Detecção de Paralelismo em Laços de Arquiteturas MIMD, SIMD e Multicore

Guido Araujo

ERAD 2011

São José dos Campos, SP

LSC COMPUTER SYSTEMS LABORATORY

IC UNICAMP

Roteiro

- Arquiteturas Paralelas
- Paralelismo em MIMD
- Paralelismo em Multicores
- Paralelismo em SIMD

MIMD (VLIW)

- Multiple Instruction Multiple Data

```
for (i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
    E[i] = D[i] << 2;
    F[i] = C[i] + E[i];
    G[i] = F[i] – 1;
}
```

C[i] = A[i] + B[i]	**E[i] = D[i] << 2**	**……**	**……**
F[i] = C[i] + E[i]	**……**	**……**	**……**
G[i] = F[i] – 1	**……**	**……**	**……**

MIMT (Multicore)

- Multiple Instruction Multiple Threads

```
for (i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
    E[i] = D[i] << 2;
    F[i] = C[i] + E[i];
    G[i] = F[i] – 1;
}
```

SIMD (Vector)

- Single Instruction Multiple Data

```
for (i = 0; i < N; i++)
    C[i] = A[i] + B[i];
```

A[ ]	A[0]	A[1]	A[2]	.....	.....	.....	A[N-2]	A[N-1]
					+			
B[ ]	B[0]	B[1]	B[2]	.....	.....	.....	B[N-2]	B[N-1]
					=			
C[ ]	C[0]	C[1]	C[2]	.....	.....	.....	C[N-2]	C[N-1]

Conceitos sobre Laços

- Variável de indução (*induction variable*)
  - Variável de contagem do laço
- Duração da viagem (*trip-count*)
  - Quantas iterações dura o laço
- Laço contável (*countable loop*)
  - Número de iterações determinada em tempo de compilação
- Laço não-contável (*uncountable loop*)
  - Número de iterações não determinada em tempo de compilação

Conceitos sobre Laços

Variável indução: i

```
for (i = 0; i < N; i++) {
        C[i] = A[i] + B[i];
        E[i] = D[i] << 2;
        F[i] = C[i] + E[i];
        G[i] = F[i] – 1;
    }
```

Countable loop

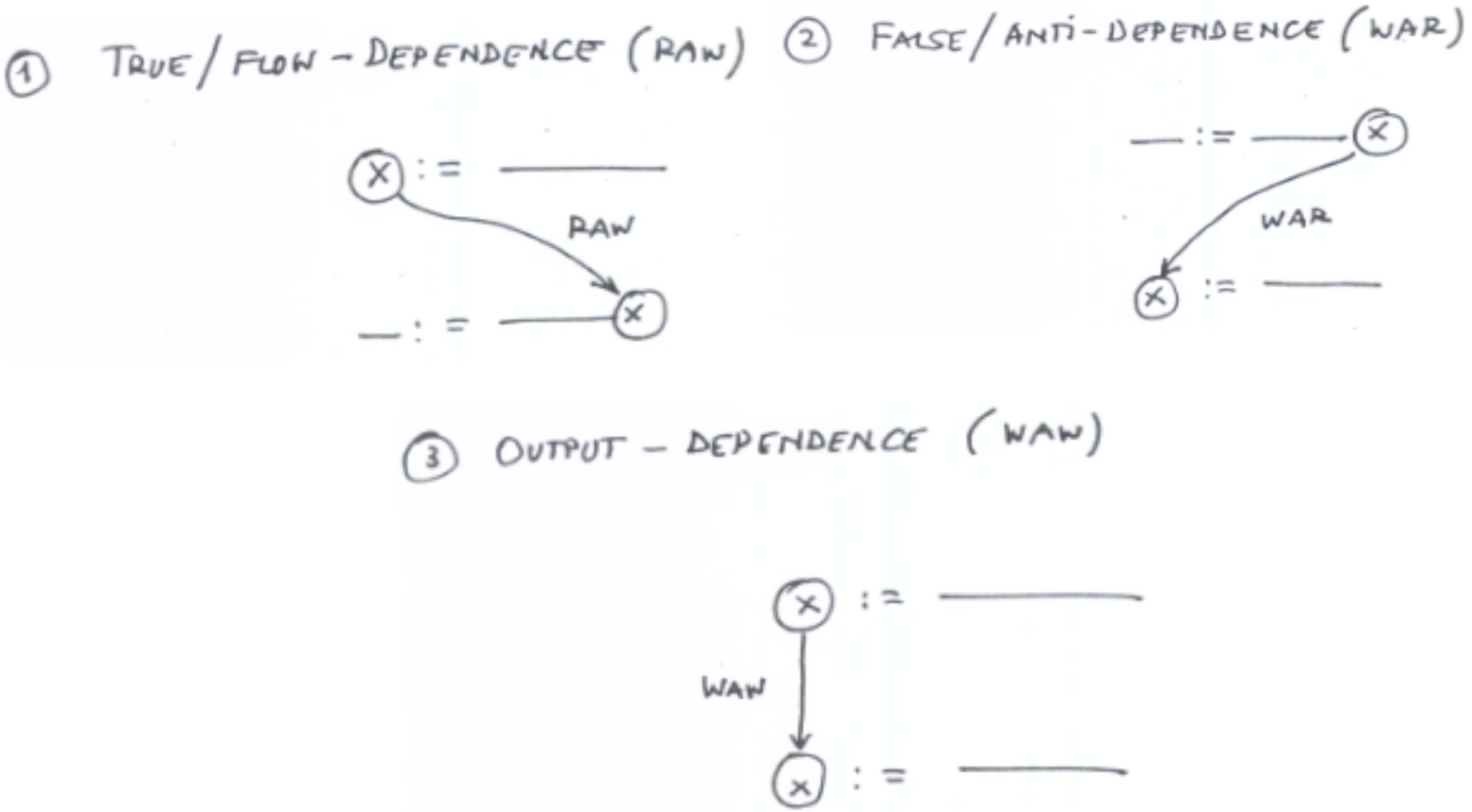
Trip-count: N

Valor de var é desconhecido

```
for (i = 0; i < var; i++) {
        C[i] = A[i] + B[i];
        E[i] = D[i] << 2;
        F[i] = C[i] + E[i];
        G[i] = F[i] – 1;
    }
```

Uncountable loop

Entendendo Dependências



Entendendo Dependências

```
for (i = 0; i < N; i++) {
        (1) C[i] = A[i] + B[i];
        (2) E[i] = D[i] << 2;
        (3) F[i] = C[i] + E[i];
        (4) G[i] = F[i] – 1;
      }
```

Reestruturando Laços

- Escalonar instruções dentro do laço de modo a otimizar o desempenho
  - Pode ser usado para esconder latência
  - Pode ser usado para extrair paralelismo

Esocndendo Latência

- Custo do acesso à memória é 2 ciclos
- Laço executa 1000 vezes

```
for (……) {                          ciclos
 (1) sub r1, r1, 1;                    1
 (2) store r5, [r3 + 256]              2
 (3) add r3, r3, 1                     1
}                          Total: 4 * 1000 = 4000
```

Reestruturando Laços (Escalonamento)

- Alterando a estrutura do laço:
  - Esconde a latência da memória
  - Melhora desempenho 4000/3000 ~ 30%

```
for (……) {                          ciclos
 (1) store r5, [r3 + 256]             2
 (2) sub r1, r1, 1;                   1
 (3) add r3, r3, 1                    1
}                              Total: 3 * 1000 = 3000
```

Extraindo paralelismo

- Separando o laços em dois cores

```
for (i = 0; i < N; i++) {
  C[i] = A[i] + B[i];
  E[i] = D[i] << 2;
  F[i] = C[i] + E[i];
  G[i] = F[i] – 1;
}
```

```
for (i = 0; i < N; i++) {
   C[i] = A[i] + B[i];
   F[i] = C[i] + E[i];
   G[i] = F[i] – 1;
}
```

```
for (i = 0; i < N; i++) {
   E[i] = D[i] << 2;
}
```

Roteiro

- Arquiteturas Paralelas
- Paralelismo em MIMD
- Paralelismo em Multicores
- Paralelismo em SIMD

MIMD (VLIW)

```
for (i = 1; i <= N; i++) {
        (1) B[i] = A[i] +1;
        (2) C[i] = B[i] << 2;
        (3) D[i] = C[i] -1;
    }
```

B[i] = A[i] + 1	.....	.....	.....
C[i] = B[i] << 2	.....	.....	.....
D[i] = C[i] - 1	.....	.....	.....

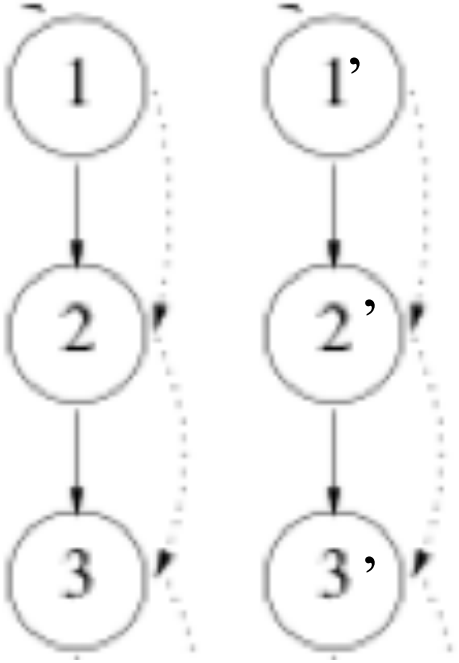
Como aumentar paralelismo?

- Desenrolando o laço….

```
for (i = 1; i <= N; i += 2) {
    (1) B[i] = A[i] +1;
    (2) C[i] = B[i] << 2;
    (3) D[i] = C[i] -1;
    (1') B[i+1] = A[i+1] +1;
    (2') C[i+1] = B[i+1] << 2;
    (3') D[i+1] = C[i+1] -1;
}
```

i = 1 { (1), (2), (3)

i = 2 { (1'), (2'), (3')



E o desempenho?

```
for (i = 1; i <= N; i += 2) {
    (1) B[i] = A[i] +1;
    (2) C[i] = B[i] << 2;
    (3) D[i] = C[i] -1;
    (1') B[i+1] = A[i+1] +1;
    (2') C[i+1] = B[i+1] << 2;
    (3') D[i+1] = C[i+1] -1;
}
```

$i = 1$

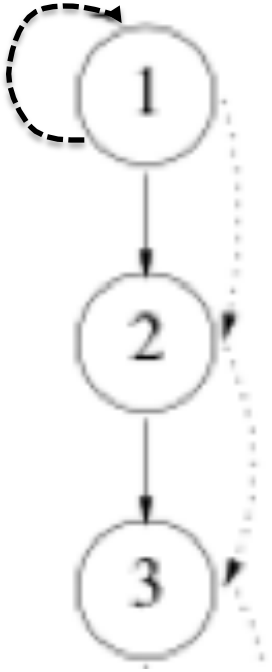
$i = 2$

Speedup:

B[i] = A[i] + 1	B[i+1] = A[i+1] + 1	.....	.....
C[i] = B[i] << 2	C[i+1] = B[i+1] << 2	.....	.....
D[i] = C[i] - 1	D[i+1] = C[i+1] - 1	.....	.....

E se existir dependência entre iterações?

```
for (i = 1; i <= N; i++) {
    (1) A[i] = A[i-1] +1;
    (2) B[i] = A[i] << 2;
    (3) C[i] = B[i] -1;
}
```

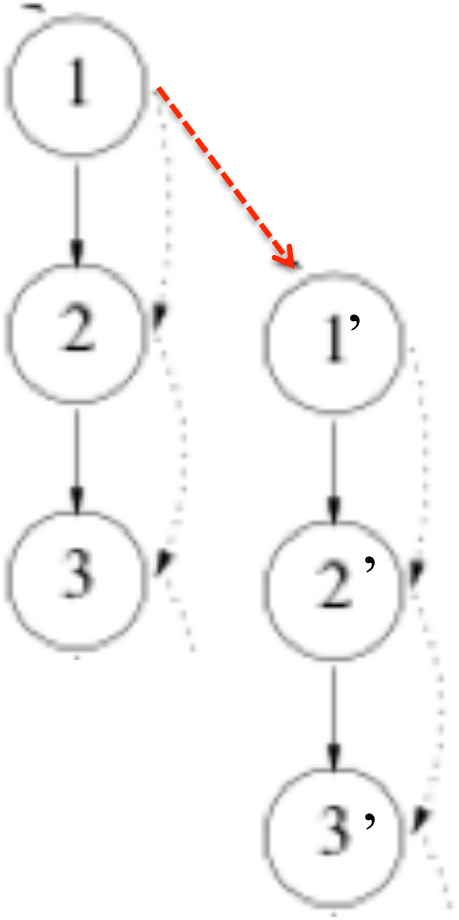


Time:

E se existir dependência entre iterações?

- Precisa garantir o escalonamento.

```
for (i = 1; i <= N; i += 2) {
    i = 1 {  (1) A[i] = A[i-1] +1;
             (2) B[i] = A[i] << 2;
             (3) C[i] = B[i] -1;
    i = 2 {  (1') A[i+1] = A[i] +1;
             (2') B[i+1] = A[i+1] << 2;
             (3') C[i+1] = B[i+1] -1;
}
```



E o desempenho?

```
for (i = 1; i <= N; i += 2) {
    (1) A[i] = A[i-1] +1;
    (2) B[i] = A[i] << 2;
    (3) C[i] = B[i] -1;
    (1') A[i+1] = A[i] +1;
    (2') B[i+1] = A[i+1] << 2;
    (3') C[i+1] = B[i+1] -1;
}
```

$i = 1$

$i = 2$

Speedup:

A[i] = A[i-1] + 1	NOP	.....	.....
B[i] = A[i] << 2	A[i+1] = A[i] + 1	.....	.....
C[i] = B[i] - 1	B[i+1] = A[i+1] << 2	.....	.....
NOP	C[i+1] = B[i+1] - 1	.....	.....

E o desempenho?

```
for (i = 1; i <= N; i += 2) {
    (1) A[i] = A[i-1] +1;
    (2) B[i] = A[i] << 2;
    (3) C[i] = B[i] -1;
    (1') A[i+1] = A[i] +1;
    (2') B[i+1] = A[i+1] << 2;
    (3') C[i+1] = B[i+1] -1;
}
```

$i = 1$ { (1), (2), (3) }

$i = 2$ { (1'), (2'), (3') }

Speedup:

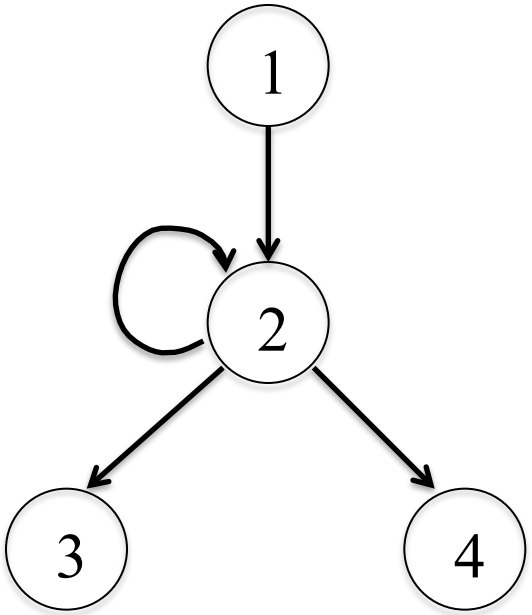
		.....	.....
		.....	.....
		.....	.....
		.....	.....

Uau!!

- Mas o que ocorre de desenrolarmos mais e mais para maximizar o paralelismo?

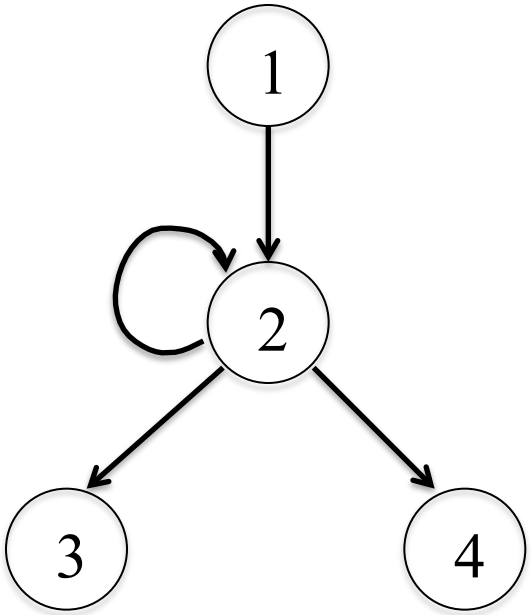
O quanto devemos desenrolar?

- Vamos ver….



O quanto devemos desenrolar?

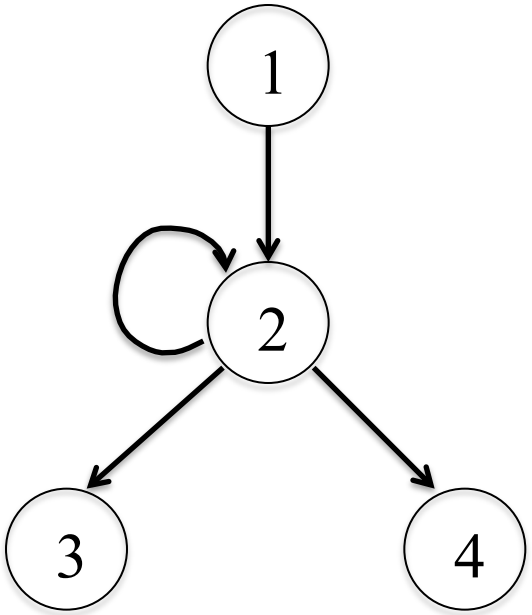
- Agora a 2a iteração



1						
2		1				
3	4					

O quanto devemos desenrolar?

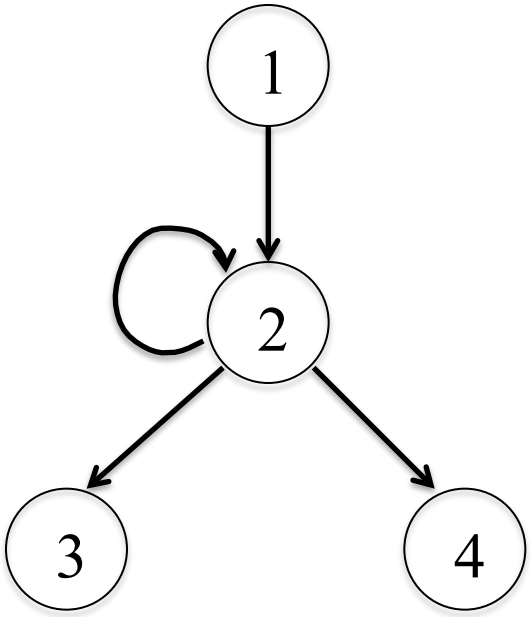
- Agora a 3a iteração



1						
2		1				
3	4	2		1		
		3	4			

O quanto devemos desenrolar?

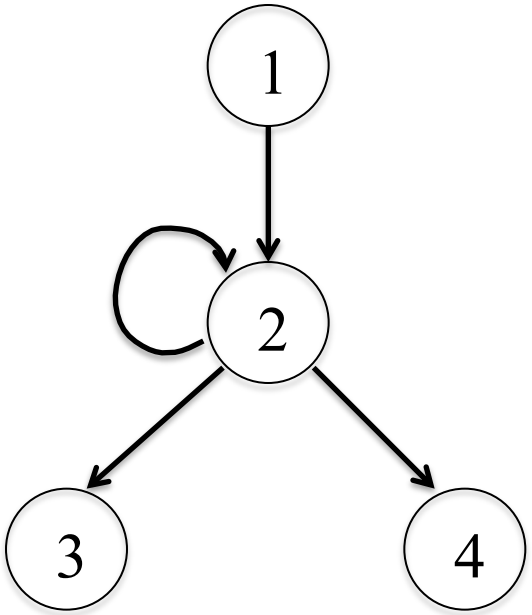
- Agora a 4a iteração



1						
2		1				
3	4	2		1		
		3	4	2		1
				3	4	

O quanto devemos desenrolar?

- Assim por diante….

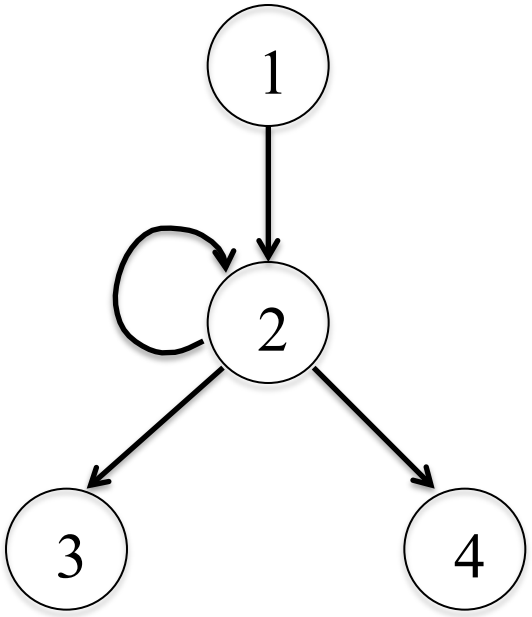


1						
2		1				
3	4	2		1		
		3	4	2		1
				3	4	2
….	….	….	….	….	….	….

Ops, parece que está aparecendo algo!!

O que será que está aparecendo?

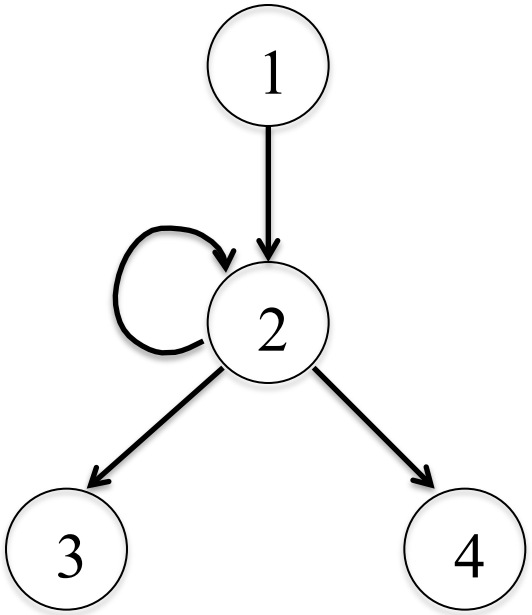
- Um padrão se repete!



1			
2	1		
3	4	2	1
3	4	2	1
3	4	2	1
....	....	....	....

E o que ocorre no final do laço?

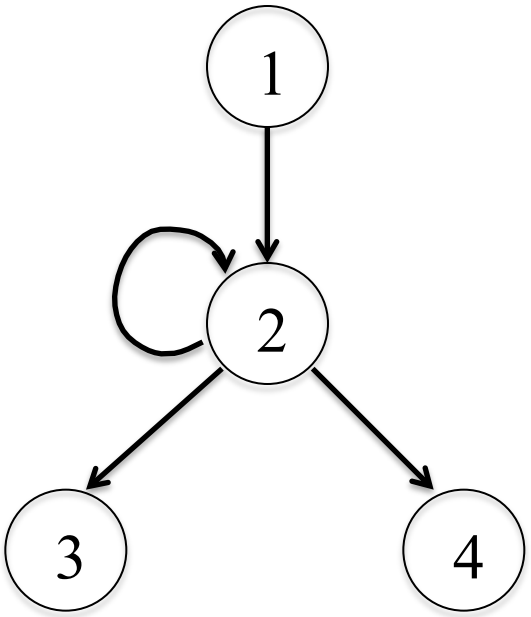
- Prólogo e epílogo



1						
2	1					
3	4	2	1			
3	4	2	1			
....	....	....	....			
3	4	2	1			
		2				
		3	4			

E como fica o desempenho final?

- Speed-up



N – 2 vezes {

1						
2	1					
3	4	2	1			
		2				
		3	4			

Speed-up: 3N/

Software Pipelining

- Esta técnica se chama software pipelining

- Será possível generalizar?
  - Qual o número de vezes que é preciso desenrolar?
  - Existe um algoritmo para isto?

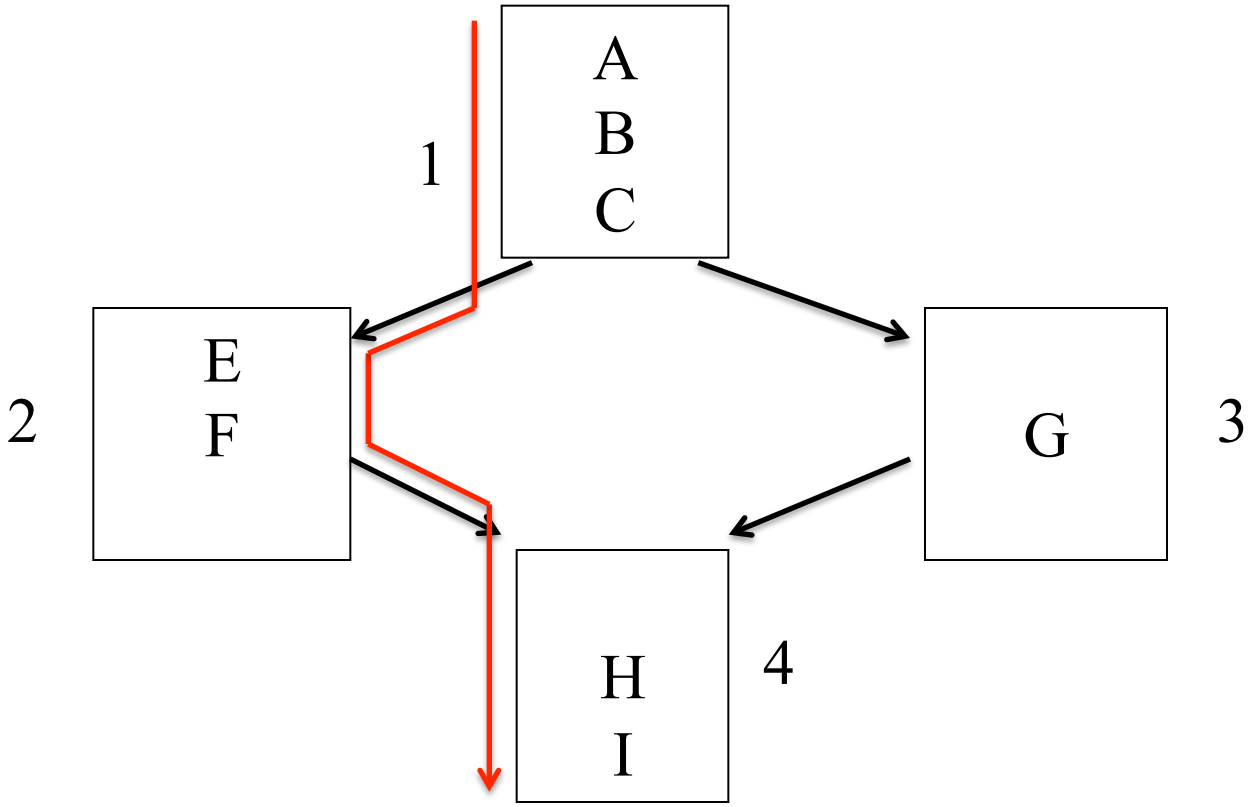
E se o laço contiver condições?

- Detectar paralelismo em tempo de execução
  - Compactar instruções nos traços que executam com mais frequência

- Considere um if no qual o then executa 90% e o else 10%

Executando no traço mais importante

```
A;
B;
C;
if (i<N) {
  E;
  F;
} else
   G;
H;
I;
```



Mesmo ciclo

A	B	C	E	F	H	I

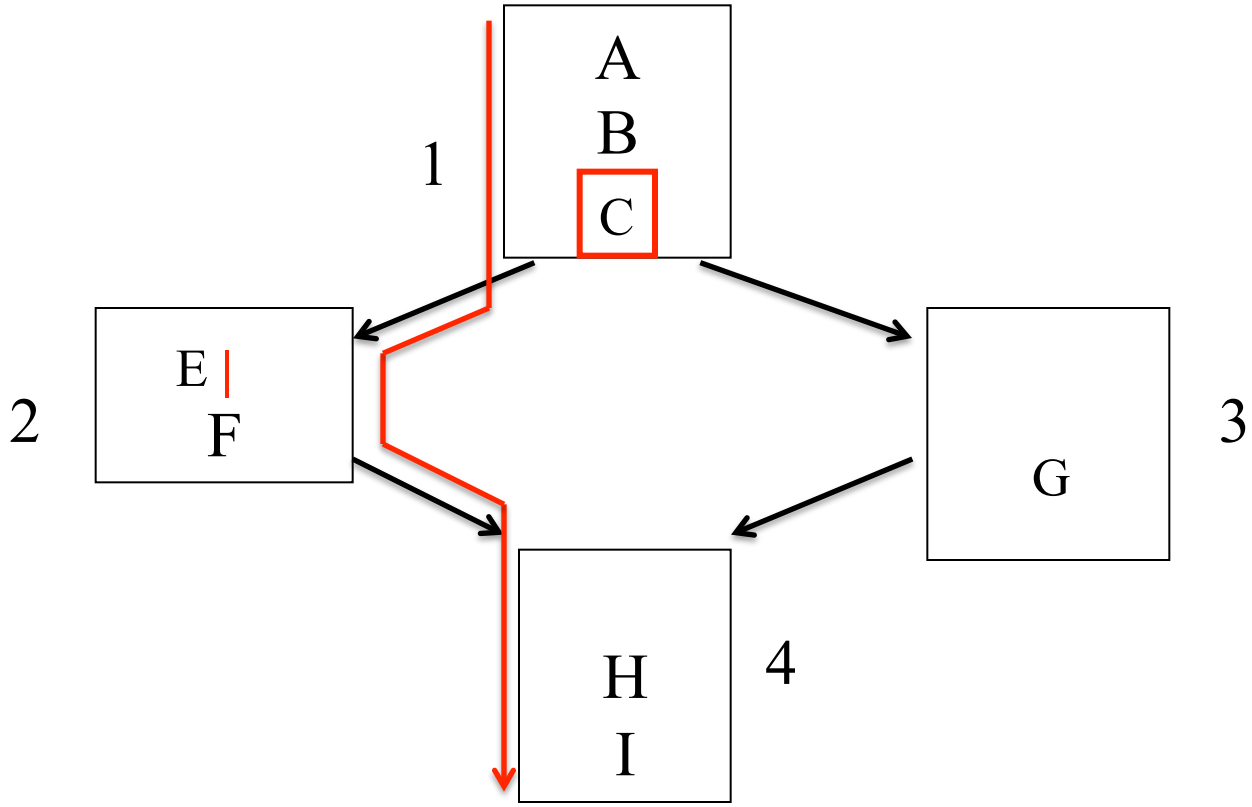
Escalonamento de Traços

- Detectar paralelismo em tempo de execução
  - Compactar instruções nos traços que executam com mais frequência

- Considere um if no qual o then executa 90% e o else 10%
  - E se sair do traço?
  - Usar código para compensar

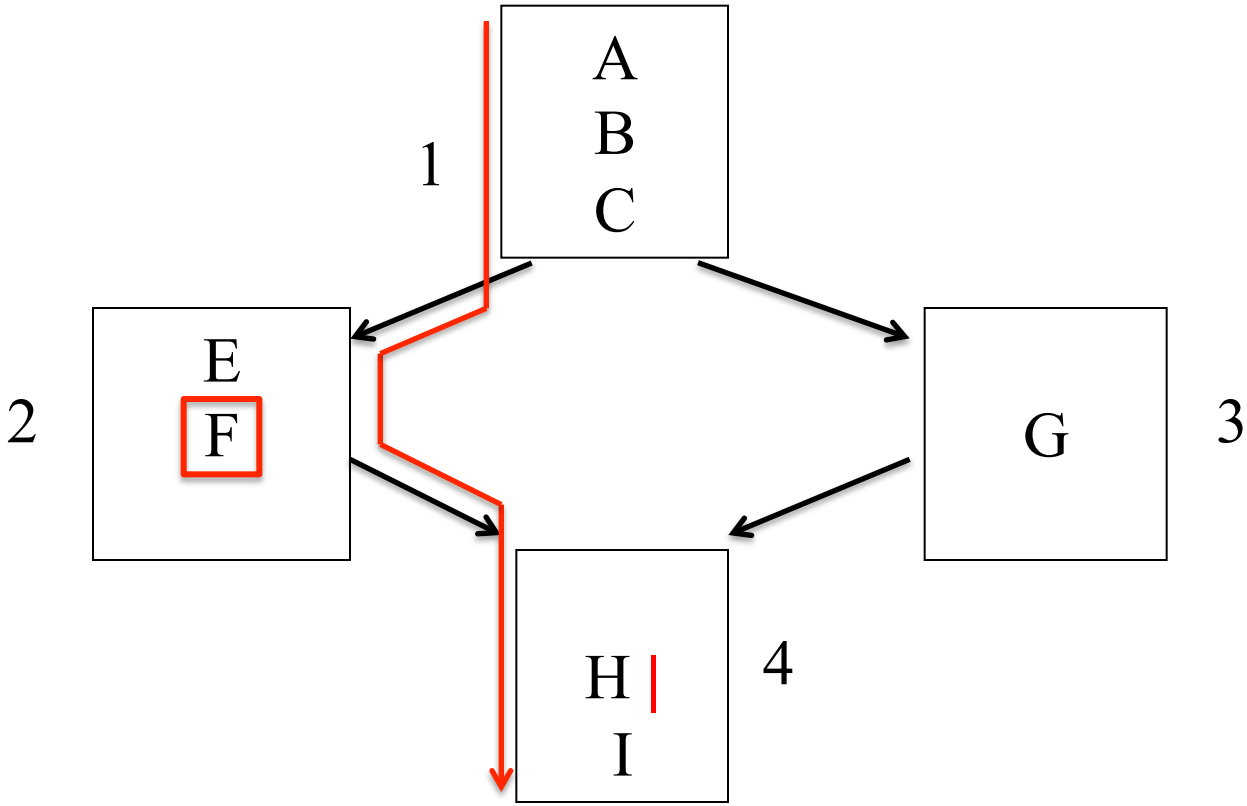
Paralelizando no traço

- Paralelizando para baixo (1)



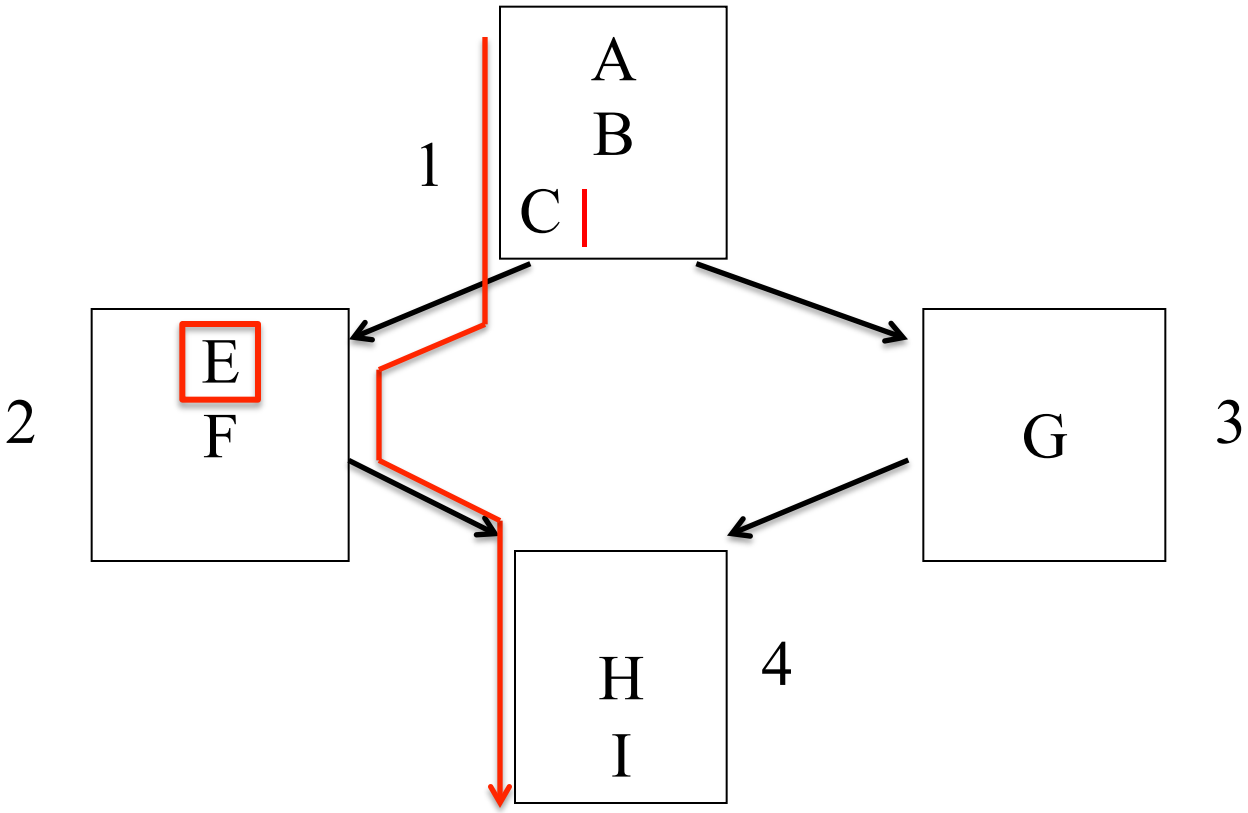
Paralelizando no traço

- Paralelizando para baixo (2)



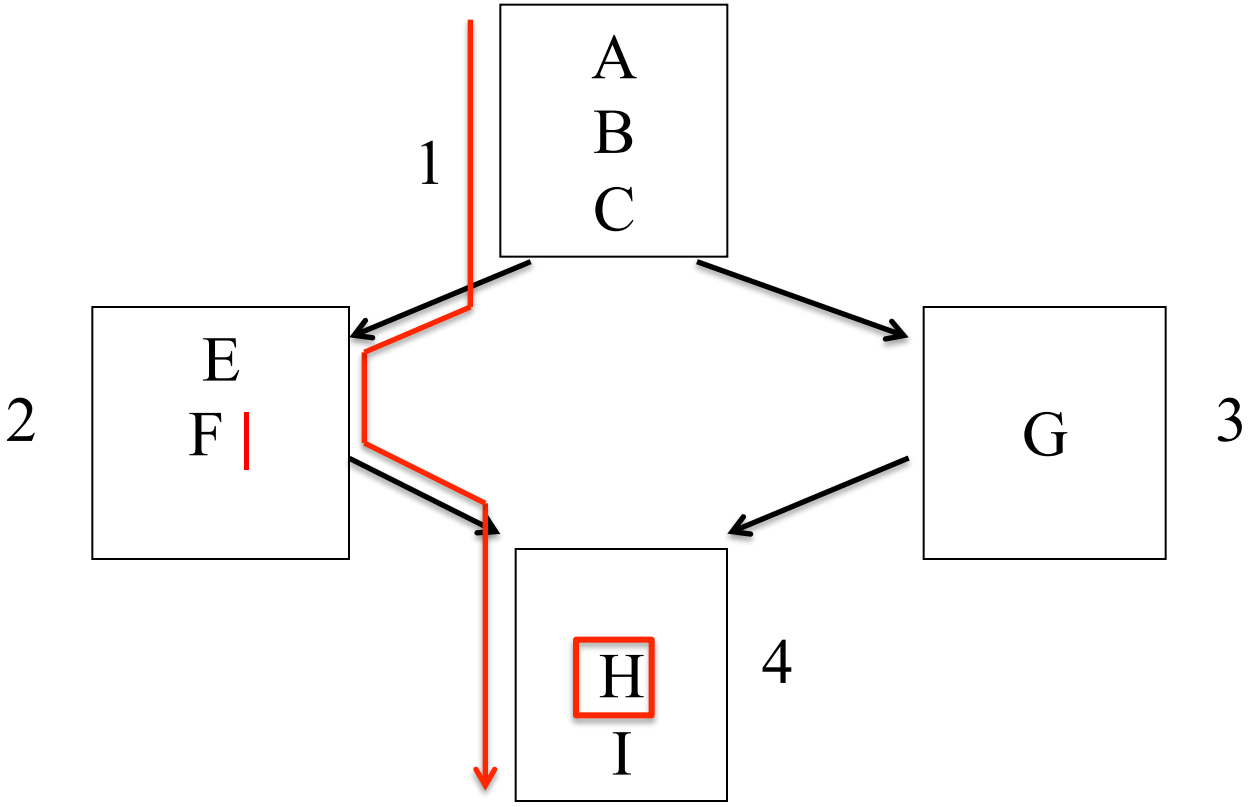
Paralelizando no traço

- Paralelizando para cima (1)

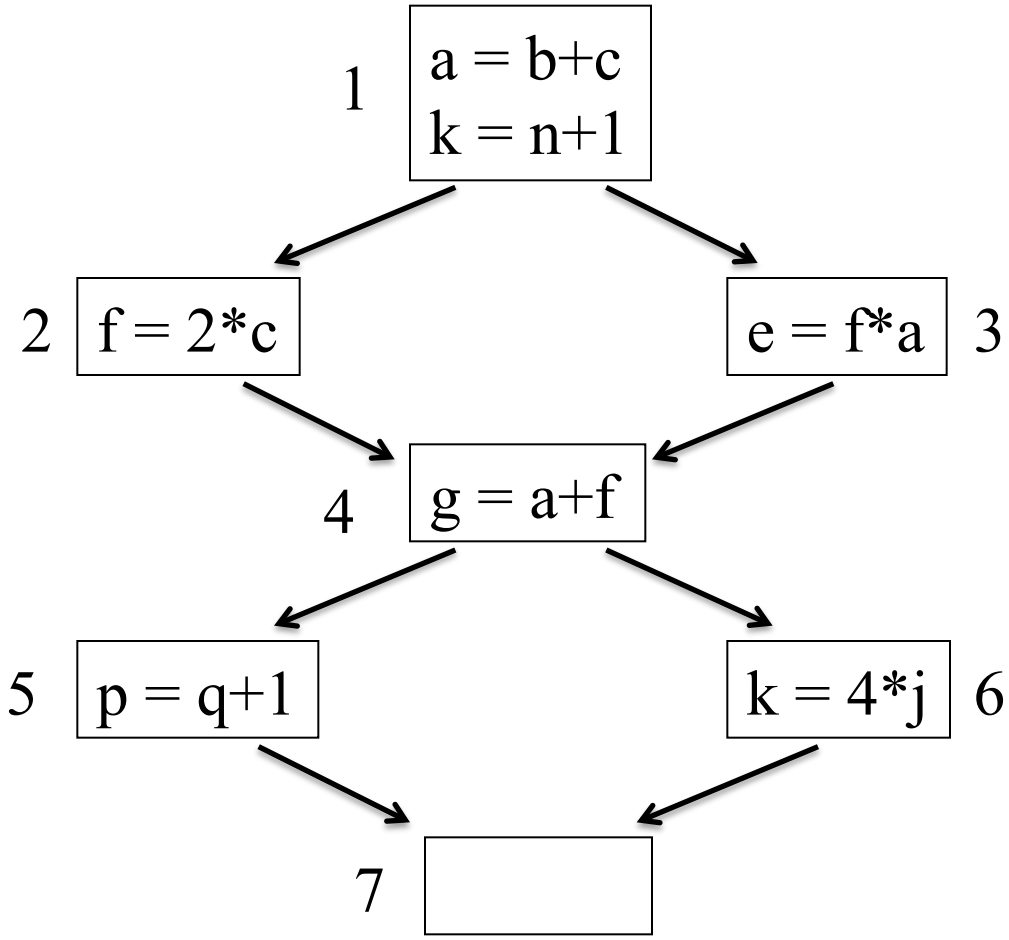


Paralelizando no traço

- Paralelizando para cima (2)

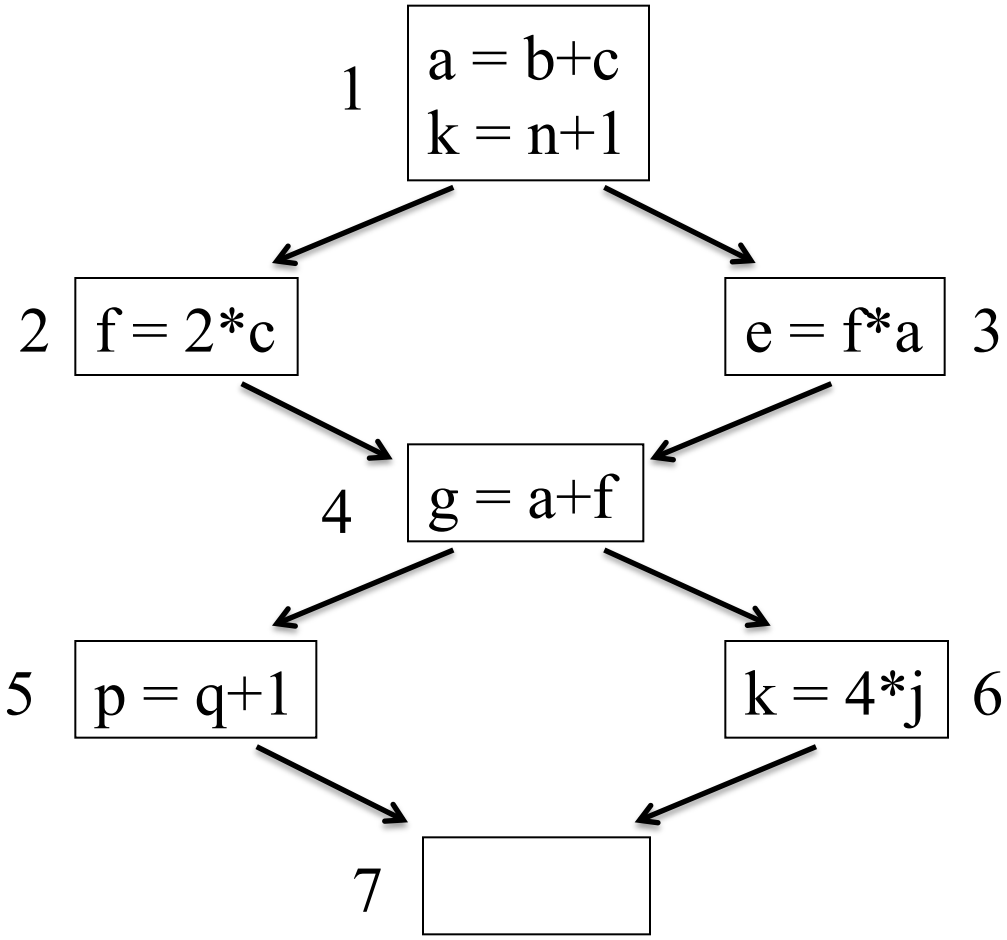


Fica mais claro com um exemplo…



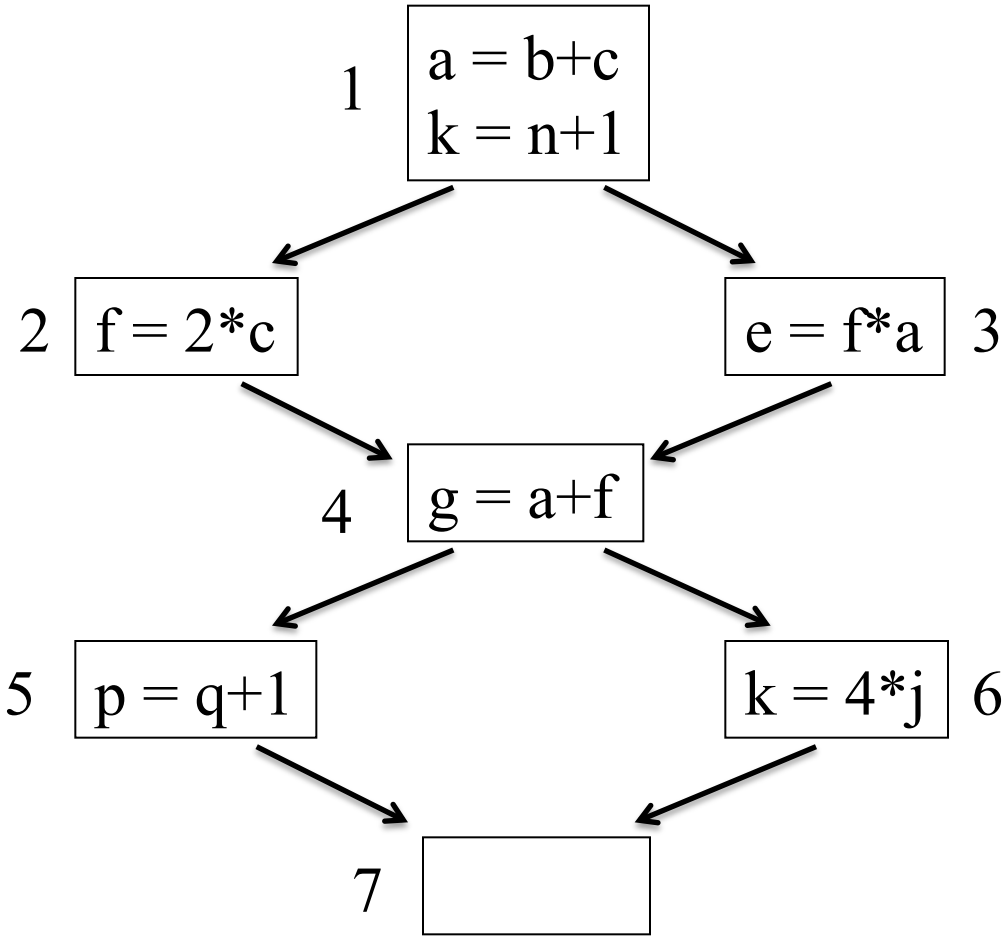
Quantos caminhos (*traços*)?

Traço A



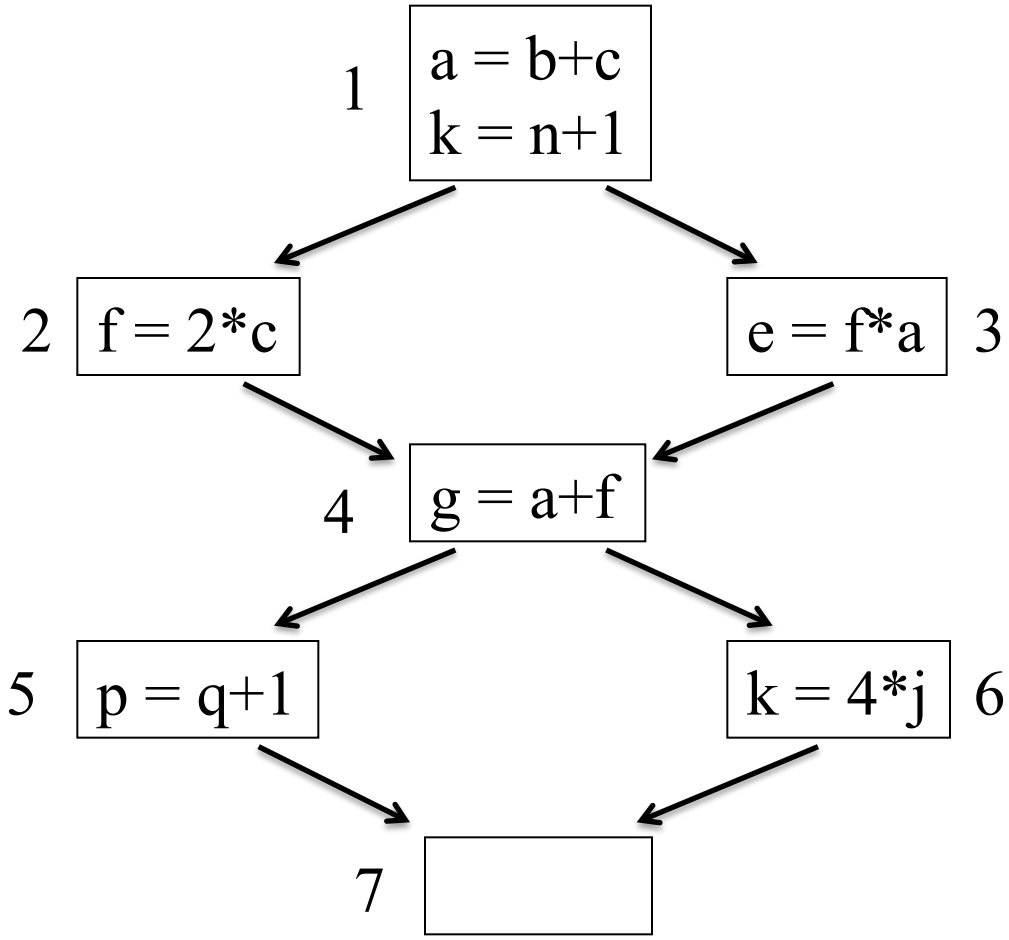
Traço A: 1, 2, 4, 6 e 7 Prob(A): 91% Tempo(A):

Traço B



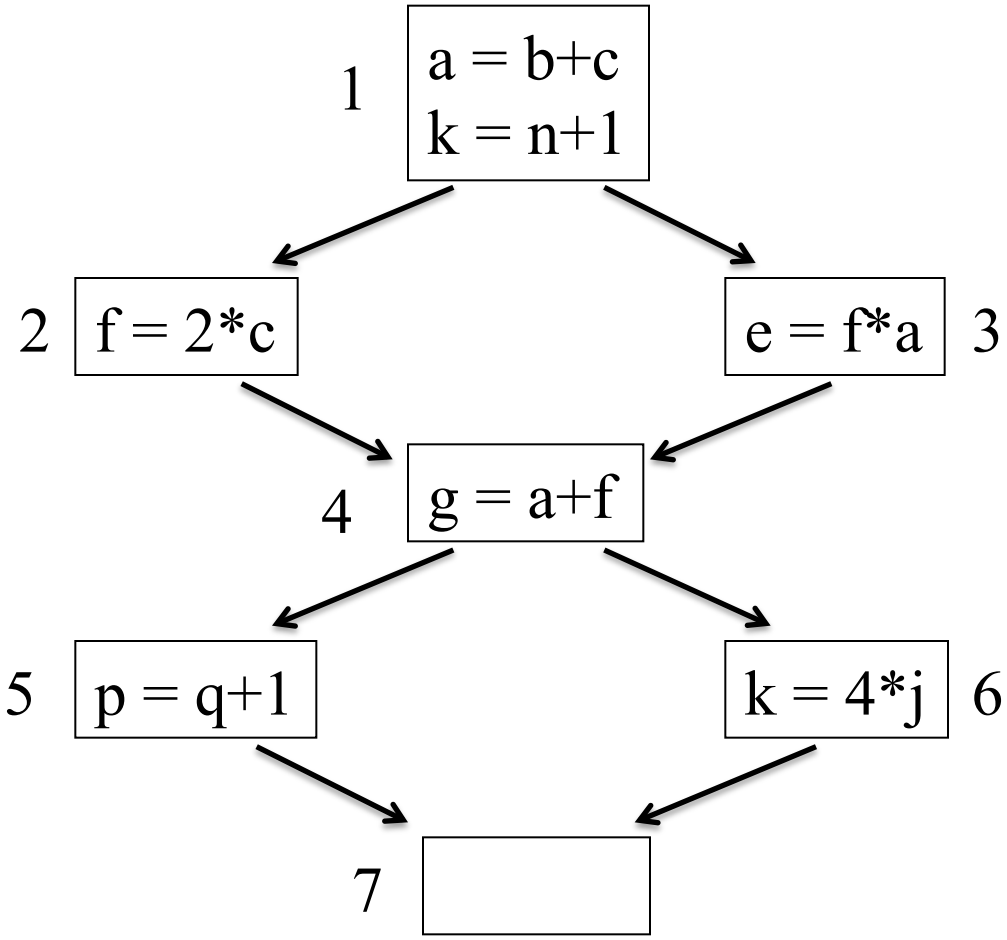
Traço B: 1, 3, 4, 5 e 7 Prob(B): 3% Tempo(B):

Traço C



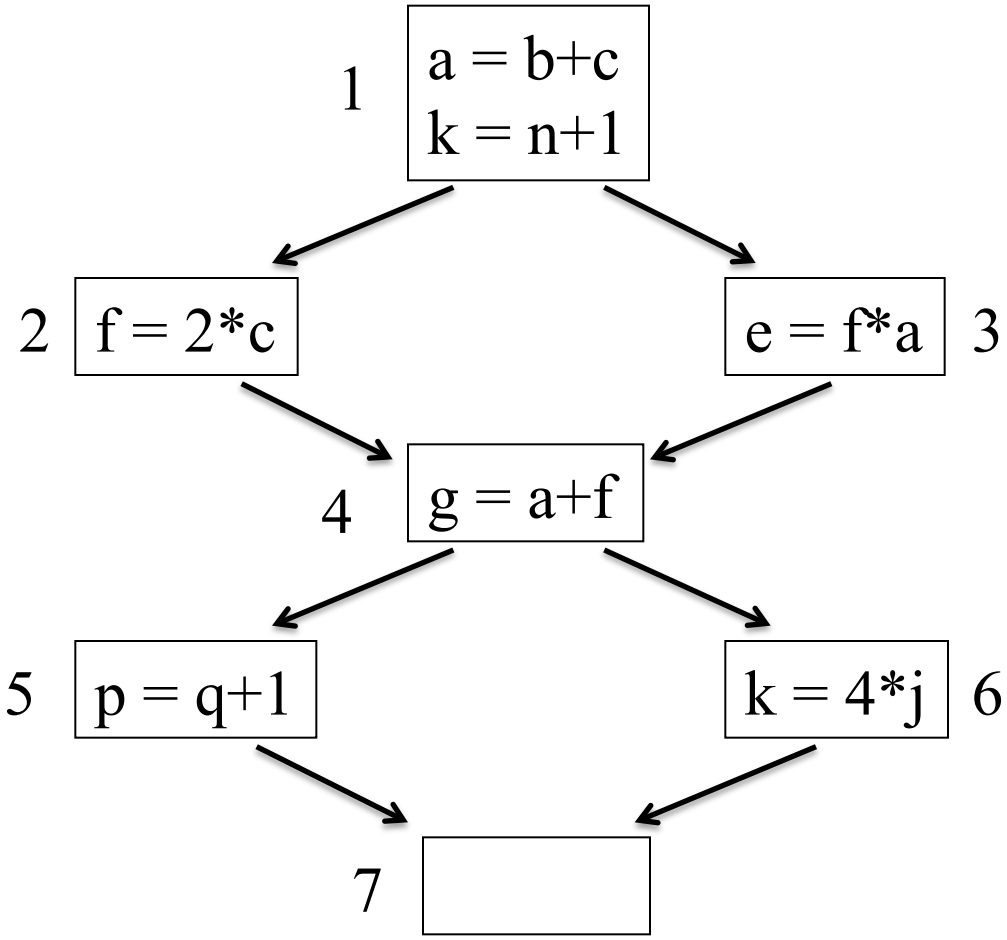
Traço C: 1, 2, 4, 5 e 7 Prob(C): 3% Tempo(C):

Traço D



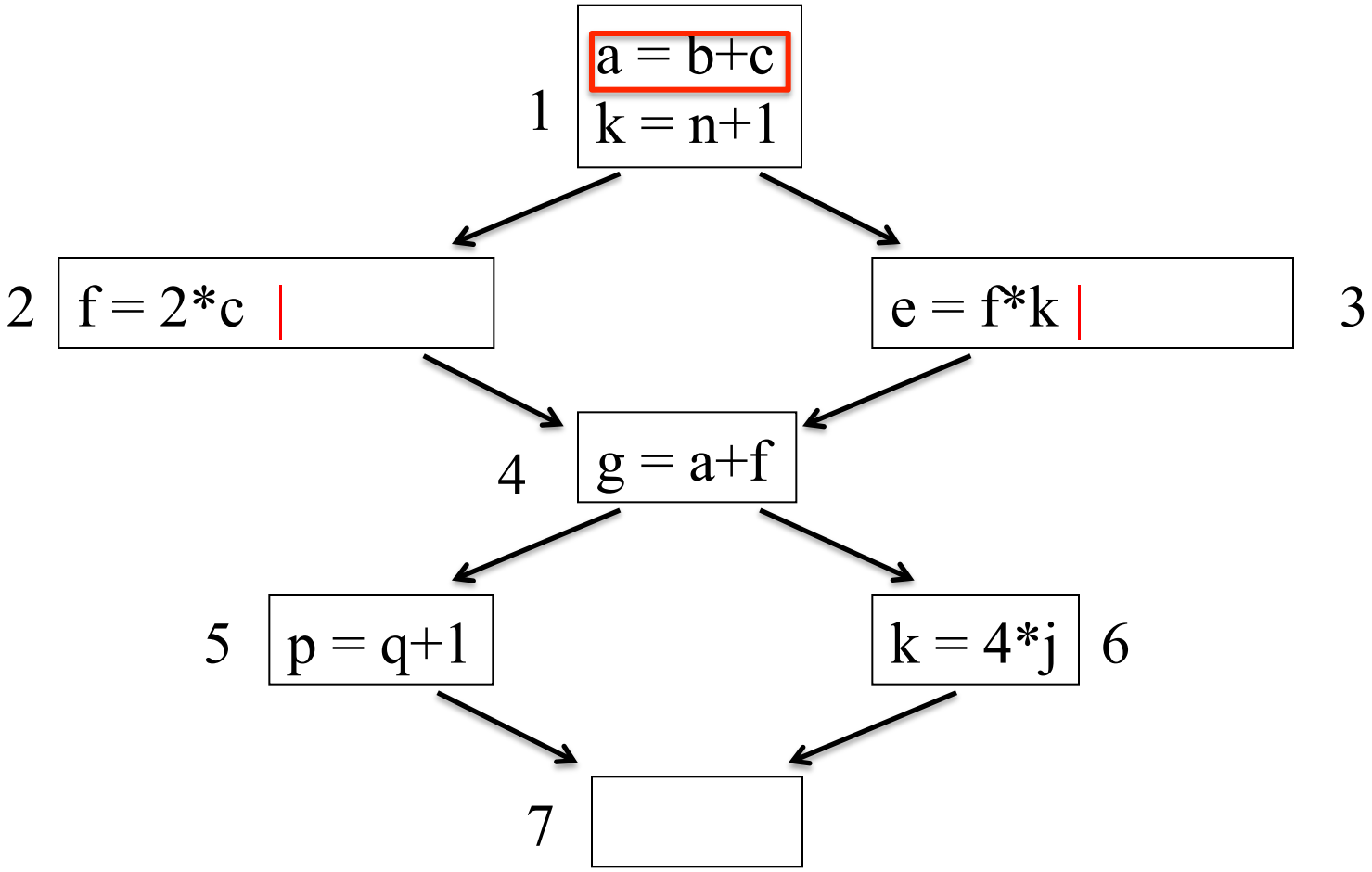
Traço D: 1, 3, 4, 6 e 7 Prob(D): 3% Tempo(D):

Tempo Total

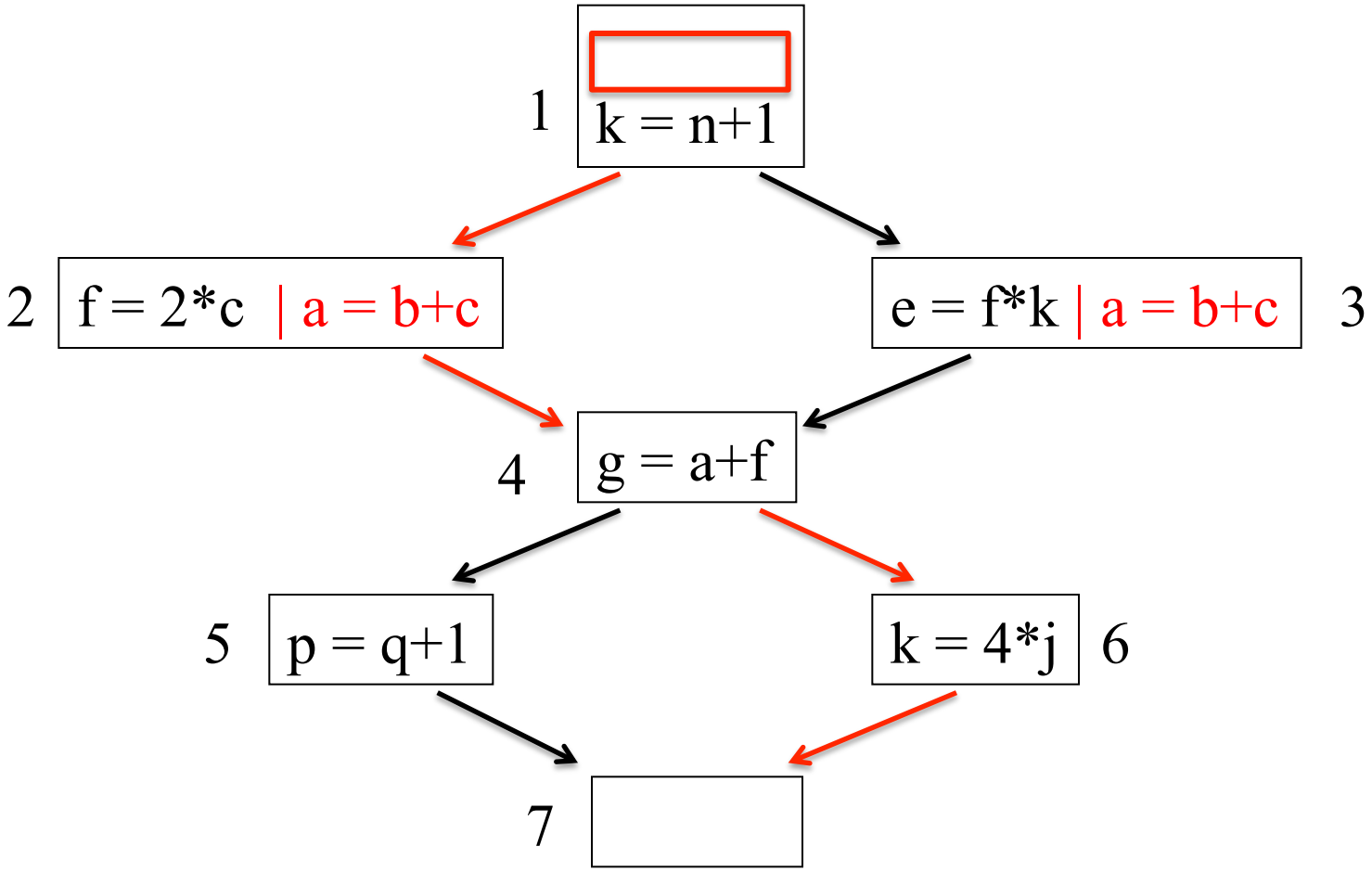


Prob(Total): Time(Total):

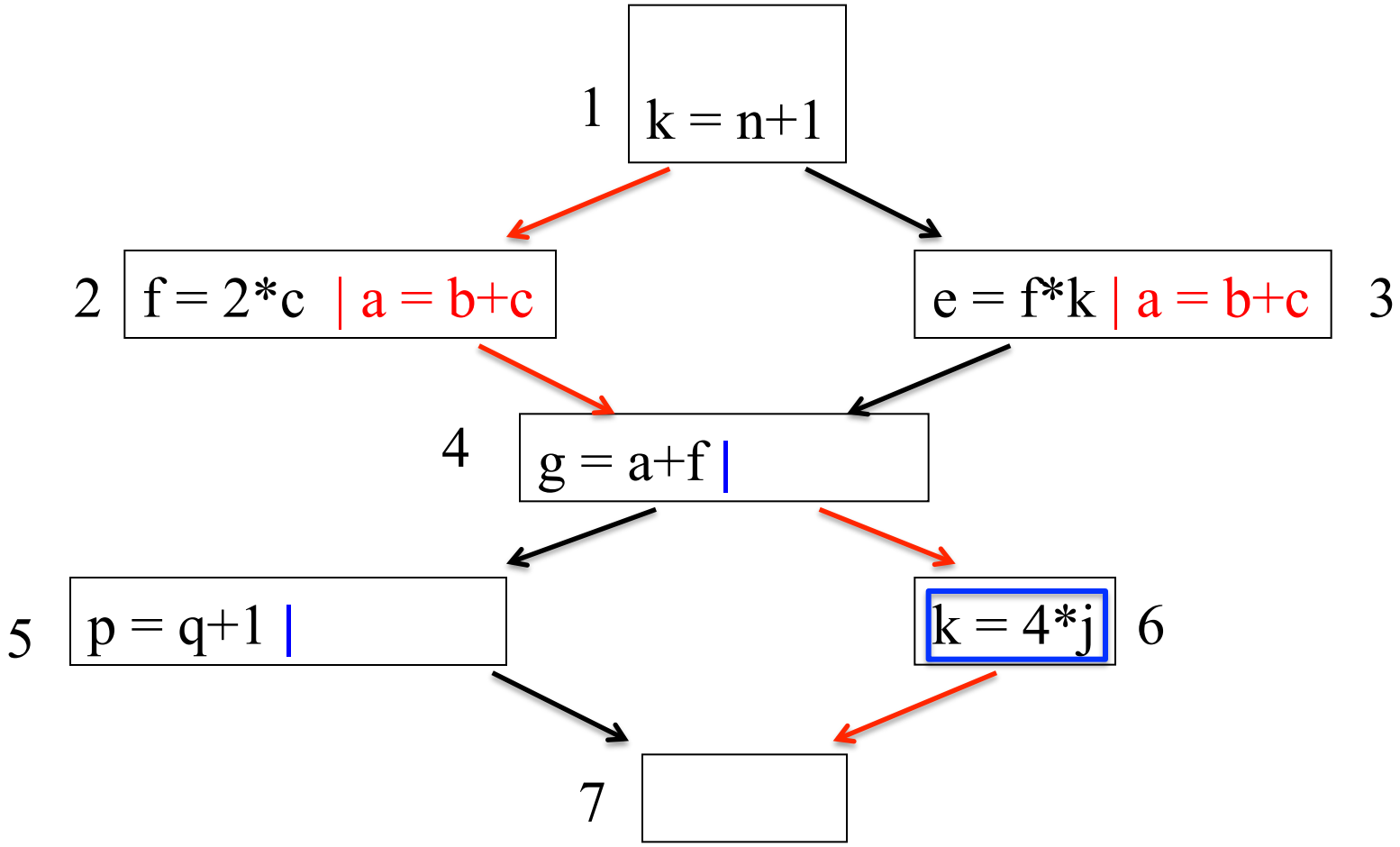
Movendo para Baixo



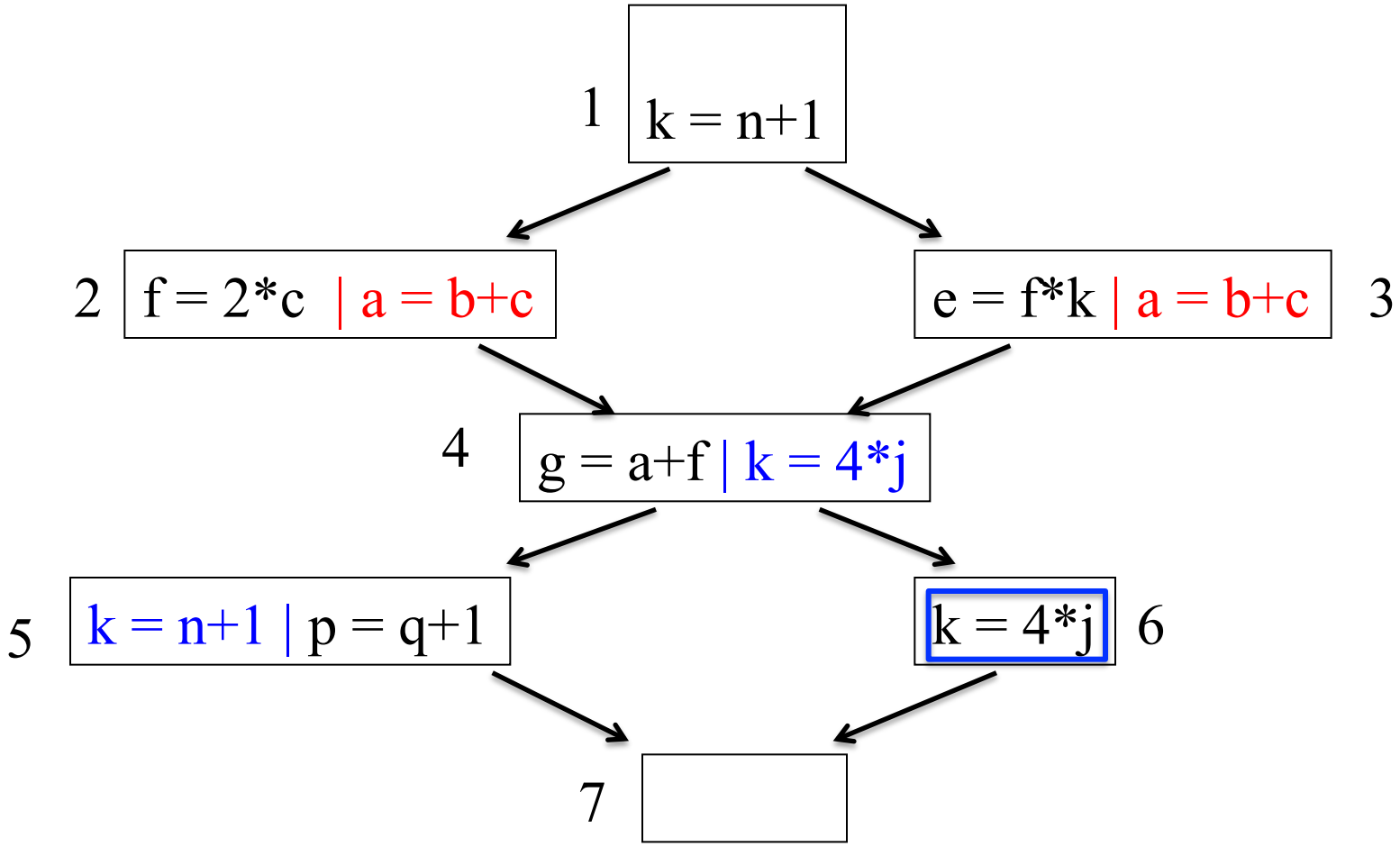
Movendo para baixo



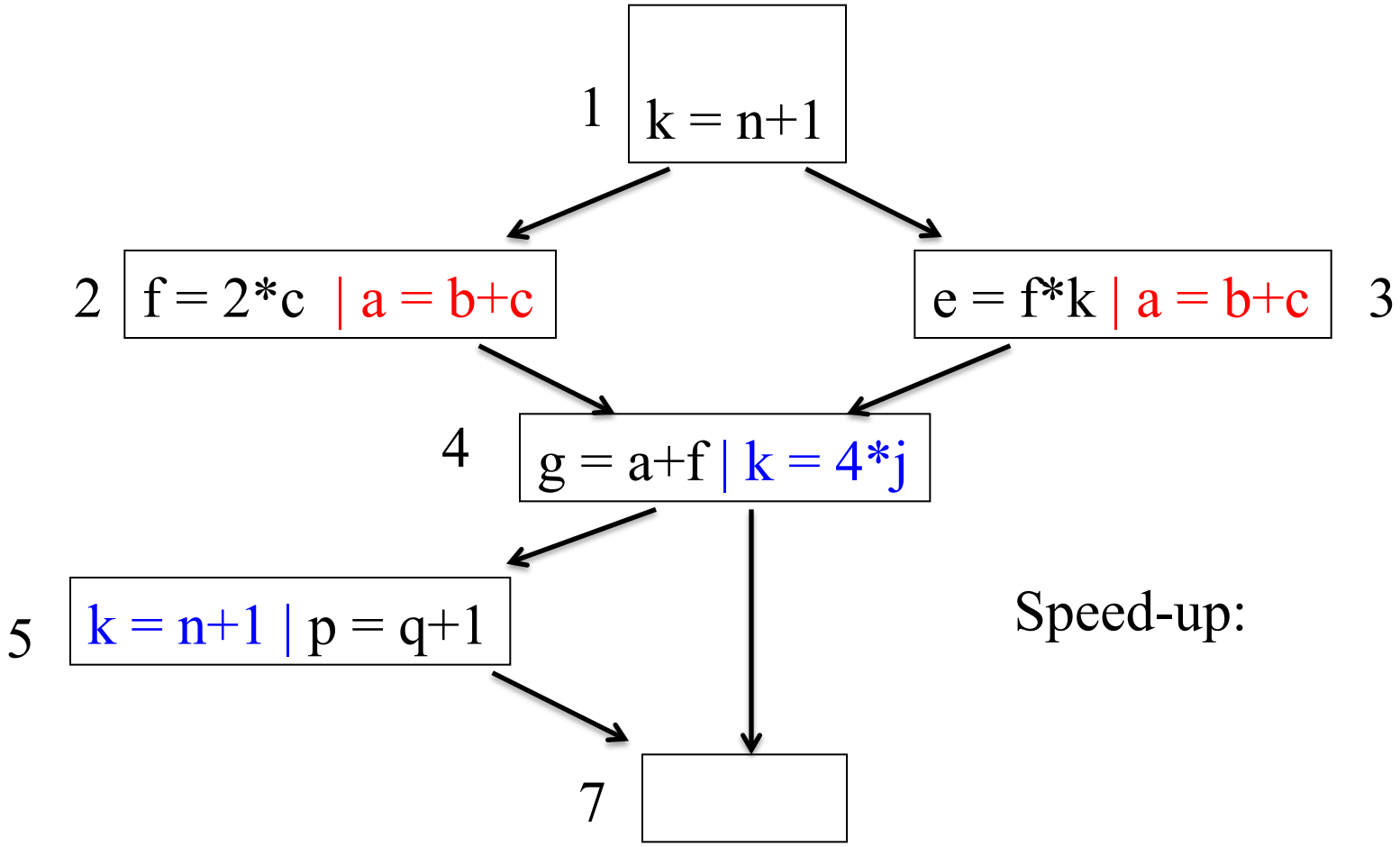
Movendo para cima



Movendo para cima



Qual será o novo tempo?



Speed-up:

Tempo(Novo):

Roteiro

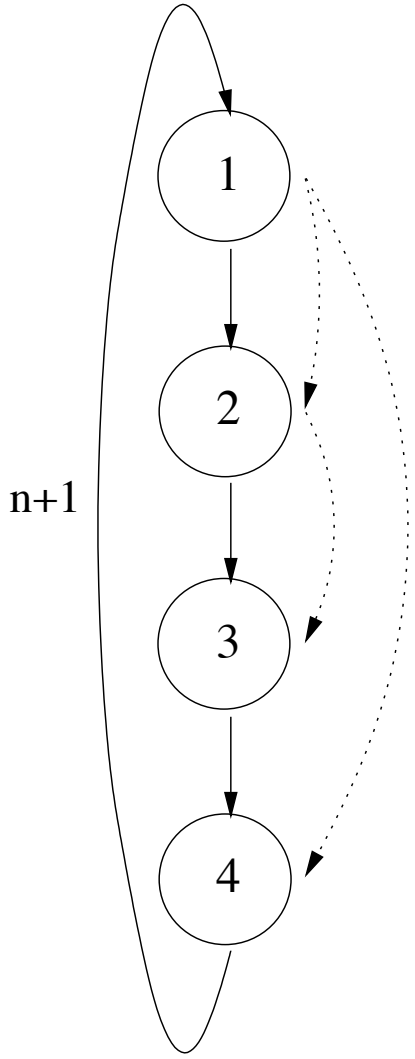
- Arquiteturas Paralelas
- Paralelismo em MIMD
- Paralelismo em Multicores
- Paralelismo em SIMD

Paralelismo em Arquiteturas MIMT

- Doall
- Doacross
- Software pipelining
- Decoupled Software Pipelining

Grafo de Dependências

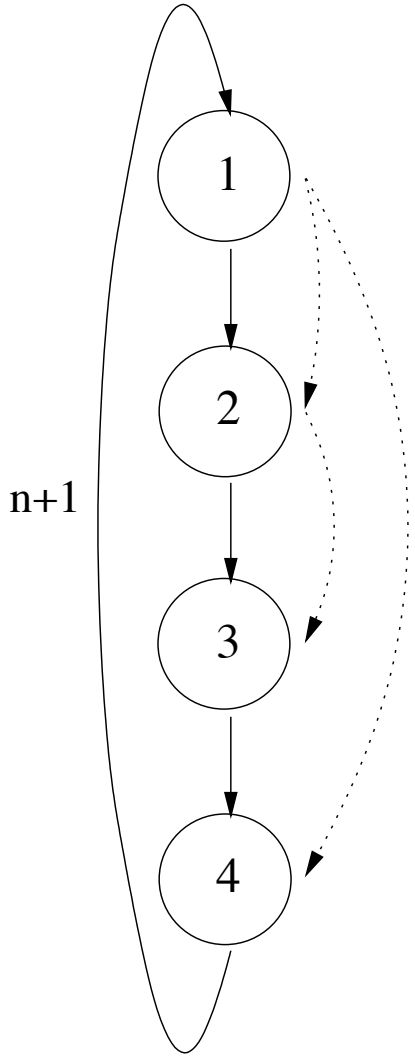
- Arestas cheias
  - Dependência de controle
- Arestas pontilhadas
  - Dependência de dados



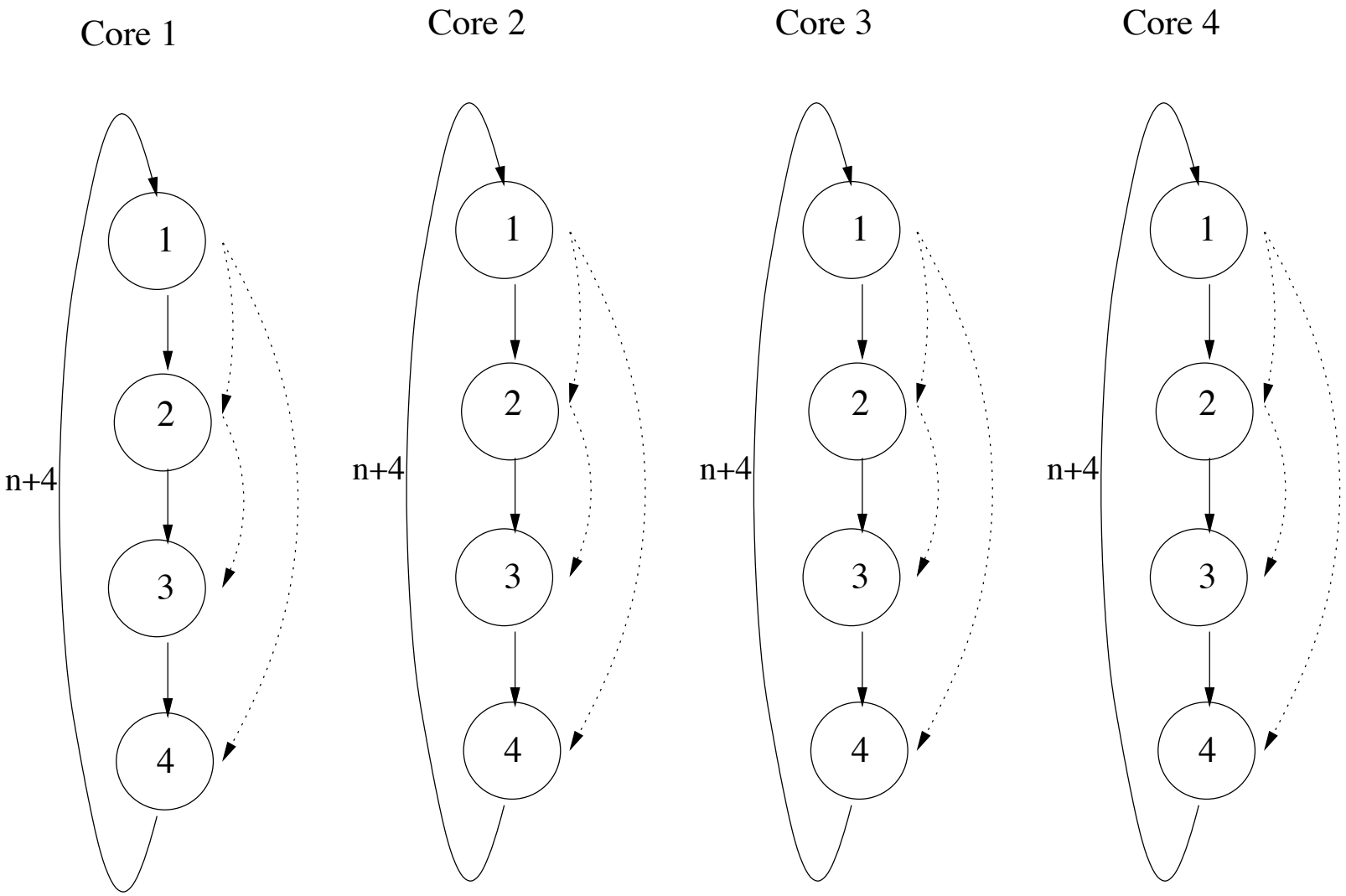
Doall

- Não existe ciclo de dependência
  - Podemos alocar um conjunto de iterações em cada núcleo
  - O laço levará N/4 para executar

```
for (i = 0; i <= N; i++) {
   (1) C[i] = A[i] + B[i];
   (2) D[i] = C[i] << 2;
   (3) E[i] = D[i] + 1;
   (4) G[i] = C[i] - 1;
 }
```



Doall



Doall

Core 1

```
for (i = 0; i < N/4; i++) {
   (1) C[i] = A[i] + B[i];
   (2) D[i] = C[i] << 2;
   (3) E[i] = D[i] + 1;
   (4) G[i] = C[i] - 1;
 }
```

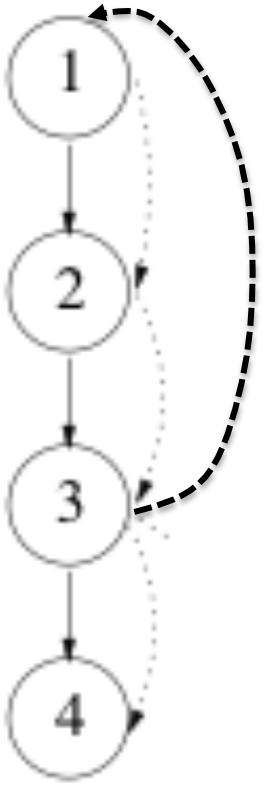
Core 2

```
for (i = N/4; i < N/2; i++) {
   (1) C[i] = A[i] + B[i];
   (2) D[i] = C[i] << 2;
   (3) E[i] = D[i] + 1;
   (4) G[i] = E[i] - 1;
 }
```

.....

Doacross

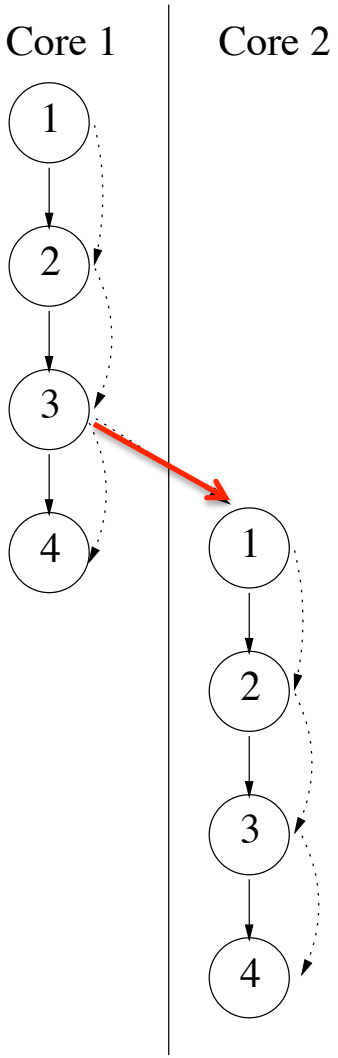
- Existe ciclo de dependência
  - Linha (1) na próxima iteração (i+1) depende de (3) nesta iteração (i)




```
for (i = 0; i < N; i++) {
   (1) C[i] = A[i] + B[i];
   (2) D[i] = C[i] << 2;
   (3) A[i+1] = D[i] + 1;
   (4) E[i] = A[i+1] - 1;
 }
```

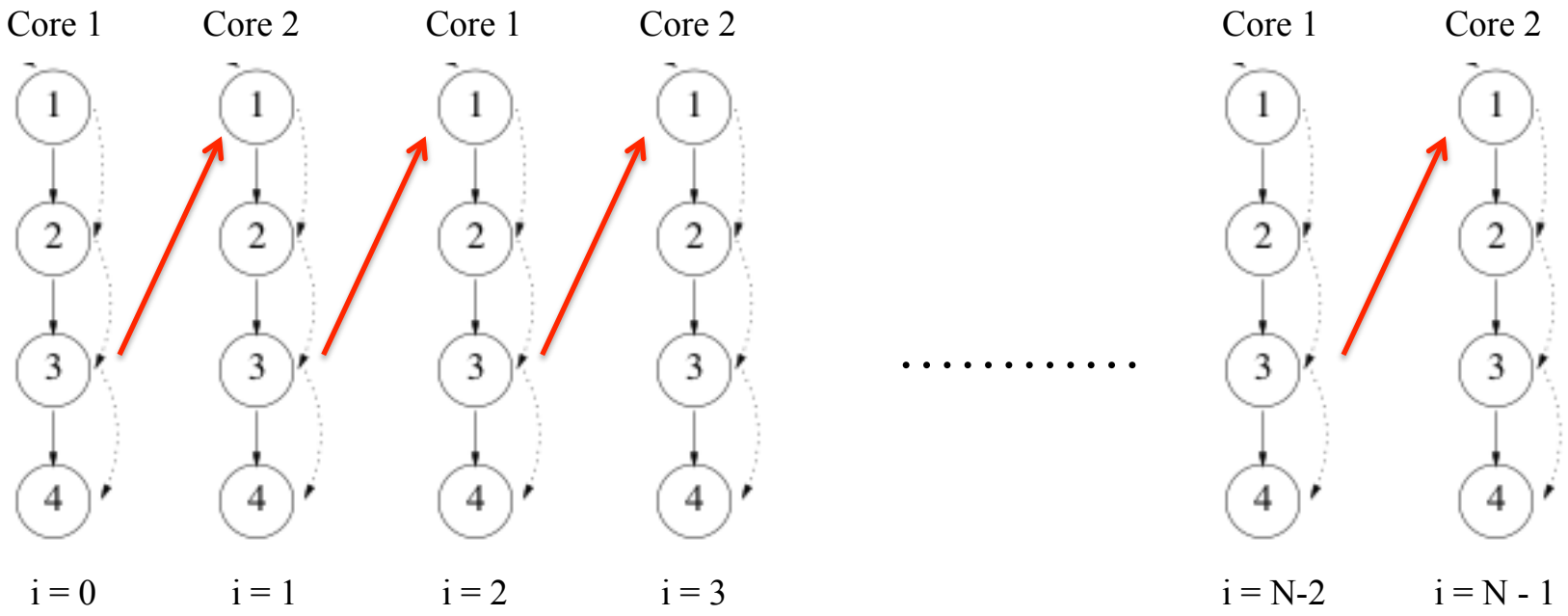
Doacross

- E se dependência “loop-carried”?
  - Replicar o laço em núcleos diferentes mas respeitando a dependência 3 -> 1



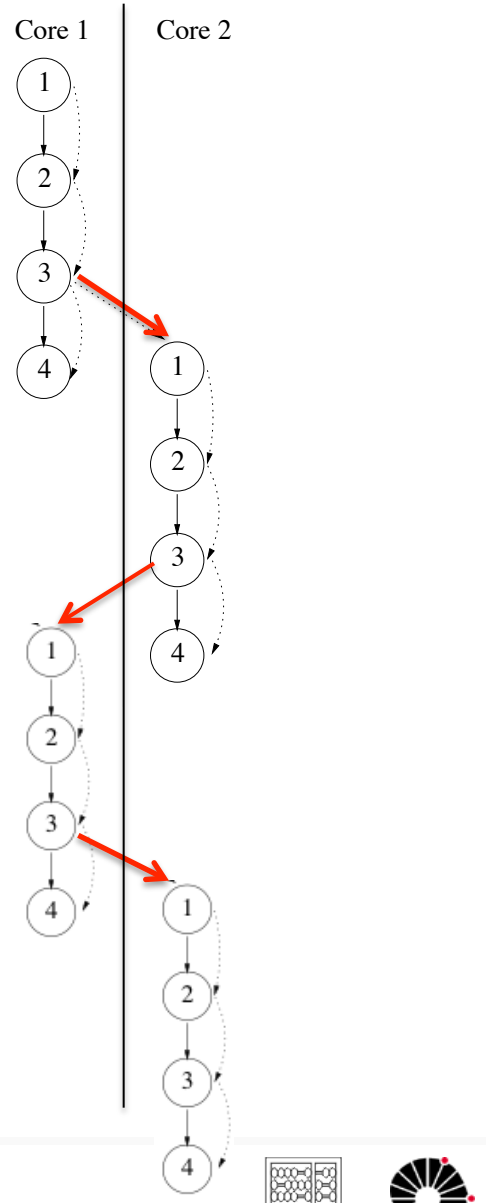

```
for (i = 0; i < N; i++) {
   (1) C[i] = A[i] + B[i];
   (2) D[i] = C[i] << 2;
   (3) A[i+1] = D[i] + 1;
   (4) E[i] = A[i+1]-1;
 }
```

Doacross (Software Pipelining)



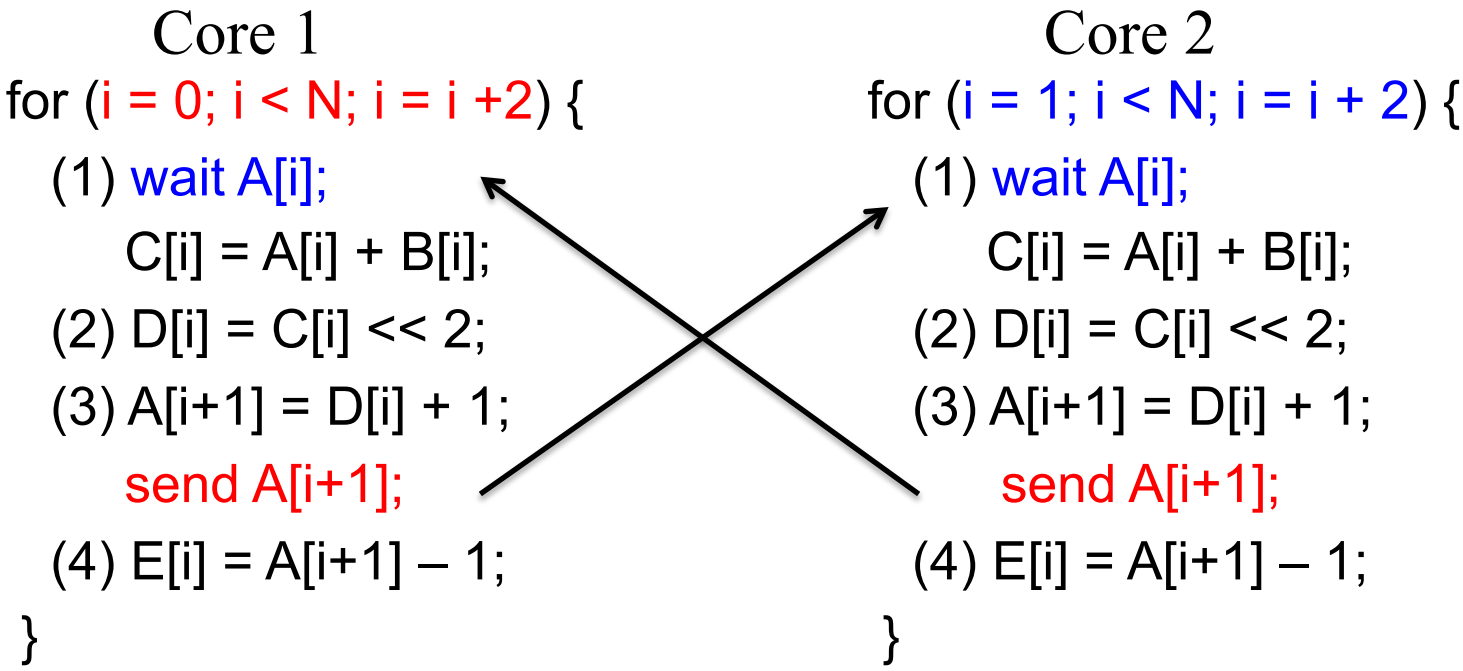
Software Pipelining

- Paralelismo
  - 4 (em i) em paralelo com 1 (em i+1)
  - Total: 3N+1 ciclos
  - Speed-up: 4N/(3N+1) ~ 33%
- Problemas
  - Dependência atravessa fila de comunicação (vai e volta)
  - send/wait bloqueia o paralelismo



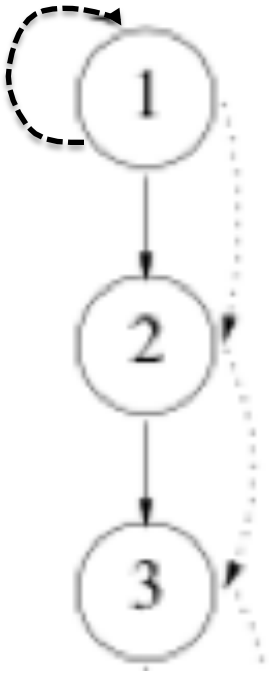
Doacross

- Dado é sincronizado entre cores
  - Core 2 espera por A[i+1] enviado por core 1.
  - Core 1 espera por A[i+2] enviado por core 2.



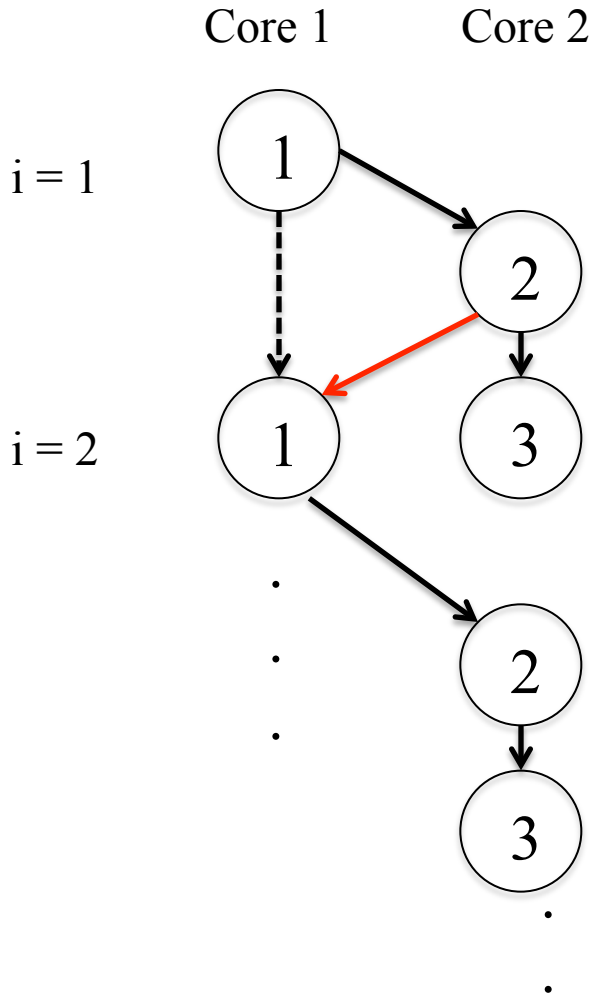
Um outro exemplo

```
for (i = 1; i < N=; i++) {
   (1) A[i] = 2*A[i-1];
   (2) C[i] = A[i] + 1;
   (3) D[i] = C[i] + 1;
}
```



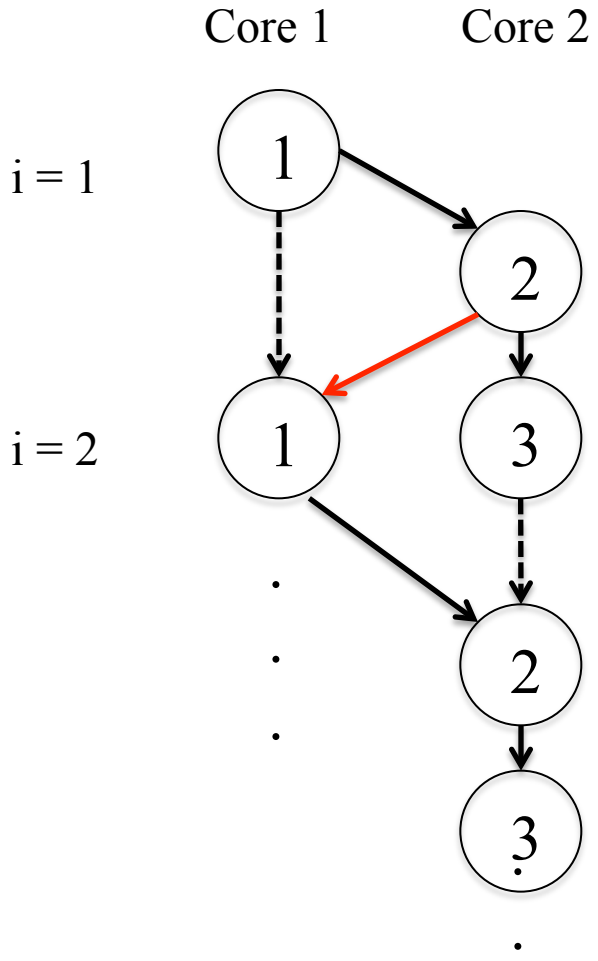
Como ficaria com Software Pipelining?

- Tem que esperar pelo início do laço no outro core!



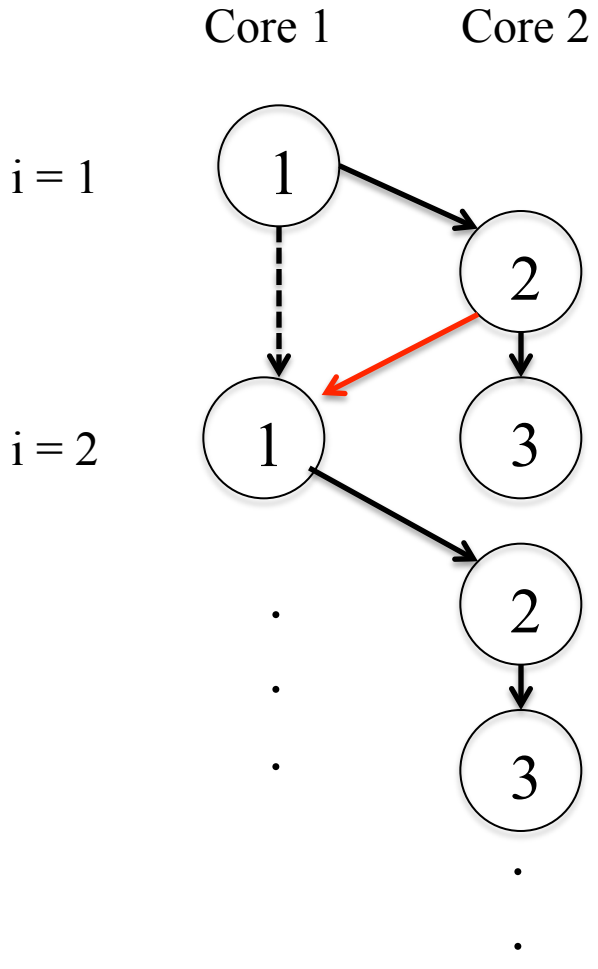
Será que dá para melhorar?

- Speed-up:
  - 3N/(2N+1) ~ 50%
- Pergunta:
  - Será que haveria uma maneira do core 1 não esperar pelo core 2?



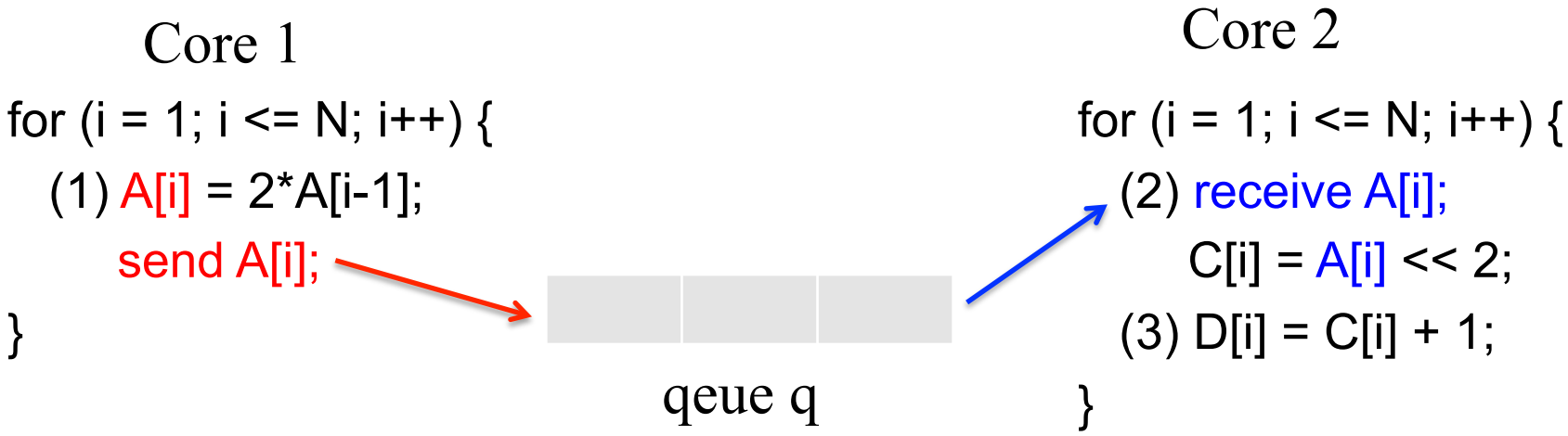
Será que dá para melhorar?

- Speed-up:
  - 3N/(2N+1) ~ 50%
- Pergunta:
  - Será que haveria uma maneira do core 2 não esperar pelo core 1?
  - E se colocarmos uma fila entre o core 1 e o core 2?

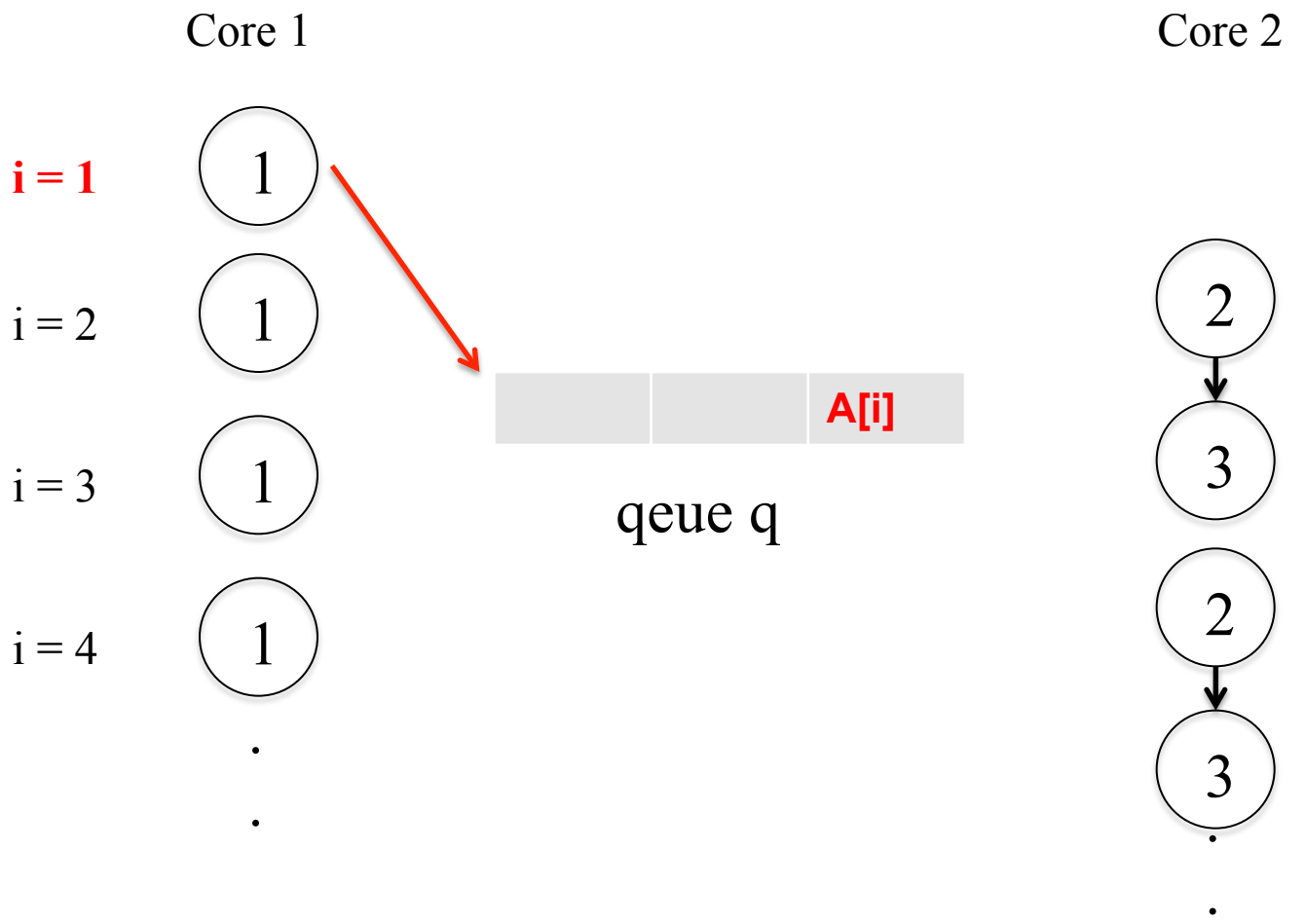


Decoupled Software Pipelining (DSWP)

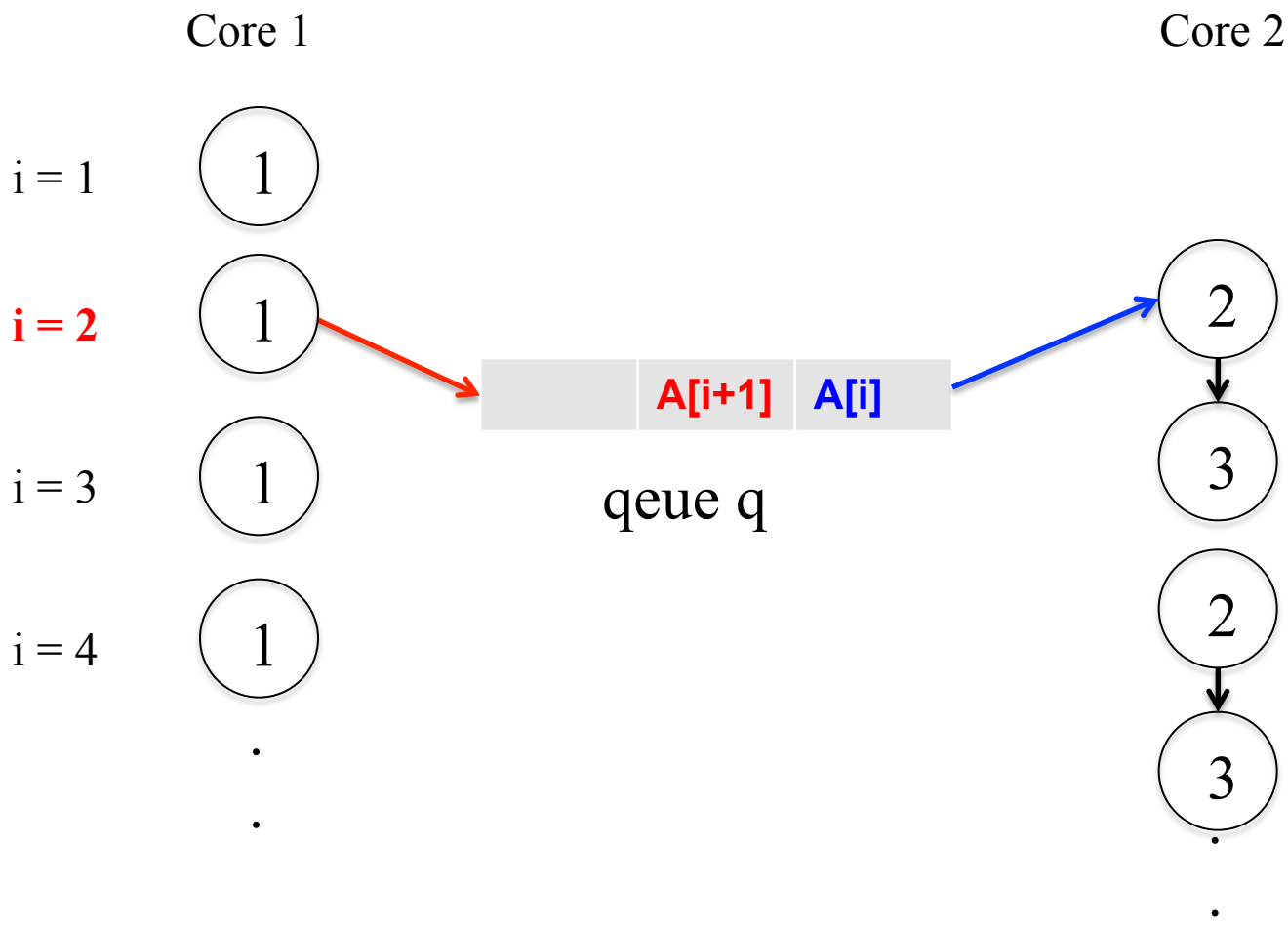
- Usando uma fila para comunicar dados
  - A[i] calculado no core 1, enviado para core 2
  - Fila desacopla execução de ambos cores!



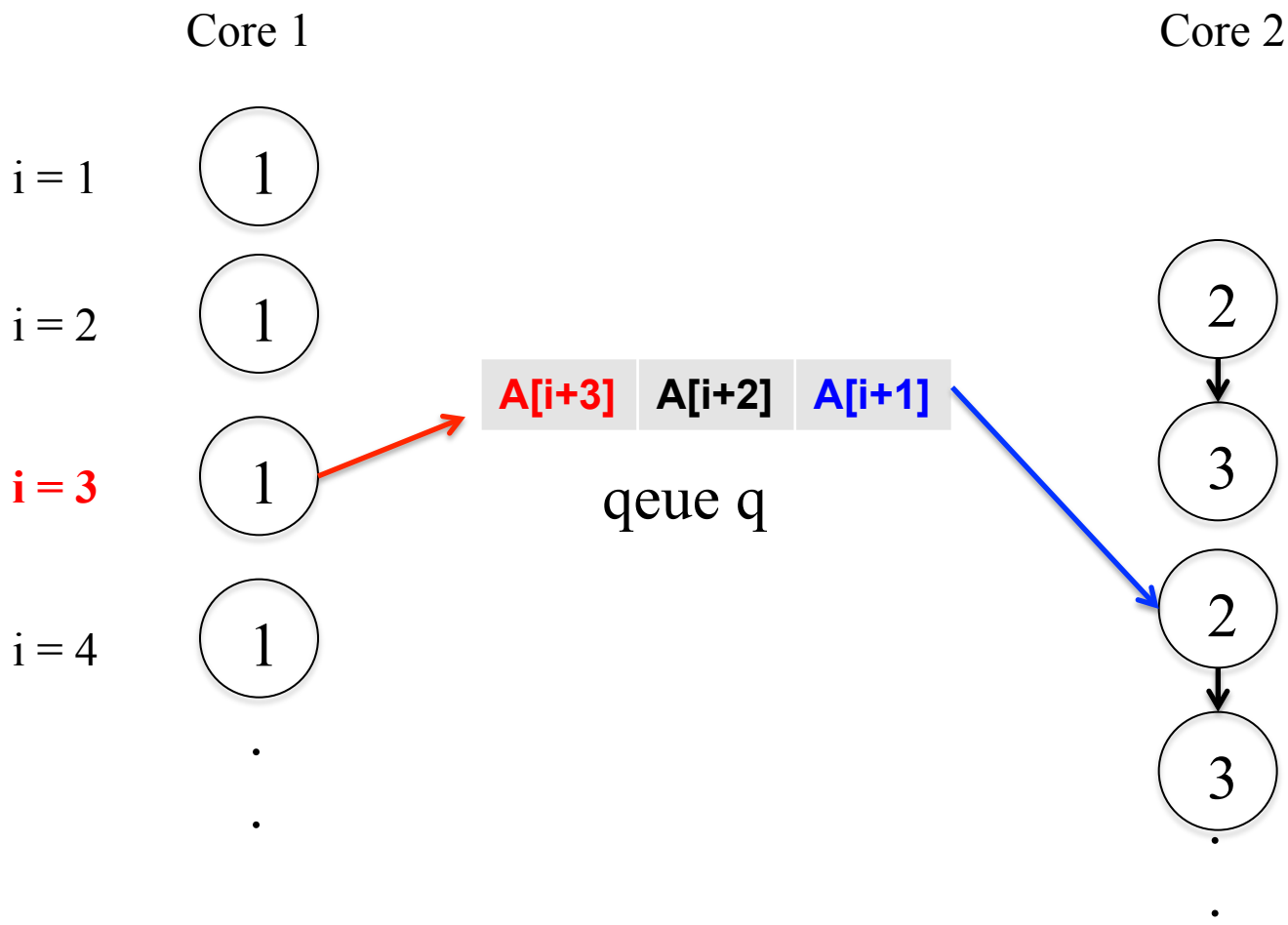
Decoupled Software Pipelining (DSWP)



Decoupled Software Pipelining (DSWP)

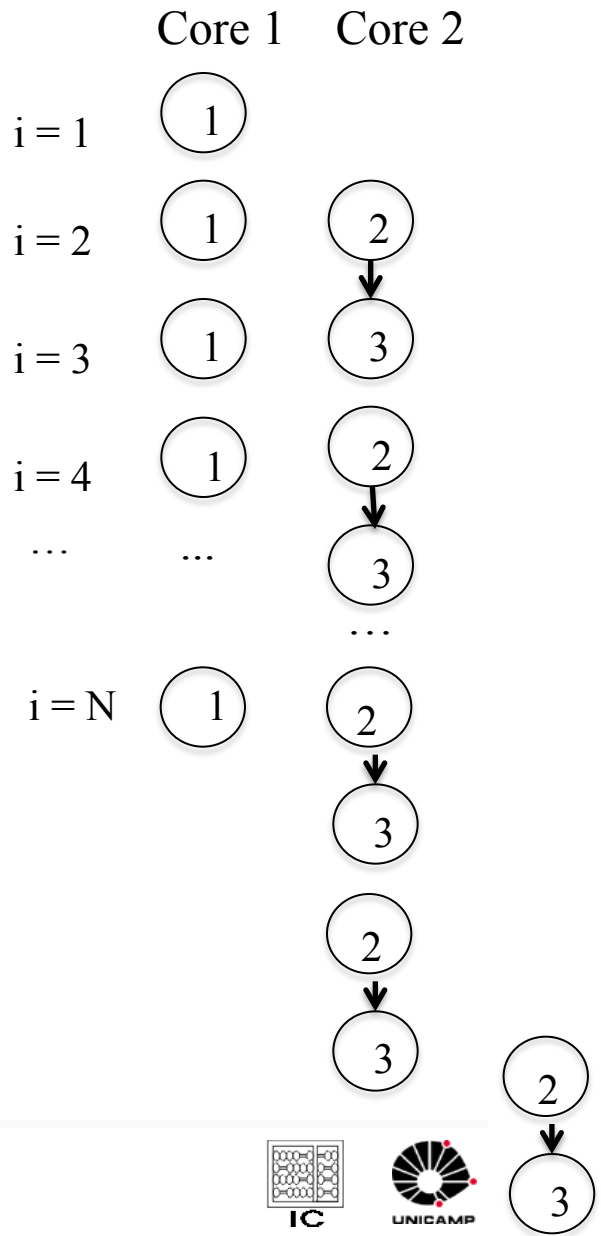


Decoupled Software Pipelining (DSWP)



Como ficaria com DSWP?

- Speed-up:
  - 3N/(N+5) ~ 300% !!



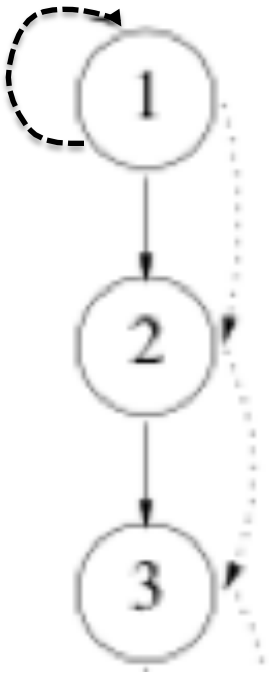
Sempre Funciona?

- Como garantir que sempre funciona?
  - A fila deve enviar dados sempre em uma direção de outro modo ocorre um travamento e voltamos à Doacross

- Separar grafo em componentes
  - Não podem existir ciclos entre componentes

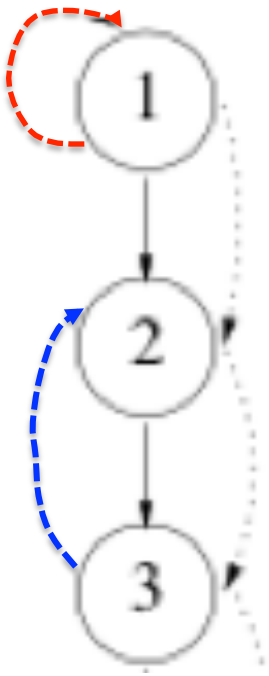
Como assim?

```
for (i = 1; i <= N; i++) {
   (1) A[i] = 2*A[i-1];
   (2) C[i] = A[i] << 2;
   (3) D[i] = C[i] + 1;
}
```



Um outro Exemplo?

```
for (i = 1; i <= N; i++) {
   (1) A[i] = 2*A[i-1];
   (2) C[i] = A[i] << 2;
   (3) C[i] = C[i-1] + 1;
}
```



É sempre possível encontrar os ciclos?

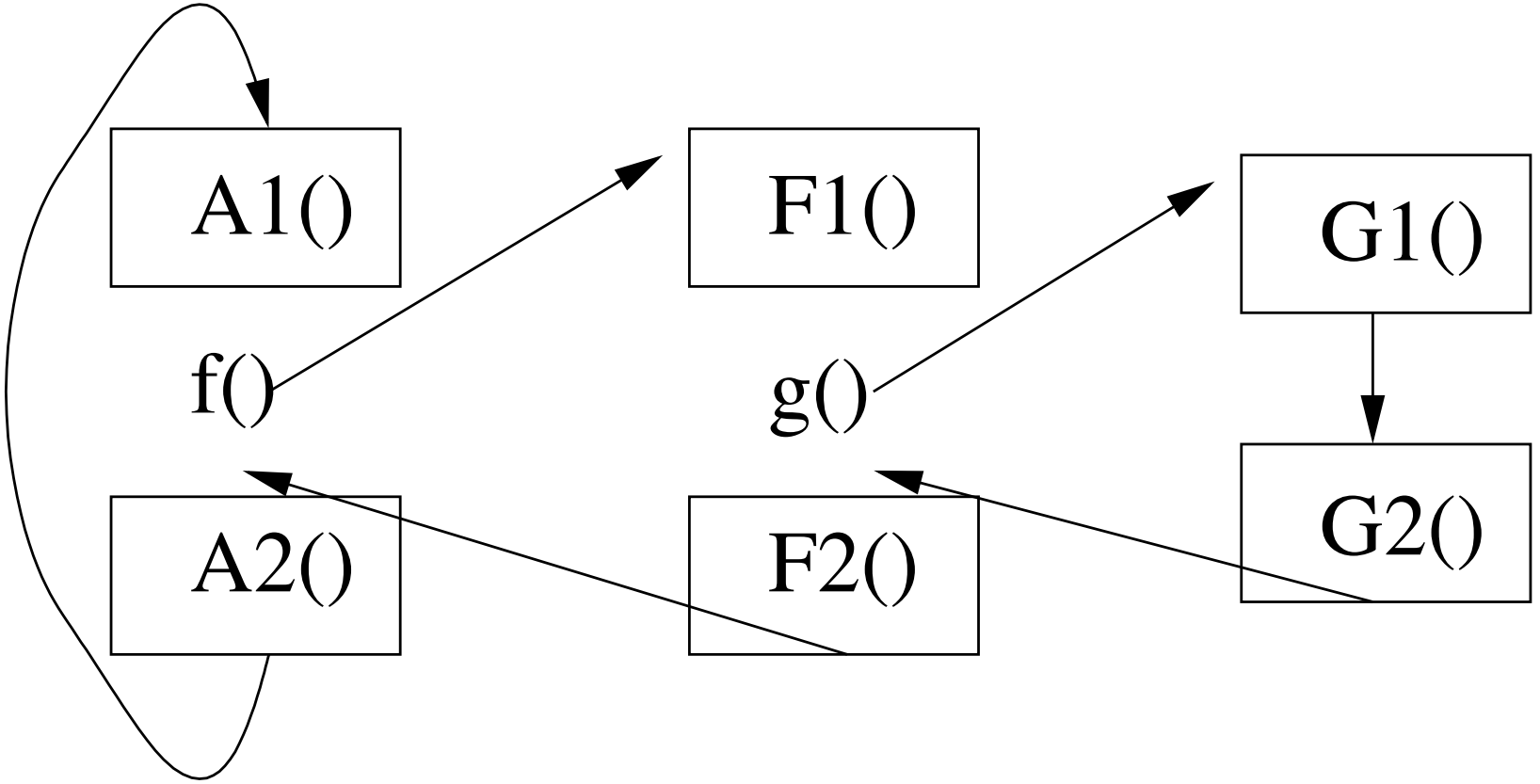
- Chamadas de procedimentos presentes

```
A() {
    for (i = 0; i < N; i++) {
      (1) A1();
      (2) f();
      (3) A2();
}
```

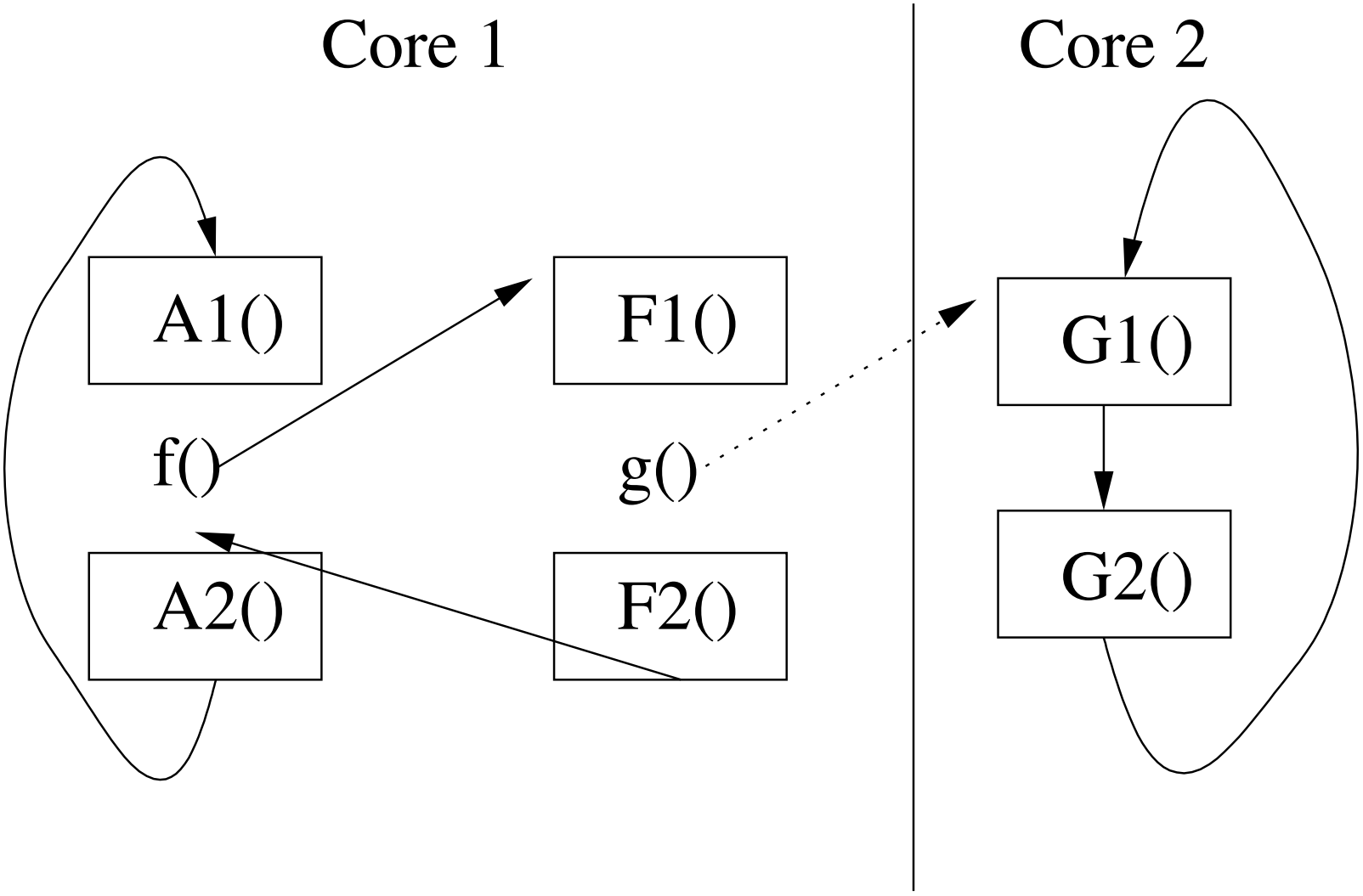
```
f() {
      (1) F1();
      (2) g();
      (3) F2();
}
```

```
g() {
      (1) G1();
      (3) G2();
}
```

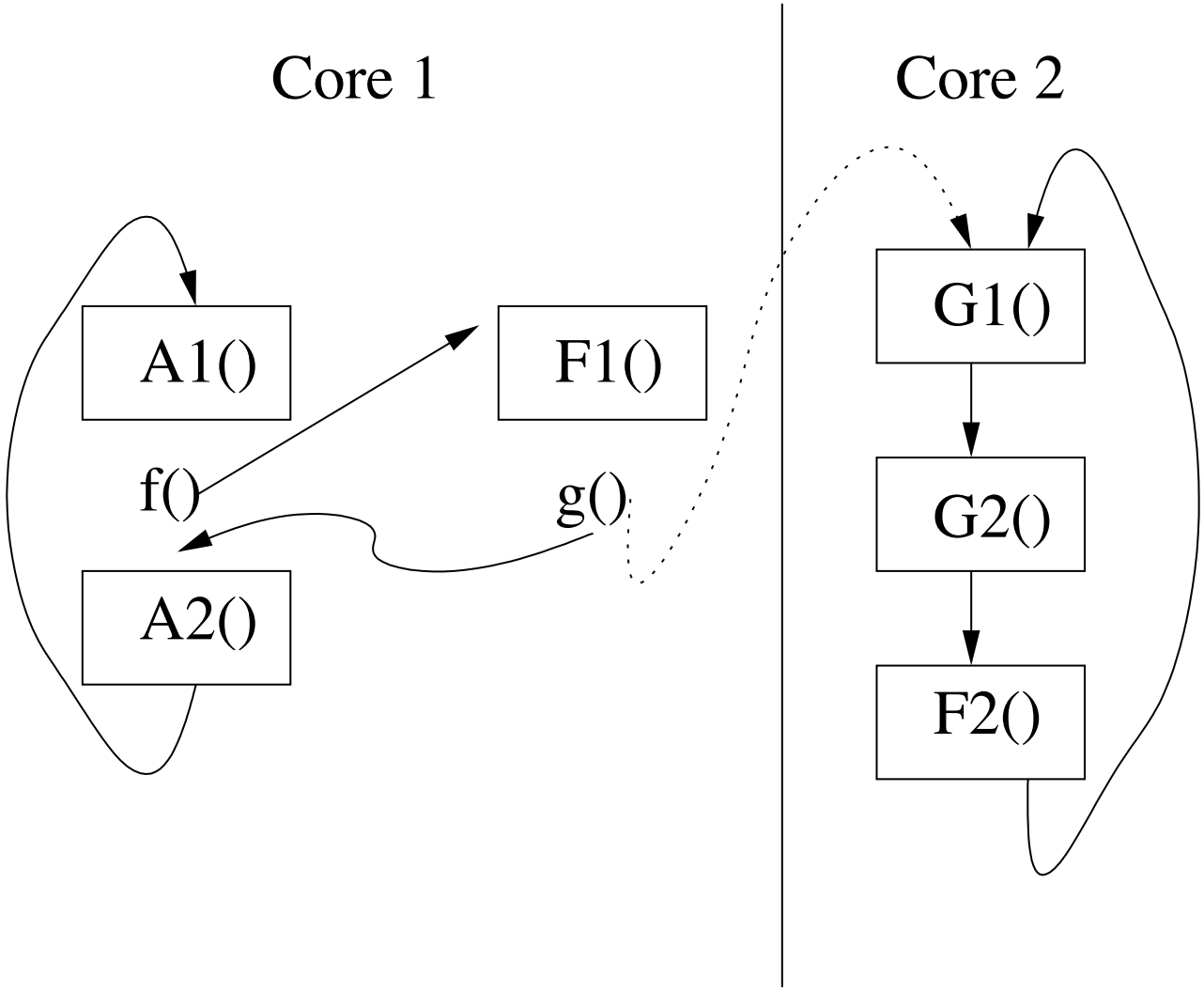
Grafo de Dependência entre Procedimentos



Separando em duas componentes

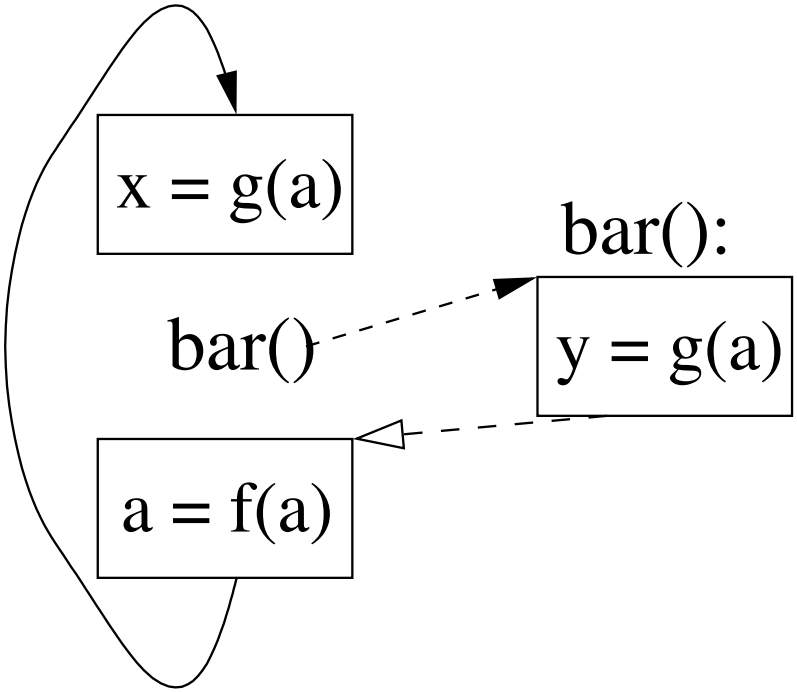


Alterando o Balanceamento



É sempre possível detectar ciclos?

- Sim, desde que seja possível resolver as dependências

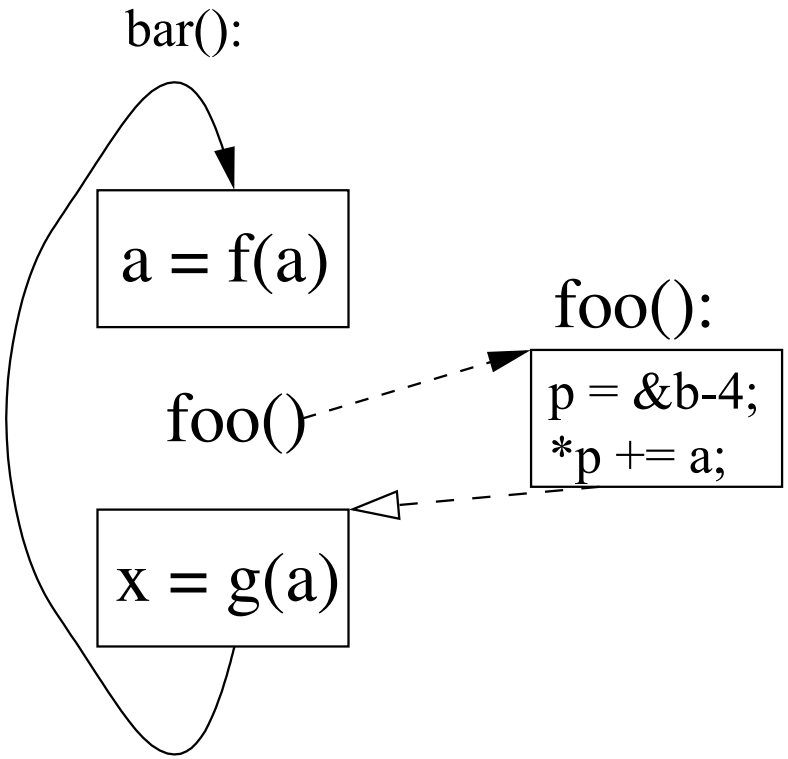


E como seria para este exemplo?

```
static int a;
static int b;

bar() {
   a = f(a);
   foo();
   x = g(a);
}

foo(y) {
    p = &b-4;
    *p += a;
}
```



Se você achar uma resposta me avise !!

Temos algumas idéias….

Roteiro

- Arquiteturas Paralelas
- Paralelismo em MIMD
- Paralelismo em Multicores
- Paralelismo em SIMD

SIMD (Vector)

- Single Instruction Multiple Data

```
for (i = 0; i < N; i++)
    C[i] = A[i] + B[i];
```

Uma única instrução

A[ ]	A[0]	A[1]	A[2]	.....	.....	.....	A[N-2]	A[N-1]
					+			
B[ ]	B[0]	B[1]	B[2]	.....	.....	.....	B[N-2]	B[N-1]
					=			
C[ ]	C[0]	C[1]	C[2]	.....	.....	.....	C[N-2]	C[N-1]

Paralelizando em Arquiteturas SIMD (Vector)

- Arquitetura Vetorial
  - Registradores vetoriais
  - Instruções vetoriais

- Paralelizando
  - Vetorizando um laço
  - Expansão de escalar
  - Redução
  - Divisão conjunto índices

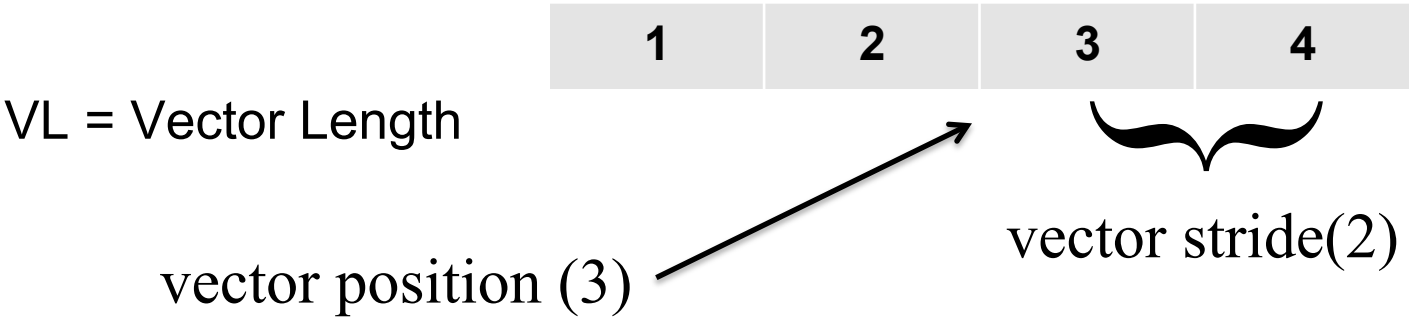
Paralelizando em Arquiteturas SIMD (Vector)

- Arquitetura Vetorial
  - Registradores vetoriais
  - Instruções vetoriais

- Paralelizando
  - Vetorizando um laço
  - Expansão de escalar
  - Redução
  - Divisão conjunto índices

Arquitetura Vetorial

- Registradores vetoriais



- Banco de registradores vetoriais
  - 8 vetores de 1 K elementos
  - 128 vetores de 64 elementos

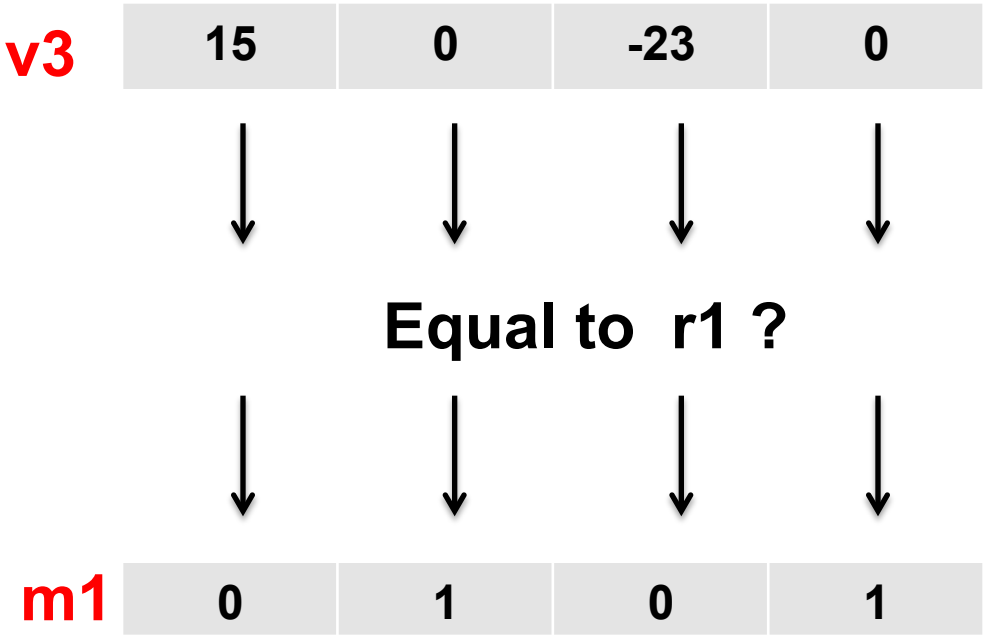
Instruções Vetoriais

- Define tamanho do vetor
  - VL = max(N, 64)
- Operação aritméticas
  - vadd v1 + v2 -> v3
- Acesso à memória
  - vfetch A[i], 1 -> v1
  - vstore v1 -> B[i], 1
- Carrega registrador com imediato
  - loadi #0 -> r1

Comparação Vetorial

- Comparação
  - loadi #0, r1

    vcmp v3, r1 -> m1



Instruções Vetoriais com Condicionais

- Operação aritmética condicional
  - vaddc v1 + v2 -> v3, m1

m1	0	1	0	1
	12	8	-4	-5
			+	
	7	12	13	6
			=	
	0	20	0	1

- Acesso condicional à memória
  - vstorec v1 -> B[1], 1, m1

Paralelizando SIMD (Vector)

- Arquiteturas Vetoriais
  - Registradores vetoriais
  - Instruções vetoriais

- Paralelizando
  - Vetorizando um laço
  - Expansão de escalar
  - Redução
  - Divisão conjunto índices

Vetorizando um Laço

- Exemplo

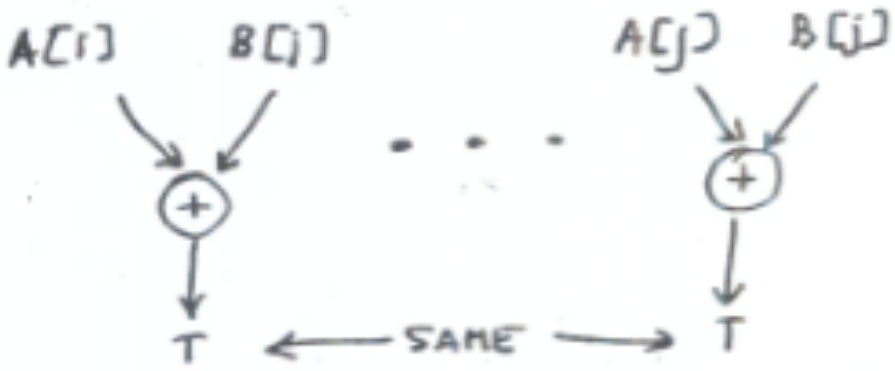
```
(1)   forall I = 1 to N do
(2)      A[I] = B[I] + C[I]
(3)      if A[I] > 0 then
(4)         B[I] = B[I] + 1
(5)      endif
(6)   endforall
```

```
for I = 1 to N by 64 do
   VL = max(N-I+1,64)
   vfetch B[I],1 -> v1
   vfetch C[I],1 -> v2
   vadd v1+v2 -> v3
   vstore v3 -> A[I],1
   loadi #0 -> r1
   vcmp v3,r1 -> m1
   loadi #1 -> r2
   vaddc v1+r1 -> v1, m1
   vstorec v1 -> B[I],1, m1
endfor
```

Expansão de Escalar

- E se o laço contiver um escalar?

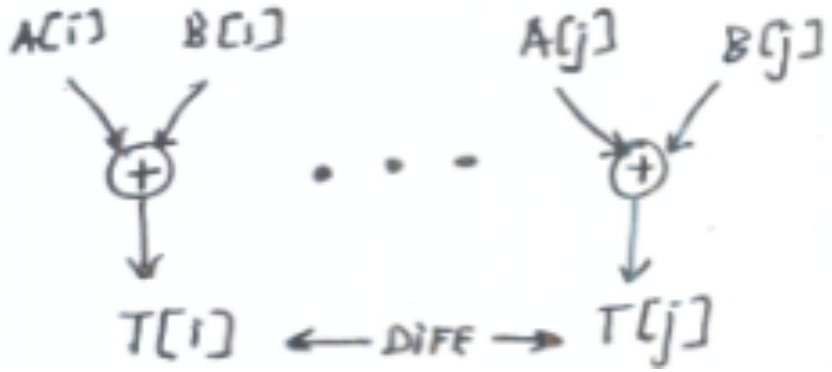
```
(1)   for I = 1 to N do
(2)      T = A(I) + B(I)
(3)      C(I) = T + 1/T
(4)   endfor
```



Expansão de Escalar (Exemplo)

- E se o laço contiver um escalar?
  - Expanda o escalar em um vetor

```
        if N >= 1 then
           allocate Tx(1:N)
(1)        for I = 1 to N do
(2)           Tx(I) = A(I) + B(I)
(3)           C(I) = Tx(I) + 1/Tx(I)
(4)        endfor
           T = Tx(N)
        endif
```



Strip-mining

- E se o registrador vetorial for menor que a viagem do laço ?
  - Tamanho registrador 64 bits.
  - Viagem do laço: N = 1024.

```
for (i = 1; i <= N; i++)
    C[i] = A[i] + B[i];
```

Strip-mining

- Dividir de modo a caber no vetor
- Usa VL para cuida do tamanho

```
for (i = 1; i <= N; i++) {
    A[i] = B[i] + C[i];
}
```

```
for (i = 1; i <= N; i = i + 64) {
    VL = min(N-i+1, 64);
    vfetch A[i],1 -> v1;
    vfetch B[i],1 -> v2;
    vadd v1 + v2 -> v3;
    vstore v3, A[i],1;
}
```

Redução

- E se o resultado do laço for um escalar?

```
(1)   forall I = 1 to N do
(2)      A[I] = B[I] + C[I]
(3)      ASUM += A[I]
(4)   endforall
```

Redução

- E se o resultado do laço for um escalar?
  - Strip-mining, seguido de redução

```
(1)   forall I = 1 to N do
(2)      A[I] = B[I] + C[I]
(3)      ASUM += A[I]
(4)   endforall
```

```
vsub v4-v4 -> v4
for I = 1 to N by 64 do          <- strip-mining
   VL = min(N-I+1,64)
   vfetch B[I],1 -> v1
   vfetch C[I],1 -> v2
   vadd v1+v2 -> v3
   vstore v3 -> A[I],1
   vadd v3+v4 -> v4
endfor
for i = 0 to min(N-1,63)         } REDUCTION
   ASUM = ASUM + v4[i]           }
endfor
```

Remoção de Condição

- E se o corpo do laço tiver condicionais
  - Remove condições de dentro do laço

```
(1)   loop
(2)      statements
(3)      if test then
(4)         then part
(5)      else
(6)         else part
(7)      endif
(8)      more statements
(9)   endloop
```

```
(3)   if test then
(1)      loop
(2)         statements
(4)         then part
(8)         more statements
(9)      endloop
(5)   else
(1)      loop
(2)         statements
(6)         else part
(8)         more statements
(9)      endloop
(7)   endif
```

Remoção de Condição (Exemplo)

```
(1)   for I = 1 to N do
(2)       for J = 2 to N do
(3)           if T[I] > 0 then
(4)               A[I,J] = A[I,J-1]*T[I] + B[J]
(5)           else
(6)               A[I,J] = 0.0
(7)           endif
(8)       endfor
(9)   endfor
```

```
(1)   for I = 1 to N do
(3)       if T[I] > 0 then
(2)           for J = 2 to N do
(4)               A[I,J] = A[I,J-1]*T[I] + B[J]
(8)           endfor
(5)       else
(2)           for J = 2 to N do
(6)               A[I,J] = 0.0
(8)           endfor
(7)       endif
(9)   endfor
```

Divisão do Conjunto de Índices

- Dividir a viagem (*trip-count*) do laço em duas
  - Permite remover condições de dentro do laço

```
(1)   for I = 1 to 100 do
(2)       A[I] = B[I] + C[I]
(3)       if I > 10 then
(4)           D[I] = A[I] + A[I-10]
(5)       endif
(6)   endfor
```

```
(1)   for I = 1 to 10 do
(2)       A[I] = B[I] + C[I]
(6)   endfor
(1)   for I = 11 to 100 do
(2)       A[I] = B[I] + C[I]
(4)       D[I] = A[I] + A[I-10]
(6)   endfor
```

É sempre possível Paralelizar?

- Condição básicas
  - Grafo de dependência no corpo do laço é acíclico.

- Por quê?
  - Uma instrução vetorial vai operar em todos elementos de um vetor
  - O vetor não pode ser alterado enquanto lido

Grafo Cíclico

- Iteração i = 0 altera A[1] ao mesmo tempo em que a iteração i = 1 lê A[1]

```
for (i = 0; i < N; i++)
    (1) A[i+1] = A[i] + B[i];
```

A[0]	A[1]	A[2]	.....	.....	.....	A[N-2]	A[N-1]
			+				
B[0]	B[1]	B[2]	.....	.....	.....	B[N-2]	B[N-1]
			=				
A[0]	A[1]	A[2]	.....	.....	.....	A[N-2]	A[N-1]

E como resolver isto?

ERAD 2012

Perguntas?

guido@ic.unicamp.br