REESTRUTURAÇÃO AUTOMÁTICA DE PROGRAMAS SEQÜENCIAIS PARA PROCESSAMENTO PARALELO

Fábio Carneiro Mokarzel
Divisão de Processamento de Dados
Instituto Tecnológico de Aeronáutica
Centro Técnico Aeroespacial
12225 – São José dos Campos, SP

Jairo Panetta
Instituto de Estudos Avançados
Centro Técnico Aeroespacial
12225 – São José dos Campos, SP

RESUMO

Este trabalho descreve diversos métodos para transformar, automáticamente, trechos de programas seqüenciais em trechos de programas paralelos equivalentes.

ABSTRACT

This work describes several methods to transform, automatically, sequential program fragments into equivalent parallel program fragments.

1. INTRODUÇÃO

As linguagens de programação hoje disponíveis em computadores paralelos podem ser grosseiramente classificadas em três grupos. O primeiro grupo é formado por novas linguagens projetadas especificamente para processamento paralelo. O segundo grupo é formado por linguagens convencionais ampliadas por primitivas para expressar paralelismo. O terceiro grupo é formado por linguagens convencionais sem extensões, onde o paralelismo é extraído automaticamente pelo compilador.

Atualmente, todos os computadores paralelos comercialmente disponíveis são programados ou por meio de linguagens convencionais com extensões ou por meio de novas linguagens paralelas. Este fato é inconveniente para os usuários, visto que o uso de tais linguagens exige alterações de monta nos seus programas. Para evitar tal esforço, deve-se utilizar linguagens convencionais sem extensões, onde a tarefa de extrair paralelismo é efetuada automaticamente pelo compilador. A importância desta opção cresce quando se lembra que há trinta anos de programas desenvolvidos em FORTRAN, alguns deles (justamente os de maior interesse) com milhões de linhas de código fonte.

Este artigo descreve diversas técnicas destinadas à reestruturação automática de programas seqüenciais visando a obtenção de um programa paralelo equivalente. A secção 2 descreve as fases de um compilador-detector de paralelismo, enquanto a secção 3 apresenta diversas técnicas utilizadas durante a reestruturação.

2. FASES DE UM DETECTOR AUTOMÁTICO

Um compilador-detector de paralelismo possue todas as fases usuais de um compilador mais duas fases exclusivas, uma de detecção de paralelismo e outra de alocação de recursos.

A fase de detecção de paralelismo produz sucessivas versões do programa fonte, aplicando transformações que mantém a semântica original. Essas transformações buscam obter a independência, quanto à ordem de execução, dos comandos que compõe o programa. Quanto maior a independência entre comandos, maior a simultaneidade na execução e maior o grau de paralelismo atingido.

As transformações utilizam o conceito de grafo de dependências, introduzido na sub-secção 2.1. As atividades da última fase do detector, relativas à alocação de recursos, são brevemente comentadas na sub-secção 2.2.

2.1. Grafo de dependências.

Grafo de dependências a nível de comandos é um grafo orientado em que os vértices são os vários comandos do programa e as arestas são as dependências entre eles. Dois comandos são ditos dependentes quando o resultado da computação que os contém pode ser modificado se a ordem original de execução desses comandos for alterada. A seguir, definem-se alguns tipos de dependências.

Sejam dois comandos de atribuição, c_1 e c_2, com c_1 executado antes de c_2. Existe uma dependência direta de c_1 para c_2 (c_1 dd c_2) se o valor atribuido à variável alvo de c_1 for utilizado por c_2. Existe uma antidependência de c_1 para c_2 (c_1 da c_2), caso c_1 utilize uma variável cujo valor é alterado por c_2. Existe uma dependência de saída de c_1 para c_2 (c_1 ds c_2), se os dois comandos possuem a mesma variável alvo da atribuição.

Existe uma dependência condicional de b para c (b dc c) se b for o cabeçalho de um comando condicional e c for um comando no escopo de b. Finalmente, sendo d o cabeçalho de um comando repetitivo e c um comando em seu escopo, existe uma dependência de repetição de d para c (d dr c).

Dependências no escopo de comandos repetitivos são munidas de uma direção ([8]) e uma distância ([9]). No comando

```
      do i = 1, n
c1:       a(i) = b(i) + 5
c2:       c(i) = a(i) * 2
      end do
```

a dependência (c_1 dd c_2) é interna à iteração i. Diz-se que sua direção é do tipo "=" e sua distância é zero. Já no comando

```
      do i = 1, n
c1:       a(i) = b(i) + 5
c2:       c(i) = a(i - 3) * 2
      end do
```

a dependência (c_1 *dd* c_2) vai da iteração i para a iteração $i-3$. Diz-se então que sua direção é do tipo "<" e sua distância é 3. Também existem direções do tipo ">" e distâncias negativas, como veremos a seguir.

Em aninhamentos de comandos repetitivos é possível existir mais do que uma direção e uma distância para cada dependência, devido aos múltiplos índices. Ordenando-se as direções e as distâncias de acordo com a seqüência de comandos repetitivos, constroem-se vetores chamados de vetor direção e vetor distância. Por exemplo, no comando

```
      do i = 1, n
         do j = 1, n
c1:         a(i, j) = b(i, j) + c(i, j)
c2:         d(i, j) = a(i - 2, j + 1) + e(i, j)
         end do
      end do
```

a dependência (c_1 *dd* c_2) tem o vetor direção (<, >) e o vetor distância $(2, -1)$.

2.2. Alocação de recursos.

Uma vez reestruturado, o programa estará particionado em conjuntos de tarefas tais que todas as tarefas de um mesmo conjunto podem ser executadas em paralelo. Já os conjuntos de tarefas devem ser executadas em uma certa ordem, devido às dependências entre os componentes desses conjuntos.

Como diversas técnicas de reestruturação não consideram o número de processadores efetivamente disponíveis, é necessário alocar tarefas a processadores em uma fase posterior à reestruturação. Além disso, para uma tarefa formada, por exemplo, por um aninhamento de comandos repetitivos, é necessário um esquema para distribuir os comandos pelos processadores visando otimizar a eficiência, o que é um problema complexo.

Recentes pesquisas ([10], [1], [11], [12]) tem sido dirigidas para este problema e muitas heurísticas tem sido produzidas, uma vez que várias fases desta atividade são problemas NP-completos. Há alternativas de escalonar processadores estaticamente (durante a compilação) ou dinamicamente (durante a execução), ou ainda de forma mista.

3. TÉCNICAS DE REESTRUTURAÇÃO

As técnicas para reestruturar programas seqüenciais podem ser classificadas em três grupos, a saber: técnicas para a geração de comandos paralelos de alto nível, técnicas para a eliminação de dependências e de cálculos desnecessários e técnicas para adequar o programa às características da máquina.

Como cada técnica resolve uma classe de dependências, há de se escolher um elenco de técnicas que explicite o maior grau possível de paralelismo. Infelizmente, há técnicas que ao eliminar um tipo de dependência introduz outro tipo, de forma que a seqüência de aplicação das técnicas também é uma questão em aberto, dependendo dos programas alvo da reestruturação. Programas distintos sugerem seqüências distintas, eventualmente com repetições de seqüências de técnicas em um processo iterativo, utilizando a estabilidade do grafo de dependências como critério de convergência ([3]). No entanto, estabelecer uma seqüência fixa contendo todas as técnicas ainda é um problema em aberto.

Recentes pesquisas tem sido dedicadas ao estudo de reestruturação interativa entre o compilador e o usuário ([13]). O compilador mostra um trecho do programa e sugere um elenco de técnicas a serem aplicadas. O usuário, com um conhecimento mínimo do assunto, seleciona uma ou mais técnicas do elenco.

A seguir, é apresentada uma lista não exaustiva de técnicas de reestruturação. Devido a limitações de espaço, não apresentaremos algorítmos para detecção de dependências. Sugerimos que os interessados no assunto consultem as referências [3], [4], [5], [6] e [7].

3.1. Geração de comandos paralelos.

Esta classe de técnicas é utilizada para gerar comandos de alto nível existentes nas extensões de linguagens convencionais para computadores vetoriais ou para multiprocessadores. Claramente, estas mesmas técnicas podem ser aplicadas para gerar o código objeto, diretamente.

3.1.1. Vetorização de comandos repetitivos.

Comandos repetitivos aninhados podem ser decompostos em diversos comandos repetitivos de menor escopo, e aqueles resultantes que não possuam ciclos de dependências em seu escopo podem ser vetorizados ([8], [3], [5]). Por exemplo, o comando repetitivo

```
      do i = 1, n
c1:      a(i) = b(i) + c(i)
c2:      b(i + 1) = a(i) + s
c3:      d(i) = a(i - 1) + r
c4:      f(i + 1) = d(i - 1) + e(i)
      end do
```

pode ser transformado em

```
      do i = 1, n
c1:      a(i) = b(i) + c(i)
c2:      b(i + 1) = a(i) + s
      end do
c3:   d(1 : n) = a(0 : n - 1) + r
c4:   f(2 : n + 1) = d(0 : n - 1) + e(1 : n)
```

onde a notação $d(1 : n)$ representa o vetor formado pelos n primeiros elementos de d.

3.1.2. Vetorização por redução.

Diversos computadores vetoriais possuem instruções de máquina que implementam operadores matemáticos de redução, como por exemplo o somatório ([8], [6], [3]). Para estas máquinas, técnicas de redução transformam o fragmento de programa

```
      do i = 1, n
c1:      a(i) = b(i) * c(i)
c2:      s = s + a(i)
      end do
```

em

```
c1:   a(1 : n) = b(1 : n) * c(1 : n)
c2:   s = s + somatorio(a(1 : n))
```

3.1.3. Distribuição de iterações por processadores.

As iterações de um comando repetitivo podem ser distribuidas pelos processadores em um multiprocessador. Algumas extensões de linguagens seqüenciais contém primitivas para tal. Por exemplo, o trecho seqüencial

$$\begin{array}{l}\textbf{do } i = 1, n\\ \quad a(i) = b(i) + c(i)\\ \quad b(i) = a(i)/2\\ \textbf{end do}\end{array}$$

pode ser convertido em

$$\begin{array}{l}\textbf{doall } i = 1, n\\ \quad a(i) = b(i) + c(i)\\ \quad b(i) = a(i)/2\\ \textbf{end doall}\end{array}$$

O comando *doall* expressa que instâncias distintas da seqüência de comandos em seu escopo podem ser executadas simultaneamente. Por exemplo, sendo p o número de processadores e $p < n$, para todo i em $\{1, 2, \ldots, p\}$ o processador i pode executar as iterações i, $i + p$, $i + 2p$, etc. O comando *doall* requer que as dependências entre os comandos em seu escopo tenham direções do tipo "=". No casos de dependências com outras direções, devem ser inseridas primitivas de sincronização ([8], [14]). Por exemplo, o fragmento de programa seqüencial

$$\begin{array}{l}\textbf{do } i = 1, n\\ \quad a(i) = b(i) + c(i)\\ \quad d(i) = a(i)/5\\ \quad e(i) = f(i) - a(i-1)\\ \textbf{end do}\end{array}$$

é equivalente à execução, em cada um dos n processadores, do trecho de programa

$$\begin{array}{l}a(i) = b(i) + c(i)\\ \textbf{if } (i < n)\ \mathit{signal}(s(i))\\ d(i) = a(i)/5\\ \textbf{if } (i > 2)\ \mathit{wait}(s(i-1))\\ e(i) = f(i) + a(i-1)\end{array}$$

onde i é a identidade de cada um dos n processadores.

3.1.4. Paralelização parcial de comandos repetitivos.

Quando existirem dependências com direções "<" e ">", as iterações de um comando repetitivo não podem ser executadas em total simultaneidade. Há casos porém, em que é possível uma simultaneidade parcial, ou seja, uma iteração pode começar a ser executada antes do término da iteração anterior, bastando respeitar as dependências existentes.

O último exemplo do ítem 3.1.3 mostra que essa paralelização parcial pode ser implementada por meio de primitivas de sincronização. Naquele caso, se todos os iniciarem a execução simultaneamente e trabalharem numa mesma velocidade, a sincronização pode ser desnecessária pois o comando fonte da dependência aparece lexicamente antes, no programa, que o comando dele dependente, podendo ser executado antes do outro começar.

A referência [14] apresenta um método para os casos em que o comando fonte da dependência aparece lexicamente depois do comando dele dependente, num ambiente em que os processadores trabalham numa mesma velocidade e começam a executar as iterações de um comando repetitivo simultaneamente.

O método introduz comandos de demora no início de cada iteração dos comandos repetitivos. Por exemplo, o fragmento de programa seqüencial

$$\begin{array}{ll}& \textbf{do } i = 1, 3\\ c_1: & \quad a(i) = b(i) + 5\\ c_2: & \quad b(i+1) = a(i) **2\\ c_3: & \quad c(i) = b(i) + f(i)\\ c_4: & \quad e(i) = c(i)/3\\ & \textbf{end do}\end{array}$$

pode ser executado em uma máquina paralela com três processadores na forma:

proc 1	proc 2	proc 3
$c_1(1)$		
$c_2(1)$		
$c_3(1)$	$c_1(2)$	
$c_4(1)$	$c_2(2)$	
	$c_3(2)$	$c_1(3)$
	$c_4(2)$	$c_2(3)$
		$c_3(3)$
		$c_4(3)$

utilizando-se o trecho de programa

$$\begin{array}{ll}& \textbf{doacross } i = 1, 3\\ & \quad \mathit{delay}((i-1) * d)\\ c_1: & \quad a(i) = b(i) + 5\\ c_2: & \quad b(i+1) = a(i) **2\\ c_3: & \quad c(i) = b(i) + f(i)\\ c_4: & \quad e(i) = c(i)/3\\ & \textbf{end doacross}\end{array}$$

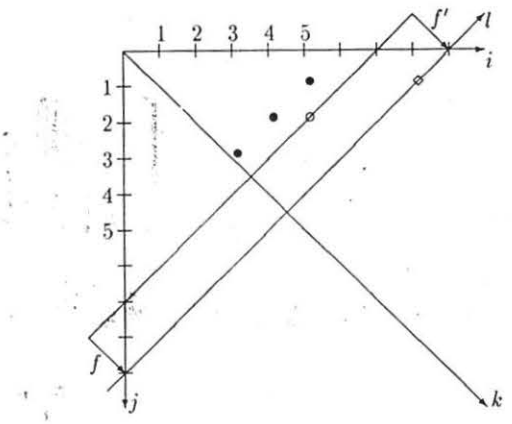
onde d representa o tempo necessário para que o processador 1 execute $c_1(1)$ e $c_2(1)$.

3.1.5. Método da frente de onda.

Usado para obter paralelismo em aninhamentos de comandos de repetição quando seu escopo contiver ciclos de dependências. Para ilustrá-lo, considere o fragmento de programa:

$$\begin{array}{l}\textbf{do } i = 0, 9\\ \quad \textbf{do } j = 0, 9\\ \quad\quad a(i,j) = a(i-1,j) + a(i,j-1)\\ \quad\quad\quad + a(i-2,j+1) + a(i+3,j-1)\\ \quad \textbf{end do}\\ \textbf{end do}\end{array}$$

Para as atribuições $i = 5$ e $j = 2$, a figura abaixo representa por o a instância $(2,5)$, por • as instâncias anteriores que influenciam o cálculo de $(2,5)$ e por ◇ a instância posterior que influenciam o mesmo cálculo.



Todas as instâncias cortadas pela "frente de onda" ff' podem ser executadas em paralelo, mediante a transformação

```
do k = -9,9
   m = 9 - abs(k)
   doall l = -m,m,2
         a((k+l+9)/2,(k-l+9)/2) =
                        a((k+l+7)/2,(k-l+9)/2)
                        +a((k+l+9)/2,(k-l+7)/2)
                        +a((k+l+5)/2,(k-l+11)/2)
                        +a((k+l+15)/2,(k-l+7)/2)
   end doall
end do
```

Este método pode ser generalizado para um maior número de aninhamentos e de comandos de atribuição, conforme citado em ([15], [16], [17], [6], [3]).

3.1.6. Reordenação de comandos de atribuição.

Possibilita a vetorização de comandos repetitivos de escopo não unitário e minimiza os efeitos negativos da sincronização de programas concorrentes ([18]). Por exemplo, o trecho seqüencial

```
      do i = 1,n
c1:      a(i) = b(i) + c(i)
c2:      d(i) = a(i+1) * *2
      end do
```

pode ser automaticamente transformado no trecho seqüencial

```
      do i = 1,n
c2:      d(i) = a(i+1) * *2
c1:      a(i) = b(i) + c(i)
      end do
```

resultando no trecho vetorizado

$$d(1:n) = a(2:n+1) * *2$$
$$a(1:n) = b(1:n) + c(1:n)$$

3.2. Eliminação de dependências desnecessárias.

Durante a elaboração de um programa seqüencial é comum atribuir múltiplas interpretações a uma mesma posição de memória ou mesmo introduzir cálculos desnecessários, degradando o potencial de paralelismo do programa pelo estabelecimento de dependências desnecessárias. Muitos destes casos podem ser detectados e resolvidos pelo compilador.

3.2.1. Uso de novos identificadores.

O uso de um mesmo identificador com propósitos diferentes em diversos pontos do programa pode introduzir antidependências e dependências de saída. Para eliminá-las, basta atribuir um identificador para cada propósito da variável ([7], [3], [8]).

Por exemplo, no trecho a seguir o identificador r tem duas interpretações, por ser alvo tanto da atribuição c_1 quanto de c_3:

```
      do i = 1,n
c1:      r = a(i) + b(i)
c2:      c(i) = r/2
c3:      r = r + 3
c4:      u = r * *2 + u
      end do
```

Essa dupla interpretação gera dependências desnecessárias como (c_1 *ds* c_3) e (c_4 *da* c_1), que podem ser eliminadas por

```
      do i = 1,n
c1:      r1 = a(i) + b(i)
c2:      c(i) = r1/2
c3:      r2 = r1 + 3
c4:      u = r2 * *2 + u
      end do
```

3.2.2. Expansão de variáveis escalares.

A atribuição de valores a variáveis escalares dentro de comandos repetitivos é também causa de antidependências e dependências de saída, além de impedir a vetorização ou exigir excessiva sincronização. Para eliminar tais dependências, substituem-se as variáveis escalares por indexadas ([7], [3], [8]), conforme ilustrado a seguir.

```
      do i = 1,n
c1:      r = s + 1
c2:      t = 2 * r
c3:      s = a(i) + b(i)
      end do

      sa(0) = s
      do i = 1,n
c1:      ra(i) = sa(i-1) + 1
c2:      ta(i) = 2 * ra(i)
c3:      sa(i) = a(i) + b(i)
      end do
      r = ra(n)
      s = sa(n)
      t = ta(n)
```

Esta expansão é apropriada para máquinas vetoriais, mas há o equivalente para multiprocessadores. Após esta expansão, recomenda-se a reordenação dos comandos de atribuição (3.1.6).

3.2.3. Fissão de comandos.

Alguns comandos repetitivos contém ciclos de dependências que podem ser eliminados através de atribuições temporárias ([7], [8], [3]). Por exemplo, o ciclo de dependências (c_1 *dd* c_2) e (c_2 *da* c_1) no trecho

```
      do i = 1,n
c1:      a(i) = b(i) - c(i)
c2:      d(i) = a(i+1) * a(i-1)
      end do
```

não existe no trecho

```
      do i = 1,n
c21:     at(i) = a(i+1)
c1:      a(i) = b(i) - c(i)
c2:      d(i) = at(i) * a(i-1)
      end do
```

devido à introdução da variável $at(i)$, o que elimina a dependência (c_2 *da* c_1).

3.2.4. Fissão de comandos repetitivos.

Há casos em que um comando repetitivo tem só duas atribuições em seu escopo, e estas formam um ciclo com uma dependência direta e uma antidependência, e este ciclo pode ser quebrado mediante uma verdadeira fissão, sem introduzir variáveis auxiliares. Por exemplo, o comando:

```
      do i = 1,50
c1:     v(-30 + 2 * i) = y(i)
c2:     z(i) = v(70 - 3 * i)
      end do
```

pode ser partido nos seguintes comandos ([5], [3]):

```
      do i = 1,20
c11:    v(-30 + 2 * i) = y(i)
      end do
      do j = 1,19
c21:    z(j) = v(70 - 3 * j)
      end do
      do k = 20,50
c22:    z(k) = v(70 - 3 * k)
      end do
      do l = 21,50
c12:    v(-30 + 2 * l) = y(l)
      end do
```

3.2.5. Substituição de variáveis por expressões.

Em caso de dependência direta entre dois comandos de atribuição, pode-se substituir a variável causadora da dependência pela expressão que a atualiza, eliminando-se a dependência ([17], [3], [8]). Por exemplo, as dependências (c_1 dd c_2) e (c_2 dd c_3) no trecho

```
c1:   a = b + c
c2:   d = a + e
c3:   f = d + g
```

foram eliminadas no trecho

```
c1:   a = b + c
c2:   d = b + c + e
c3:   f = b + c + e + g
```

o que torna c_1, c_2 e c_3 independentes.

3.2.6. Eliminação de variáveis de indução.

Variáveis de indução são funções dos índices de comandos repetitivos, com valores formando uma progressão aritmética. Quando aparecem como subscritos de variáveis indexadas, induzem falsas dependências que podem ser eliminadas ([8], [18]), como é o caso de $i2$ no exemplo a seguir:

```
do i = 1,n
   i2 = 2 * i - 1
   a(n - i) = b(i) + c(i2)
end do

do i = 1,n
   a(n - i) = b(i) + c(2 * i - 1)
end do
i2 = 2 * n - 1
```

3.2.7. Eliminação de pseudo-variáveis de indução.

Às vezes uma variável não é de indução unicamente pelo seu valor inicial, como é o caso da variável j no exemplo a seguir. O método de eliminação é similar ao anterior ([8], [18]).

```
j = k
do i = 1,n
   a(i) = b(j) + b(j + 1)
   j = i
end do

j = k
if (n ≥ 1) then
   a(1) = b(j) + b(j + 1)
end if
do i = 2,n
   a(i) = b(i - 1) + b(i)
end do
```

3.2.8. Eliminação de recorrências de índices.

Alguns índices de variáveis indexadas são obtidos por recorrências internas a comandos repetitivos que geram falsas dependências ([18]). Por exemplo:

```
j = 0
do i = 1,n
   j = j + i
   a(i) = a(i) + b(j)
end do

do i = 1,n
   a(i) = a(i) + b(i * (i + 1)/2)
end do
```

3.2.9. Versões de comandos repetitivos.

Seja o comando repetitivo:

```
      do i = 1,n
c1:     a(i) = b(i) + c(i)
c2:     d(i) = a(i - m) + e(i)
      end do
```

A dependência direta deixa de existir caso $m < 0$ e a antidependência deixa de existir caso $m \geq 0$, permitindo a seguinte vetorização:

```
if (m ≥ 0) then
  a(1 : n) = b(1 : n) + c(1 : n)
  d(1 : n) = a(1 - m : n - m) + e(1 : n)
else
  d(1 : n) = a(1 - m : n - m) + e(1 : n)
  a(1 : n) = b(1 : n) + c(1 : n)
end if
```

Pode-se também criar várias versões de um comando repetitivo, gerando códigos escalar, vetorial, concorrente e vetorial-concorrente, decidindo-se por um deles em tempo de execução, conforme sugerido na referência [19].

3.2.10. Desdobramento de comandos repetitivos.

Algumas vezes é util isolar algumas iterações iniciais ou finais de um comando repetitivo para aplicar outras técnicas. A referência [3] mostra que esta técnica pode substituir variáveis por expressões dentro de comandos repetitivos. Já [18] a aplica para eliminar pseudo-variáveis de indução e cálculos excessivos, como para eliminar a função *min* no exemplo:

```
do j = 1, n, k
   do i = j, min(j + k − 1, n)
      a(i) = b(i) + c(i)
   end do
end do

n1 = trunc(n/k)
n2 = n1 * k
do j = 1, n2, k
   do i = j, j + k − 1
      a(i) = b(i) + c(i)
   end do
end do
do i = n2 + 1, n
   a(i) = b(i) + c(i)
end do
```

3.2.11. Retrocesso de ciclos.

No comando a seguir,

```
      do i = 5, n
c1:      a(i) = b(i − 4) + c(i)
c2:      b(i) = a(i − 5) − d(i)
      end do
```

as dependências (c_1 *dd* c_2) e (c_2 *dd* c_1) têm distâncias 5 e 4. Pode-se dividir as iterações deste comando em grupos livres de dependências de acordo com estas distâncias ([18]), como em:

```
do j = 5, n, 4
   do i = j, j + 3
      a(i) = b(i − 4) + c(i)
      b(i) = a(i − 5) − d(i)
   end do
end do
```

o que permite a vetorização da repetição mais interna.

3.2.12. Testes em tempo de execução.

O desconhecimento de certos valores em tempo de compilação, tais como limites de comandos repetitivos, condições de comandos condicionais, etc, introduz certas dependências que podem deixar de existir uma vez conhecidos seus valores em tempo de execução. Recentes estudos ([18]) tem introduzido testes no programa a ser compilado, com a finalidade de eliminar essas dependências e poderão alterar bastante o cenário atual.

3.3. Adaptação do programa à máquina.

Arquiteturas finitas sempre impõem restrições à execução paralela de programas, seja pelo número de processadores, pela taxa de transferência de memória ou do sistema de entrada e saída, ou seja pelo modo de interconexão dos seus vários componentes. Cada compilador deve esconder estas restrições do usuário e produzir código que utilize as características favoráveis das máquinas alvo.

3.3.1. Reordenação de comandos aninhados.

A troca de posição de cabeçalhos em um aninhamento de comandos repetitivos pode ser providencial em várias circunstâncias. Quando o comando repetitivo mais interno do aninhamento não pode ser vetorizado, pode-se trazer para a sua posição outro comando repetitivo "vetorizável", como em:

```
do i = 1, n
   do j = 1, n
      a(i, j) = a(i, j − 1) + b(j)
   end do
end do

do j = 1, n
   a(1 : n, j) = a(1 : n, j − 1) + b(j)
end do
```

3.3.2. Particionamento de comandos repetitivos.

Esta técnica transforma um comando repetitivo simples num aninhamento de dois níveis, agrupando as iterações em conjuntos de tamanho aproximadamente igual. Ela é útil para ajustar vetores ao tamanho de registradores vetoriais e para gerar comandos concorrentes em número igual ao dos processadores de um multiprocessador ([18], [8]). Segue-se uma ilustração para máquinas com registradores vetoriais de 64 elementos:

```
do i = 1, n
   a(i) = b(i) + c(i)
end do

do j = 1, n, 64
   m = min(n, j + 63)
   do i = j, m
      a(i) = b(i) + c(i)
   end do
end do
```

3.3.3. Colapso de comandos repetitivos.

Quanto maiores os vetores manipulados por máquinas vetoriais, melhor é o desempenho dessas máquinas, desde que esses vetores caibam nos seus registradores vetoriais. Esta técnica transforma um aninhamento de dois níveis num comando repetitivo simples, linearizando as variáveis indexadas, com o objetivo de aumentar o comprimento dos vetores ([8]). Por exemplo:

```
do i = 0, 7
   do j = 0, 7
      a(i, j) = b(i, j) + c(i, j)
   end do
end do

do k = 0, 63
   a(k) = b(k) + c(k)
end do
```

3.3.4. Linearização de comandos repetitivos.

Aninhamentos de comandos repetitivos totalmente paralelos podem ser transformados num só comando, sem linearizar suas variáveis indexadas. Isto é útil em computadores capazes de executar em paralelo apenas comandos repetitivos desaninhados. Além disso, a alocação de recursos é mais complexa e ineficiente para aninhamentos muito grandes ([18], [12]). Como exemplo, o aninhamento

```
doall j = 1,n
    doall k = 1,n
        a(j,k) = ***
    end doall
end doall
```

pode ser assim linearizado :

```
doall i = 1,n**2
    a(teto(i/n),i - n*piso((i - 1)/n)) = ***
end doall
```

Há casos em que aninhamentos de comandos repetitivos não totalmente paralelos podem também ser linearizados.

4. CONCLUSÕES

O trabalho de reestruturação de um programa é extenso, complexo e indefinido, devido à falta de um critério para determinar as técnicas a serem utilizadas e a ordem de aplicação. Esta coletânea, ainda que extensa, é incompleta, e muitas outras técnicas poderiam ter sido mencionadas. Os algorítmos e as implementações destas técnicas são bem documentados na literatura, tendo sido objeto de diversas teses de doutorado ([5], [15], [21], [22], [23], [24], [25], [26]).

Além disso, outras atividades de um compilador-detector de paralelismo, tais como o estudo de alocação de recursos e de sincronização de programas reestruturados, despontam como fértil campo de pesquisas presentes e futuras.

AGRADECIMENTOS

Os autores agradecem a Sueli Maria Vicente pela prestimosa colaboração prestada durante a preparação da versão final deste trabalho.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Polychronopoulos, C. D. e Banerjee, U., "Processor allocation for horizontal and vertical parallelism and related speedup bounds", IEEE Transactions on Computers, 36(4), pg.410–420, abril de 1987.

[2] Amdahl, G. M., "Validity of the single processor approach to achieving large scale computing capabilities", Proceedings of the AFIPS Computing Conference, 30, 1967.

[3] Mokarzel, F. C., "Compilador de programas seqüenciais para multiprocessamento: análise e metodologia para sua implementação", Tese de Mestrado em CiΦncias, Instituto Tecnológico de Aeronáutica, SΣo JosΘ dos Campos, SP, outubro de 1984.

[4] Mokarzel, F. C., " Compilação de programas seqüenciais para multiprocessamento", Anais do I Simpósio Brasileiro de Arquitetura de Computadores e Processamento Paralelo, 25–37, Gramado, RS, maio de 1987.

[5] Banerjee, U., "Speedup of ordinary programs", Ph.D. Thesis, University of Illinois, Urbana, IL, outubro de 1979.

[6] Kuck, D. J., "A survey of parallel machine organization and programming", Computer Surveys, 9(1), pg 29–59, março de 1977.

[7] Kuck, D. J. e Kuhn, R. H. e Padua, D. A., e Leasure, B. e Wolfe, M. J., "Dependence graphs and compiler optimizations", Proceedings of the 8th ACM Symposium on Principles of Programming Languages, pg. 207–218, Williamsburg, VA, janeiro de 1981.

[8] Padua, D. A. e Wolfe, M. J., "Advanced compiler optimizations for supercomputers", Communications of the ACM, 29(12), pg 1184–1201, dezembro de 1986.

[9] Padua, D. A. e Kuck, D. J. e Lawrie, D. H., "High-speed multiprocessors and compilation techniques", IEEE TC, 29(9), pg. 763–776, setembro de 1980.

[10] Polychronopoulos, C. D. e Kuck, D. J., "Guided self-scheduling: a practical scheduling scheme for parallel supercomputers", IEEE Transactions on Computers, 36(12), pg. 1425–1439, dezembro de 1987.

[11] Cytron, R. G., "Useful parallelism in a multiprocessor environment", Proceedings of the 1985 International Conference on Parallel Processing, 450-457, agosto de 1985.

[12] Polychronopoulos, C. D., "Loop coalescing: a compiler transformation for parallel machines", Proceedings of the 1987 International Conference on Parallel Processing, 235–242, agosto de 1987.

[13] Gannon, D. e Atapattu, D. e Lee, M. H. e Shei, B., "A software tool for building supercomputer applications", Technical Report 224, Indiana University, Bloomington, IN, agosto de 1987.

[14] Cytron, R. G., "Doacross: beyond vectorization for multiprocessors", Proceedings of the 1986 International Conference on Parallel Processing, 836-844, agosto de 1986.

[15] Muraoka, Y., "Parallelism exposure and exploitation in programs", Ph.D. Thesis, University of Illinois, Urbana, IL, fevereiro de 1971.

[16] Lamport, L., "The parallel execution of DO loops", Communications of the ACM, 17(2), pg 83–93, fevereiro de 1974.

[17] Kuck, D. J. e Muraoka, Y. e Chen, S. C., "On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup", IEEE Transactions on Computers, 21(12), pg 1293–1310, dezembro de 1972.

[18] Polychronopoulos, C. D., "Automatic restruturing of Fortran programs for parallel execution", 1987 Conference on Parallel Processing in Science and Engineering, Bonn, WG, junho de 1987.

[19] Byler, M. e Davies, J. e Huson,C. e Leasure,B. e Wolfe,M.J., "Multiple version loops", Proceedings of the 1987 International Conference on Parallel Processing, pg. 312–318, agosto de 1987.

[20] Wolfe, M. J., "Advanced loop interchange", Proceedings of the 1986 International Conference on Parallel Processing, pg 536–543, agosto de 1986.

[21] Padua, D. A., "Multiprocessors: discussion of some theoretical and practical problems", Ph.D. Thesis, University of Illinois, Urbana, IL, outubro de 1979.

[22] Wolfe, M. J., "Optimizing supercompilers for supercomputers", Ph.D. Thesis, University of Illinois, Urbana, IL, outubro de 1982.

[23] Polychronopoulos, C. D., "On program restructuring, scheduling, and communication for parallel processor systems", Ph.D. Thesis, University of Illinois, Urbana, IL, agosto de 1987.

[24] Cytron, R. G., "Compile-time scheduling and optimization for asynchronous machines", Ph.D. Thesis, University of Illinois, Urbana, IL, outubro de 1984.

[25] Allen, J. R., "Dependence analysis for subscripted variables and its application to program transformations", Ph.D. Thesis, Rice University, Houston, TX, abril de 1983.

[26] Veidenbaum, A. "Program optimization and architecture desing issues for high-speed multiprocessors", Ph.D. Thesis. University of Illinois, Urbana, IL, 1985.