop* L_k é justamente o de L_{k+1}. O corpo do *loop* mais interno é visto como o corpo do aninhamento.

Dado um comando S pertencente a um programa, define-se como *nível* de S ou *nest (S)* o conjunto de todos os *loops* que contém esse comando.

Dois comandos S e T, não necessariamente distintos, num programa, definem três aninhamentos disjuntos: um aninhamento L, que engloba os dois comandos, um aninhamento L_S formado pelos *loops* que contém S mas não T e um aninhamento L_T formado pelos *loops* que contém T mas não S. Os aninhamentos L, L_S e L_T são os *maiores possíveis.* Se o corpo de H for formado por n *loops* a divisão será a seguinte:

L = (L_1, L_2, ..., L_e);

L_S = (L_{e+1}, L_{e+2}, ..., L_m);

L_T = (L_{m+1}, L_{m+2}, ..., L_n);

Assim:

nest(S) = (L, L_S), com m *loops*;

nest(T) = (L, L_T), com (n - m + e) *loops*;

O conjunto de índices de nest(S) é (I, I_S) e uma iteração qualquer denota-se por (i, i_S), onde i = (i_1, i_2, ..., i_e) e i_S = (i_{e+1}, i_{e+2}, ..., i_m). Para T, nest(T) = (I, I_T) e uma iteração qualquer é representada por (j, j_T), onde j = (j_1, j_2, ..., j_e) e j_T = (j_{m+1}, j_{m+2}, ..., j_n).

Dados dois vetores i = $(i_1, i_2, \ldots, i_k)$ e j = $(j_1, j_2, \ldots, j_k)$ define-se a relação $<_u$ como:

$$\text{i} <_u \text{j} \quad \text{se e só se } i_1 = j_1, i_2 = j_2, \ldots, i_{u-1} = j_{u-1} \text{ e } i_u < j_u.$$

Por exemplo, tem-se (1,3) $<_1$ (2,-4) e (1,2) $<_2$ (1,3), em R^2. Se a definição de $<_u$ for estendida para o caso de $u = m+1$, então i $<_{m+1}$ j significa que $i_r = j_r$ para cada r no intervalo $1 \leq \text{u} \leq \text{m}$.

Usa-se a notação $S < T$ para dizer que o comando S precede lexicamente o comando T dentro do programa. Esta é uma informação importante para se estabelecer uma *ordem de execução* entre comandos. O fato de $S < T$ não significa obrigatoriamnte que o comando S seja executado antes do comando T. Para se descobrir isso as condições abaixo devem ser verificadas.

S e T são dois comandos de atribuição num programa. Uma instância S(i, i_S) é executada antes da instância de T(j, j_T) se e só se:

1- S e T não tiverem índices em comum (nest(S) = (L_S) e nest(T) = (L_T)) e $S < T$;

2- $i \leq j$ para $S < T$;

3- $i < j$ para $S \geq T$;

A notação acima é bastante geral e cada caso deve ser examinado em separado, como nos exemplos a seguir.

EXEMPLO:

```
DO    i := 1,IMAX BEGIN
S:    A(i) := B(i) + C(i);
END

DO    j := 1,JMAX BEGIN
T:    C(j) := K(j) + B(j);
END
```

Como os comandos S e T não possuem nenhum índice em comum todas as instâncias do primeiro *loop* são executadas antes das instâncias do segundo. Este exemplo se enquadra no caso 1 das condições acima.

EXEMPLO:

```
DO    i :=   1,100 BEGIN
      DO     j := 1,50 BEGIN
      S:     A(i,j) := B(i) * C(3i,j);
      T:     K(i,j) := A(i+1, j) + C(3i,j);
      END
END
```

Lexicamente, $S < T$. Uma instância de S é representada pelos valores (i_1, j_1) e de T pelos valores (i_2,j_2). Qualquer instância de S será executada antes de T se $(i_1, j_1) \leq (i_2, j_2)$. Por exemplo, a instância S(1,3) é executada antes de T(2,3). Na verdade basta que o vetor i = (i_1, j_1) seja $<_1$ que o vetor j = $(1_2, j_2)$ para que qualquer instância de S seja executada antes de T. Se fixarmos o valor do índice mais externo (i) fazendo $i_1 = i_2$, então basta que i$<_2$j ou ainda i$<_3$ para que a afirmativa acima seja verdadeira.

Como $T > S$, qualquer instância de T será executada antes de S se $(i_2,j_2) < (i_1,j_1)$. Na verdade é suficiente que j $<_1$ i, não importando o valor de j_1 e j_2. Por exemplo, a instância T(9,40) é executada antes de S(10,10). Se fixarmos o valor do índice mais externo (i) basta que j $<_2$ i para garantir que qualquer instância de T seja executada antes de S. Este exemplo se enquadra no caso 2 das condições acima.

A informação do fluxo de dados é gerada em dois níveis de complexidade: informação local, envolvendo apenas pares de comandos no programa e informação global, envolvendo o fluxo através do programa inteiro. A análise local é chamada de *teste de dependência de dados* e a análise global é chamada de *partição pi* sendo definida depois.

São dados os dois comandos S e T, não necessariamente distintos, num *aninhamento*, e uma instância de cada um, S(i) e T(j). Define-se a análise

local:

S(i) e T(j) estão envolvidos numa *dependência de saída* se e só se (1) S(i) for executada antes de T(j) e (2) OUT(S(i)) $\cap$ OUT(T(j)) $\neq \vee$. Simboliza-se S δ° T e graficamente $\multimap$.

S(i) e T(j) estão envolvidos numa *antidependência* se e só se (1) S(i) for executada antes de T(j) e (2) IN(S(i)) $\cap$ OUT(T(j)) $\neq \vee$. Simboliza-se S $\overline{\delta}$ T e graficamente $\not\rightarrow$.

S(i) e T(j) estão envolvidos numa *dependência direta* se e só se (1) S(i) for executada antes de T(j) e (2) OUT(S(i)) $\cap$ IN(T(j)) $\neq \vee$. Além disso não pode existir nenhuma instância k de outro comando Q(k) que seja executada depois de S(i) e antes de T(j) tal que S δ° Q. Simboliza-se S δ T e graficamente $\rightarrow$.

O teste da condição (1) é chamado *teste de prioridade* no qual se verifica a *ordem de execução* dos dois comandos envolvidos. O teste para a condição (2) é chamado *teste de interseção* e serve para estudar a interseção entre os conjuntos de entrada e saída dos dois comandos envolvidos.

Cada dependência tem associado a si um *sentido*, o qual mostra a sua origem e o seu destino variando com o tipo de dependência. O *sentido* é sempre fixo podendo ser *para a frente* ou *para trás.* Para se mudar o *sentido* de uma dependência é necessário inverter-se a ordem léxica de execução dos dois comandos envolvidos.

O *sentido* para cada dependência:

Numa *dependência direta* (S δ T), o *sentido* vai do comando que escreve numa posição de memória (S) para o comando que lê a mesma posiçao (T);

Numa *antidependência* (S $\overline{\delta}$ T), o *sentido* é do comando que lê uma posição de memória (S) para o comando que escreve na mesma posição (T);

Numa *dependência de saída* (S δ° T), o *sentido* é do comando que escreve pele primeira vez numa posição de memória (S) para o comando que reescreve a mesma posição (T);

O resultado da análise de dependências é uma ferramenta chamada *grafo de dependências* que será usada em todo o processo de otimização. Nesse grafo cada nó representa um comando e os arcos representam as dependências entre os comandos. O *sentido* de cada arco está associado ao tipo de dependência que ele representa. Além disso, os arcos podem ser de três tipos diferentes, de acordo com cada dependência existente.

A importância do *sentido* se baseia no fato de que para se vetorizar um *loop* todas as dependências devem ter *sentido* para a frente. Algumas técnicas têm como objetivo apenas mudar o *sentido* das dependências.

As dependências que se encontram no escopo de comandos repetitivos são munidas de algumas informações adicionais, além de *sentido*, como *distância* e *direção*. Isso é importante porque estamos interessados não só nas relações de dependência entre comandos mas também nas relações entre instâncias dos comandos. Algumas técnicas, para serem aplicadas necessitam dessas informações.

Para cada dependência de dados que envolve os comandos S(i_1, i_2, ..., i_k) e T(j_1, j_2, ..., j_k), num aninhamento de nível K, saindo de S (*origem*) e chegando em T (*destino*), define-se a r-ésima *distância* ϕ_r como $\phi_r = j_r - i_r$, $(1 \leq r \leq k)$. O vetor $< \phi_1, \phi_2, \ldots, \phi_k >$ é chamado *vetor de distâncias*, podendo ser formado por valores positivos, negativos e zeros.

Os vetores (i_1, i_2, ..., i_k) e (j_1, j_2, ..., j_k) representam valores particulares dos índices dos K *loops* que formam o aninhamento. Esses valores fazem com que os subscritos das variáveis indexadas envolvidas numa dependência sejam iguais.

A *distância* pode ser constante ou variável e indica o número de iterações necessárias para que dois comandos causem alguma dependência.

Para algumas técnicas de otimização, mais importante que o valor

da *distância*, que nem sempre é constante, é o seu sinal. A forma encontrada para se tratar isso é salvar-se o sinal da *distância* num vetor chamado *vetor de direção*. Cada elemento desse vetor pode ser +, -, 0 ou representado por <, >, =.

Além disso, a *direção* mostra a *iteração* de onde parte a dependência e a *iteração* aonde ela chega. Se as iterações forem iguais, a *direção* é 0 ou =; se a iteração da origem é menor que a iteração da chegada, a *direção* é + ou <; se a iteração da origem for maior que a iteração da chegada, a *direção* é - ou >.

As definições acima são muito importantes para se verificar se uma determinada técnica pode ou não ser aplicada sobre um par de dependências.

EXEMPLO:

```
FOR   I := 1,50 BEGIN
S1:   C(I) := A(I+4);
S2:   A(I) := C(I) + 2;
END
```

Uma iteração do comando S_1, para o vetor A, é representada por I = j ($A \in IN(S_1)$) e do comando S_2, para o mesmo vetor, por I = i ($A \in OUT(S_2)$) . Assim para que haja interseção entre os dois comandos em relação ao uso do vetor A:

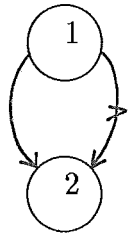
$$\mathrm{i} = \mathrm{j} + 4 \Longrightarrow \textit{i sempre maior que j;}$$

Como j é sempre menor que i, o comando S_1 lê antes do comando S_2 escrever a mesma posição de A. Então a dependência em relação ao uso de A é uma *antidependência* e o seu sentido é de S_1 para S_2. Assim essa dependência vai de uma iteração I = j para a iteração I = i. Como j é sempre *menor* que i, então a *direção* é <.

A *distância* é a diferença entre o valor de i e de j para que i = j+4. Assim ela é constante e igual a 4.

A *dependência direta* causada pelo uso do vetor C tem *sentido para*

frente, *distância* igual a 0, pois ocorre dentro da mesma iteração, e *direção* estacionária igual a =. O *grafo de dependência* deste exemplo está na figura II.1.



$S_1 \overline{\delta}_{<} S_2$
$S_1 \delta_{=} S_2$

Figura II.1: Grafo de Dependência

EXEMPLO:

```
FOR  I :=   1,N BEGIN
     FOR    J := 2,N BEGIN
     S1:    A(I,J) := H(I,J) + B(I,J);
     S2:    C(I,J) := A(I,J-1) + D(I+1,J);
     S3:    D(I,J) := 0.1 * A(I+1,J-1);
END
     END
```

Uma iteração do comando S_1 é (I,J) = (i_1,j_1), do comando S_2 é (I,J) = (i_2,j_2) e do comando S_3 é (I,J) = (i_3,j_3). Existem duas dependências causadas pelo uso de A e uma dependência pelo uso de D. Elas serão analisadas separadamente.

- A(I,J) := ...
  ... := A(I,J-1) + ...

$$j_1 = j_2 - 1 \Longrightarrow j_1 \text{ sempre } \textit{menor} \text{ que } j_2;$$

Como $i_1 = i_2$ e $j_1 < j_2$ a dependência entre os dois comandos é uma *dependência direta* cujo sentido é do comando S_1 para o comando S_2 (*para a frente*). Sendo assim, ela vai da iteração menor (i_1,j_1) para a iteração (i_2,j_2). A *direção* é $<$ e a *distância* é 1, em relação a J. Como os comandos se encontram num aninhamento e possuem *nível* igual a 2 *direção* e *distância* devem ser representadas por um vetor.

O *vetor de distância* é <0, 1> e o *vetor direção* é (=,<). A dependência pode ser representada por $S_1\delta_{(=,<)}S_2$.

- A(I,J) := ...
  ... := A(I+1,J-1) + ...

$$i_1 = i_3 + 1 \Longrightarrow i_1 \text{ sempre } \textit{maior} \text{ que } i_3;$$

$$j_1 = j_3 - 1 \Longrightarrow j_1 \text{ sempre } \textit{menor} \text{ que } j_3;$$

Como $i_1 > i_3$ a dependência entre os comandos é uma *antidependência* com *sentido* de S_3 para S_1 (*para trás*). Sendo assim, a dependência vai da iteração menor (I,J) = (i_3,j_3) para a iteração (I,J) = (i_1,j_1). Em relação a I a *direção* é < e a *distância* é 1. Já em relação a J, como $j_3 > j_1$ a *direção* é > e a *distância* é -1.

O *vetor de distância* é <1, -1> e o *vetor de direção* é (<, >) e a dependência pode ser representada por $S_3\overline{\delta}_{(<,>)}S_1$.

- ... := D(I+1,J)
  D(I,J) := ...

$$i_3 = i_2 + 1 \Longrightarrow i_3 \text{ sempre } \textit{maior} \text{ que } i_2;$$

Como $i_3 > i_2$ e $j_3 = j_2$ a dependência causada pelo uso do vetor D é uma *antidependência* cujo *sentido* é do comando S_2 para o comando S_3 (*para a frente*). Sendo assim, ela vai da iteração menor (i_2,j_2) para a iteração maior (i_3,j_3). A *direção* é < e a *distância* é 1, em relação a I, e 0, em relação a J.

O *vetor distância* é $< 1, 0 >$ e o *vetor direção* é $(<, =)$. A seguir, na figura II.2 é mostrado o *grafo de dependência* correspondente.

Uma coisa que deve ser observada é que se os subscritos só envolvem equações do tipo (a $\pm$ I) é simples verificar-se se há alguma dependência, o seu tipo e os vetores de *direção* e de *distância* associados. Quando os subscritos são expressões do tipo (aI $\pm$ b) a identificação da dependência, caso exista, e tudo o que é associado a ela se torna bem mais complicada.

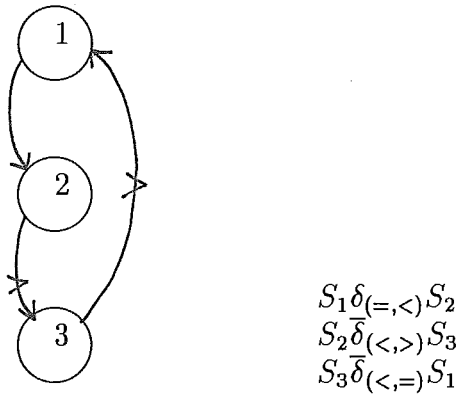


Figura II.2: Grafo de Dependência

EXEMPLO:

```
FOR   I := 1,50 BEGIN
S1:   A(2I) := B(I) + 2;
S2:   C(I) := A(2I+1) + 2;
END
```

Neste exemplo, a princípio imagina-se que exista uma dependência entre os dois comandos dentro do *loop*. Mas ao fazer-se uma análise mais cuidadosa dos subscritos observa-se que os dois comandos nunca acessarão posições de memória iguais. Com esse tipo de subscrito a análise fica mais difícil.

Além disso, no caso de comandos dentro de aninhamentos, para se identificar o tipo de dependência deve-se estudar o primeiro índice cuja *distância* seja diferente de 0.

Define-se a análise global:

Um *pi-bloco* é um conjunto máximo de comandos ciclicamente conectados por uma dependência de dados. Assim cada *pi-bloco* pode ser um comando independente (não envolvido por nenhum ciclo) ou um con-

junto de comandos ligados por um ciclo de dependências. A *partição pi* é a divisão do programa em *pi-blocos* e o *grafo parcialmente ordenado* é a tradução do grafo de dependência de dados em relação à *partição pi.* Isso implica que para cada dependência entre os comandos S_i e S_j no grafo de dependências existirá um arco no *grafo parcialmente ordenado* entre os *pi-blocos* que contém S_i e S_j.

Um dos objetivos da aplicação das técnicas de otimização é tentar quebrar os ciclos existentes. Os ciclos que não podem ser quebrados são chamados de *recorrências* e representam conjuntos de comandos que não podem ser executados em paralelo.

Capítulo III

Técnicas de Otimização

Serão descritas a seguir as técnicas mais comuns de otimização para vetorizar e paralelizar programas seqüenciais. Existem diferentes transformações e cada uma delas explora um tipo de paralelismo diferente. Apesar das técnicas serem independentes entre si a ordem em que são aplicadas é significante. Podem-se aplicar as mesmas técnicas num mesmo programa em ordens diferentes que serão obtidos programas sintaticamente distintos, com quantidades de paralelismo que variam e que podem levar a *desempenhos* diferentes na mesma máquina. Por isso a melhor ordem de se aplicarem essas transformações é ainda um problema em aberto.

Para a aplicação das transformações que serão descritas assume-se que o *grafo de dependências* com todas as informações necessárias sobre o programa já foi construído pelo compilador.

Para se representar a *vetorização* de alguns blocos será usada a seguinte notação:

$$X\ (1{:}N) := \ldots$$

a qual mostra que os N primeiros elementos do vetor X serão acessados de uma só vez.

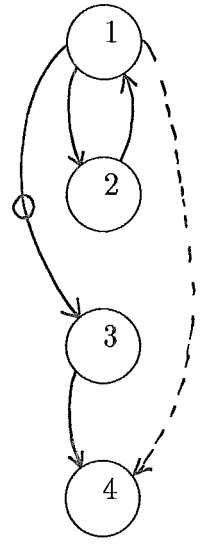
III.1 Renomeação de Variáveis

Algumas variáveis são usadas com fins distintos em diferentes pontos do programa aparecendo no *conjunto de saída* de diversos comandos, podendo introduzir no *grafo de dependências* algumas *antidependências* e *dependências de saída* falsas. Em programas seqüenciais o problema não é grave, mas quando se pensa em paralelizá-los ou vetorizá-los pode-se estar impondo restrições.

O objetivo principal da técnica é eliminar as falsas dependências atribuindo diferentes nomes para essas variáveis simplificando o *grafo de dependências.*

EXEMPLO I:

```
DO    i := 1,IMAX BEGIN
1     K := X(i) + Y(i+1);
2     X(i+2) := K + 1;
END

DO    j := 1,JMAX BEGIN
3     K := Z(j) * 100;
4     P(j) := K * 10;
END
```

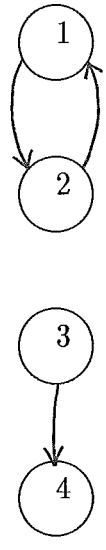


$S_1 \delta_= S_2$
$S_1 \delta^o_= S_3$
$S_2 \delta_= S_1$
$S_3 \delta_= S_4$

Figura III.1: Grafo de Dependência

Na figura III.1 há uma *dependência de saída* entre os comandos S_1 e S_3 causada pelo aparecimento da variável K nos respectivos conjuntos de saída desses comandos. Com a renomeação dessa variável em um dos *loops* a dependência será eliminada e os blocos ficarão independentes. A dependência tracejada é falsa pois o valor de K usado no comando S_4 é o calculado no comando S_3 e não no comando S_1 (figura III.2).

```
DO    i := 1,IMAX BEGIN
1     K := X(i) + Y(i+1);
2     X(i+2) := K + 1;
END

DO    j := 1,JMAX BEGIN
3     KMOD := Z(j) * 100;
4     P(j) := KMOD * 10;
END
```



$S_1 \delta_{=} S_2$
$S_2 \delta_{=} S_1$
$S_3 \delta_{=} S_4$

Figura III.2: Grafo de Dependência

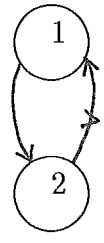
III.2 Quebra de Comando

Esta técnica tem como objetivo principal quebrar ciclos existentes no *grafo de dependências* trabalhando em cima de *antidependências* e de *dependências de saída* falsas. A princípio, quando há algum ciclo no *grafo de dependências* a vetorização

ou paralelização é impossível. Quando os ciclos envolvem apenas *dependências diretas* eles realmente não podem ser quebrados, mas quando envolvem *antidependências* alguma otimização deve ser feita podendo resultar numa paralelização parcial dos *loops*. A *quebra de comando* se implementa com a introdução de novos comandos de atribuição e de novas variáveis no programa fonte.

EXEMPLO I:

```
DO    i := 1,N
1     A(i) := B(i) + C(i);
2     D(i) := A(i-1) + A(i+1);
END
```



$S_1 \delta_< S_2$
$S_2 \overline{\delta}_< S_1$

Figura III.3: Grafo de Dependência

Na figura III.3 existe uma *antidependência* entre os comandos S_1 e S_2 com *sentido para trás* ($S_2\ \overline{\delta}\ S_1$) e uma *dependência direta* entre os mesmos comandos, com *sentido para frente* ($S_1\ \delta\ S_2$). Essas duas dependências provocam um ciclo envolvendo esses comandos. É exatamente esse ciclo que a técnica tenta quebrar.

```
DO    i := 1,N
3     T(i) := A(i+1);
1     A(i) := B(i) + C(i);
2     D(i) := A(i-1) + T(i);
END
```

Com a quebra do comando S_2 e a criação de um novo comando, o S_3, o ciclo desaparece na figura III.4. Como todos as dependências têm o mesmo *sentido*, o *loop* pode então ser vetorizado. A *dependência direta* não foi alterada.

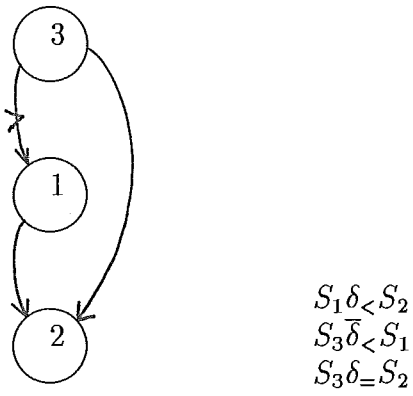


Figura III.4: Grafo de Dependência

```
3  T(1:N) := A(2:N+1);
1  A(1:N) := B(1:N) + C(1:N);
2  D(1:N) := A(0:N-1) + T(1:N);
```

III.3 Reordenação de Comandos

Um compilador sempre possui várias alternativas na ordem de execução dos comandos de um *loop*. Para que a reordenação não altere os resultados finais não devem existir dependências com *sentido para a frente* unindo os comandos que vão ser reordenados, no *grafo de dependências*. Na verdade, a reordenação tem como objetivo trocar o *sentido* da dependência entre dois comandos para que possam ser vetorizados. Para um *loop* ser vetorizado todas as dependências devem ter o mesmo *sentido, para a frente.*

Apesar de ser uma transformação bastante simples, nem sempre é permitido aplicá-la. A restrição é mostrada a seguir:

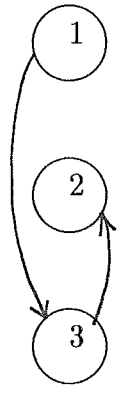
> se um *loop* L contém os comandos S_j e S_k, onde S_j vem antes de S_k, e existe alguma dependência com *direção* (=) entre esses dois comandos, então S_k não pode ser colocado antes de S_j.

Se o *grafo de dependências* é acíclico, então uma reordenação correta dos comandos sempre permitirá que o *loop* seja vetorizado.

Um outro objetivo da reordenação é a minimização da comunicação entre comandos que pertençam a blocos que são executados em paralelo. O compilador (otimizador) pode detectar no programa fonte tarefas que são independentes e por isso podem ser executadas por processadores diferentes ao mesmo tempo. Isso pode acontecer mesmo que exista alguma dependência refletida na comunicação entre essas tarefas. Essa dependência afeta o tempo de execução e a correta reordenação de comandos pode reduzir bastante o atraso provocado pela espera da tarefa em obter algum dado que será produzido num instante mais à frente por outra tarefa.

EXEMPLO I:

```
DO    i := 1,N
1     A(i) := D(i) * T;
2     B(i) := (C(i) + E(i)) / 2;
3     C(i+1) := A(i) + 1;
END
```



$S_1 \delta_{=} S_3$
$S_3 \delta_{<} S_2$

Figura III.5: Grafo de Dependência

No grafo de dependência da figura III.5 existe uma *dependência direta* entre os comandos S_2 e S_3 com *direção* $<$ e *sentido para trás* e uma *dependência direta* entre os comandos S_1 e S_3. Pela regra acima, os comandos S_2 e S_3 podem ser reordenados.

```
DO    i := 1,N
1     A(i) := D(i) * T;
3     C(i+1) := A(i) + 1;
2     B(i) := (C(i) + E(i)) / 2;
END
```

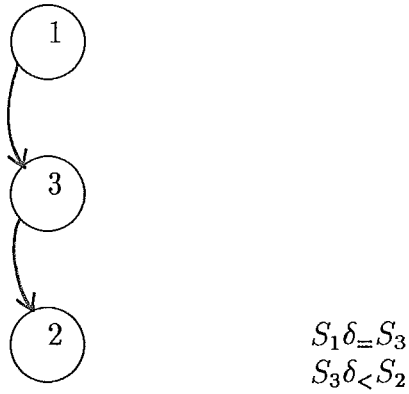


Figura III.6: Grafo de Dependência

Após a aplicação da técnica todas as dependências da figura III.6 possuem o mesmo *sentido para a frente* e o bloco pode ser vetorizado.

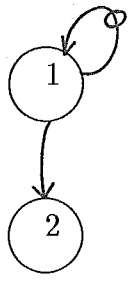
```
1  A(1:N) := D(1:N) * T;
3  C(2:N+1) := A(1:N) * 1;
2  B(1:N) := (C(1:N) + E(1:N)) / 2;
```

III.4 Expansão de Variáveis Escalares

O objetivo desta técnica é transformar em vetoriais variáveis escalares que são usadas dentro de blocos do tipo *DO*. Com isso eliminam-se *dependências de saída* e *antidependências* e diminui o número de arcos no *grafo de dependências* passando cada iteração dos blocos a ter seu próprio conjunto de posições de memória. Esse problema não ocorre quando vetores são usados já que em cada iteração uma posição diferente de memória é acessada.

EXEMPLO I:

```
DO    j := 1,JMAX BEGIN
1     KMOD := Z(j) * 100;
2     P(j) := KMOD * 10;
END
```

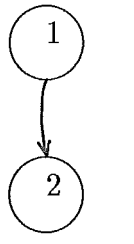


$S_1 \delta_{=} S_2$

Figura III.7: Grafo de Dependência

O uso da variável escalar *KMOD* causa uma *dependência de saída* que aparece na figura III.7, envolvendo o comando S_1, pois em cada iteração do *loop* as posições de escrita e de leitura são sempre as mesmas o que impede a vetorização do *loop*. A variável *KMOD* deve ser expandida (temporariamente).

```
KMODTEMP   := 0;
DO         j := 1,JMAX BEGIN
1          KMODTEMP(j) := Z(j) * 100;
2          P(j) := KMODTEMP(j) * 10;
END
KMOD       := KMODTEMP(jmax)
```



$S_1 \delta_{=} S_2$

Figura III.8: Grafo de Dependência

Agora, na figura III.8, só existe uma *dependência direta* com *sentido para frente* envolvendo os dois comandos. Assim o bloco pode ser vetorizado.

```
1  KMODTEMP(1:JMAX) := Z(1:JMAX) * 100;
2  P(1:JMAX) := KMODTEMP(1:JMAX) * 10;
```

III.5 Substituição para Frente de Comandos

Esta transformação tem como objetivo principal eliminar alguns arcos de *dependência direta* no *grafo de dependências* do programa através da substituição de uma variável

pela sua expressão correspondente.

Um método simples de substituição é mostrado a seguir: se existirem dois comandos de atribuição S_j e S_k, onde o lado esquerdo de S_j é X o qual é usado no lado direito de S_k, e existe uma *dependência direta* $S_j \delta S_k$, então pode-se substituir o lado direito de S_j no lado direito de S_k. Assim, elimina-se esse arco de dependência do *grafo de dependências.*

Apesar de ser uma técnica simples, para se garantir que o resultado final depois da execução dos comandos está correto é necessária a seguinte definição:

> o comando S_j *domina* o comando S_k se qualquer caminho que vá do início do programa até S_k passe por S_j. Todo o comando *domina* a si próprio e o ponto de entrada *domina* todos os comandos.

Assim, o lado direito de S_j só pode ser substituído no lado direito de S_k se não houver nenhum outro comando S_m entre S_j e S_k, tal que, $S_j \ \overline{\delta} \ S_m$ ou $S_j \ \delta^\circ \ S_m$ e se a execução de S_j *dominar* a execução de S_k.

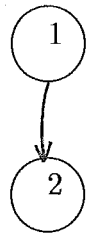
Há situações em que a técnica de *reordenação de comandos* deve ser aplicada antes para que a *substituição para a frente* possa ser feita.

EXEMPLO I:

```
DO    i := 1,IMAX BEGIN
1     K := 10 - i;
2     X(i+2) := P(K) + 1;
END
```

A variável K é usada como índice do vetor P e é calculada no comando anterior provocando uma *dependência direta* que aparece na figura III.9, unindo os dois comandos ($S_1 \delta S_2$). Se a técnica for aplicada o resultado será o seguinte:

```
DO    i := 1,IMAX BEGIN
1     K := 10 - i;
2     X(i+2) := P(10-i) + 1;
END
```



$S_1 \delta_= S_2$

Figura III.9: Grafo de Dependência

A única dependência que existia foi eliminada e o *loop* pode ser vetorizado. O comando S_1 deve ser eliminado (*remoção de código morto* ou *código redundante*).

III.6 Loop Blocking

Esta técnica que cria a partir de um loop simples um aninhamento duplo de *loops* é usada para gerenciar o uso de registradores vetoriais os quais servem para guardar operandos vetoriais. Com a sua aplicação e o aparecimento de outro loop mais interno as operações são realizadas num passo k que é exatamente o tamanho dos registradores vetoriais.

EXEMPLO I:

```
DO    i := 1,N
1     A(i) := B(i) + C(i);
END
```

Se cada registrador vetorial for de tamanho K ($K < N$) o bloco será dividido em dois loops que funcionarão com passo igual a K.

```
DO    i :=   1,N,K
      DO     j := i,MIN (i + K, N)
             A(j) := B(j) + C(j);
      END
END
```

III.7 Fusão de Loops

Um par de *loops* que tenham os mesmos espaços de iteração podem ser agrupados num único *loop* com o propósito de aumentar o número de variáveis referenciadas num mesmo bloco. Isso é importante para reduzir o tráfego entre memória e registradores e para utilizar aproximadamente o número máximo de registradores na execução de cada *loop*. Descobrir uma fusão ótima que minimize o número de *loops* é um problema difícil.

Apesar de não ser uma técnica difícil de ser aplicada, existe uma regra que deve ser obedecida.

> Se, dados dois *loops* L_1 e L_2 com os comandos $S_j \in L_1$ e $S_k \in L_2$, houver alguma relação de dependência do tipo $S_j\gamma(>)S_k$, onde γ representa qualquer uma das três dependências, então a fusão é ilegal.

EXEMPLO I:

```
DO    i := 1,N
1     A(i) := D(i) + 2;
END

DO    i := 1,N
2     B(i) := A(i) + 1;
END
```

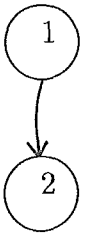
$S_1 \delta_{=} S_2$

Figura III.10: Grafo de Dependência

Na figura III.10 há uma *dependência direta* com *direção* (=) e *sentido para a frente* envolvendo os comandos S_1 e S_2, ($S_1\delta_{=}S_2$). A *fusão dos loops* é legal e o *grafo de dependências* se mantém.

EXEMPLO II:

```
DO    i := 1,N
1     A(i) := D(i) + 2;
END

DO    i := 1,N
2     B(i) := A(i+1) + 1;
END
```



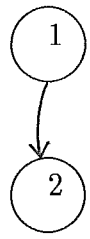
$S_1\delta_{>}S_2$

Figura III.11: Grafo de Dependência

Na figura III.11 existe uma *dependência direta* entre os comandos S_1 e S_2 com *direção* (>), *distância* negativa (-1) e *sentido para a frente* ($S_1\delta(>)S_2$). A *fusão* dos dois loops é ilegal pois o *grafo de dependências* ficaria diferente passando a existir uma *antidependência* do comando S_2 para o comando S_1.

EXEMPLO III:

```
DO    i := 1,N
1     A(i) := B(i) + C(i);
END

DO    i := 1,N
2     D(i) := A(i-1) * 2;
END
```



$S_1 \delta_< S_2$

Figura III.12: Grafo de Dependência

Na figura III.12 existe uma *dependência direta* entre os comandos S_1 e S_2 com *direção* ($<$) e *sentido para frente* ($S_1 \delta_< S_2$), o que não impede que a técnica seja aplicada. O resultado é o seguinte:

```
DO    i := 1,N
1     A(i) := B(i) + C(i);
2     D(i) := A(i-1) * 2;
END
```

Depois da fusão dos dois blocos num só, são feitas apenas N operações de *incremento - teste* em relação ao índice do *loop*. Inicialmente eram feitas 2N operações do mesmo tipo.

III.8 Distribuição do Loop

Esta técnica deve ser aplicada depois que outras técnicas de otimização tenham sido usadas, como por exemplo, reordenação e quebra de comandos, renomeação e expansão de variáveis. Isto porque ela trabalha em cima do *grafo parcialmente ordenado*, formado pelos *pi-blocos*. O *grafo de dependências* já deve ter sido construído, analisado e sofrido algumas modificações resultantes de otimizações. Com isso os *pi-blocos* serão formados por *comandos independentes* e *recorrências*.

Um comando múltiplo *DO*, por exemplo, pode ser quebrado em vários *loops* menores, formados por *comandos independentes* ou por *recorrências*. Como o objetivo principal desta técnica é vetorizar o programa fonte, os *comandos independentes* dão origem a operações vetoriais e as *recorrências* mantém a sua forma

seqüencial original já que são formadas por comandos que não podem ser executados em paralelo.

EXEMPLO I:

```
DO    i := 1,N
1     A(i-1) := K(i-1) + 1;
2     Y(i-1) := A(i-1) + 2;
3     Z(i) := Y(i-1) + V(i);
4     T(i-1) := X(i+1);
5     K(i) := T(i-1) + X(i-1);
6     X(i) := W(i) + 1;
7     W(i+1) := X(i) + 1;
END
```

Este *loop* já sofreu algumas otimizações e se encontra quase na sua forma final, para ser distribuído. O resultado dessa distribuição serão os comandos vetorizados (*comandos independentes*) e os blocos que não podem ser vetorizados (*recorrências*).

No *grafo de dependências* da figura III.13 há um ciclo inquebrável envolvendo os comandos S_6 e S_7. O ciclo é inquebrável pois os *sentidos* das dependências são opostos. Os outros comandos formam unidades independentes podendo ser vetorizados. O bloco inteiro deve ser reordenado para que todas as dependências tenham o mesmo *sentido* (*para a frente*) e a técnica seja aplicada. O resultado obtido é o seguinte:

```
DO    i := 1,N
4     T(i-1) := X(i+1);
6     X(i) := W(i) + 1;
7     W(i+1) := X(i) + 1;
5     K(i) := T(i-1) + X(i-1);
1     A(i-1) := K(i-1) + 1;
2     Y(i-1) := A(i-1) + 2;
3     Z(i) := Y(i-1) + V(i);
END
```

Agora todas as dependências da figura III.14 têm o mesmo *sentido* e a técnica então pode ser aplicada. Os únicos comandos que não podem ser vetorizados são o S_6 e o S_7 porque formam uma *recorrência*. O resultado final é o seguinte:

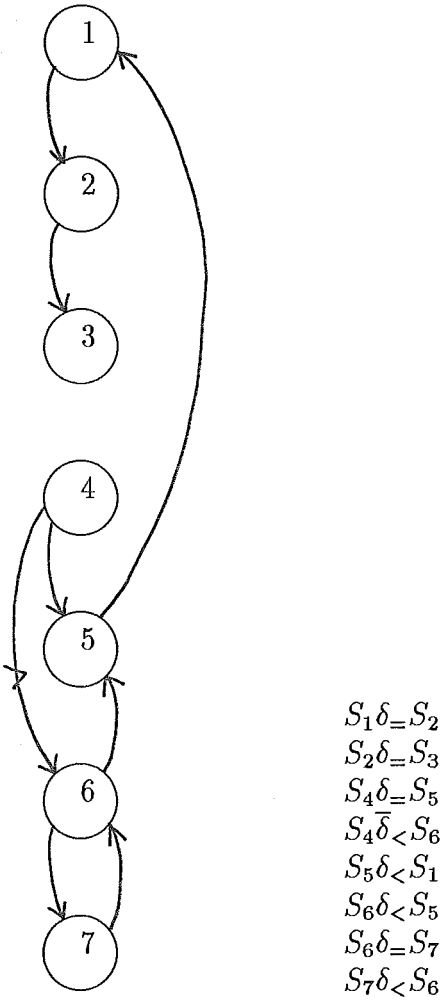


Figura III.13: Grafo de Dependência

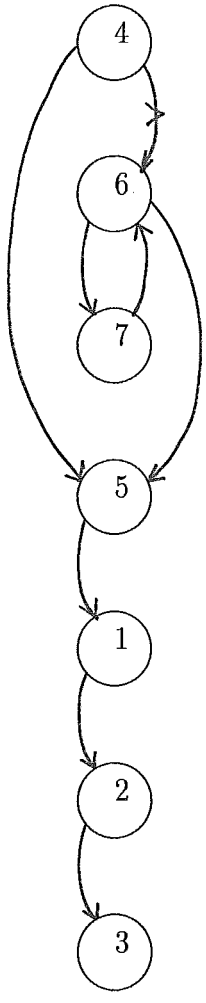


Figura III.14: Grafo de Dependência

```
4       T(0:N-1) := X(2:N+1);
        Bloco 1

DO      i := 1,N BEGIN
6       X(i) := W(i) + 1;
7       W(i+1) := X(i) + 1;
END
        Bloco 2

5       K(1:N) := T(0:N-1) + X(0:N-1);
1       A(0:N-1) := K(0:N-1) + 1;
2       Y(0:N-1) := A(0:N-1) + 2;
3       Z(1:N) := Y(0:N-1) + V(1:N);
        Bloco 3
```

O bloco inicial foi otimizado mas deve-se observar que no final o vetor X é usado tanto de forma seqüencial como vetorial. Isso talvez seja uma restrição para a linguagem Actus, a qual será descrita no capítulo seguinte. A vetorização impõe um paralelismo vetorial entre as iterações e não entre os comandos. Este exemplo é suficiente pois mostra tanto o caso dos *comandos independentes* como das *recorrências.*

III.9 Redução de Ciclo

Em muitos *loops* seriais existem ciclos impossíveis de serem quebrados mesmo aplicando-se todas as técnicas descritas até aqui, por envolverem *dependências diretas.* Esta técnica tem como objetivo tentar extrair algum paralelismo que possa estar implícito nesses *loops* seriais. Para isso continuaremos a usar o *grafo de dependências* e os conceitos de *direção*, *sentido* e de *distância.*

A técnica de *redução de ciclo* é bastante útil em casos de dependência entre comandos com *distância* maior que um, transformando um *loop* serial num aninhamento duplo perfeito: o *loop* mais externo serial e o mais interno paralelo. A idéia é que apesar de haver uma *dependência direta* entre os comandos S_1 e S_2 ($S_1 \delta S_2$) existem instâncias de S_1 e de S_2 que não estão envolvidas na dependência. Isso acontece porque a *distância* é maior que um. A técnica deve extrair dos comandos essas instâncias livres da dependência e criar um *loop* interno paralelo.

A princípio consideram-se os ciclos de dependência onde todas as dependências têm *distâncias* constantes e maiores que um ($\lambda > 1$). Para que as *distâncias* sejam sempre constantes é necessário que os subscritos que fazem referência aos elementos das variáveis vetoriais sejam do tipo ($I \pm a$), onde $a > 0$.

Como resultado final tem-se um *loop* simples com um ciclo de dependência entre seus comandos transformado num aninhamento duplo onde o *loop* mais interno deve ser do tipo *DOALL*. Se as *distâncias* das dependências envolvidas no ciclo forem diferentes a técnica seleciona a menor para garantir que não haverá nenhum conflito.

EXEMPLO I:

```
DO    i := 3,N
1     A(i) := B(i-2) - 1;
2     B(i) := A(i-3) * K;
END
```

$S_1 \delta_< S_2$
$S_2 \delta_< S_1$

Figura III.15: Grafo de Dependência

Na figura III.15 as dependências que envolvem os comandos S_1 e S_2 formam um ciclo inquebrável. A menor *distância* entre as duas dependências é $\lambda = 2$.

```
DO    j := 3,N   ,2
      DOALL  i := j,j+1
1            A(i) := B(i-2) - 1;
2            B(i) := A(i-3) * K;
END
```

III.10 Desdobramento de Loop

Esta técnica é usada para extrair paralelismo de *loops* que devem ser executados na seqüência em que aparecem no programa fonte. Isso se consegue através de uma sobreposição das iterações destes *loops*. Existem dependências entre os *loops* que impedem que a paralelização seja total.

Um *loop* será visto como um *bloco*, o k-ésimo bloco será representado como $\beta_k(*)$ e $\beta_k(i)$ será a i-ésima iteração desse bloco. O espaço iterativo do bloco será de 1 a N_k. A detecção de dependências entre comandos que ficam em *loops* diferentes se faz da mesma maneira como se os comandos pertencessem ao mesmo *loop*.

A aplicação da técnica será inicialmente em dois blocos: B_1 e B_2. A transformação produzirá um novo *loop* serial onde cada passo executará uma iteração de B_1 ($\beta_1(i)$) e uma iteração de B_2 ($\beta_2(i-k)$). O valor de k indica a partir de que iteração do bloco B_2 os dois blocos executarão em paralelo. Este valor deve ser o mínimo dentre os valores encontrados para uma dada dependência. Se entre dois loops existir mais de uma dependência então vários Ks serão encontrados. O valor resultante será o *máximo* dentre os valores encontrados.

A seguir é dada uma demonstração da técnica.

```
DO    I=1, N1
      X(f(I)) := ...
END

DO    I=1, N2
      ...:= X(g(I))
END
```

Com a transformação passa-se a ter:

```
DO    I=1, N1
      COBEGIN
                X(f(I)) := ...
                IF (I > K) THEN ...:=X(g(I-K));
      COEND
END
```

DO I=$N_1 - K + 1, N_2$
...:= X(g(I-K))
END

Se $f(i) = ai + b$ e $g(j) = cj + d$ o problema se reduz a descobrirmos os valores de i_0 e j_0 tais que $f(i_0) = g(j_0)$. Assim o valor de K a ser calculado é dado pela seguinte fórmula:

$$k > \left(\frac{c-a}{c} \right)^+ (N-1) + \frac{c-a+d-b}{c}$$

A definição da fórmula geral x^+ é dada por:

$$x^+ = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Capítulo IV

As Técnicas de Otimização e a Linguagem Actus

IV.1 Características da Linguagem Actus

Actus [12] [13] é uma linguagem de programação paralela síncrona que segue uma filosofia da programação seqüencial: a criação de um programa deve independer da arquitetura da máquina. Ela permite que o programador expresse diretamente o paralelismo e controle o processamento paralelo através de estruturas de contrôle da própria linguagem. Mais especificamente, Actus é uma linguagem Pascal-*like* com estruturas de dados e de contrôle para implementar o paralelismo. Isso foi feito para que a linguagem se orientasse para a programação científica e obtivesse as vantagens dos processadores matriciais e vetoriais.

Actus é diferente das outras linguagens de programação para processadores matriciais e vetoriais no seguinte sentido: se baseia na expressão do paralelismo do problema e apresenta algumas características que facilitam a expressão desse paralelismo.

Esse tipo de computadores paralelos foram desenvolvidos para realizarem a mesma operação num conjunto de dados independentes em paralelo. Por isso, o próprio dado deve indicar a sua *extensão máxima de paralelismo* no momento da sua declaração.

Num processador matricial, a *extensão de paralelismo* indica o número

de processadores que trabalham, logicamente, sobre uma dada estrutura de dados ao mesmo tempo. Essa extensão pode ser maior, menor ou igual ao número de processadores existentes. Num processador vetorial, a *extensão de paralelismo* é o tamanho da estrutura de dados usada pelo processador.

É muito importante entender que a *extensão de paralelismo* é o conceito central da linguagem e que em Actus pode ser controlada através dos *conjuntos de índices* que são formados por valores enumeráveis (inteiros ou caracteres) e identificam os elementos que serão acessados em paralelo na execução de um comando ou de um conjunto de comandos.

A *extensão máxima de paralelismo* é indicada no momento da declaração das variáveis e as únicas estruturas de dados que permitem isso são os vetores e as matrizes.

EXEMPLO:

```
const
        N = 200;
var
        PARALLEL  :  array [1:N] of REAL;
        SCALAR    :  array [1..N] of REAL;
```

O uso de *dois pontos* (:) indica que a *extensão máxima de paralelismo* para o *array* paralelo PARALLEL é N, ou seja, N elementos podem ser acessados em paralelo. Já os elementos do *array* SCALAR só podem ser manipulados um a um.

Num mesmo comando todos os operandos paralelos devem ter a mesma *extensão de paralelismo* pois as operações são aplicadas elemento por elemento, simultaneamente. Isto é, elementos individuais dos *arrays* são emparelhados e o comando é executado. Por exemplo, a expressão

$$A[1{:}50] \star B[1{:}50]$$

faz com que cada elemento de A seja multiplicado pelo elemento correspondente de B, em paralelo.

Uma matriz pode ter N dimensões, mas no máximo duas paralelas. Existem as *constantes paralelas* que são usadas em comandos de atribuição para inicializar vetores paralelos. Elas podem ter qualquer valor inicial, final e incremento constante.

EXEMPLO:

```
const
        N = 200;
parconst
        SEQUENCE  =  1:50;
        PC1       =  100:[-2]84;
```

No exemplo acima a constante paralela SEQUENCE é inicializada com os valores seqüenciais 1, 2, 3, ..., 50 e PC1 é inicializada com nove valores diferentes (100, 98, 96, 94, ..., 86, 84).

A *extensão de paralelismo* pode ser trocada cada vez que a estrutura de dados é usada. Isso é feito dentro do comando *USING* através do uso dos *conjuntos de índices.* Cada comando em Actus que contém variáveis paralelas deve estar associado a um único comando *USING* e a uma única *extensão de paralelismo* que deve ser menor ou igual do que a extensão declarada para as variáveis envolvidas.

EXEMPLO:

```
index
        CONT1 =   3:100;
        CONT2 =   INTEGER;
var
        PARAL     : array [1:N] of REAL;
        VPAR      : array [3:100] of REAL;
        SCALAR    : array [1..N] of REAL;

begin

        using     CONT1
                  PARAL[CONT1] := PARAL[CONT1] * 2;
        using     CONT2 := 1:200
                  VPAR[CONT2] := VPAR[CONT2] + PARAL[CONT2];

end
```

No exemplo acima, existem dois *conjuntos de índices*: o CONT1 que não é redefinível e o CONT2 que é redefinível. Esses *conjuntos de índices* indicam a *extensão de paralelismo* que será usada pelos comandos que se encontram dentro de um bloco inicializado por um USING. Como se pode ver cada USING tem associada uma *extensão de paralelismo* diferente mas sempre dentro dos limites impostos pela *extensão máxima de paralelismo* definida na declaração de cada variável paralela.

Numa linguagem seqüencial, os blocos formados por um comando de *loop* (por exemplo, um DO) são executados seqüencialmente e controlados por um índice que normalmente é usado como subscrito de vetores. Esse índice tem um valor inicial, final e um incremento. Os vetores ou as matrizes têm seus elementos acessados um a um. Em Actus, comandos que se encontram dentro de um bloco inicializado por um *USING* são executados também seqüencialmente, porém não são controlados por nenhum índice e sim por um *conjunto de índices* que indica a *extensão de paralelismo* que atuará sobre os comandos. Não há a idéia de *instâncias* de comandos. As matrizes e os vetores paralelos têm seus elementos, dentro dos limites da *extensão de paralelismo*, acessados em paralelo. Isso não acontece numa linguagem seqüencial.

A linguagem possui também comandos condicionais (*if,else, case*) e de repetição (*for, repeat, while*) tanto seqüenciais como paralelos. A idéia de que os comandos repetitivos podem ser paralelos significa que dentro ou fora do escopo

desses comandos existe um *USING* com índices que indexarão algumas variáveis. Na realidade esses comandos repetitivos são escalares e não têm nenhuma semelhança com comandos do tipo *DOALL* ou *COBEGIN*.

EXEMPLO:

```
index
        I1, I2   = 1:5;
var
        A, B     : array [1:5,1:5] of INTEGER;

begin
        using    I1, I2 do
                 while A[I1,I2] > 0 do
                 begin
                 B[I1,I2] := B[I1,I2] + B[I1,I2] * A[I1,I2];
                 A[I1,I2] := A[I1,I2] - 4;
                 end;
end;
```

O exemplo acima tem o efeito de, repetidamente, executar o corpo do comando *while* enquanto pelo menos um dos elementos de A for maior que zero. A *extensão de paralelismo* antes da primeira avaliação da expressão boleana é 1:5,1:5. A cada iteração do corpo do *loop* os elementos de A que têm valor menor ou igual a zero são retirados da *extensão de paralelismo*. A execução do *loop* só terminará quando a *extensão de paralelismo* estiver vazia. Neste exemplo pode-se perceber a diferença entre o comando *while* paralelo e o seqüencial que é igual ao de todas as linguagens seqüenciais conhecidas.

EXEMPLO:

```
index
        I, J    = 1:50;
var
        A, B    : array [1:50,1:50] of REAL;

begin
        using   I, J do
                if A[I,J] > 0.0 then
                B[I,J] := B[I,J] - 0.5;
                else
                B[I,J] := B[I,J] + 0.5;
end;
```

Aqui os elementos da matriz B para os quais os elementos correspondentes de A têm valores positivos (se houver algum) serão decrementados de 0.5 e os elementos restantes (se houver algum) serão incrementados de 0.5. Assim a execução do comando IF paralelo é bastante diferente da execução do mesmo comando seqüencial: tanto a parte do *then* como do *else* são executadas não havendo exclusão entre elas.

Como já se pôde observar a linguagem Actus tem características próprias bem diferentes das linguagens seqüenciais, para as quais foram criadas as técnicas de otimização já descritas. Na linguagem Actus os blocos que contém os comandos podem ser de três tipos diferentes:

1- blocos do tipo *USING*;

Exemplo:

```
USING   I := 1:N
        X[I] := J[I] * 2;
```

2- blocos repetitivos seqüenciais (sem a presença do comando *USING*);

Exemplo:

```
FOR   K := 1,N DO
      A[K] := Y[K + 1];
```

3- blocos repetitivos paralelos (com um comando *USING* dentro ou fora do seu escopo);

Exemplo:

```
FOR   J := 1:N
      USING    I := J:N
               X[I] := K[I] + X[I shift 1];
```

A divisão dos blocos em três grupos tem como objetivo facilitar a comparação entre linguagens seqüenciais e a linguagem Actus. Assim quando as técnicas são aplicadas sobre comandos que estão num bloco do grupo 1 deve-se prestar muita atenção na *extensão de paralelismo* e no uso das variáveis antes e depois do local da otimização; se os comandos estão num bloco do grupo 2 é como se se tentasse otimizar um bloco dentro de uma linguagem seqüencial; se os comandos estão num bloco do grupo 3, além da *extensão de paralelismo*, deve-se prestar atenção nas dependências que são causadas pelas várias iterações.

As técnicas são aplicadas para se detectarem dependências entre blocos pois o principal objetivo da otimização é descobrir blocos que podem ser executados em paralelo. Um segundo objetivo da otimização é encontrar blocos que possuam comandos que podem ser vetorizados (usando-se as prerrogativas da própria linguagem) e passaram despercebidos pelo programador.

Como a linguagem tem uma característica de paralelismo tão vantajosa a aplicação das técnicas sofre muitas mudanças inclusive com restrições em algumas delas. Um outro detalhe da linguagem é não conter primitivas de sincronização já que processadores vetoriais e matriciais não têm esse problema tão comum em arquiteturas formadas por processadores independentes. Como se deseja paralelizar blocos e distribuí-los numa rede de processadores foram criadas, para este trabalho, as seguintes notações, onde o uso de ( ) indica um bloco de comandos:

1- SEQ: indica a execução seqüencial de vários comandos;

Exemplo:
(1 **SEQ** 2 **SEQ** 3) → os comandos 1, 2, 3 são executados seqüencialmente;

(1 **SEQ** 2) **SEQ** (3 **SEQ** 4) → os quatro comandos são seqüenciais e pertencem a dois blocos diferentes;

2- PAR: indica a execução paralela de comandos pertencentes a blocos diferentes;

Exemplo:
(1) **PAR** (2 **SEQ** 3) → o comando 1 (*bloco 1*) é executado em paralelo com os comandos 2 e 3 (*bloco 2*), estes seqüenciais;

3- SINC: dispara a execução de um determinado conjunto de comandos após a execução do último comando antes da primitiva; o conjunto disparado será executado em paralelo com os comandos seqüenciais a esse último comando;

Exemplo:
(4) (**SINC** (6 **SEQ** 7 **SEQ** 8)) **SEQ** 1 **SEQ** 2 **SEQ** 3 → o comando 4 (*bloco 1*) dispara a execução dos comandos seqüenciais 6,7,8 (*bloco 2*) e executa seqüencialmente com os comandos 1,2,3 (*bloco 1*); os comandos 6,7,8 executam em paralelo com os comandos 1,2,3;

Um aspecto que deve ser atentamente observado quando se deseja aplicar alguma *técnica de otimização* é a *extensão de paralelismo* que envolve as variáveis paralelas nos blocos. O *teste de dependência de dados* continua a ser feito da mesma forma, mas a *análise global* sofre algumas modificações, devido à própria

estrutura da linguagem. No caso de blocos que pertencem ao grupo 1, uma *região fortemente conectada* não será formada por um ciclo. Basta que exista pelo menos uma dependência unindo dois comandos para que eles já não possam ser executados em paralelo. Isso é conseqüência do fato de não existirem dependências com *sentido para trás* e de se querer paralelizar esses comandos. No caso de blocos que pertencem aos grupos 2 e 3, as definições iniciais de *direção, sentido* e *distância* continuam valendo. Para a aplicação das técnicas assume-se que o *grafo de dependências* com todas as informações necessárias sobre o programa já foi construído pelo compilador.

Para cada uma das técnicas apresentadas neste capítulo foi construído um algoritmo correspondente. Os algoritmos se encontram num nível alto pois o objetivo é dar uma idéia de como um otimizador pode ser construído. No apêndice foram colocadas as definições da pseudo-linguagem criada para se escreverem os algoritmos.

IV.2 Renomeação de Variáveis

Devido à sua simplicidade e ao seu objetivo, acabar com falsas dependências e tornar o programa mais legível, não há necessidade de se fazer alguma alteração para se usar esta técnica em programas escritos na linguagem Actus. Podemos aplicá-la tanto em variáveis escalares como em vetores (seqüenciais ou paralelos) apesar de o primeiro caso ser mais simples.

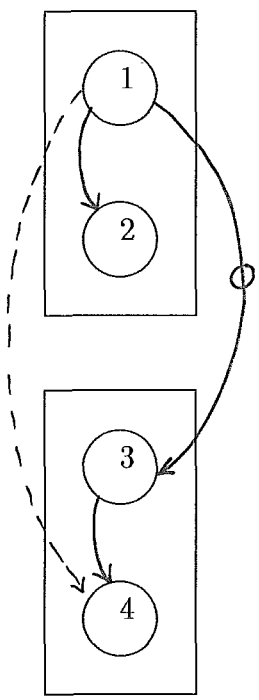
ALGORITMO:

Algoritmo: renomeação de variáveis;
Entrada: GD, programa P, variável A;

início
se ((A = escalar) *ou* ((A $\neq$ escalar) *e*
(ext-par-bloco(origem) = ext-par-bloco(destino)))
então faça
descobrir no GD todas as *ds* que envolvem A;
se (num(ds) = 0) *então* faça
descobrir no GD as *dd* que envolvem A;
se (num(dd) = 0) *então* faça
se (num(ad) $\neq$ 0) *então* faça
substituir no comando(origem) A por A_0;
fazer isto para todos os comando(origem) cujo comando(destino)
seja o mesmo;
senão (não faça nada);
senão faça
substituir A por A_i (i=1) no comando(origem) e no
comando(destino) de cada *dd*;
se (num(ad) $\neq$ 0) *então* faça
substituir no comando(origem) A por A_0;
fazer isto para todos os comando(origem) cujo
comando(destino) seja o mesmo;
senão (não faça nada);
fim-se;
senão faça
num(ds) = n;
para i = 1, ..., n+1 faça
substituir A por A_i, t.q., A $\in$ OUT(cmd);
substituir A por A_i em comando(destino) de *dd*, t.q.
comando(origem) = cmd;
se ($\exists$ *ad* cujo comando(destino) = comando(origem)
da primeira *ds*) *então* faça
substituir A por A_0 no comando(origem) das *ad*;
fim-se;
senão (não faça nada);
fim-se;
final;

EXEMPLO I:

```
USING   m := 2:MLIM DO
1       EM := X - 1;
2       U[m] := Z[m] * EM;
END
        Bloco 1

USING   n := 1:NX DO
3       EM := WGHT * (X - 1);
4       POLY[n] := Z[n] * EM;
END
        Bloco 2
```



$S_1 \delta S_2$
$S_1 \delta^{\circ} S_3$
$S_3 \delta S_4$

Figura IV.1: Grafo de Dependências

O objetivo ao se analisar o *grafo de dependências* é descobrir falsas *dependências de saída* e *antidependências*, causadas pelo uso da mesma variável em pontos diferentes do programa.

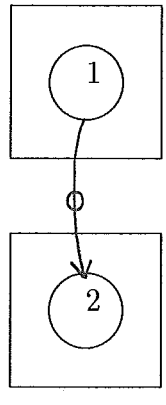
Neste exemplo, como se pode ver no grafo da figura IV.1, existe uma *dependência de saída* entre os comandos S_1 e S_3 (S_1 δ° S_3) causada pelo uso da mesma variável escalar *EM* nos dois blocos e uma falsa *dependência direta* entre os comandos S_1 e S_4. Na verdade o comando S_4 depende da execução do comando S_3. Essa *dependência de saída* não permite que os dois blocos executem em paralelo pois ambos escrevem na mesma posição de memória.

```
USING   m := 2:MLIM DO
1       EM := X - 1;
2       U[m] := Z[m] * EM;
END
        Bloco 1

USING   n := 1:NX DO
3       EN := WGHT * (X - 1);
4       POLY[n] := Z[n] * EN;
END
        Bloco 2
```

Os dois blocos podem executar em paralelo pois não existem mais dependências entre eles. A execução dos comandos será a seguinte: [(1 SEQ 2) PAR (3 SEQ 4)

EXEMPLO II:

```
USING   m := 1:400
1       X[m] := Y[m] + R[m];

        Bloco 1

USING   n := 1:200
2       X[n] := U[n] + A[n];

        Bloco 2
```



$S_1 \delta^{\circ} S_2$

Figura IV.2: Grafo de Dependências

Inicialmente, os dois blocos são executados seqüencialmente e existe uma *dependência de saída* envolvendo os comandos S_1 e S_2 (figura IV.2). Os dois blo-

cos possuem diferentes *extensões de paralelismo* o que faz com que essa dependência não envolva todos os elementos. A técnica de *renomeação* pode ser aplicada.

```
USING   m := 1:400
1       X[m] := Y[m] + R[m];

        Bloco 1

USING   n := 1:200
2       XM[n] := U[n] + A[n];

        Bloco 2
```

A única dependência que existia foi eliminada, o que significa que os blocos podem executar em paralelo: [(1) PAR (2)]. Deve-se observar que não há muito sentido em se aplicar esta técnica em vetores que são recalculados em diferentes pontos do programa. Só há utilidade se for usada toda a *extensão de paralelismo* correspondente ao vetor, como pode ser visto no exemplo a seguir. Neste exemplo, o valor final dos primeiros 200 elementos do vetor X se encontra na verdade em XM.

EXEMPLO III:

```
USING   m := 1:400
1       X[m] := Y[m] + R[m];
2       C[m] := X[m] + Y[m];

        Bloco 1

USING   n := 1:200
3       Z[n] := X[n] + A[n];

        Bloco 2

USING   p := 1:400
4       X[p] := H[p] * 10;

        Bloco 3
```

Neste exemplo, o vetor X é calculado, em toda a sua *extensão de paralelismo*, pelo bloco 1 e recalculado pelo bloco 3, também em toda a sua *extensão*

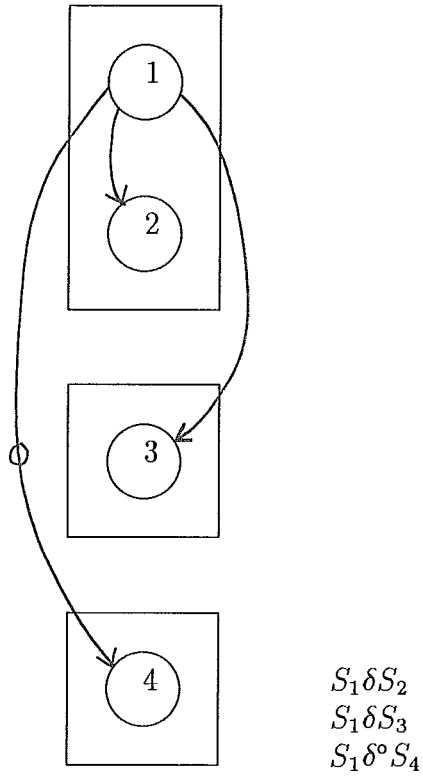


Figura IV.3: Grafo de Dependências

de paralelismo. Por isso aparece no grafo da figura IV.3 uma *dependência de saída* unindo os dois blocos.

Então, no bloco 4, o vetor X pode ser renomeado assim como em todos os pontos do programa em que ele é recalculado (em toda a sua extensão). Isso deve eliminar algumas dependências do grafo.

```
USING   m := 1:400
1       X[m] := Y[m] + R[m];
2       C[m] := X[m] + Y[m];

        Bloco 1

USING   n := 1:200
3       Z[n] := X[n] + A[n];

        Bloco 2

USING   p := 1:400
4       XM[p] := H[p] * 10;

        Bloco 3
```

Agora o bloco 4 é totalmente independente dos outros blocos podendo ser executado concorrentemente com eles (figura IV.4). Neste caso não existe o problema do exemplo anterior pois foi usada toda a *extensão de paralelismo* referente ao vetor.

IV.3 Quebra de Comando

A idéia de se aplicar esta técnica na linguagem Actus é um pouco diferente da idéia inicial. Agora o objetivo principal é quebrar a *extensão de paralelismo* de blocos que estão dentro de um comando *USING* para torná-los paralelos ou para que outras técnicas possam ser aplicadas posteriormente. Para se quebrar a *extensão de paralelismo* deve-se observar o *grafo de dependências* e a interseção das *extensões de paralelismo* entre os blocos em questão. Essa técnica é mais usada então em blocos pertencentes ao grupo 1.

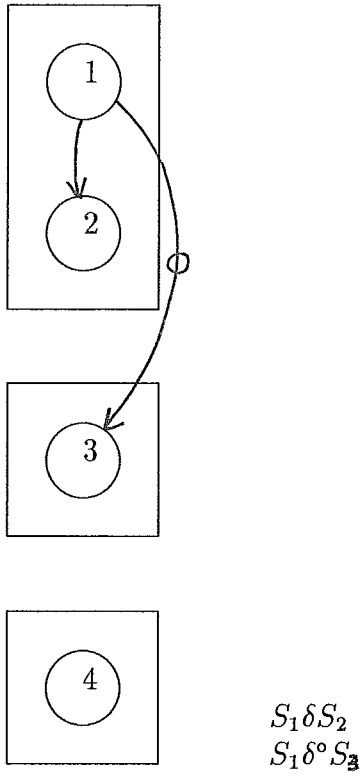


Figura IV.4: Grafo de Dependências

Se um bloco inicializado pelo comando *USING* for visto como um comando a técnica também pode ser aplicada para separar comandos independentes dentro do mesmo bloco. Esses comandos podem ser executados em paralelo com o bloco inicial.

No caso de blocos que pertençam ao grupo 2 a técnica se aplica da mesma forma que no caso de linguagens não vetoriais. No final o bloco é vetorizado usando-se as vantagens da linguagem Actus.

ALGORITMO:

Algoritmo: quebra de comando;
Entrada: GD, programa P;

início;
descobrir no GD todas as depêndencias entre blocos que envolvem vetores ou arrays; (2 a 2)
se (ext-par-bloco(origem) = ext-par-bloco(destino)) *então* (não faça nada);
senão faça
mantém ext-par-bloco(menor);
cria novo comando de loop;
repete comandos dentro de loop;
se ($\exists$ *shifts*) *então* faça
ext-par-bloco(maior) = ext-par-bloco(menor);
ext-par-bloco(novo) = ext-par-bloco(maior)-ext-par-bloco(menor);
sai;
senão faça
ext-par-bloco(maior) = faixa de elementos acessados no bloco(menor) pela variável com menor (*shift*);
ext-par-bloco(novo) = faixa de elementos acessados que sobraram;
sai;
fim-se;
fim-se;
final;

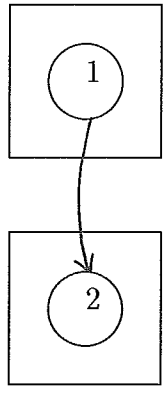
EXEMPLO I:

USING k := 1:400
1 X[k] := Q $\star$ Y[k] $\star$ ...;

Bloco 1

USING k := 1:[5]996
2 Q_1[k] := Z[k] $\star$ X[k] + Z[k shift 1] $\star$ X[k shift 1]

Bloco 2



$S_1 \delta S_2$

Figura IV.5: Grafo de Dependências

No *grafo de dependências* da figura IV.5 existe uma *dependência direta* entre os dois blocos, mas a *interseção* das *extensões de paralelismo* mostra que essa dependência só é verdadeira para os primeiros quatrocentos elementos do vetor X. Para os restantes elementos a dependência é falsa.

Como não existe nenhuma dependência com *sentido para trás* dentro do próprio comando S_2 a técnica pode ser aplicada.

USING k := 1:400
1 X[k] := Q ⋆ Y[k] ⋆ ...;

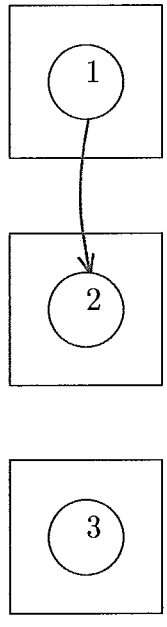
Bloco 1

USING k_1 := 1:[5]396
2 $Q_1[k_1]$:= Z[k_1] ⋆ X[k_1] +
Z[k_1 shift 1] ⋆ X[k_1 shift 1]

Bloco 2

USING k_2 := 401:[5]996
3 $Q_1[k_2]$:= Z[k_2] ⋆ X[k_2] +
Z[k_2 shift 1] ⋆ X[k_2 shift 1]

Bloco 3



$S_1 \delta S_2$

Figura IV.6: Grafo de Dependências

No *grafo de dependência* da figura IV.6 a *dependência direta* só envolve os elementos de X que pertencem à interseção entre as *extensões de paralelismo* dos blocos 1 e 2. Com isso o bloco 3 ficou totalmente independente e pode ser executado em paralelo com qualquer um dos outros dois. A execução será:
[(1 PAR 3) SEQ 2 ou (1 SEQ 2) PAR 3].

O objetivo da técnica continua sendo eliminar falsas dependências para se poder otimizar blocos, só que agora essas dependências podem ser de qual-

quer tipo e deve-se fazer a quebra sobre a *extensão de paralelismo*. Isso quando se tratarem de blocos pertencentes ao grupo 1.

IV.4 Reordenação de Comandos

No caso desta *técnica de otimização* a idéia original de sua aplicação se mantém, porém com algumas modificações, na linguagem Actus.

É o caso dos blocos pertencentes ao grupo 1. Não existe nenhuma dependência entre comandos ou entre estes blocos com *sentido para trás*. Assim, o objetivo neste caso é apenas diminuir o tempo de espera de comunicação entre os blocos, quando estes forem executados em paralelo. Isso é feito também com a introdução da notação SINC. Não existe nenhuma restrição quanto às *extensões de paralelismo* a não ser que sua interseção deve ser diferente de vazio.

Para os blocos pertencentes ao grupo 2 os objetivos são resolver o problema da comunicação e agora também mudar o sentido das dependências (*para trás*) para que os blocos possam ser vetorizados.

ALGORITMO:

Algoritmo: reordenação de comandos;
Entrada: GD;

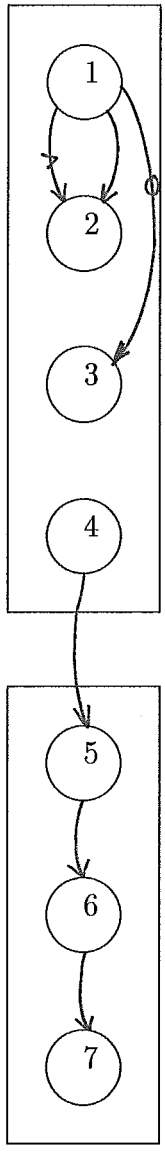
início;
descobrir no GD todas as dependência-sentido(PT); {dentro do mesmo bloco}
caso bloco(tipo)
tipo-1: *início*
para (todas as dependência-sentido(PT)) faça
se ($\nexists$ dependência-sentido(PT)) *então*
(não faça nada);
se (dependência-sentido(PT) $\in$ ciclo) *então*
(não faça nada):
reordena comando(origem) e comando(destino);
fim-início;

tipo-3: *início*
(não faça nada);
fim-início;
fim-caso;

descobrir no GD todas as dependências entre blocos;
se ($\nexists$ dependência entre blocos) *então* (não faça nada);
se (bloco(tipo) = tipo-2 *ou* bloco(tipo) = tipo-3) *então* (não faça nada);
reordena em bloco(origem) o comando(origem) afastando-o ao máximo do comando(destino);
faz o mesmo para bloco(destino) e comando(destino);
final;

EXEMPLO I:

```
USING   k := 1:100 DO
1       A[k] := B[k] - Z[k];
2       B[k] := 2 * A[k] + C[k];
3       A[k] := A[k] + Z[k];
4       D[k] := D[k] * C[k];
END
        Bloco 1

USING   j := 1:100 DO
5       D[j] := D[j] * 5;
6       E[j] := E[j] + D[j];
7       F[j] := F[j] * E[j];
END
        Bloco 2
```



$S_1 \overline{\delta} S_2$
$S_1 \delta S_2$
$S_1 \delta^{\circ} S_3$
$S_4 \delta S_5$
$S_5 \delta S_6$
$S_6 \delta S_7$

Figura IV.7: Grafo de Dependências

Entre os dois blocos da figura IV.7 só existe uma dependência , a qual é causada pelo uso do vetor D. Se os dois blocos fossem executados em paralelo na ordem em que se encontram os comandos, com alguma sincronização, não haveria ganho nenhum porque o último comando do bloco 1 produz o valor do vetor D que será usado pelo bloco 2. Aplicando a técnica de *reordenação de comandos* o resultado é o seguinte:

```
USING   k := 1:100 DO
4       D[k] := D[k] * C[k];
1       A[k] := B[k] - Z[k];
2       B[k] := 2 * A[k] + C[k];
3       A[k] := A[k] + Z[k];
END
        Bloco 1

USING   j := 1:100 DO
5       D[j] := D[j] * 5;
6       E[j] := E[j] + D[j];
7       F[j] := F[j] * E[j];
END
        Bloco 2
```

O *grafo de dependências* resultante da otimização (figura IV.8) não se altera, em relação ao da figura IV.7, o que já era esperado, pois as dependências continuam existindo, entre os mesmos comandos. O que se verifica é que agora depois que o bloco 1 calcula o valor do vetor D o bloco 2 pode ser disparado. Os dois blocos executarão em paralelo da seguinte maneira:
[4 (SINC (5 SEQ 6 SEQ 7)) SEQ (1 SEQ 2 SEQ 3)].

IV.5 Expansão de Variáveis Escalares

Na linguagem Actus esta técnica é útil apenas quando aplicada em blocos que pertencem ao grupo 2, pois esses blocos são semelhantes aos descritos para as linguagens seqüenciais. Para blocos pertencentes aos grupos 1 e 3 não faz muito sentido a aplicação da técnica. No primeiro caso, porque não existe um *loop* nem a noção de espaço de iteração. No segundo caso porque, mesmo havendo um *loop*, este já se encontra vetorizado.

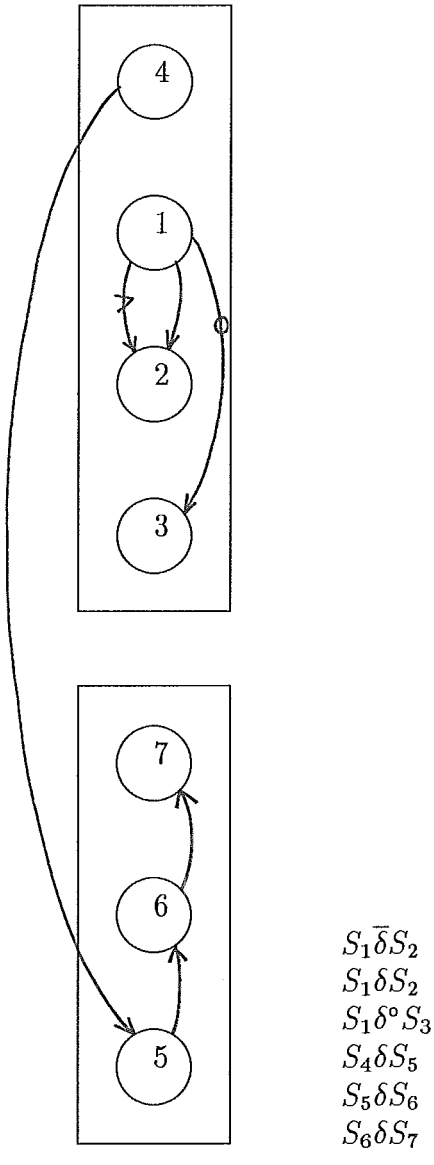


Figura IV.8: Grafo de Dependências

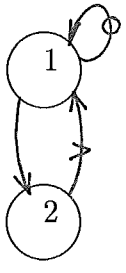
ALGORITMO:

Algoritmo: expansão de variáveis escalares;
Entrada: GD;

início;
descobrir no GD todas as dependências que envolvem variáveis escala
dentro do mesmo bloco);
substituir cada variável escalar (no comando(origem)) por vetor com
mesmo nome;
indexar vetor com o mesmo índice do loop;
fazer o mesmo no comando(destino);
final;

EXEMPLO I:

```
FOR   m := 1,400 DO
1     EM := X[m] - 1;
2     U[m] := Z[m] ⋆ EM;
END
```



$S_1 \delta S_2$
$S_2 \overline{\delta} S_1$

Figura IV.9: Grafo de Dependências

Há uma *dependência de saída* do comando 1 em relação a si mesmo pois ele sempre escreve na mesma posição de memória (a variável *EM*). Aparece, no grafo da figura IV.9, um ciclo entre os comandos S_1 e S_2 causado também pelo uso da variável *EM*. A *dependência direta* ocorre porque, na mesma iteração, o comando S_1 calcula o valor de *EM* e o comando S_2 usa esse valor e a *antidependência* é porque o comando S_2 usa o valor de *EM* o qual será modificado pelo comando S_1 na próxima iteração. A presença do ciclo impede a vetorização. Com a técnica o resultado é o seguinte:

```
FOR   m := 1,400 DO
1     EM[m] := X[m] - 1;
2     U[m] := Z[m] * EM[m];
END
```

O *grafo de dependências* fica mais simples com apenas uma *dependência direta* que não pode ser eliminada. Assim o bloco pode ser vetorizado.

```
USING   m := 1:400 DO
1       EM[m] := X[m] - 1;
2       U[m] := Z[m] * EM[m];
END
```

A aplicação da técnica otimizou bastante o bloco inicial. Este é um exemplo que poderia passar desapercebido pelo programador.

IV.6 Substituição para a Frente de Comandos

A idéia principal da técnica é eliminar *dependências diretas* do *grafo de dependências.* No caso de programas escritos em Actus devemos manter as restrições iniciais de *dominadores.*A substituição tanto pode ser feita dentro do mesmo bloco ou entre blocos visando a vetorização de comandos e a paralelização de blocos.

A aplicação da técnica num bloco do grupo 1 quando o *grafo de dependências* mostrar que isso é permitido, não terá nenhum problema pois dentro do bloco a *extensão de paralelismo* não muda.

Se a técnica for aplicada em comandos que se encontram dentro de um bloco do grupo 2 nenhuma alteração precisa ser feita pois é como se esse bloco pertencesse a uma linguagem seqüencial.

Quando a substituição desejada é entre blocos a *extensão de paralelismo* de ambos os blocos tem de ser comparada, assim como as dependências causadas pelos comandos envolvidos. Deve-se verificar também se há necessidade de se aplicar antes as técnicas de *quebra de comando* ou de *reordenação de comandos.* A primeira, para quebrar a *extensão de paralelismo* no caso de haver alguma diferença e a segunda para mudar o *sentido* de algumas dependências que aparecem

no *grafo de dependências* e que impediriam a aplicação da técnica. A reordenação pode ser feita mesmo no caso dos comandos pertencerem a blocos diferentes (ver a seção *reordenação de comandos*).

As variáveis envolvidas na substituição tanto podem ser escalares como vetores sendo que, o primeiro caso, é bem mais fácil de tratar.

ALGORITMO:

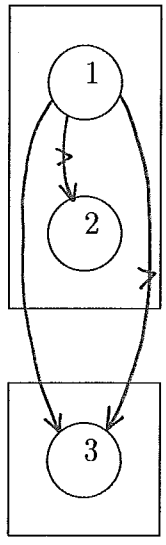
Algoritmo: substituição para a frente de comandos;
Entrada: GD;

início;
descobrir no GD todas as dd(PF);
para (cada dd(PF)) faça
se (∃ entre comando(origem) (O) e comando(destino(D) algum comando (M) tal que O e M sejam unidos por uma ad(PF,=) ou por uma ds(PF,=)) *então*
(não faça nada);
substituir no comando(destino) a variável pelo lado direito do comando(origem);
fim-para;
final;

EXEMPLO I:

```
USING   i := 1:N DO
1       A[i] := C[i] + B[i];
2       C[i] := E[i];
END
                Bloco 1

USING   j := 1:N DO
3       B[j shift 1] := A[j] + 2;
END
                Bloco 2
```

A *substituição de comando* deve ser aplicada, na figura IV.10, entre os comandos S_1 e S_3 envolvendo o vetor A. Para que seja possível a aplicação da técnica o comando S_1 tem de *dominar* o comando S_3 e não pode existir nenhum comando S_2 com uma relação de *antidependência* vinda do comando S_1. Neste caso a segunda hipótese não é verdadeira e se a substituição for feita o resultado final será diferente. A técnica de *reordenação de comandos* deve ser usada para mudar a



$S_1 \overline{\delta} S_2$
$S_1 \delta S_3$
$S_1 \overline{\delta} S_3$

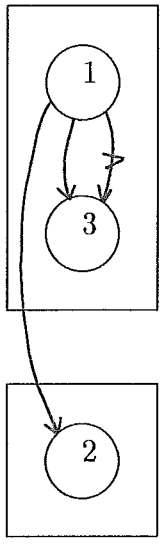
Figura IV.10: Grafo de Dependências

ordem de execução dos comandos S_2 e S_3.

```
USING   i := 1:N DO
1       A[i] := C[i] + B[i];
3       B[i shift 1] := A[i] + 2;
END
        Bloco 1

USING   j := 1:N DO
2       C[j] := E[j];
END
        Bloco 2
```

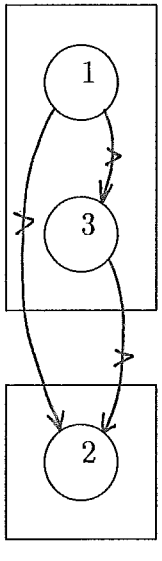
Com o grafo da figura IV.11 a *substituição de comandos* pode ser feita.

```
USING   i := 1:N DO
1       A[i] := C[i] + B[i];
3       B[i shift 1] := (C[i] + B[i]) + 2;
END
        Bloco 1

USING   j := 1:N DO
2       C[j] := E[j];
END
        Bloco 2
```



$S_1 \overline{\delta} S_2$
$S_1 \delta S_3$
$S_1 \overline{\delta} S_3$

Figura IV.11: Grafo de Dependências



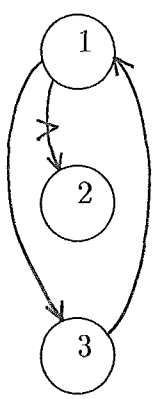
$S_1 \overline{\delta} S_2$
$S_1 \overline{\delta} S_3$
$S_3 \overline{\delta} S_2$

Figura IV.12: Grafo de Dependências

Antes de ser aplicada a técnica de *substituição de comandos* podia-se explorar algum paralelismo usando a sincronização: quando o comando S_1 tivesse terminado de calcular o vetor A o comando S_3 pertencente ao bloco 2 podia começar a sua execução. Isso ocorreria paralelamente à execução do comando S_2 pertencente ao bloco 1. Depois da *substituição* (figura IV.12) isso não pode mais ser feito pois os dois comandos do bloco 1 utilizam valores antigos do vetor C. Esta otimização perdeu o paralelismo inicial que havia entre os dois blocos, não havendo nenhuma vantagem na aplicação da técnica, neste exemplo.

EXEMPLO II:

```
FOR   i := 1 TO N DO
1       A[i] := C[i] + B[i];
2       C[i] := E[i];
3       B[i + 1] := A[i] + 2;
END
```



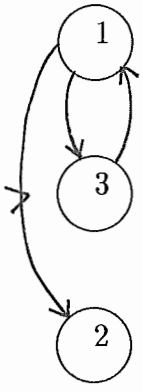
$S_1 \overline{\delta} S_2$
$S_1 \delta S_3$
$S_3 \delta S_1$

Figura IV.13: Grafo de Dependências

Neste exemplo o bloco pertence ao grupo 2, ou seja, é *seqüencial.* Assim, o objetivo é vetorizá-lo assumindo que o programador não percebeu o paralelismo existente.

Como há uma *antidependência* entre os comandos S_1 e S_2 (figura IV.13), a substituição do comando S_1 no comando S_3 não pode ser feita imediatamente. Antes a técnica de *reordenação de comandos* deve ser usada, entre os comandos S_2 e S_3.

```
FOR   i := 1 TO N DO
1       A[i] := C[i] + B[i];
3       B[i + 1] := A[i] + 2;
2       C[i] := E[i];
END
```

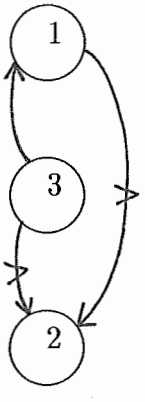


$S_1 \overline{\delta} S_2$
$S_1 \delta S_3$
$S_3 \delta S_1$

Figura IV.14: Grafo de Dependências

Agora que as regras para a *substituição* estão satisfeitas a técnica pode ser aplicada (figura IV.14).

```
FOR   i := 1 TO N DO
1       A[i] := C[i] + B[i];
3       B[i + 1] := (C[i] + B[i]) + 2;
2       C[i] := E[i];
END
```

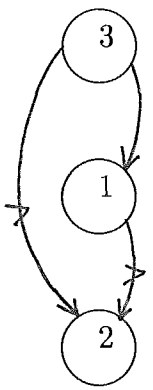


$S_3 \overline{\delta} S_2$
$S_3 \delta S_1$
$S_1 \overline{\delta} S_2$

Figura IV.15: Grafo de Dependências

Depois da otimização, o *grafo de dependências* possui uma dependência com *sentido para trás*, do comando S_3 para o S_1 (figura IV.15). Por isso a técnica de *reordenação de comandos* deve ser usada novamente.

```
FOR   i := 1 TO N DO
3     B[i + 1] := (C[i] + B[i]) + 2:
1     A[i] := C[i] + B[i];
2     C[i] := E[i];
END
```



$S_3 \overline{\delta} S_2$
$S_3 \delta S_1$
$S_1 \overline{\delta} S_2$

Figura IV.16: Grafo de Dependências

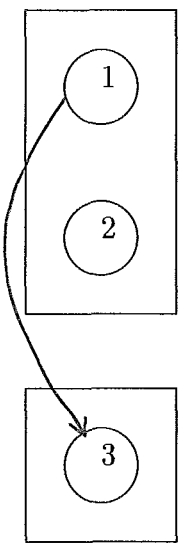
Todas as dependências possuem agora o mesmo *sentido* tornando possível a vetorização do bloco (figura IV.16).

```
USING   i := 1:N DO
3       B[i shift 1] := (C[i] + B[i]) + 2;
1       A[i] := C[i] + B[i];
2       C[i] := E[i];
END
```

Uma coisa que deve ser vista é que o bloco resultante não pertence mais ao grupo 2 e sim ao grupo 1 e as variáveis que eram seqüenciais se tornaram paralelas. O uso mais adiante destas variáveis deve ser observado para que não existam possíveis conflitos.

EXEMPLO III:

```
USING   i := 1:N DO
1       A[i] := Z[i] + W[i];
2       C[i]:= E[i];
END
        Bloco 1

USING   j := 1:N DO
3       B[j] := A[j] + 2
END
        Bloco 2
```



$S_1 \delta S_3$

Figura IV.17: Grafo de Dependências

A *substituição dos comandos* deve ocorrer entre os comandos S_1 e S_3, e analisando o *grafo de dependências* da figura IV.17 verifica-se que não há nenhuma restrição para a aplicação da técnica pois a única dependência existente envolve os comandos S_1 e S_3.

```
USING   i := 1:N DO
1       A[i] := Z[i] + W[i];
2       C[i]: E[i];
END
        Bloco 1

USING   j := 1:N DO
3       B[j] := (Z[j] + W[j]) + 2
END
        Bloco 2
```

Todas as dependências foram eliminadas. Antes da técnica ser aplicada os dois blocos podiam ser executados em paralelo, com algumas restrições de sincronização e agora podem ser executados em paralelo sem nenhuma restrição. Se os blocos envolvidos não tiverem a mesma *extensão de paralelismo* a técnica de *quebra de comando* deve ser usada antes.

EXEMPLO IV:

```
USING   k := 1:400 DO
1       X[k] := Y[k] * (R[k] + Z[k shift 1]);

        Bloco 1

USING   m := 1:120 DO
2       X[m] := U[m] * (R[m] + Z[m]);

        Bloco 2

USING   l := 1:300 DO
3       W[l] := X[l] + A[l shift 2];

        Bloco 3
```

Inicialmente os três blocos são executados seqüencialmente e possuem *extensões de paralelismo* diferentes. A princípio não há nenhuma restrição no *grafo de dependências* (figura IV.18) para a aplicação da técnica. Existe uma *dependência de saída* envolvendo os comandos S_1 e S_2 e algumas falsas dependências. A diferença neste exemplo é que tratam-se de vetores paralelos com *extensões de paralelismo* diferentes de bloco para bloco. Por isso nem todas as dependências que parecem falsas são realmente falsas, o que pode ser visto no *grafo de dependências.* Não existe, por exemplo, a *dependência direta* entre os comandos S_1 e S_3 nos 120 primeiros elementos já que estes são novamente calculados no comando S_2. O mesmo acontece com os blocos 2 e 3 em relação aos últimos 180 elementos que são calculados pelo comando S_1 e com os blocos 1 e 2 (a *dependência de saída* só existe para os 120 primeiros elementos).

Para se aplicar a técnica da *substituição para a frente* deve-se antes usar a técnica de *quebra de comando* para se ajustarem as *extensões de paralelismo.*

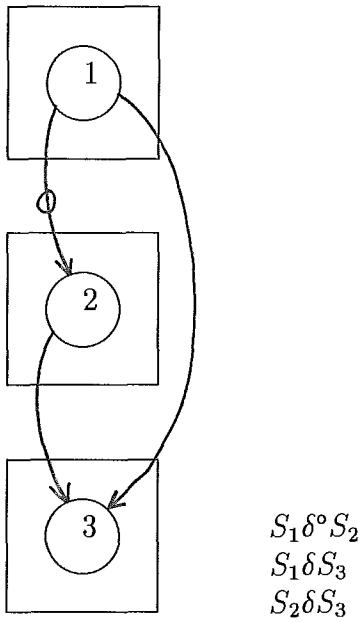


Figura IV.18: Grafo de Dependências

```
USING   k1 := 1:120 DO
1       X[k1] := Y[k1] * (R[k1] + Z[k1 shift 1]);

        Bloco 1a

USING   k2 := 121:400 DO
2       X[k2] := Y[k2] * (R[k2] + Z[k2 shift 1]);

        Bloco 1

USING   m := 1:120 DO
3       X[m] := U[m] * (R[m] + Z[m]);

        Bloco 2

USING   l1 := 1:120 DO
4       W[l1] := X[l1] + A[l1 shift 2];

        Bloco 3a

USING   l2 := 121:300 DO
5       W[l2] := X[l2] + A[l2 shift 2];

        Bloco 3b
```

Só com a aplicação da técnica da *quebra de comando* se conseguiu

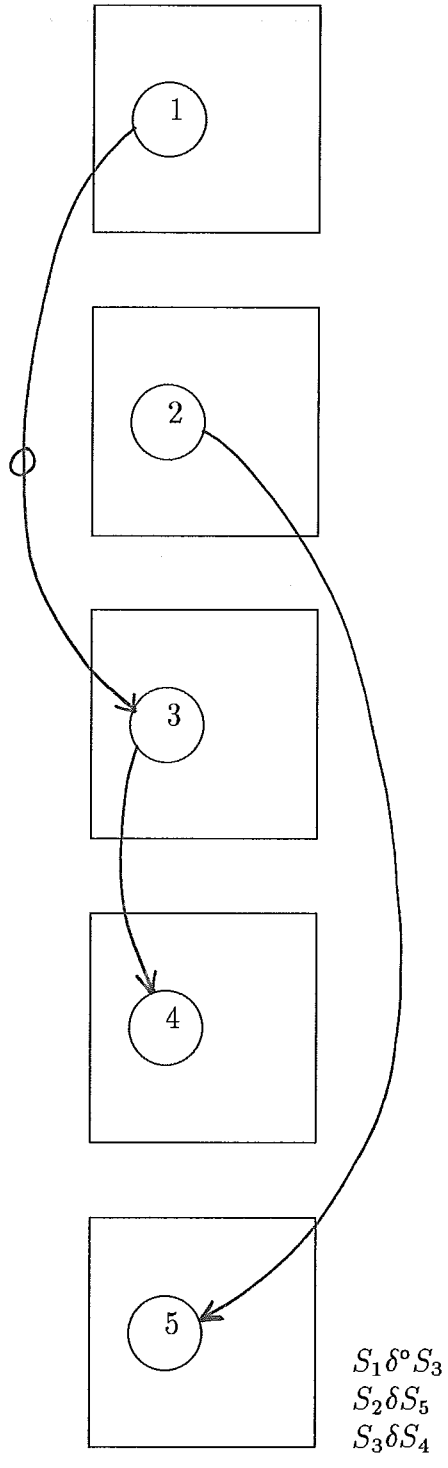


Figura IV.19: Grafo de Dependências

otimizar de alguma forma o conjunto formado pelos blocos (figura IV.19). Os seguintes blocos podem executar em paralelo: [(1 PAR 2) SEQ (3 PAR 5) SEQ 4]. A seguir aplica-se a *substituição de comandos.*

```
USING   k1 := 1:120 DO
1       X[k1] := Y[k1] * (R[k1] + Z[k1 shift 1]);

        Bloco 1a

USING   k2 := 121:400 DO
2       X[k2] := Y[k2] * (R[k2] + Z[k2 shift 1]);

        Bloco 1b

USING   m := 1:120 DO
3       X[m] := U[m] * (R[m] + Z[m]);

        Bloco 2

USING   l1 := 1:120 DO
4       W[l1] := (U[l1] * (R[l1] + Z[l1])) + A[l1 shift 2];

        Bloco 3a
USING   l2 := 121:300 DO
5       W[l2] := (Y[l2] * (R[l2] + Z[l2 shift 1])) + A[l2 shift 2];

        Bloco 3b
```

No *grafo de dependências* da figura IV.20 alguns arcos foram eliminados com a substituição e o conjunto formado pelos blocos foi novamente otimizado. A execução dos blocos pode ser da seguinte forma agora: [(1 PAR 2) SEQ (3 PAR 4 PAR 5)].

O uso *para a frente* do vetor paralelo X deve ser bem estudado para não haver nenhum conflito nas *extensões de paralelismo.* Na realidade não há motivo para existirem esses conflitos: no futuro qualquer extensão pode ser usada pelo vetor X, até seu limite máximo, pois ele foi totalmente calculado mesmo que de forma quebrada.

Uma coisa que se pode concluir é que a aplicação da técnica de *substituição para a frente* em variáveis escalares é bem mais simples que em variáveis paralelas, justamente pela *extensão de paralelismo.*

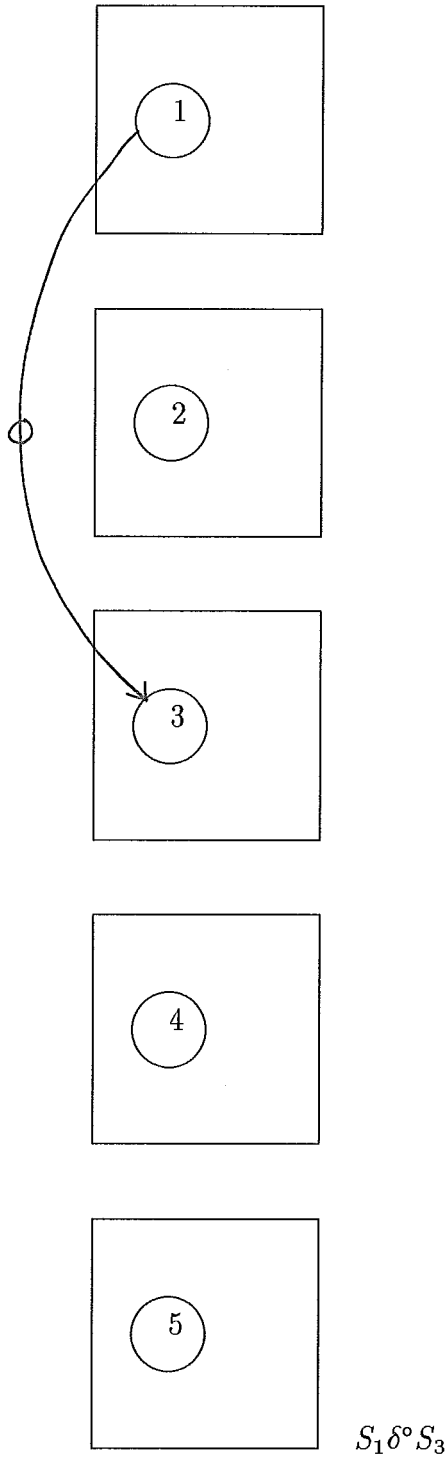


Figura IV.20: Grafo de Dependências

IV.7 Loop Blocking

O objetivo inicial da técnica é criar, a partir de um loop, um aninhamento duplo, para otimizar o tratamento de registradores vetoriais internos.

A arquitetura para a qual o trabalho todo está voltado possui na verdade um processador de altíssima velocidade que simula um processador vetorial. Não existem então, registradores vetoriais com um tamanho interno fixo, não havendo necessidade de alguma preocupação com eles. Assim, inicialmente, a técnica parece não ter aplicação na linguagem Actus para a arquitetura proposta.

Uma outra idéia para a aplicação da técnica é de se quebrarem vetores (ou instruções vetoriais) longos transformando-os em vetores menores e atribuí-los a processadores diferentes. Com isso seria atingido o objetivo de otimizar programas escritos em Actus: aproveitar a vetorização da linguagem e introduzir paralelismo entre blocos usando vários processadores. Essa idéia é bem semelhante à de distribuir os comandos de um loop (sem vetorizá-lo) usando um comando do tipo *DOALL*. As iterações do loop são distribuídas pelos processadores.

EXEMPLO I:

```
USING   I := 1:N DO
1       A[I] := B[I] + C[I];
END
```

Se o bloco for executado numa rede formada por P processadores, onde $N > P$, pode-se distribuir essa instrução vetorial aplicando a técnica:

```
K :=    TRUNC (N/P);
DOALL   I := 1:P          DO
        USING J :=        (I-1)K + 1:IK DO
        1                 A[J] := B[J] + C[J];
END

USING   J:=PK+1:N
        A[J]:=B[J]        + C[J];
```

É importante observar que não há na linguagem Actus comandos como o *DOALL*. Para o trabalho não há nenhuma restrição pois esta é exatamente a proposta: apontar quais são os comandos necessários para a aplicação de todas as técnicas.

Neste caso não houve qualquer preocupação com *balanceamento de carga* na rede de processadores. A carga foi dividida pelos P processadores e um deles executou o que ficou sobrando.

IV.8 Distribuição do Loop

Para se aplicar a técnica na linguagem Actus algumas modificações devem ser feitas na idéia inicial.

No caso de blocos pertencentes ao grupo 2, a técnica será aplicada normalmente, pois esse tipo de bloco representa uma estrutura seqüencial. Neste caso cada comando pode formar um *pi-bloco* a ser vetorizado ou não, seguindo os critérios da linguagem.

Quando a análise é feita em cima de blocos pertencentes ao grupo 1, a dependência a ser estudada é entre blocos e não mais entre comandos. Assim, para simplificar, cada bloco *USING* é visto como um comando simples e tenta-se fazer a distribuição desses blocos. O objetivo é paralelizá-los sempre que possível. Com isso, ao invés de existirem blocos (na verdade linhas de comandos) para serem vetorizados, existirão blocos totalmente independentes que serão executados em paralelo.

Esta técnica deve ser aplicada depois que todas as outras técnicas básicas foram usadas. Quando se chega a este ponto da otimização, basta separar os blocos (pi-blocos) e de alguma forma implementar o paralelismo. Os blocos que não puderem ser executados em paralelo, como conseqüência de pelo menos uma dependência entre eles, formam uma *região fortemente conectada* (recorrência).

Nos exemplos vistos até agora, de aplicação de outras técnicas, a técnica de distribuição já vinha sendo usada no final de cada exemplo para demons-

trar o paralelismo entre blocos. Na realidade, a partir de agora, que já se tomou conhecimento da técnica, os blocos que executarão em paralelo serão marcados no *grafo de dependências.*

É importante mostrar também que, no caso dos blocos pertencentes ao do grupo 1 não existirão ciclos porque também não existirão dependências com *sentido para trás.* Qualquer dependência entre os blocos, que não puder ser resolvida com sincronização, será uma restrição à paralelização dos blocos.

Quando a técnica é aplicada em blocos pertencentes ao grupo 3, a idéia é a mesma e cada bloco *USING* será visto como um pi-bloco. O objetivo continuará a ser o de paralelizar blocos independentes.

ALGORITMO:

Algoritmo: distribuição do loop;
Entrada: GD;

início;
{a análise é feita bloco a bloco}
para (cada bloco) faça
caso bloco(tipo)
tipo-1: *início*
separa comandos que não são origem e
não são destino; (comandos independentes)
comandos restantes (comandos dependentes);
fim-início;

tipo-2: *início*
se (todas as dependências que $\notin$ a ciclo tem o mesmo
sentido (PF) e GD não pode mais ser mudado)
então faça
separa comandos que $\in$ a nenhum ciclo; (comandos
independentes)
comandos restantes (recorrências);
fim-início;

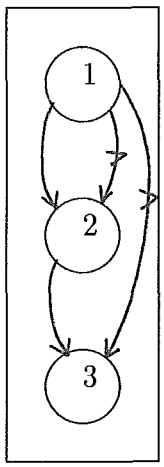
tipo-3: *início*
(não faça nada);
fim-início;
fim-caso;
fim-para;

se ($\exists$ bloco(tipo-1) que não são unidos por nenhuma dependência) *então*
faça
blocos independentes (PAR);
para (blocos restantes) faça
blocos não-independentes (PAR/SINC);
final;

EXEMPLO I:

```
FOR   i := 1,N
      USING   j := i:N DO
      1       A[j shift 1] := B[j shift -1] + C[j];
      2       B[j] := A[j] * K;
      3       C[j] := B[j] - Z[j];
END
```

A *antidependência* e a *dependência direta*, causadas pelo vetor C, que aparecem na figura IV.21, se confundem devido à própria estrutura da linguagem.



$S_1 \delta S_2$
$S_1 \overline{\delta} S_2$
$S_1 \overline{\delta} S_3$
$S_2 \delta S_3$

Figura IV.21: Grafo de Dependências

Não existe nenhuma otimização possível em cima desse grafo. Ele é formado por várias *regiões fortemente conectadas*. O *loop* não pode ser distribuído e nenhum de seus comandos pode ser colocado em paralelo com algum outro.

EXEMPLO II:

```
USING   I := 1:N DO
1       A[I] := B[I] + C[I];
2       X[I] := A[I] * 4;
3       H[I] := Z[I] * k;
END
        Bloco 1

USING   J := 1:N DO
4       Y[J] := C[J] * K;
5       P[J] := D[J] + A[J];
END
        Bloco 2
```

Os comandos S_3 (bloco 1) e S_4 (bloco 2), na figura IV.22, são independentes, dentro dos seus respectivos blocos podendo ser colocados em outros blocos a serem criados e executados em paralelo. No grafo há duas *dependências diretas*: uma envolvendo os comandos S_1 e S_2 (bloco 1) que não pode ser eliminada e a outra entre os comandos S_1 (bloco 1) e S_5 (bloco 2). Esta última pode ser otimizada através de algum mecanismo de sincronização.

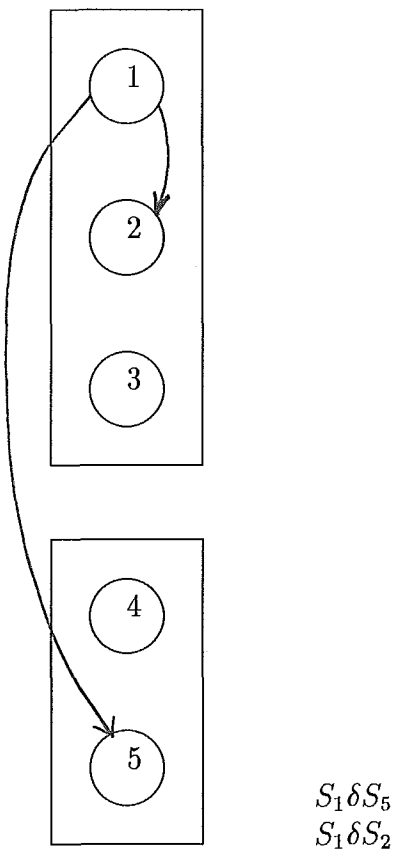


Figura IV.22: Grafo de Dependências

Depois de otimizado, o bloco 1 inicial ficará apenas com dois comandos e outros dois blocos serão criados. O resultado é mostrado a seguir:

```
USING   I := 1:N DO
1       A[I] := B[I] + C[I];
2       X[I] := A[I] * 4;
END
        Bloco 1

USING   J := 1:N DO
3       H[J] := Z[J] * K;
END
        Bloco 1'

USING   D := 1:N DO
4       Y[J] := C[J] * K;
END
        Bloco 2'

USING   M := 1:N DO
5       P[J] := D[J] + A[J];
END
        Bloco 2
```

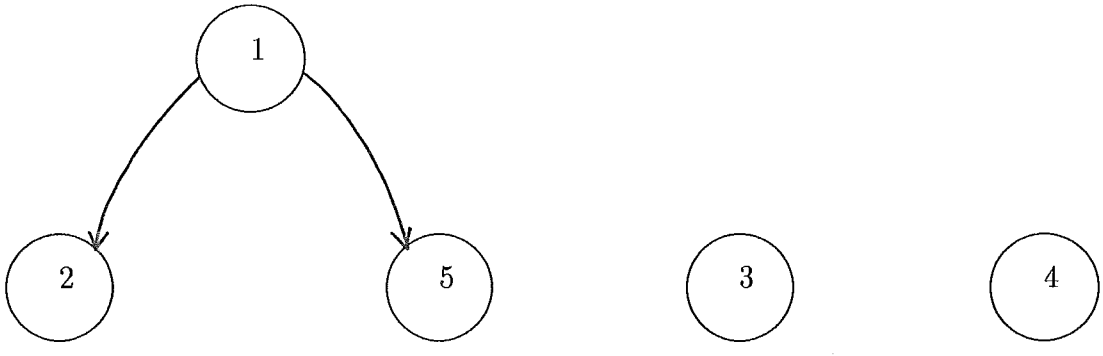


Figura IV.23: Grafo de Dependências

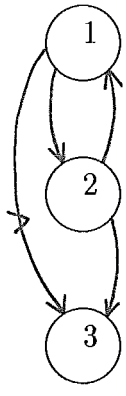
Com a otimização, dois blocos (1' e 2') se tornaram totalmente independentes dos outros e um terceiro bloco (bloco 2) parcialmente independente, em

relação ao bloco 1 (figura IV.23). A execução será a seguinte:
[(1 SEQ (2 PAR 5 PAR 3 PAR 4))].

Não há necessidade de nenhuma sincronização visto que só o comando S_1 causa algum tipo de dependência.

EXEMPLO III:

```
FOR   i := 1,N DO
1     A(i+1) := B(i-1) + C(i);
2     B(i) := A(i) * K;
3     C(i) := B(i) - 1;
END
```



$S_1 \delta S_2$
$S_1 \overline{\delta} S_3$
$S_2 \delta S_1$
$S_2 \delta S_3$

Figura IV.24: Grafo de Dependências

Na figura IV.24 existe um ciclo formado por duas *dependências diretas*, envolvendo os comandos S_1 e S_2, que é inquebrável, criando uma *recorrência* e há também um *comando independente* (S_3). A técnica deve então ser aplicada tendo em vista que o bloco do exemplo pertence ao grupo 2.

```
FOR     i := 1,N DO
1       A(i+1) := B(i-1) + C(i);
2       B(i) := A(i) * K;
END
        Bloco 1

USING   J := 1:N
3       C[J] := B[J] - 1;

        Bloco 2
```

O comando S_3 foi vetorizado o que faz com que os vetores C e B sejam usados de forma seqüencial e paralela.

EXEMPLO IV:

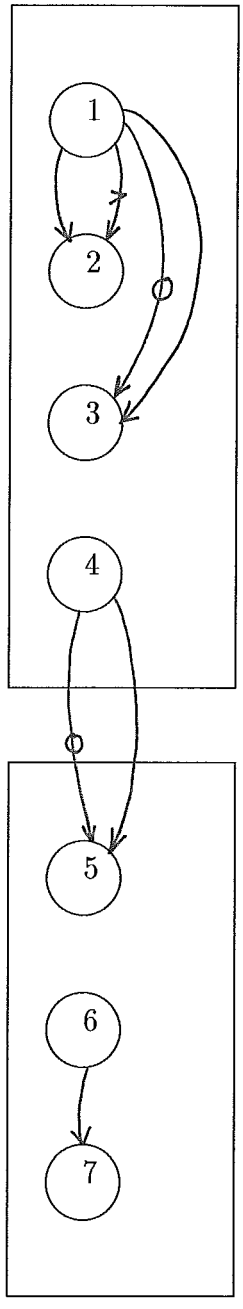
```
USING   k := 1:100 DO
1       A[k] := B[k] - Z[k];
2       B[k] := 2 * A[k] + C[k];
3       A[k] := A[k] + Z[k];
4       D[k] := D[k] * C[k];
END
        Bloco 1

USING   J := 1:100 DO
5       D[J] := D[J] * 5;
6       E[J] := E[J] + Z[J];
7       F[J] := F[J] * E[J];
END
        Bloco 2
```

Foi mostrada anteriormente uma solução para este exemplo, na qual com a técnica de *reordenação de comandos* e a introdução de alguma sincronização os blocos podiam executar em paralelo. Agora será dada outra solução com o uso da técnica de *distribuição de comandos.*

No *grafo de dependências* da figura IV.25 há blocos isolados que podem formar *blocos independentes* e serem executados em paralelo havendo um ganho maior do que o obtido com a outra otimização sem a necessidade da *reordenação de comandos.*

Nas linguagens seqüenciais, uma *dependência direta* entre dois comandos não impede a vetorização, apenas no caso de dependências para trás ou ciclos. Em Actus, quando há alguma dependência ligando dois comandos, estes não poderão ser executados em paralelo. Pode ser que algum tipo de sincronização resolva, mas, mesmo assim, os comandos não executarão em paralelo.

Neste exemplo:



$S_1 \delta S_2$
$S_1 \overline{\delta} S_2$
$S_1 \delta^\circ S_3$
$S_1 \delta S_3$
$S_4 \delta^\circ S_5$
$S_4 \delta S_5$
$S_6 \delta S_7$

Figura IV.25: Grafo de Dependências

```
USING   k := 1:100 DO
1       A[k] := B[k] - Z[k];
2       B[k] := 2 * A[k] + C[k];
3       A[k] := A[k] + Z[k];
END
                Bloco 1

USING   J := 1:100 DO
4       D[J] := D[J] * C[J];
5       D[J] := D[J] * 5;
END
                Bloco 1'

USING   I := 1:100 DO
6       E[I] := E[I] + Z[I];
7       F[I] := F[I] * E[I];
END
                Bloco 2
```

Os três blocos resultantes são independentes e podem ser executados totalmente em paralelo:
[(Bloco 1) PAR (Bloco 1') PAR (Bloco 2)].

Os comandos, dentro de cada bloco, formam uma *região fortemente conectada*, embora não apareça nenhum ciclo, porque não podem ser executados em paralelo.

EXEMPLO V:

```
FOR   i := 1,N   DO
      USING      I := 1:N DO
      1          A[I] := B[I] + C[I];
      2          D[I] := A[I] * 5;
      3          K[I] := Z[I] + E[I];
      END
                 Bloco 1

FOR   j := 1,N   DO
      USING      J := 1:N DO
      4          R[J] := K[J] * 2;
      5          G[J] := H[J] * 10;
      6          M[J] := G[J] + H[J];
      END
                 Bloco 2
```

No *grafo de dependências* (figura IV.26) há três conjuntos de comandos independentes entre si que podem ser executados em paralelo. Para isso são criados três blocos distintos aproveitando que as *extensões de paralelismo* combinam.

```
FOR  i := 1,N   DO
     USING      I := 1:N DO
     1          A[I] := B[I] + C[I];
     2          D[I] := A[I] * 5;
     END
                Bloco 1

FOR  j := 1,N   DO
     USING      J := 1:N DO
     3          K[J] := Z[J] + E[J];
     4          R[J] := K[J] * 2;
     END
                Bloco 1'

FOR  v := 1,N   DO
     USING      V := 1:N DO
     5          G[J] := H[J] * 10;
     6          M[J] := G[J] + H[J];
     END
                Bloco 2
```

A execução dos blocos será a seguinte:
[(Bloco 1) PAR (Bloco 1') PAR (Bloco 2)].

Este exemplo podia também ser resolvido com o uso da técnica de *reordenação de comandos* mas o ganho alcançado seria menor. Essa técnica traria vantagens se os comandos ou os blocos não fossem totalmente independentes.

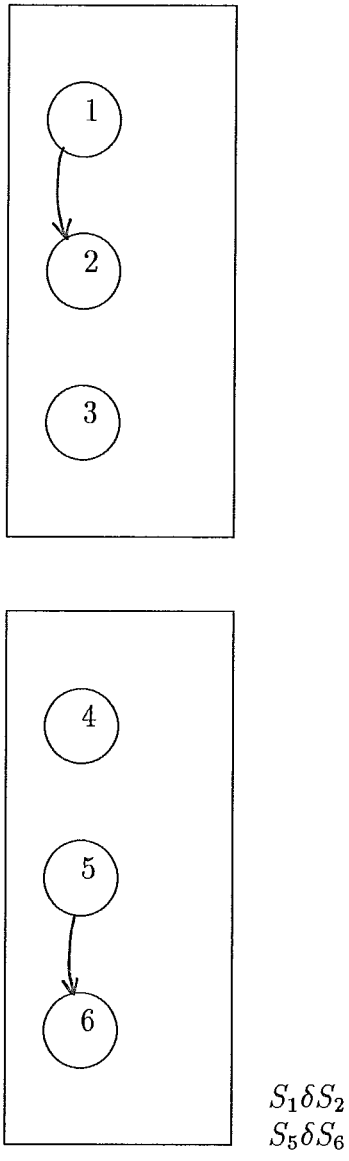


Figura IV.26: Grafo de Dependências

Capítulo V

Conclusão

Inicialmente pensou-se em fazer uma aplicação prática de todas as técnicas de otimização que se adaptassem à linguagem Actus. Essa aplicação seria em cima de uma linguagem intermediária [14] gerada pelo compilador para a linguagem Actus. Como a linguagem intermediária não foi concluída a tempo optou-se por um estudo profundo de um número expressivo de técnicas de otimização, pela adaptação de algumas técnicas à linguagem Actus e pela eleboração dos algoritmos apresentados. Além disso o trabalho revisa e sintetiza as técnicas de *vetorização* e de *paralelização* espalhadas pela literatura. Nem todas as técnicas de otimização estão presentes neste trabalho devido à sua complexidade. Foram escolhidas as mais utilizadas na literatura e as que tinham maior chance de se adaptarem à estrutura da linguagem Actus.

Uma das vantagens obtidas com o trabalho é que este estudo pode ser estendido a outras linguagens paralelas tipo FORTRAN 90 ou FORTRAN 8X e não só à linguagem Actus. Por isso algumas técnicas tiveram de ser modificadas pois agora a linguagem alvo não é mais uma linguagem puramente seqüencial mas uma linguagem que já contém algum tipo de paralelismo embutido, o vetorial.

Os algoritmos aqui apresentados, apesar de serem apenas um *protótipo* dos verdadeiros algoritmos para cada técnica de otimização, servirão de base para um próximo trabalho que será constituído da própria implementação do otimizador.

O que se pode concluir é que a reestruturação automática de progra-

mas é uma tarefa extensa e complexa devido ao grande número de técnicas existentes e à falta de um critério para determinar quais as técnicas que devem ser aplicadas e a sua ordem. Os resultados finais podem ser muito diferentes no que se refere à quantidade de paralelismo alcançada.

O objetivo inicial do trabalho, conhecimento e possibilidade da aplicação de um otimizador numa linguagem com características vetoriais, foi alcançado. Há agora a necessidade de se avaliar, na prática, a eficácia das técnicas na linguagem Actus. Deve ser feito também um estudo sobre a alocação estática e dinâmica de processadores para a distribuição das tarefas criadas pelas notações *SINC* e *PAR*, num sistema paralelo distribuído. Além disso continua em aberto o problema de se saber a melhor ordem para a aplicação de um conjunto de técnicas.

Referências Bibliográficas

[1] Constantine D. Polychronopoulos, "Parallel Programming And Compilers", Kluwer Academic Publishers, 1988.

[2] Constantine D. Polychronopoulos, "Loop Coalescing: a Compiler Tranformation for Parallel Machines", Proceedings of the 1987 International Conference on Parallel Processing, pags. 235-242, Ago. 1987.

[3] Constantine D. Polychronopoulos, "Automatic Restruturing of Fortran Programs for Parallel Execution", 1987 Conference on Parallel Processing in Science and Engineering, Bonn, Jun. 1988.

[4] Constantine D. Polychronopoulos, "Toward Auto-scheduling Compilers", The Journal of Supercomputing, vol.2, pags. 297-330, Nov. 1988.

[5] David J. Kuck, Robert H. Kuhn, Bruce Leasure e Michael Wolfe, "The Structure of an Advanced Retargetable Vectorizer", IEEE Transactions on Computers, pags. 163-178, 1980.

[6] David J. Kuck, Robert H. Kuhn, David A. Padua, Bruce Leasure e Michel Wolfe, "Dependence Graphs and Compiler Optimizations", 1981 ACM, pags. 207-218.

[7] Michael J. Wolfe, "Optimizing Supercompilers for Supercomputers", Department of Computer Science, University of Illinois at Urbana-Champaign.

[8] Michael J. Wolfe, "Data Dependence and Program Restructuring", The Journal of Supercomputing, vol. 4, pags. 321-344, Jan. 1991.

[9] David A. Padua e Michel J. Wolfe, "Advanced Compiler Optimizations for Supercomputers", Communications of the ACM, vol. 29, num. 12, Dez. 1986.

[10] Utpal Banerjee, "An Introdution to a Formal Theory of Dependence Analysis", The Journal of Supercomputing, vol. 2, pags. 133-149, Out. 1988.

[11] Fábio Carneiro Mokarzel e Jairo Panetta, "Reestruturação Automática de Programas Seqüenciais para Processamento Paralelo.

[12] R. H. Perrot, "Parallel Programming", Addison-Wesley, 1987.

[13] R. H. Perrot, R. W. Lyttle, e P. S. Dhillon, "The Design and Implementation of a Pascal Based Language for Array Processor Architecture", Journal of Parallel and Distributed Computing 4, 266-287 (1987).

[14] C. L. Sales, L. M. R. Eizirik e C. L. Amorim, "Uma Linguagem Intermediária para Compilar ACTUS II em OCCAM 2".

Apêndice A

Definições

Neste apêndice são mostradas as definições da pseudo-linguagem criada para se escreverem os algoritmos do capítulo IV, assim como o significado de algumas estruturas de dados.

- *GD* = grafo de dependências;
- *variáveis* = podem ser *escalar* ou *vetorial* ($\neq$ *escalar*);
- *dd* = dependência direta;
- *ad* = antidependência;
- *ds* = dependência de saída;
- *num(dep)* = quantidade de um certo tipo de dependência (dep);
- *comando(origem)* = comando, no GD, onde se origina um arco de dependência;
- *comando(destino)* = comando, no GD, onde termina um arco de dependência;
- *bloco(origem)* = bloco que contém o comando(origem) para uma dada dependência;
- *bloco(destino)* = bloco que contém o comando(destino) para uma dada dependência;
- *OUT(cmd)* = conjunto de saída de um comando (cmd);

- *IN(cmd)* = conjunto de entrada de um comando (cmd);
- *ext-par-bloco* = extensão de paralelismo de um bloco;
- *bloco(tipo)* = define o tipo de bloco que pode ser 1 (USING), 2 (SEQÜENCIAL) ou 3 (USING + SEQÜENCIAL);
- *dep-sentido(PT)* = dependência com sentido *para trás*;
- *dep-sentido(PF)* = dependência com sentido *para a frente*;
- *ciclo* = ciclo envolvendo comandos e arcos de dependências;
- *dd(PT)* = dependência direta com sentido *para trás*;
- *ad(PT)* = antidependência com sentido *para trás*;
- *ds(PT)* = dependência de saída com sentido *para trás*; (idem *(PF)* = *para a frente*)
- { } = indicam comentário;