Paralelização Automática

Debora M. R. de Medeiros
Diego H. de Campos
Giampaolo L. Libralão
Murilo C. Naldi
Thiago Fernando Alves

Introdução

- Esconder paralelismo do programador
- Programa seqüencial
  - Técnicas para detectar:
    - Possibilidades de paralelismo
    - Transformações que permitam a paralelização

Introdução

- Detecção de paralelismo
  - Dois comandos podem ser executados em paralelo se produzirem os mesmos resultados quando executados em qualquer ordem
    - Entradas de um não são saídas de outro
    - Saídas de um não são saídas de outro

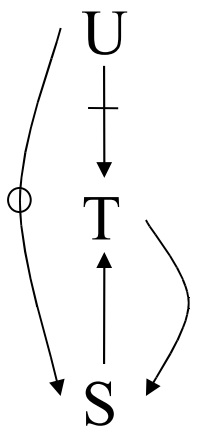
Dependência

- Dependência de dados
- Três tipos:
  - Dependência de fluxo
    - Saída de um é entrada de outro
  - Antidependência
    - Um escreve em uma posição previamente lida por outro
  - Dependência por saída
    - Mesma variável de saída

Dependência

- Grafo de dependência
  - Dependência de fluxo
  - Antidependência
  - Dependência por saída

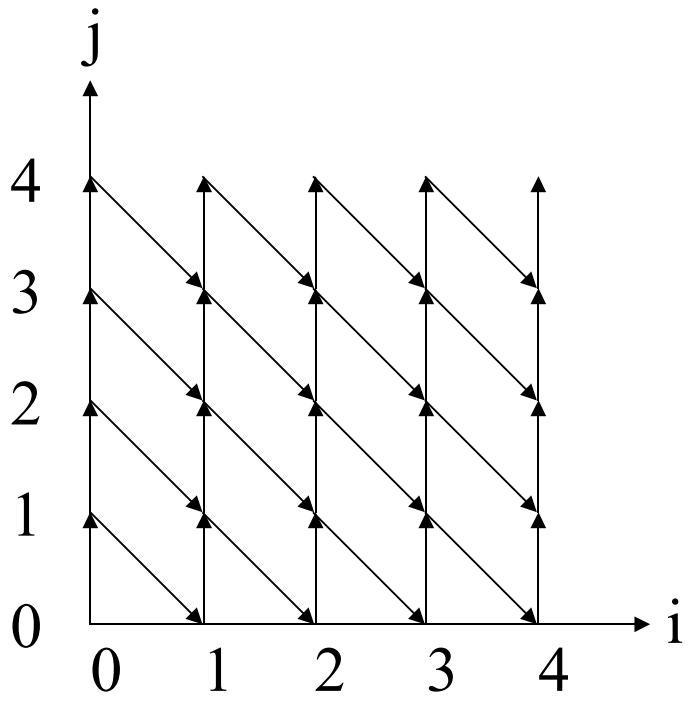
```
Para i=2..200
    S: A[i] = B[i] + C[i]
    T: B[i+2] = A[i-1] + C[i-1]
    U: A[i+1] = B[2i+3] + 1
Fim Para
```



Dependência

- Grafo de dependências entre iterações

```
Para i=0..4
    Para j=0..4
        S: A[i+1][j] = B[i][j]+C[i][j]
        T: B[i][j+1] = A[i][j+1]+1
    Fim Para
Fim Para
```



Dependência

- Dependência de controle
  - Grafo de fluxo de controle com um *sink*
  - Um nó Y pós-domina um nó X se todos os caminhos de X para o *sink* incluem Y
  - Um nó T é dependente de controle de S se:
    - Há um caminho de S para T cujos nós internos são pós-dominados por T
    - T não pós-domina S

Dependência

- Dependência de controle para dependência de dados

```
Se A != 0
    C = C+1
    D = C/A
Senão
    D = C
Fim Se
X = C + D
```

```
b = [A != 0]
C = C+1 quando b
D = C/A quando b
D = C quando não b
X = C+D
```

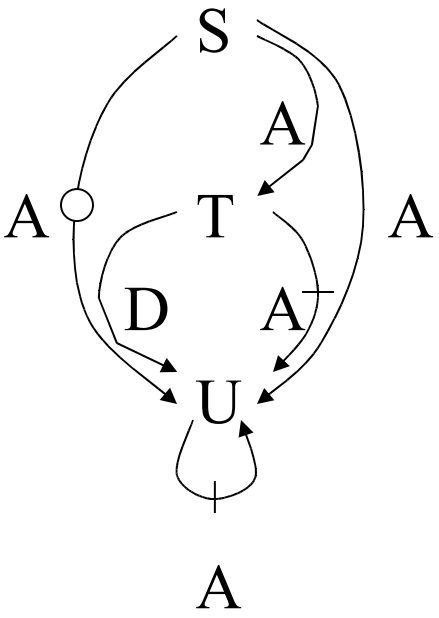
Eliminar Dependências

- *Renaming*
  - Diferentes nomes para usos diferentes de uma mesma variável

S: A = B + C

T: D = A + E

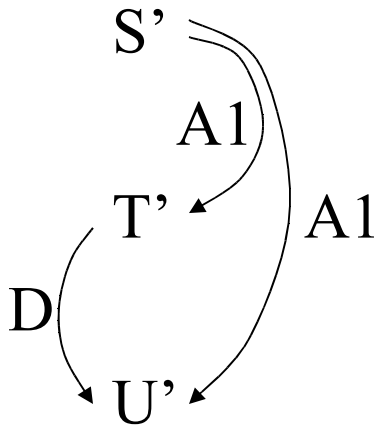
U: A = A + D



S': A1 = B + C

T': D = A1 + E

U': A = A1 + D



Eliminar Dependências

- *Forward Substitution*
  - Copiar o lado direito de uma atribuição no lado direito de outra atribuição

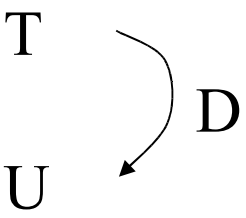
T: D = B + C + E

U: A = B + C + D

→

T': D = B + C + E

U': A = (B + C) + (B + C + E)



T'

U'

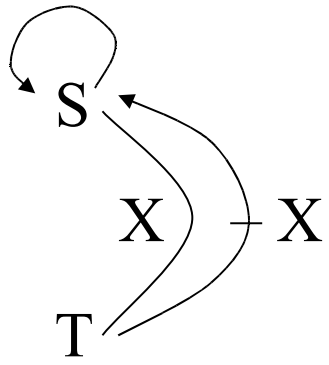
Eliminar Dependências

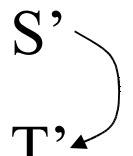
- *Scalar Expansion*
  - Transformar uma variável em um *array*

```
Para i=1..N faça
    S: X = C[i]
    T: D[i] = X + 1
Fim para
```

→

```
Para i=1..N faça
    S: X[i] = C[i]
    T: D[i] = X[i]  + 1
Fim para
```





Detectar Dependências

Para I=2..200

S: X[3I-5] = B[I]+1

T: C[I] = X[2I+6]+D[I-1]

Fim Para

Mesmas posições de memória

3i-5 = 2i+6

3i-2j = 11

m = maior divisor comum entre 3 e 2 = 1

(i,j) = ((2/1)t+i1 , ((3/1)t+j1)

(i1,j1) = ((6-5)i0/m , (6-5)j0/m)

3i0 – 2j0 = m

(i,j) = ((2/1)t+11 , ((3/1)t+11)

2<=2t+11<=200 -> -4<=t<=94

2<=3t+11<=200 -> -3<=t<=63

Detectar Dependências

Intersecção

$-3 <= t <= 63$

Ponto de intersecção entre

$i(t) = 2t + 11$ e $j(t) = 3t + 11$

é em $t=0$

$1 <= t <= 63 \rightarrow i(t) < j(t)$

$t = -3, -2, -1 \rightarrow i(t) > j(t)$

T dependente de S

$\{(S(2t+11), T(3t+11)): 0 <= t <= 63\}$

$=\{(S(11),T(11))...(S(137),T(200))\}$

S antidependente de T

$\{(T(3t+11), S(2t+11)): -3 <= t <= -1\}$

$=\{(T(2),S(5)),(T(5),S(7)),$
$(T(9),S(9))\}$

Transformação Serial-Paralela

- Técnicas de transformação utilizadas para evidenciar o paralelismo intrínseco

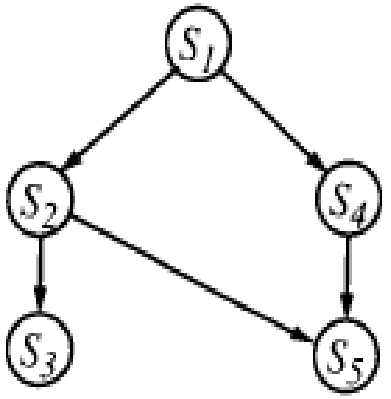
- São dividas em duas classes:
  - Paralelização de código acíclico
  - Paralelização de laços DO

Códigos acíclicos

- Não existe ciclos no grafo de fluxo do programa
- Divide as instruções em blocos, conforme a granulação desejada
- Pode-se empregar as construções COBEGIN/COEND e sincronização POST/WAIT

Códigos acíclicos - Exemplo

$$
\begin{aligned}
S_1 &: \quad A = 1 \\
S_2 &: \quad B = A + 1 \\
S_3 &: \quad C = B + 1 \\
S_4 &: \quad D = A + 1 \\
S_5 &: \quad E = D + B
\end{aligned}
$$



Códigos acíclicos - Exemplo

S_1 : $A = 1$

cobegin

S_2 : $B = A + 1$

post (e)

S_3 : $C = B + 1$

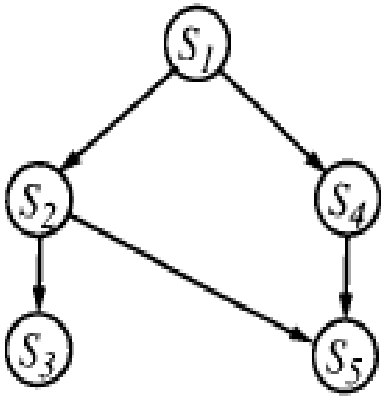
$\|$

S_4 : $D = A + 1$

wait (e)

S_5 : $E = D + B$

coend



Métodos de definição de ordem de execução (scheduling)

- Paralelização pela granulação
- Compactação de código
- Trace scheduling
- Filtragem de scheduling

Paralelização pela granulação

- Varia conforme a arquitetura empregada
- Pode-se empregar uma granulação mais grossa
- Posterior balanceamento de carga

Compactação de código

- Empregando o conceito de macronós (estruturas delimitadas por COBEGIN/COEND)
- Independência
- Cada componente forma uma árvore de instruções condicionais (if) e instruções de salto (goto)
- Procura evitar a atribuição de uma variável duas ou mais vezes no mesmo macronó, assim como dependência entre eles
- Computadores superescalares e VLIW

Trace scheduling

- Busca bastante refinada:
  - Divisão em blocos seqüências
  - Identificação de dependências
  - Agrupamento em blocos
  - Posterior execução em paralelo
- Problema: saltos condicionais
  - avalia a probabilidade de execução dos saltos
  - obtém-se um longo caminho de instruções

Trace scheduling

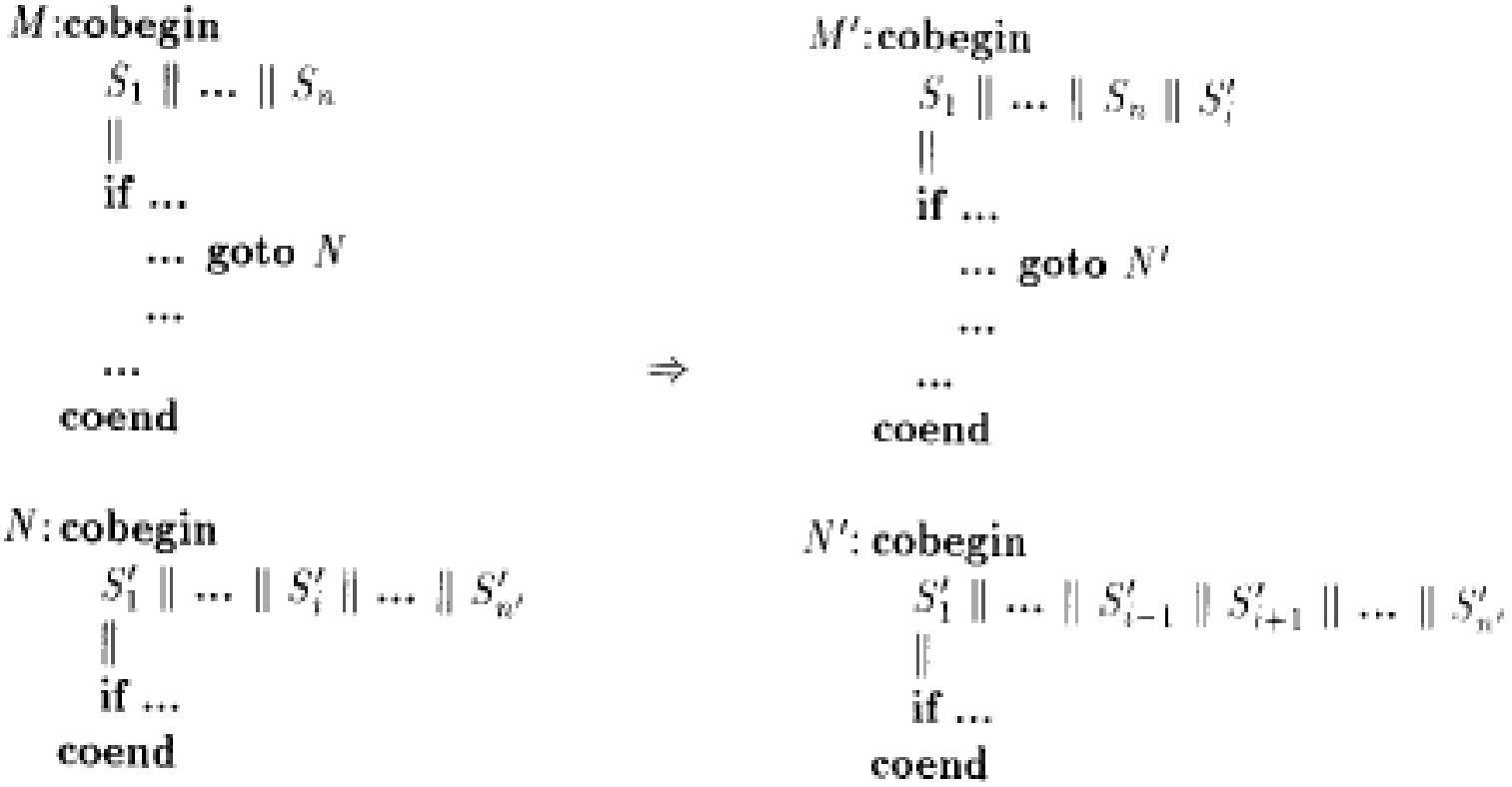
- Se houver falha na avaliação?
  - condições de contorno -> utilização do código original

- Alternativa: reorganização de pequenos blocos, para aumentar o número de instruções independentes

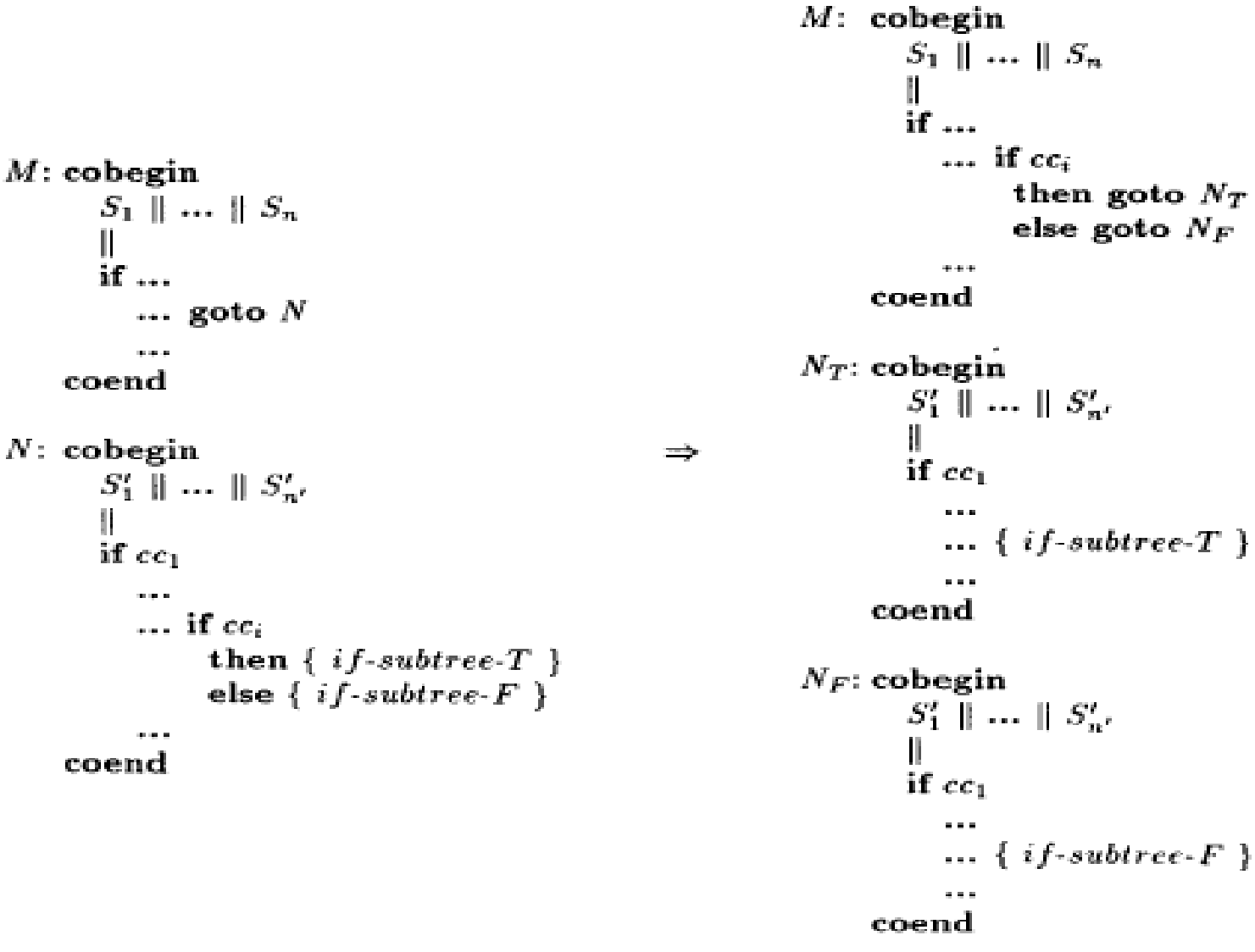
Filtragem de scheduling

- Três transformações elementares, envolvendo macronós próximos
  - (moveop) desloca uma atribuição de um macronó para o macronó anterior, no grafo de fluxo de controle
  - (movecj) desloca uma subárvore de uma árvore condicional de um macronó para o macronó anterior
  - (unify) efetua a movimentação para um macronó pai de seus sucessores

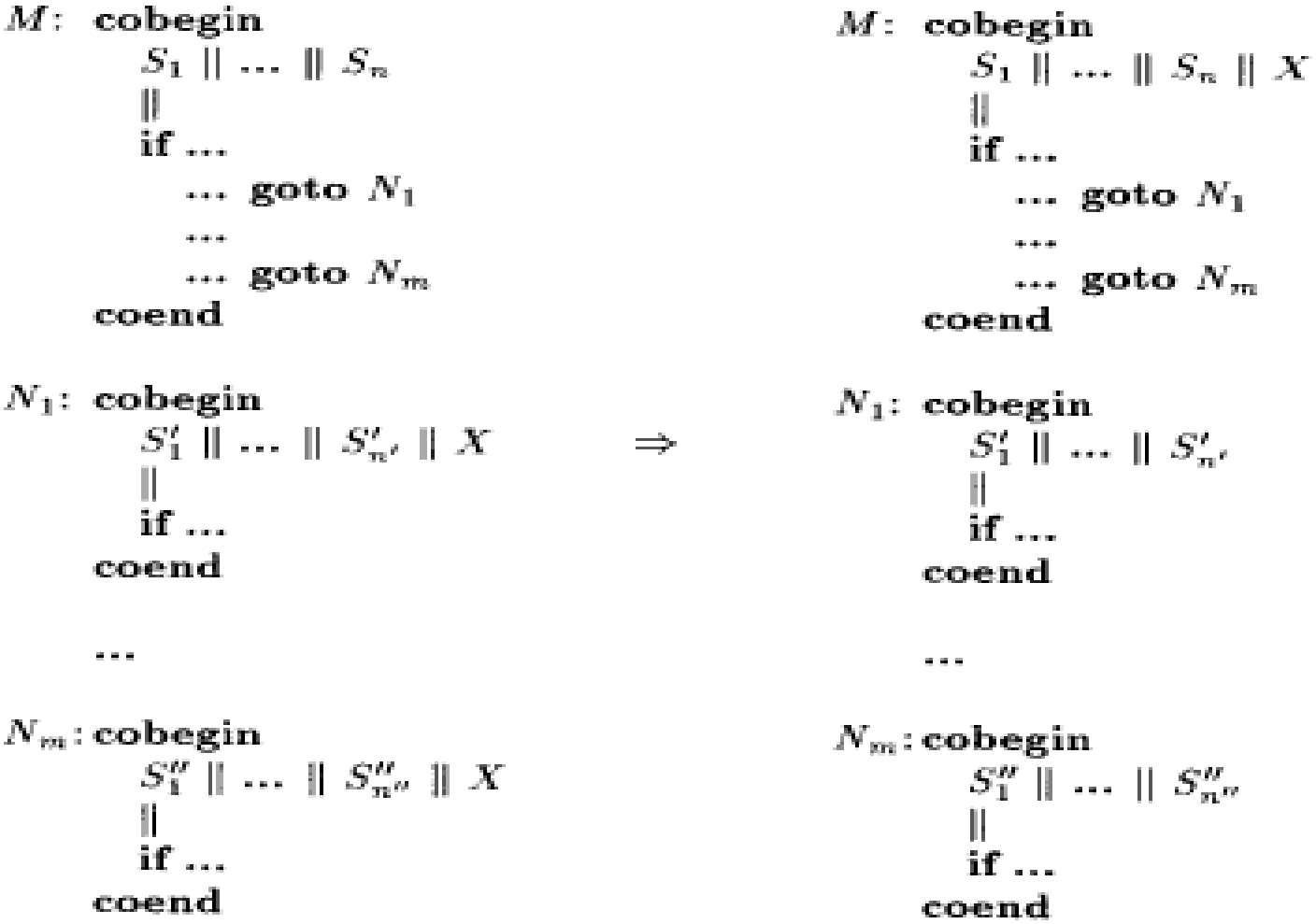
moveop - Exemplo



movecj - Exemplo



unify - Exemplo



Tipos de Laços

- Técnicas baseadas em laços DO
- Dois grupos distintos: código heterogêneo e código homogêneo

Código Heterogêneo

- Paralelização do código pela dependência
- Utilização da técnica de skewing para redução de dependência
- Garantir a independência das instruções

Skewing - Exemplo

do $I = 0, N$

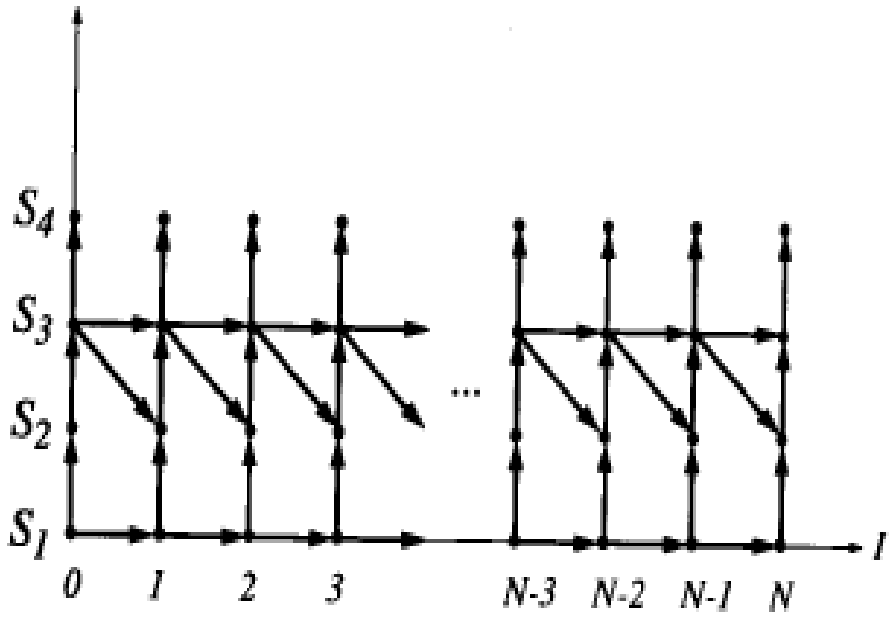
S_1: $A(I) = F_1(A(I-1))$

S_2: $B(I) = F_2(A(I), C(I-1))$

S_3: $C(I) = F_3(C(I-1), B(I))$

S_4: $D(I) = F_4(D(I), C(I))$

enddo



Skewing - Exemplo

do $K = 2, N$

cobegin

S_1: $A(K) = F_1(A(K-1))$

$\|$

S_2: $B(K-1) = F_2(A(K-1), C(K-2))$

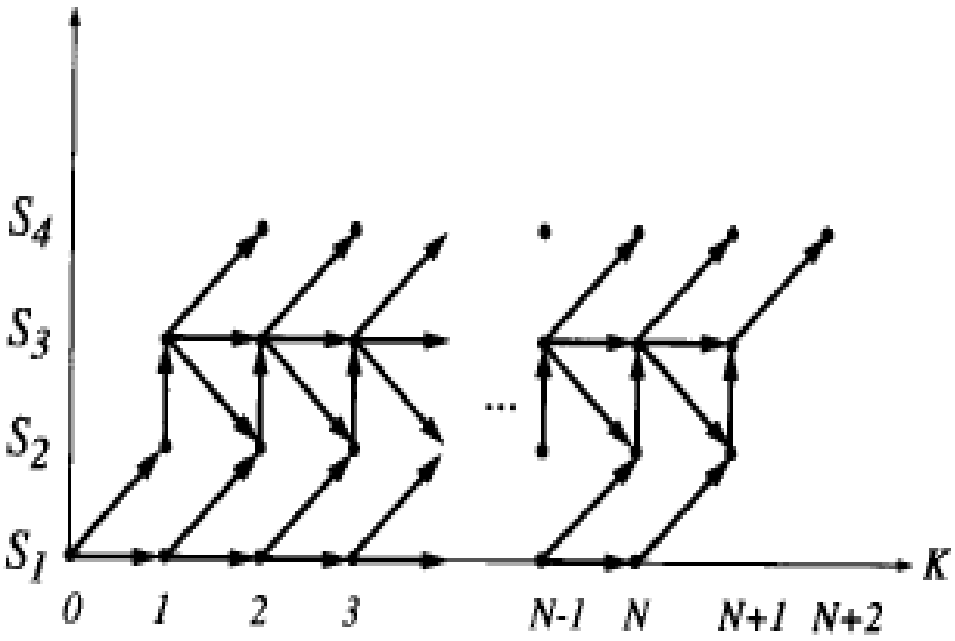
S_3: $C(K-1) = F_3(C(K-2), B(K-1))$

$\|$

S_4: $D(K-2) = F_4(D(K-2), C(K-2))$

coend

enddo



Código Heterogêneo

- Desdobramento parcial (partial enroling) do laço
  - Leva em consideração a distância de dependência entre os comandos
  - Pode ser utilizada uma substituição a frente (forward substitution)
    - v = *expressão*

Partial Enroling-Exemplo

do $I = 0, N$

S_1: $A(I) = F_1(A(I-3), C(I))$

S_2: $D(I) = F_2(A(I), D(I-2))$

enddo

do $I = 0, N, 2$

cobegin

S_1: $A(I) = F_1(A(I-3), C(I))$

S_2: $D(I) = F_2(A(I), D(I-2))$

$\|$

S_1': $A(I+1) = F_1(A(I-2), C(I+1))$

S_2': $D(I+1) = F_2(A(I+1), D(I-1))$

coend

enddo

Código Heterogêneo

- Pipeline de software
- Visa transformar os laços DO em código paralelo de fina granulação
- Macronós sem dependência de dados/controle são executados primeiro
- Macronós que tenham dependência direta em seguida e assim sucessivamente
- Problemas com ramificações

Código Homogêneo

- Construí-se blocos seqüenciais homogêneos -> pode ou não existir dependências de interação cruzada
- Caso exista, a dependência deve ser tratada
- A primeira técnica envolve a separação do código que não haja dependência cruzada seguido pelas demais instruções, por ordem de dependência

Código Homogêneo

- Outro método utiliza cópia, para evitar toda dependência cruzada de dados
- Modificações nos índices de acesso a memória (wait and post)

Código Homogêneo

- Outra técnica seria particionar os laços paralelos homogêneos
- Particionamento na qual um laço é dividido em múltiplos outros
- MDC(distância) = Max(divisões)
- Avaliação da ordem é realizada através da análise do grafo de fluxo

Particionamento - Exemplo

do $I = 0, N$

S_1: $A(I) = F_1(A(I-4), C(I))$

S_2: $D(I) = F_2(A(I), D(I-2))$

enddo

cobegin

do $I = 0, N, 2$

S_1: $A(I) = F_1(A(I-4), C(I))$

S_2: $D(I) = F_2(A(I), D(I-2))$

enddo

||

do $I = 1, N, 2$

S_1: $A(I) = F_1(A(I-4), C(I))$

S_2: $D(I) = F_2(A(I), D(I-2))$

enddo

coend

Localidade e Overhead de processadores

- Uma solução para minimizar o overhead é a utilização da fusão de laços
- Pode ser usado a técnica do *tiling*
  - permite a redução do sincronismo através de instruções de bloqueios condicionais
  - melhoramento da localidade, na tentativa de manter independência entre os macronós e aumentar o pararelismo

Tiling - Exemplo

```
do I1 = 0, N
   do I2 = 0, N
S1:      B(I2, I1) = F1(A(I1, I2))
   enddo
enddo
```

```
do J1 = 0, N, IB
   do J2 = 0, N, IB
      do I1 = J1, min(J1 + IB - 1, N)
         do I2 = J2, min(J2 + IB - 1, N)
S1:         B(I2, I1) = F1(A(I1, I2))
         enddo
      enddo
   enddo
enddo
```

Avaliação Dinâmicas

- Como escolher o caminho correto que um algoritmo serial seguirá?
  - Alternativa seria que o compilador inserisse alguns testes em pontos dúbios do código

Avaliação Dinâmicas

- Problemas com ponteiros e construções recursivas.
  - Avaliações automáticas podem ser feitas sobre ponteiros, porém seriam necessárias técnicas de identificações de algoritmos
- Funções recursivas
  - expandidas a partir do seu valor de entrada inicial
  - reduzidas pela aplicação de uma função (passíveis de paralelização)

Comparação de Performance

Estudo de caso – Polaris

- Teste realizado numa máquina Cedar
- Cedar – 4 clusters de 8 processadores cada
- Parte da memória é compartilhada entre todos os processadores e parte é compartilhada apenas entre os processadores de um mesmo cluster

Estudo de caso – Polaris

- Testes realizados com um “benchmark perfeito” – conjunto de programas representando aplicações comuns
- Os testes indicaram que houve um ganho significativo no speedup de pequenos benchmarks, mas o ganho para programas reais foi pequeno

Estudo de caso – Polaris

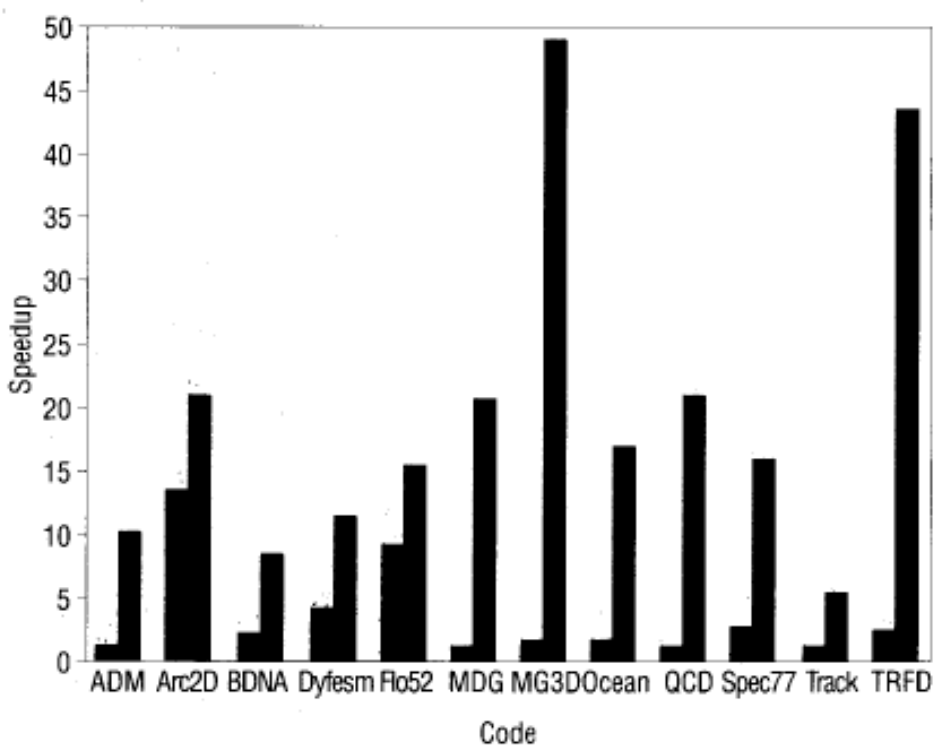


Figure 1. Speedups of automatically and manually parallelized versions of the Perfect Benchmarks on the Cedar machine.

Estudo de caso – Polaris

- Inicialmente foi feita a paralelização manual destes programas, e, com poucas exceções, apenas as transformações que um compilador paralelizador implementam foram feitas
- Além destas, apenas as baseadas no programa fonte foram implementadas, ficaram de fora as transformações que dependiam do conhecimento da aplicação

Estudo de caso – Polaris

- Essas técnicas de transformação foram implementadas num compilador chamado Polaris, que numa versão preliminar conseguiu paralelizar metade dos programas mostrados na figura 1 tão bem quanto a paralelização manual
- Examinando com um pouco mais de cautela, concluiu-se que com algumas técnicas a mais seria possível paralelizar mais 2 dos 6 programas.

Conclusões

- As técnicas de paralelização automáticas evoluíram nos últimos anos
- Porém, ainda não alcançaram as técnicas de paralelização manual
- Além disso a paralelização automática não consegue tratar muito bem transformações que dependem do conhecimento da aplicação

Bibliografia

- **Automatic Program Parallelization**
  - UTPAL BANERJEE, RUDOLF EIGENMANN, ALEXANDRU NICOLAU, DAVID A. PADUA
- **Automatic Detection of Parallelism**
  - William blume, Rudolf Eigenmann, Jay Hoeflinger, David Padua, Paul Petersen, Lawrence Rauchwerger, and Peng Tu
- **Algoritms and Applications on Vector and Parallel Computers**
  - H.J.J. t Riele, Th.J.Dekker,H.A. van der Vorst
- **Are Parallel Workstations the Right Target for Paralleling Compilers?**
  - Rudolf Eigenmmann, Insung Park, Micheal J. Voss