Otimização de Programas ACTUS

Paula Marisa da C. P. F. Maciel[1]
Claudio Amorim[2]

Sumário

Este artigo propõe uma aplicação de técnicas de otimização de linguagens seqüenciais 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 serão descritas no artigo.

Abstract

In this article we propose the aplication of optimization techniques used in sequential languages into a vector language: ACTUS II. There were few necessary changes for adjusting the technics to the caracteristics of ACTUS II and it was reached a great amount in terms of parallelism. The project which the optimizer belongs to is a compiler for ACTUS II. The ACTUS II language, the optimizations techniques and their aplications in ACTUS II will be discussed in this article.

[1]Engenheira Eletrônica, Aluna de Mestrado da COPPE - Prog. de Engenharia de Sistemas e Computação (UFRJ); áreas de interesse: Arquiteturas de Computadores e Computação Paralela

[2]M.Sc. (COPPE - 1979), PhD (Imperial College - 1984); áreas de interesse: Processamento Paralelo e Vetorial; Professor Adjunto (COPPE/UFRJ);

COPPE - Universidade Federal do Rio de Janeiro
Caixa Postal 68516 - CEP 21945 - Rio de Janeiro - RJ
E - Mail : COS99284@UFRJ (BITNET)
Esse trabalho foi parcialmente financiado pelo CNPq, Capes e FINEP.

1 Introdução

São vários os motivos que levaram os pesquisadores a estudarem técnicas de detecção e exploração 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 não é expresso nos programas já existentes e estes não podem ser abandonados devido ao volume de trabalho e informação que representam procurou-se então desenvolver técnicas que detectassem o paralelismo e otimizassem os programas.

Este trabalho 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. Por isso a linguagem escolhida para se aplicarem as técnicas de otimização foi ACTUS II [4] [5]. Os objetivos principais são: descobrir blocos e comandos que podem ser executados em paralelo e descobrir blocos que podem ser vetorizados (usando-se as prerrogativas da própria linguagem) e passaram desapercebidos pelo programador. As técnicas analisadas compreendem: renomeação de variáveis, expansão de variáveis escalares, substituição para a frente de comandos, quebra de comando, reordenação de comandos, *loop blocking* e distribuição do *loop*. O artigo se propõe a fazer uma breve descrição da linguagem Actus II e das técnicas de otimização acima e a aplicação dessas mesmas técnicas na linguagem Actus II, que é o objetivo principal do trabalho. Em todo o artigo usaremos ACTUS para substituir ACTUS II.

2 Dependências de Dados

Para que se possa fazer a reestruturação automática de um programa é necessário que se computem as dependências de dados, as quais mostram como os dados são calculados e usados durante a execução do programa. Um programa deve ser visto como uma seqüência de comandos S_i. IN (S_i) representa o conjunto de variáveis lidas pelo comando S_i e OUT (S_i) representa o conjunto de variáveis escritas pelo comando S_i. Para cada dependência de dados que envolve os comandos $S_i(I_1, \ldots, I_k)$ e $S_j(J_1, \ldots, J_k)$ define-se a r-ésima *distância* ϕ_r como $\phi_r = j_r - i_r$, $(1 \leq r \leq k)$. O valor de ϕ_r pode ser positivo, negativo ou zero e o seu sinal descreve a *direção* da dependência da seguinte maneira: se o sinal é positivo a *direção é para a frente* e representa-se (<); se o sinal é negativo a *direção é para trás* e representa-se (>); se a *distância* é zero a *direção é estacionária* e representa-se (=). Existem dois tipos de análises para se verificar se há alguma dependência entre os comandos: a análise local, chamada de *teste de dependência de dados* [2] e a análise global, chamada de *partição pi* [2].

As principais dependências são de dados:

1- Dois comandos S_i e S_j estão envolvidos numa *dependência direta* se e só se (1) S_i for executado antes de S_j e (2) OUT (S_i) $\cap$ IN (S_j) $\neq$ V. Simboliza-se $S_i \delta S_j$ e graficamente $\longrightarrow$.

2- Dois comandos S_i e S_j estão envolvidos numa *antidependência* se e só se (1) S_i for executado antes de S_j e (2) IN (S_i) $\cap$ OUT (S_j) $\neq \vee$. Simboliza-se $S_i \bar{\delta} S_j$ e graficamente $\not\rightarrow$.

3- Dois comandos S_i e S_j estão envolvidos numa *dependência de saída* se e só se (1) S_i for executado antes de S_j e (2) OUT (S_i) $\cap$ OUT (S_j) $\neq \vee$. Simboliza-se S_i δ^o S_j e graficamente $\multimap$.

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. Esses arcos podem ser de cinco tipos diferentes, de acordo com os tipos de dependências existentes. Um dos objetivos da aplicação das técnicas de otimização é quebrar ciclos de dependências existentes. As *recorrências* representam conjuntos de comandos que não podem ser executados em paralelo. Existem várias transformações e cada uma delas explora um tipo diferente de paralelismo. Apesar das técnicas serem independentes entre si, a ordem em que são aplicadas é significante. Podem-se aplicar as mesmas técnicas no 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.

3 A Linguagem Actus

Actus II 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 controle da própria linguagem. Mais especificamente, Actus é uma linguagem *PASCAL-like* com estruturas de dados e de controle 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. 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. É muito importante entender que a *extensão de paralelismo* é o conceito central da linguagem e pode ser controlada através dos *conjuntos de índices* que são formados por valores inteiros e identificam os elementos que serão acessados em paralelo. Num mesmo comando todos os operandos devem ter a mesma *extensão de paralelismo* pois as operações são aplicadas elemento por elemento, simultaneamente. 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.

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 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. 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.

4 As Técnicas de Otimização e a Linguagem Actus

Como já se pôde observar na seção anterior, a linguagem Actus tem características próprias bem diferentes das linguagens seqüenciais, para as quais foram criadas as técnicas de otimização. Na linguagem Actus trabalharemos com três tipos de blocos:

1- comandos que se encontram dentro de blocos do tipo USING;

2- comandos que se encontram dentro de blocos repetitivos seqüenciais (sem a presença do comando *USING*);

3- comandos que se encontram dentro de blocos repetitivos paralelos (com um comando *USING* dentro ou fora do escopo do bloco);

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 estivéssemos tentando 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.

Devido à própria estrutura do comando *USING*, a aplicação das técnicas sofre muitas mudanças inclusive com restrições em algumas delas. Isso acontece também, porque em Actus não existem primitivas para expressar paralelismo assíncrono nem primitivas de sincronização já que processadores vetoriais e matriciais não têm esse problema tão comum em arquiteturas formadas por processadores independentes. Como se quer paralelizar blocos e distribuí-los numa rede de processadores foram criadas, para este trabalho, as seguintes primitivas:

1- SEQ: indica e execução seqüencial de vários comandos;
Ex.:
(1 **SEQ** 2 **SEQ** 3), os comandos 1, 2, 3 executam seqüencialmente;
(1 **SEQ** 2) **SEQ** (3 **SEQ** 4), quatro comandos seqüenciais pertencentes a dois blocos diferentes;

2- PAR: indica a execução paralela de comandos pertencentes a blocos diferentes;
Ex.:
(1) **PAR** (2 **SEQ** 3), o comando 1 (bloco 1) executa 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 seqüencias a esse último comando;
Ex.:
(4) (**SINC** (6 **SEQ** 7 **SEQ** 8)) **SEQ** 1 **SEQ** 2 **SEQ** 3, tem-se o comando 4 (bloco 1) disparando os comandos seqüenciais 6,7,8 (bloco 2) e executando seqüencialmente com os comandos 1,2,3 (bloco 1); os comandos 6,7,8 executam em paralelo com os comandos 1,2,3;

Uma coisa que tem de ser atentamente observada 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 mudanças, devido à própria estrutura da linguagem. No caso de blocos pertencentes 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 voltadas para trás e de se querer paralelizar esses comandos. No caso de blocos pertencentes aos grupos 2 e 3, as definições iniciais 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.

4.1 Renomeação de Variáveis

O objetivo principal é eliminar *dependências de saída* causadas pelo uso da mesma variável no *conjunto de saída* de comandos diferentes. A técnica atribui nomes diferentes para essas variáveis simplificando o *grafo de dependências*.

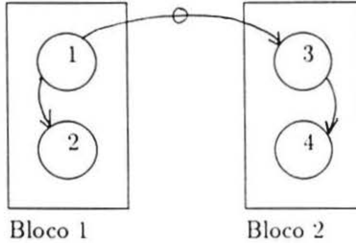
Devido à sua simplicidade não há necessidade de se fazer alguma alteração para se usar esta técnica em programas escritos na linguagem Actus. Pode ser aplicada tanto em variáveis escalares como em vetores (seqüenciais ou paralelos) apesar do primeiro caso ser mais simples.

EXEMPLO I:

```
USING   m := 1:400 DO              USING   n := 1:120 DO
1       EM := X - 1;               3       EM := WGHT * (X - 1);
2       U[m] := Z[m] * EM;         4       POLY[n] := Z[n] * EM;
END                                END

        Bloco 1                            Bloco 2
```

• **GRAFO DE DEPENDÊNCIAS - Fig. 1**



Bloco 1 Bloco 2

Fig. 1

Existe uma *dependência de saída* entre os comandos 1 e 3 causada pelo uso da mesma variável escalar *EM* nos dois blocos e uma falsa *dependência direta* entre os comandos 1 e 4. Na verdade o comando 4 depende da execução do comando 3. Essa *dependência de saída* não permite que os dois blocos executem em paralelo pois ambos escrevem na mesma posição de memória.

```
USING   m := 1:400 DO              USING   n := 1:120 DO
1       EM := X - 1;               3       EN := WGHT * (X - 1);
2       U[m] := Z[m] * EM;         4       POLY[n] := Z[n] * EN;
END                                END

        Bloco 1                            Bloco 2
```

Agora 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   k := 1:400                 USING   m := 1:200
1       X[k] := Y[k] + R[k];       2       X[m] := U[m] + A[m];

        Bloco 1                            Bloco 2
```

• **GRAFO DE DEPENDÊNCIAS - Fig. 2**

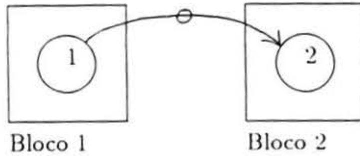


Fig. 2

Inicialmente os dois blocos são executados seqüencialmente e existe uma *dependência de saída* envolvendo os comandos 1 e 2. Os dois blocos possuem diferentes *extensões de paralelismo* o que faz com que essa dependência não envolva todos os elementos. A técnica de *substituição* deve então ser aplicada.

```
USING   k := 1:400                  USING   m := 1:200
1       X[k] := Y[k] + R[k];        2       XM[m]  :=  U[m]  +
                                            A[m];

        Bloco 1                             Bloco 2
```

Com a aplicação da técnica, a única dependência que existia foi eliminada, o que significa que os blocos podem ser executados em paralelo. A execução poderá ser a seguinte: [(1) PAR (2)]. No final deve-se observar que o valor final do vetor X[ ] se encontra na verdade em XM[ ].

4.2 Expansão de Variáveis Escalares

O objetivo desta técnica é transformar em vetorias variáveis escalares que são usadas dentro de blocos do tipo *DO*. Com isso diminui-se o número de arcos no *grafo de dependências* do programa, eliminando-se dependências de saída e *antidependências*, pois cada iteração dos blocos tem seu próprio conjunto de posições de memória. Esse problema não ocorre quando usamos vetores já que em cada iteração uma posição diferente do vetor é acessada.

Na linguagem Actus a técnica será útil apenas quando aplicada em blocos pertencentes 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*, o bloco já se encontra vetorizado.

EXEMPLO I:

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

• GRAFO DE DEPENDÊNCIAS - Fig. 3

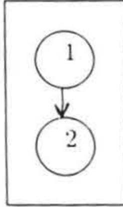


Fig. 3

Existe uma *dependência de saída* do comando 1 em relação a si mesmo pois ele sempre escreve na mesma posição de memória (a variável EM). Existe um ciclo entre os comandos 1 e 2 causados também pelo uso da variável EM. A *dependência direta* ocorre porque, na mesma iteração, o comando 1 calcula o valor de EM e o comando 2 usa esse valor e a *antidependência* é porque o comando 2 usa o EM o qual será calculado novamente pelo comando 1 na próxima iteração. A presença do ciclo impede qualquer vetorização. Aplicando a técnica obteremos o seguinte resultado:

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

O *grafo de dependências* fica muito mais simples contendo apenas uma *dependência direta*, envolvendo a variável EM, que não pode ser eliminada. Com isso pode-se vetorizar todo o bloco.

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

A aplicação da técnica conseguiu otimizar bastante o bloco inicial. Este é um exemplo de um caso que poderia passar dasapercebido pelo programador.

4.3 Substituição para a Frente de Comandos

Esta transformação tem como objetivo principal eliminar alguns arcos de *dependência direta* no *grafo de dependências* do programa pois substitui uma variável pela sua expressão correspondente. Apesar de ser um método simples para que o resultado após a *substituição* seja correto, 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, $S_j \ \bar{\delta} \ S_m$ ou $S_j \ \delta^{o} \ S_m$ e se a execução de S_j dominar a execução de S_k. Podem existir situações em que a técnica de *reordenação de*

comandos deve ser aplicada antes para que a substituição possa ser feita.

No caso de programas escritos em Actus deve-se manter a mesma restrição para que o resultado final seja correto. A substitução tanto pode ser feita dentro do mesmo bloco ou entre blocos visando a vetorização ou a paralelização de comandos e dos blocos. Só deve haver preocupação com as *extensões de paralelismo* quando a otimização for entre blocos. As variáveis envolvidas na substituição tanto podem ser escalares como vetores paralelos sendo que o primeiro caso é bem mais fácil de tratar.

EXEMPLO I:

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

• GRAFO DE DEPENDÊNCIAS - Fig. 4

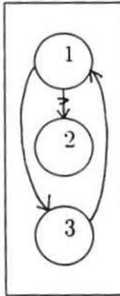


Fig. 4

Como o bloco pertence ao grupo 2 o objetivo é vetorizá-lo assumindo que o paralelismo que existe dentro do bloco não foi percebido pelo programador. Há uma *dependência direta* com direção *para trás* causada pelo vetor B[ ] e uma *antidependência* entre os comandos 1 e 2. Essa última dependência não permite que a *substitução* do comando 1 no comando 3 não seja feita imediatamente. Os comandos 2 e 3 devem ser reordenados com a técnica de *reordenação de comandos.*

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

Como a dependência que envolve os comandos 1 e 3 com direção *para trás* continua

existindo a técnica de *reordenação de comandos* deve ser novamente aplicada. Com isso todas as dependências passarão a ter a mesma direção e o bloco resultante pode ser totalmente vetorizado.

```
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
```

Este é um exemplo em que a aplicação das técnicas trouxe uma grande otimização em relação ao bloco inicial.

EXEMPLO II:

```
USING     i := 1:N DO                 USING     j := 1:N DO
1         A[i] := Z[i] + W[i];        3         B[j] := A[j] + 2;
2         C[i] := E[i];               END
END

          Bloco 1                               Bloco 2
```

• **GRAFO DE DEPENDÊNCIAS - Fig. 5**

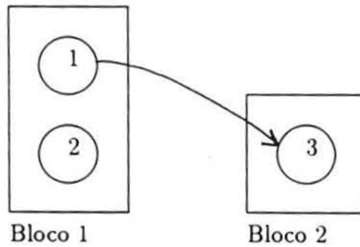


Fig. 5

A *substituição dos comandos* deve ocorrer entre os comandos 1 e 3, não havendo neste exemplo nenhuma restrição.

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

          Bloco 1                               Bloco 2
```

Antes de se aplicar essa técnica os dois blocos podiam ser executados em paralelo, com alguma sincronização. Com a otimização, o paralelismo continua a existir mas agora com a vantagem de não precisar de nenhuma sincronização porque todas as dependências foram eliminadas.

4.4 Quebra de Comando

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* ou *dependências de saída* alguma otimização pode ser feita podendo resultar numa paralelização parcial dos *loops*. A *quebra de comando* introduz novos comandos de atribuição e novas variáveis no programa fonte.

A idéia de se aplicar esta técnica na linguagem Actus é um pouco diferente da idéia de aplicá-la numa linguagem seqüencial. No caso de blocos que pertencem ao grupo 2 não há necessidade de se modificar em nada a idéia inicial pois é como se estivesse trabalhando com uma linguagem não vetorial. No final o bloco poderá ser vetorizado usando-se as vantagens da linguagem Actus.

A alteração se torna necessária quando se trabalha em cima de blocos que pertencem aos grupos 1 e 3. Isso porque nesses casos o objetivo é se quebrar a *extensão de paralelismo* dos comandos *USING* para tornar paralelos alguns blocos e também para que outras técnicas possam ser aplicadas posteriormente. Para se quebrar a *extensão de paralelismo* deve-se observar no *grafo de dependências* se há alguma dependência entre os blocos e a interseção das *extensões de paralelismo* entre eles. Uma restrição para a aplicação da técnica é que não devem existir comandos dentro do mesmo bloco unidos por uma dependência com direção *para trás*. Se isso ocorrer o resultado final será alterado. Já se pôde observar em exemplos anteriores como a técnica pode ser usada para permitir a aplicação de outras técnicas de otimização.

EXEMPLO I:

```
USING   k := 1:400                    USING   m := 1:[5]996
1       X[k] := Q[k] + Y[k];          2       Q1[m] := X[m] + Z[m];

        Bloco 1                               Bloco 2
```

- **GRAFO DE DEPENDÊNCIAS - Fig. 6**

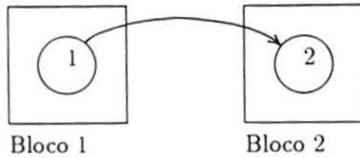


Fig. 6

Há uma *dependência direta* entre os dois blocos, mas a interseção entre as *extensões de paralelismo* mostra que essa dependência só é verdadeira para os primeiros 400 elementos de X[ ]. Para os elementos restantes a dependência é falsa.

```
USING   k := 1:400          USING   m := 1:[5]396
1       X[k] := Q[k] +      2       Q1[m] := X[m] + Z[m];
        Y[k];

        Bloco 1                     Bloco 2
```

```
USING   n := 401:[5]996
3       Q1[n] := X[n] + Z[n];

        Bloco 3
```

Depois da aplicação da técnica existirá apenas a *dependência direta* que 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 a seguinte:
[(1) PAR (3)] SEQ (2) ou [(1) SEQ (2)] PAR (3) .

4.5 Reordenação de Comandos

O objetivo principal da *reordenação de comandos* é trocar a direção de dependências entre comandos para que eles possam ser vetorizados. Apesar de ser uma transformação bastante simples, nem sempre é permitido mudar-se a ordem de execução de alguns comandos. Isso porque não pode haver uma dependência com direção (=) unindo dois comandos que vão ser reordenados. Um outro objetivo da reordenação é a minimizar o tempo de espera de uma tarefa (execução de um bloco) que necessita de algum dado que será produzido por uma outra tarefa que está sendo executada num processador diferente.

Em Actus, quando a técnica é aplicada em blocos pertencentes ao grupo 1, o objetivo principal é preparar esses blocos para a aplicação de outra técnica ou agrupar todas as dependências num bloco só para que possa existir algum paralelismo entre blocos. Isso

porque quando se tratamos com esse grupo não há necessidade de se trocar o sentido das dependências e eles já se encontram vetorizados. A reordenação pode ser feita tanto entre blocos como dentro do mesmo bloco. Em exemplos anteriores pode se verificar a aplicação da técnica.

Quando a técnica é aplicada em blocos do grupo 2 o objetivo é a vetorização desses blocos exatamente como no caso das linguagens seqüenciais.

Também em Actus a *reordenação de comandos* pode ser usada com o objetivo de diminuir o tempo de espera de comunicação entre os blocos que são executados parcialmente em paralelo. Nesse caso os comandos envolvidos podem pertencer ao mesmo bloco ou a blocos diferentes sob a mesma *extensão de paralelismo.*

EXEMPLO I:

```
USING   k := 1:100                        USING   i := 200:300 DO
1       X[k] := Q * (Y[k] + Z[k]);        2       Q := SUM (Z[k]);
                                          3       Q1 := Z[k] * X[k];

        Bloco 1                                   Bloco 2
```

- **GRAFO DE DEPENDÊNCIAS - Fig. 7**

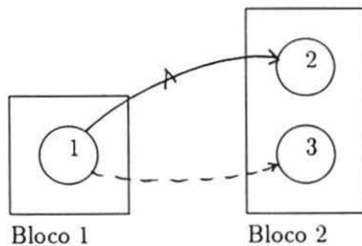


Fig. 7

Há uma *dependência direta* entre os comandos 1 e 3. A interseção entre as *extensões de paralelismo* dos dois blocos é vazia o que mostra que essa dependência é falsa. Existe também uma *antidependência* entre os comandos S_1 e S_2 envolvendo a variável escalar Q. Essa dependência é verdadeira pois não depende de nenhuma *extensão de paralelismo.* Os dois blocos só podem ser executados em paralelo se alguma primitiva de sincronização for introduzida. Não haverá nenhum ganho porque o bloco 2 tem de esperar o bloco 1 terminar toda a sua execução pois ele modifica o valor da variável Q. Para resolver aplica-se a técnica de *reordenação de comandos* invertendo a ordem de execução dos comandos 2 e 3. O *grafo de dependências* não se altera significativamente pois nenhuma dependência foi eliminada.

O bloco 2 pode ser dividido em dois blocos e com isso a execução será a seguinte: [(1 PAR 3) SEQ (2)]. Se isso não tivesse sido feito, haveria necessidade de uma primitiva

de sincronização *SINC* para existir algum paralelismo.

Uma coisa que se deve observar é que, em Actus, a aplicação das técnicas se confunde muito. A técnica de *quebra de comando* é uma das mais utilizadas e possui diversos fins. Neste exemplo que acabamos de ver a técnica que se utilizou, na segunda parte da otimização, não foi a *reordenação de comandos* mas a *distribuição de loop*. Isso porque se verificou que os comandos que pertenciam ao bloco 2 inicial eram totalmente independentes e podiam ser executados em paralelo. Neste caso bastaria que apenas a segunda técnica fosse aplicada para o problema estar resolvido. Essa versatilidade da aplicação das técnicas se dá devido à própria estrutura da linguagem e não diminui em nada a eficiência das técnicas estudadas.

EXEMPLO II:

USING	k := 1:100 DO	USING	i := 200:300 DO
1	A[k] := B[k] - Z[k];	5	D[j] := D[j] + 5;
2	B[k] := 2 * A[k] + C[k];	6	E[j] := E[j] + Z[j];
3	A[k] := A[k] + Z[k];	7	F[j] := F[j] + E[j];
4	D[k] := D[k] - C[k];		
END		END	
	Bloco 1		Bloco 2

- **GRAFO DE DEPENDÊNCIAS - Fig. 8**

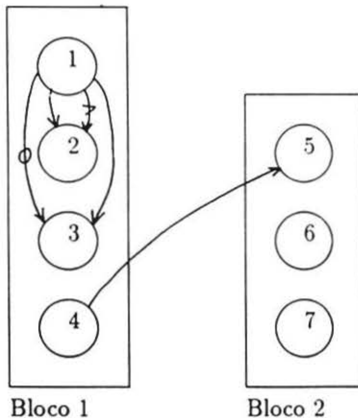


Fig. 8

O *grafo de dependências* mostra apenas uma *dependência direta* entre os dois blocos causada pelo vetor D[ ]. Se esses blocos fossem executados em paralelo, na ordem em que se encontram os comandos, não haveria ganho nenhum pois o último comando do bloco 1 produz o valor do vetor D[ ] que será utilizado no bloco 2. Aplicando a *reordenação de comandos* obtem-se o seguinte resultado:

```
USING     k := 1:100 DO              USING     i := 200:300 DO
4         D[k] := D[k] - C[k];       6         E[j] := E[j] + Z[j];
1         A[k] := B[k] - Z[k];       7         F[j] := F[j] + E[j];
2         B[k] := 2 * A[k] + C[k];   5         D[j] := D[j] + 5;
3         A[k] := A[k] + Z[k];
END                                  END
          Bloco 1                              Bloco 2
```

Como os comandos foram apenas reordenados, o *grafo de dependências* resultante da otimização não se altera. A dependência continua existindo entre os mesmos comandos mas verifica-se que quando o bloco 1 calcular o valor do vetor D[ ] o bloco 2 pode ser disparado e com isso os dois blocos executarão em paralelo. Isso é resolvido com um comando de *sincronização*. Apesar do bloco 2 ter sido também reordenado, não havia essa necessidade. A execução será a seguinte: [(4) (SINC (6 SEQ 7 SEQ 5)) PAR (1 SEQ 2 SEQ 3)].

4.6 Loop Blocking

O objetivo inicial é criar-se, a partir de um *loop*, um aninhamento duplo, para otimizar o tratamento de registradores internos. Com a aplicação da técnica e o aparecimento de um *loop* mais interno, as iterações são realizadas num passo k que é exatamente o tamanho dos registradores vetoriais.

A arquitetura para a qual o trabalho todo está voltado possui na verdade um processador de alto desempenho (*Intel i860*) que simula um processador vetorial. Não existem então, registradores vetoriais com um tamanho interno fixo, não havendo necessidade de nos preocuparmos com eles. Assim, inicialmente, a técnica parece não ter aplicação na linguagem Actus para a arquitetura proposta. Mas uma adaptação da idéia inicial, para a aplicação da técnica, é de se quebrarem blocos com grandes *extensões de paralelismo* transformando-os em blocos menores. Com isso os blocos seriam distribuídos por processadores diferentes e executados em paralelo. Essa idéia é bem semelhante à de distribuir os comandos de um *loop* (sem vetorizá-lo) usando um comando *DOALL*.

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);
```

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


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

O resultado obtido mostra os comandos distribuídos pelos P processadores. Como não há na linguagem, ainda, nenhum comando para expressar paralelismo entre iterações não se consegue perceber o ganho com a aplicação da técnica. Mas a execução será a seguinte: $\underbrace{[(1)PAR(1)PAR \ldots PAR(1)]}_{P vezes}$. Neste caso não houve preocupação com *balanceamento de carga* na rede. Apenas se dividiu a carga pelos P processadores e um deles executa o que ficou sobrando.

4.7 Distribuição do Loop

Esta técnica deve ser aplicada depois que outras técnicas de otimização tenham sido usadas porque trabalha em cima do *grafo parcialmente ordenado*, formado pelos *pi-blocos*. Cada *pi-bloco* é formado por um *comando independente* ou uma *recorrência*. Um comando múltiplo *DO*, por exemplo, pode ser quebrado em vários *pi-blocos*. 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.

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 não sofre alteração pois esse tipo de bloco representa uma estrutura seqüencial. No caso da análise feita em cima de blocos pertencentes ao grupo 1, deve-se estudar as dependências com dois enfoques diferentes: verificando dependências entre comandos de um mesmo bloco ou verificando dependências entre blocos. No primeiro caso, o objetivo é paralelizar os *comandos independentes*. No segundo caso, o objetivo é paralelizar blocos. A análise em cima dos comandos é feita da mesma forma que nas linguagens seqüenciais, com uma diferença: não pode haver nenhuma dependência (de nenhum tipo) ligando esses comandos. Na análise em cima dos blocos, cada um deles será visto como um comando e a distribuição visa paralelizar os blocos que forem independentes. Os blocos que não puderem ser executados em paralelo, como conseqüência de pelo menos uma dependência entre eles, formam uma *região fortemente conectada*. Pode se tentar resolver esse problema usando-se algum tipo de sincronização como já foi visto.

Nos exemplos mostrados até agora, de aplicação de outras técnicas, a técnica de *distribuição do loop* já vinha sendo usada no final de cada exemplo para demonstrar o paralelismo entre blocos.

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.

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.

EXEMPLO I:

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

- GRAFO DE DEPENDÊNCIAS - Fig. 9

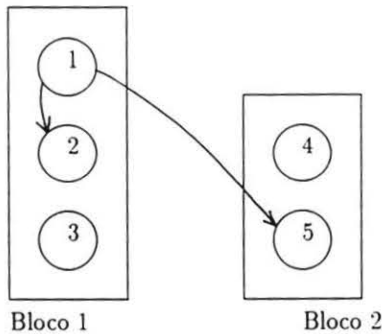


Fig. 9

No *grafo de dependências* verifica-se que os comandos 3 (bloco 1) e 4 (bloco 2) são independentes, entre si e dentro dos seus respectivos blocos. Por isso podem ser colocados em blocos separados e executados em paralelo. No grafo existe também uma *dependência direta* envolvendo apenas comandos do bloco 1 (1 dd 2) que não pode ser eliminada e uma *dependência direta* envolvendo comandos dos dois blocos (1 dd 5). Com algum mecanismo de sincronização talvez se possa superar essa última dependência.

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

```
USING    W := 1:N                  USING    M := 1:N
4        Y[W] := C[W] * K;         5        P[M] := D[M] + A[M];
         Bloco 2a                           Bloco 2
```

Como esta técnica não elimina nenhuma dependência o grafo não se altera. Dois blocos ficam totalmente independentes (1a e 2a) e um terceiro bloco (bloco 2) parcialmente independente, em relação ao bloco 1. A execução será a seguinte:
[(3) PAR (4) PAR (1) (SINC (5)) SEQ 2)].

EXEMPLO II:

```
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
```

- **GRAFO DE DEPENDÊNCIAS - Fig. 10**

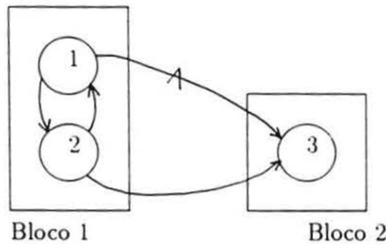


Fig. 10

No grafo há a presença de um ciclo formado por duas *dependências diretas* e envolvendo os comandos 1 e 2. Esse ciclo é inquebrável formando assim uma *recorrência*. Existe também um *comando independente* (3). A técnica deve então ser aplicada tendo em vista que o bloco do exemplo pertence ao grupo 2.

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

Como resultado final o comando 3 foi vetorizado.

5 Conclusões

O objetivo inicial do trabalho, otimizar programas escritos em ACTUS II, foi alcançado pois todas as técnicas puderam se adaptar à linguagem e produziram ganho em termos de paralelismo. Não existem resultados numéricos porque a primeira parte do trabalho foi puramente teórica. Atualmente está-se escrevendo os algoritmos para a implementção das técnicas e estudando of problema do balanceamento de carga para uma rede de processadores. O resultado final pode ficar em duas formas: um *feed-back* para o programador, que teria de voltar ao programa fonte para modificá-lo; ou um arquivo de saída com todas as otimizações aproveitando as características paralelas assíncronas da linguagem OCCAM 2. Esta última forma inclui a implementação das primitivas SEQ, PAR e SINC através de OCCAM 2.

6 Referências

Referências

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

[2] 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.

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

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

[5] 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).

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