Detecção de Paralelismo em Laços de Arquiteturas MIMD e Multicore

Guido Araujo

WAMCA/WSCAD 2011

Vitória, ES

LSC COMPUTER SYSTEMS LABORATORY

IC UNICAMP

This WAMCA/WSCAD Session

- 8:30 – 10:00 (this talk in Portuguese)
  - Loop Parallelism Techniques for VLIW and Multicore Architectures

    Guido Araujo, UNICAMP

- 10:00 – 10:30
  - Large Scale Kronecker Product on Supercomputers

    Claude Tadonki, University of Orsay

Roteiro

- Arquiteturas Paralelas
- Paralelismo em MIMD
- Paralelismo em Multicores

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]

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] = E[i] + 1;
    G[i] = F[i] – 1;
}
```

C[i] = A[i] + B[i]	**E[i] = D[i] << 2**	**……**	**……**
F[i] = E[i] + C[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;
}
```

Entendendo Dependências

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

Roteiro

- Arquiteturas Paralelas
- Paralelismo em MIMD
- Paralelismo em Multicores

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

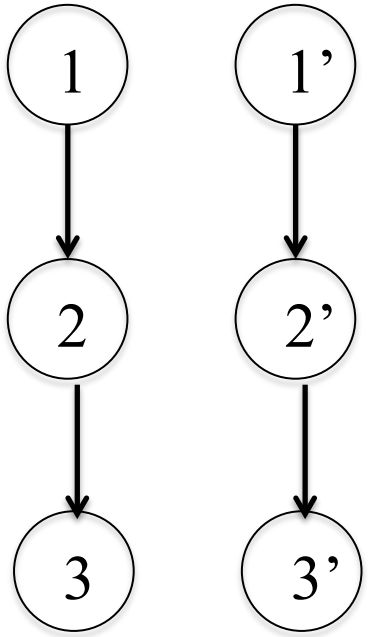
Como aumentar paralelismo?

- 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

i = 2



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;
}
```

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

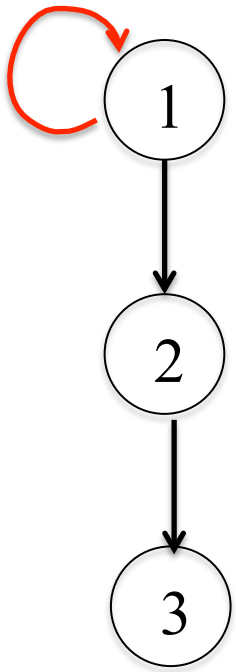
$i = 2$ { (1'), (2'), (3')

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;
}
```

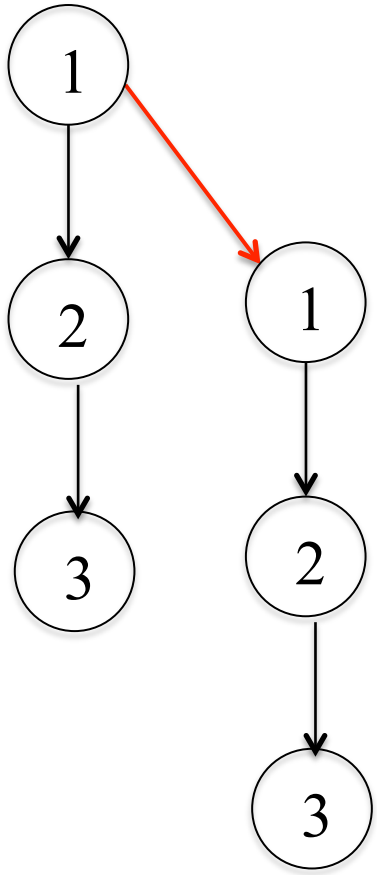


Time:

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;
}
```

Speedup: 3N/

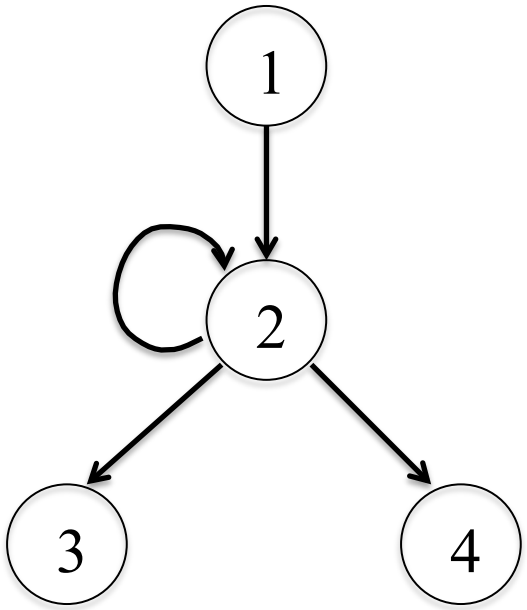
A[i] = A[i-1] + 1	**NOP**	**……**	**……**
B[i] = A[i] << 2	**A[i+1] = A[i] + 1**	**……**	**……**
C[i] = B[i] - 1	**B[i+1] = A[i+1] << 2**	**……**	**……**
NOP	**C[i+1] = B[i+1] - 1**	**……**	**……**

Uau!!

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

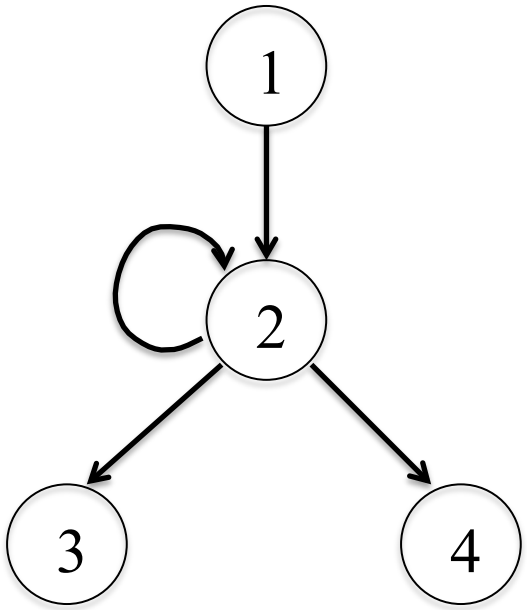
O quanto devemos desenrolar?

- Vamos ver….



O quanto devemos desenrolar?

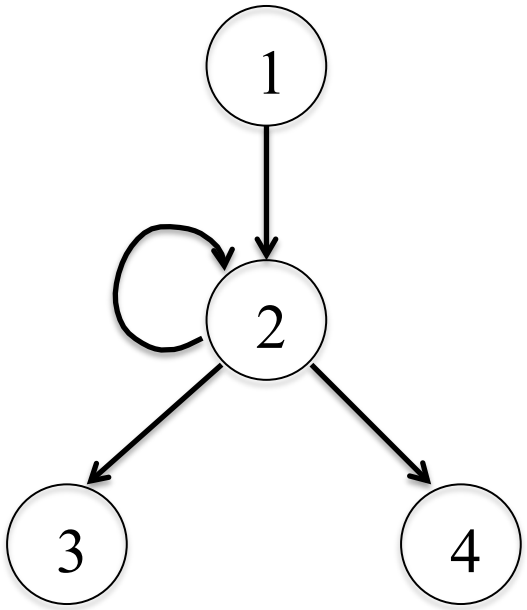
- Agora a 2a iteração



1						
2		1				
3	4					

O quanto devemos desenrolar?

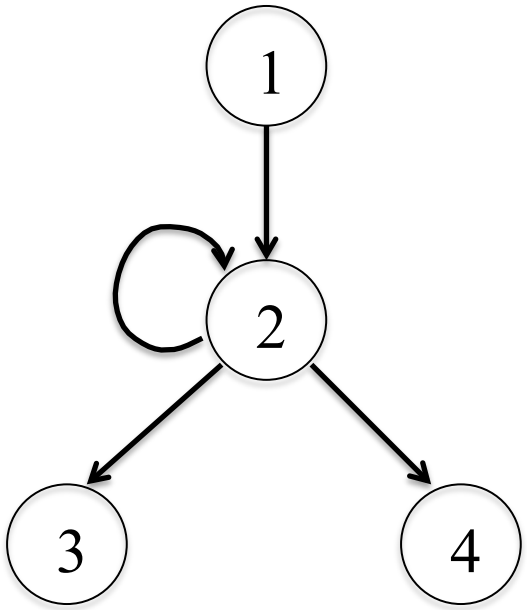
- Agora a 3a iteração



1						
2		1				
3	4	2		1		
		3	4			

O quanto devemos desenrolar?

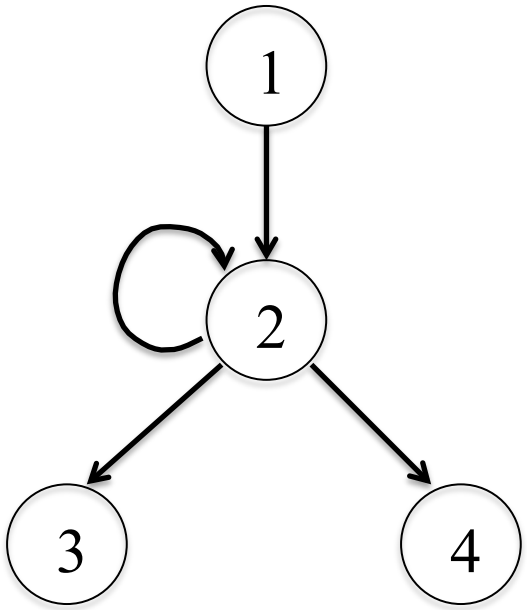
- Agora a 4a iteração



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

O quanto devemos desenrolar?

- Assim por diante….

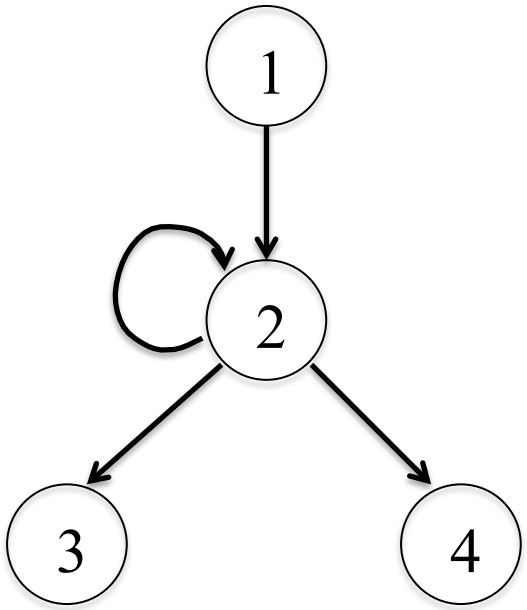


1						
2		1				
3	4	2		1		
		3	4	2		1
				3	4	2
….	….	….	….	….	….	….

Ops, parece que está aparecendo algo!!

O que será que está aparecendo?

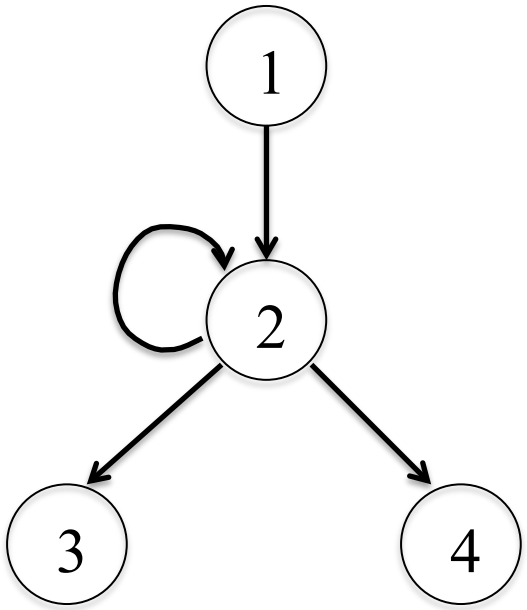
- Um padrão se repete!



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

E o que ocorre no final do laço?

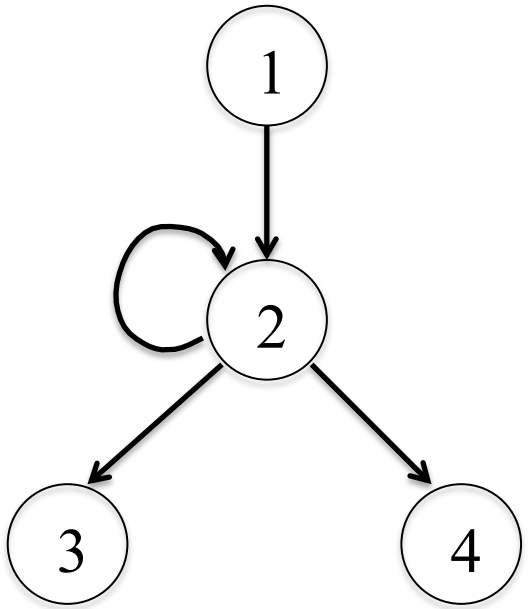
- Prólogo e epílogo



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?

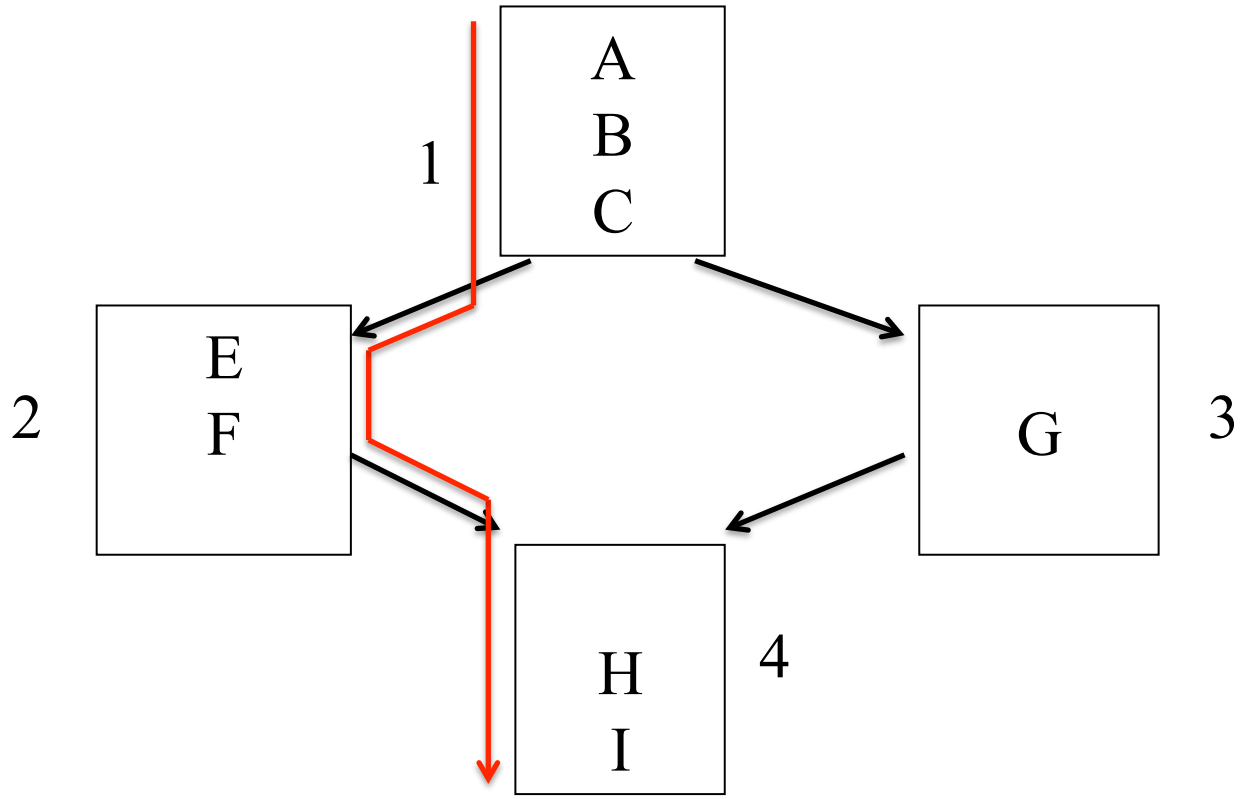
E se o laço contiver condições?

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

- Considere em 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

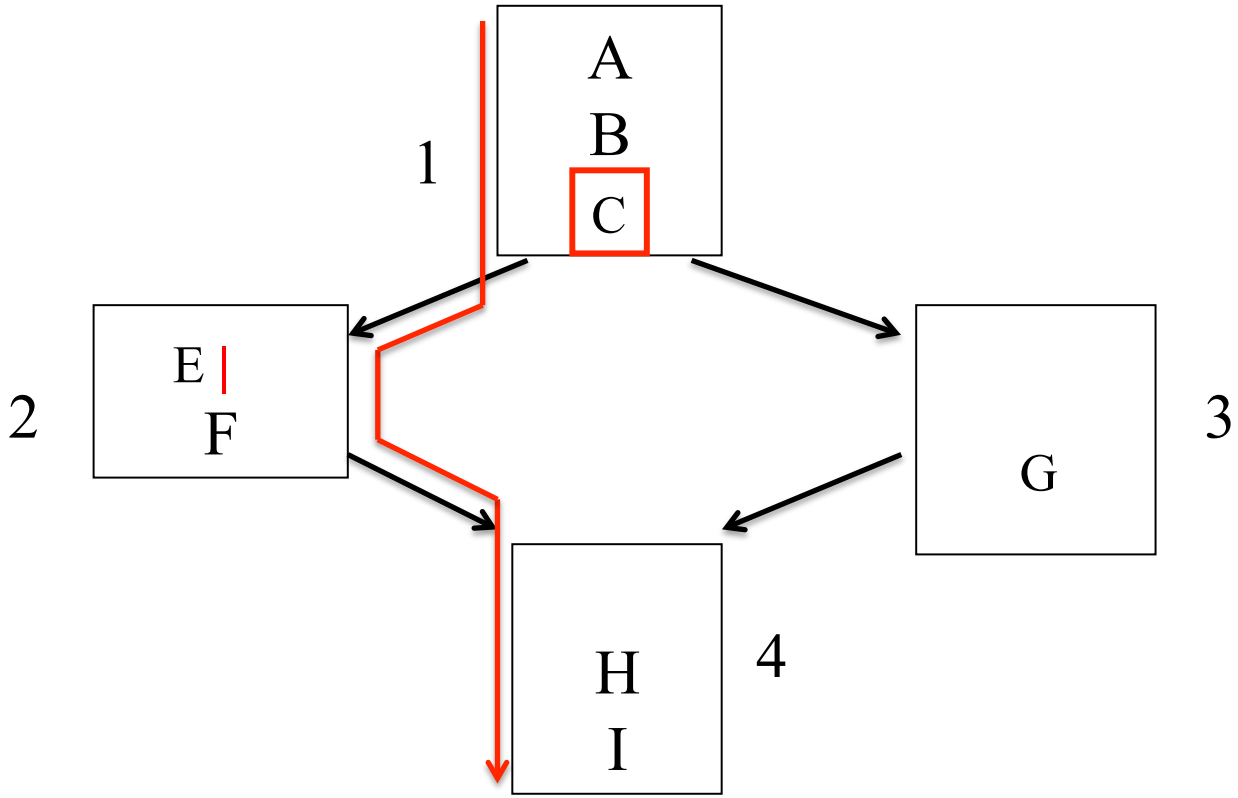
Escalonamento de Traços

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

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

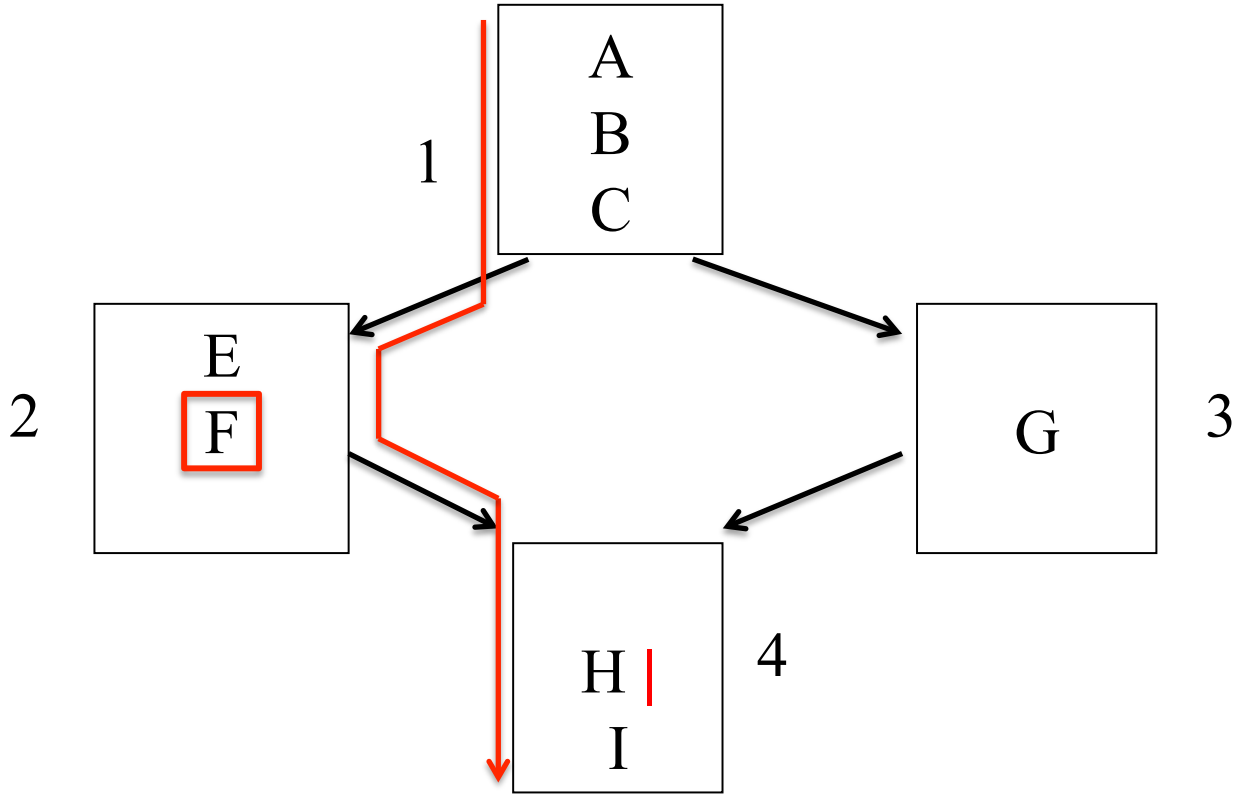
E se sair to traço?

- Paralelizando para baixo (1)



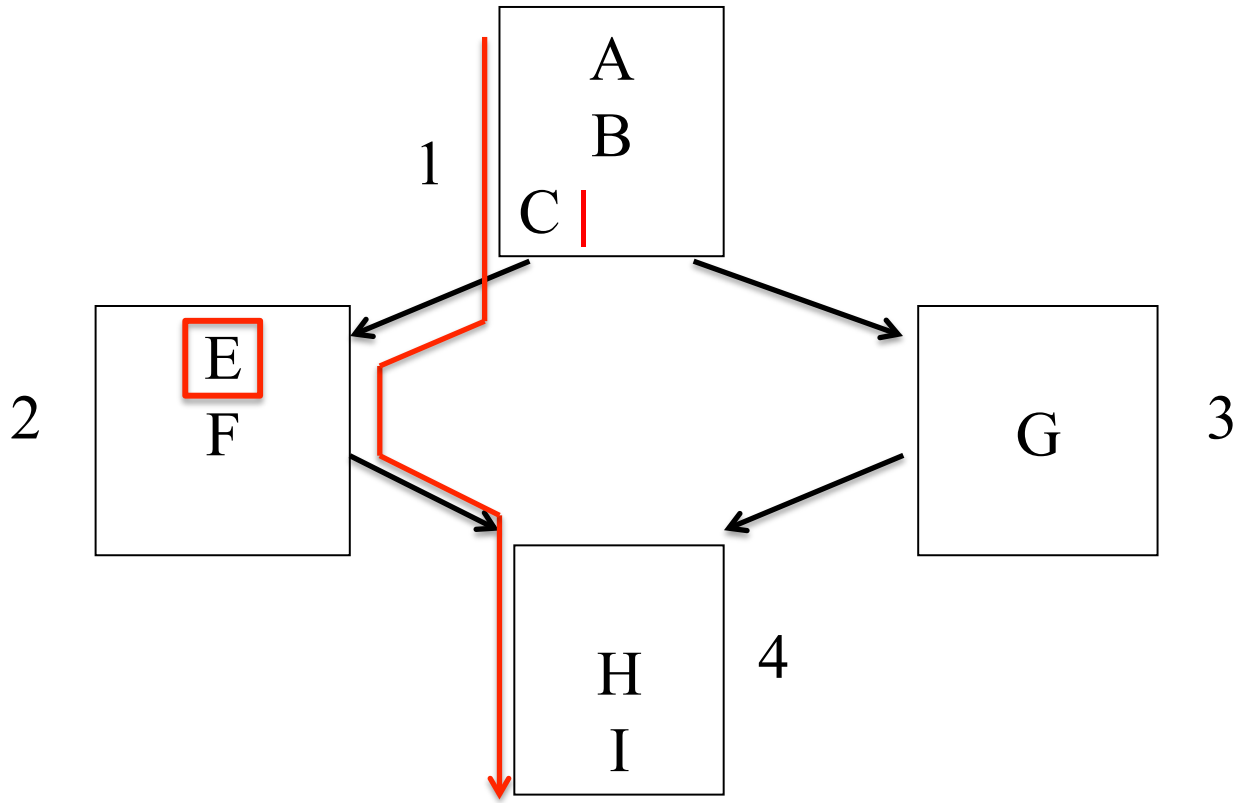
E se sair to traço?

- Paralelizando para baixo (2)



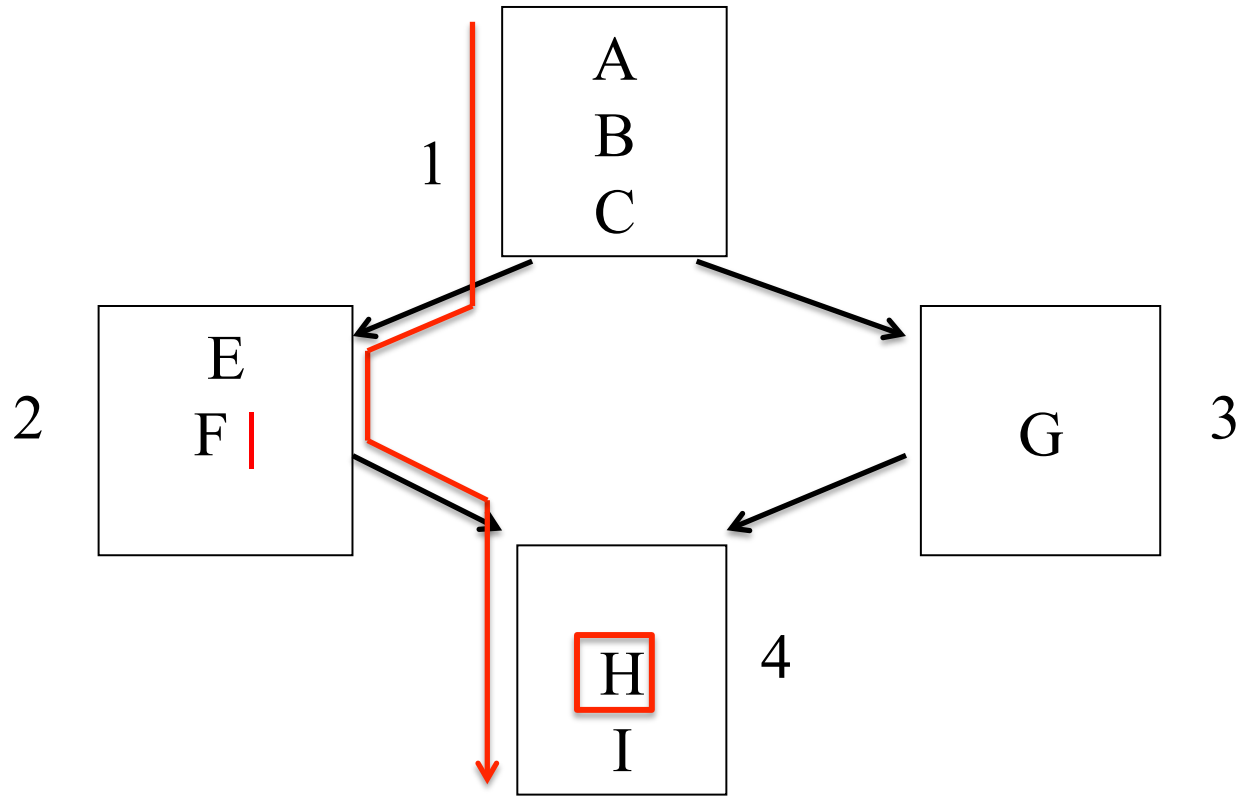
E se sair to traço?

- Paralelizando para cima (1)

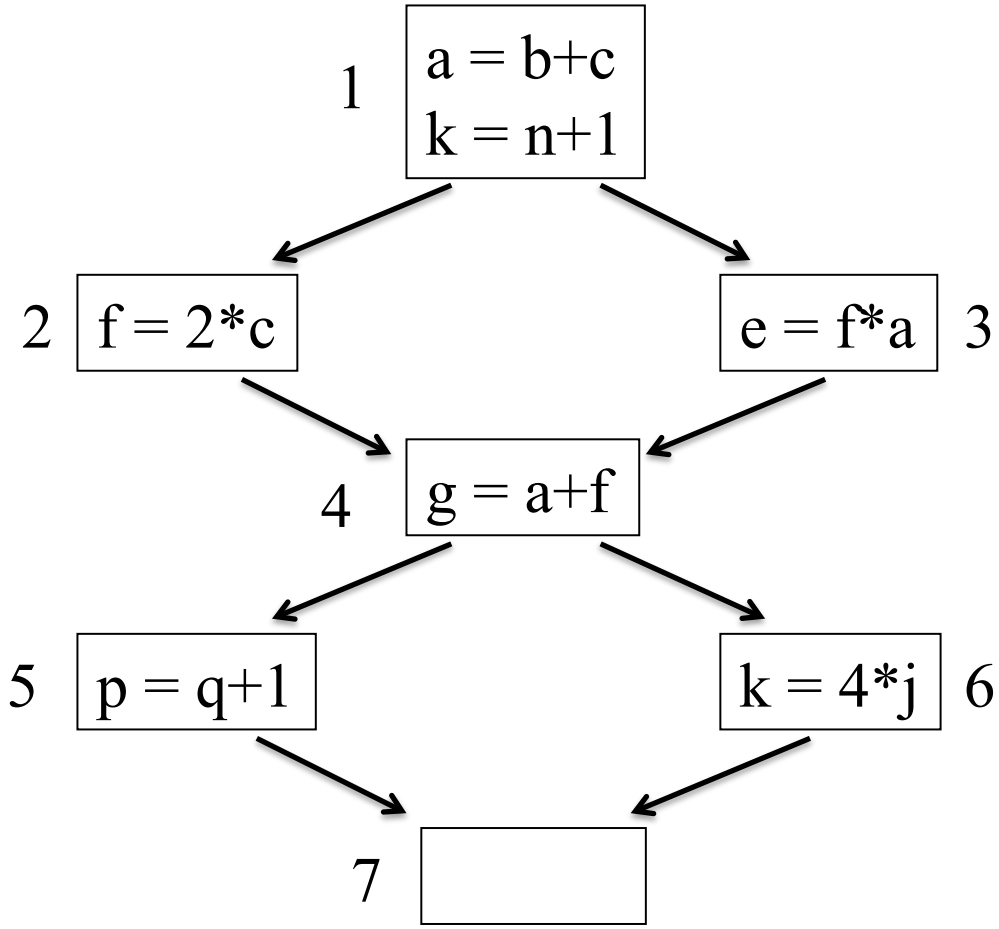


E se sair to traço?

- Paralelizando para cima (2)

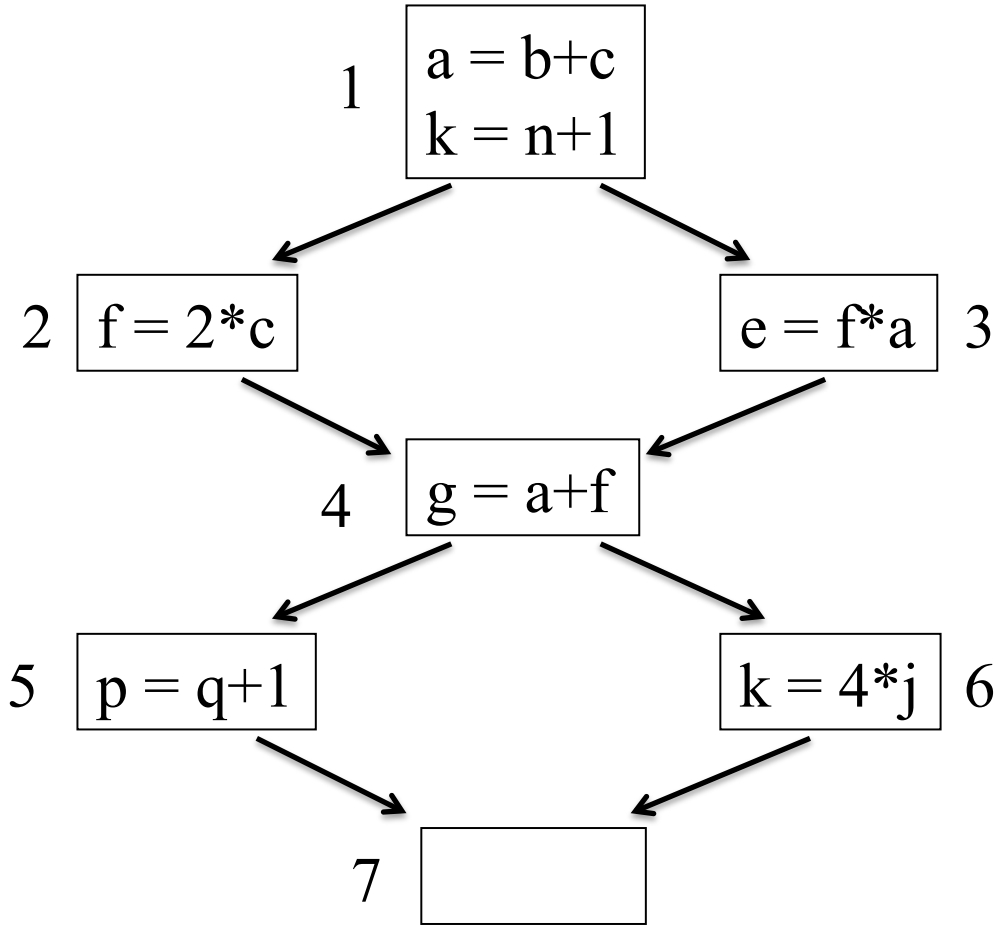


Fica mais claro com um exemplo…



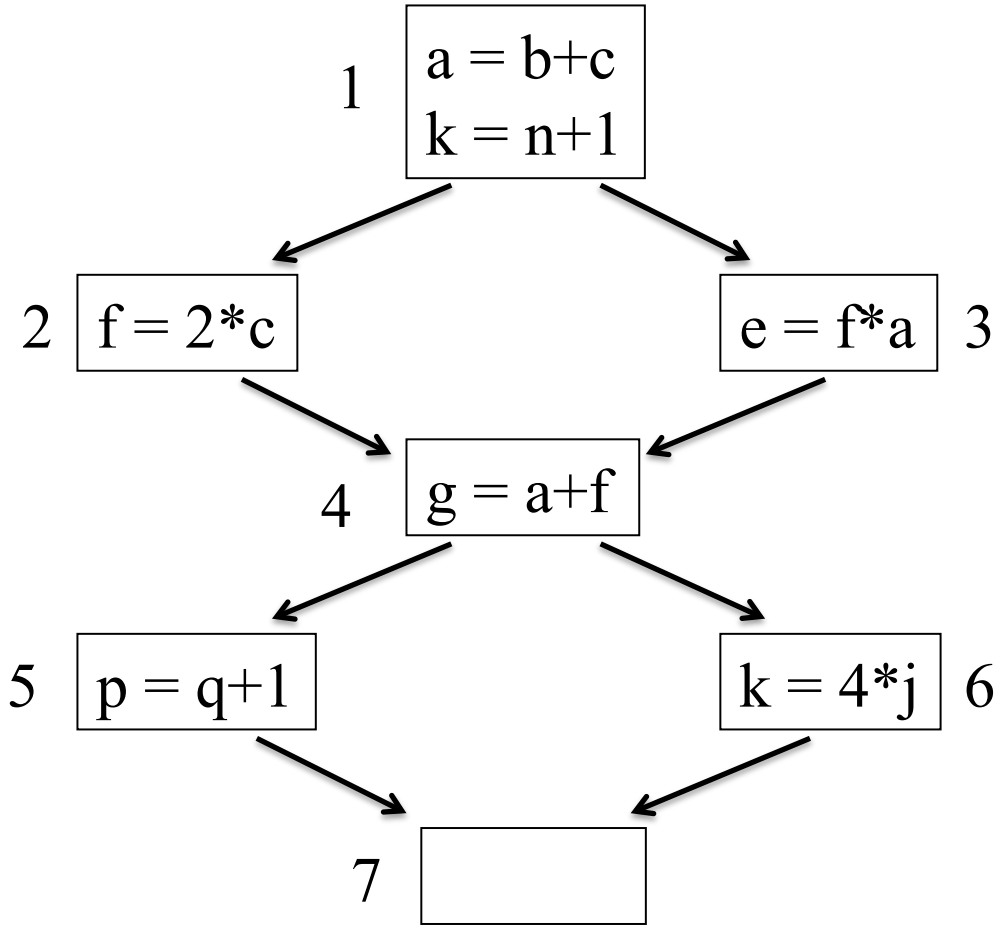
Quantos caminhos (*traços*)?

Traço A



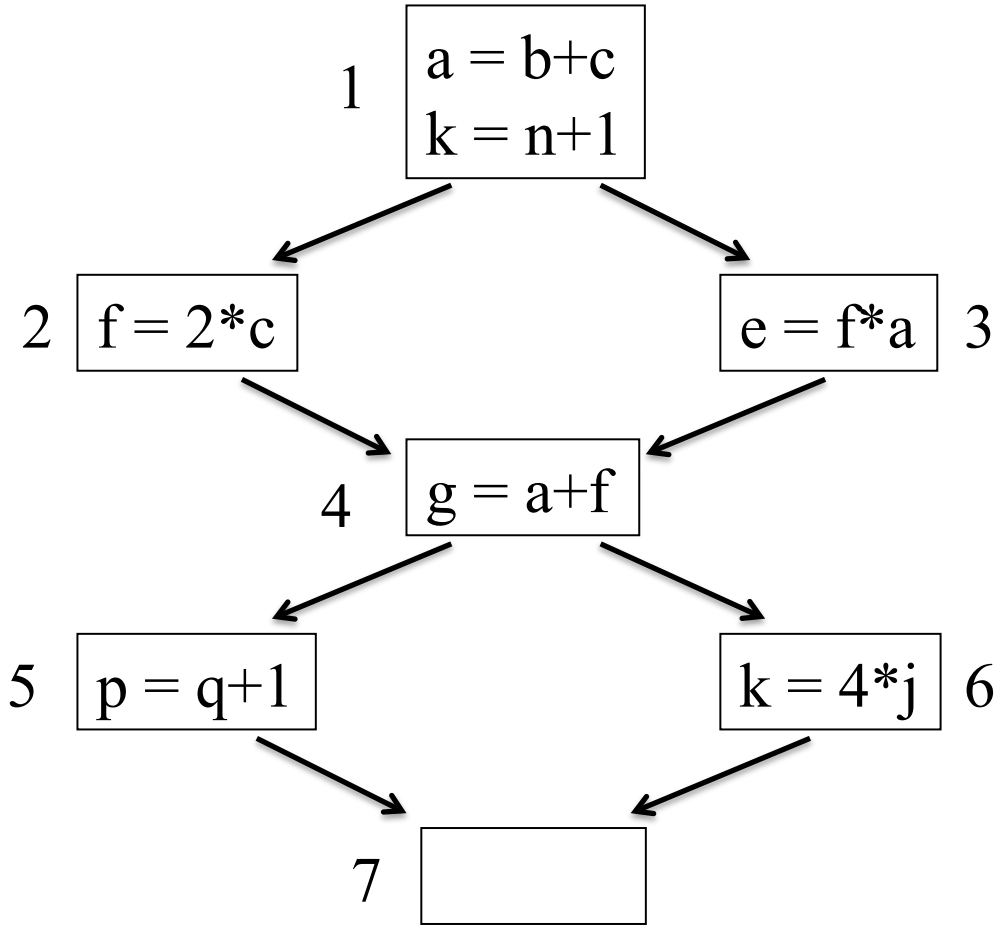
Traço A: 1, 2, 4, 6 e 7 Prob(A): 91% Tempo(A):

Traço B



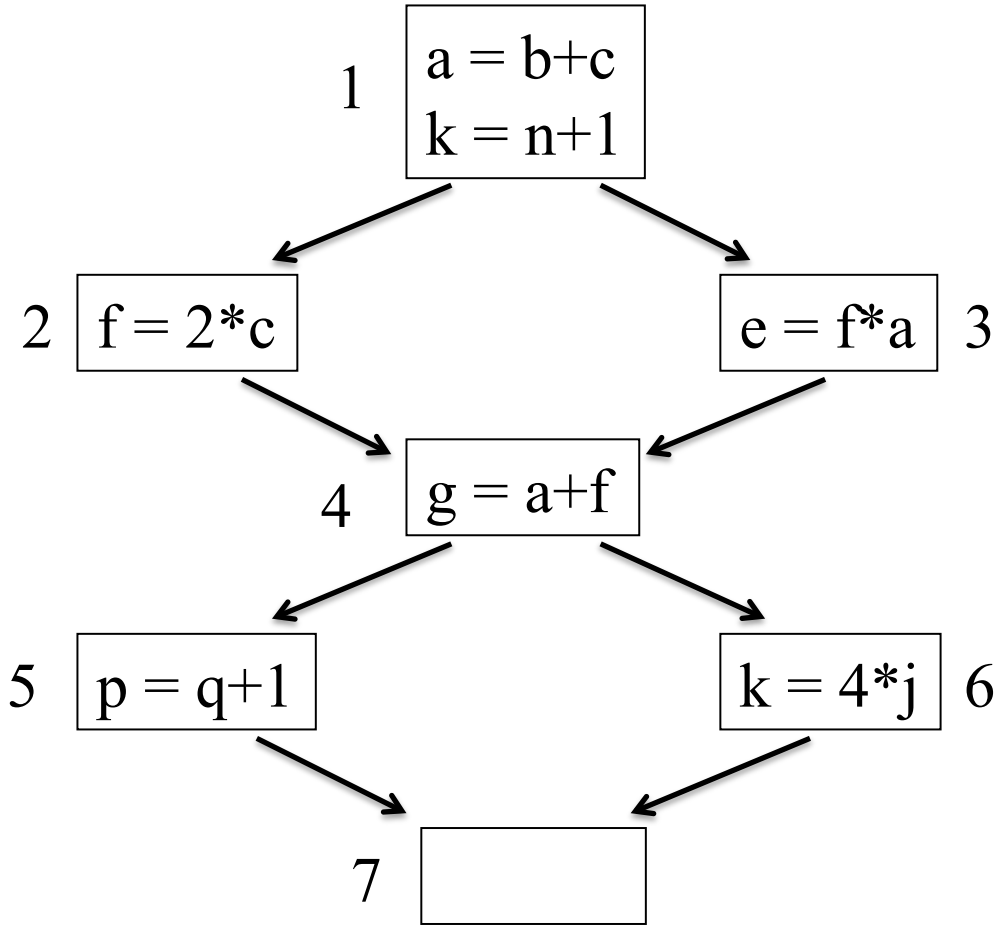
Traço B: 1, 3, 4, 5 e 7 Prob(B): 3% Tempo(B):

Traço C



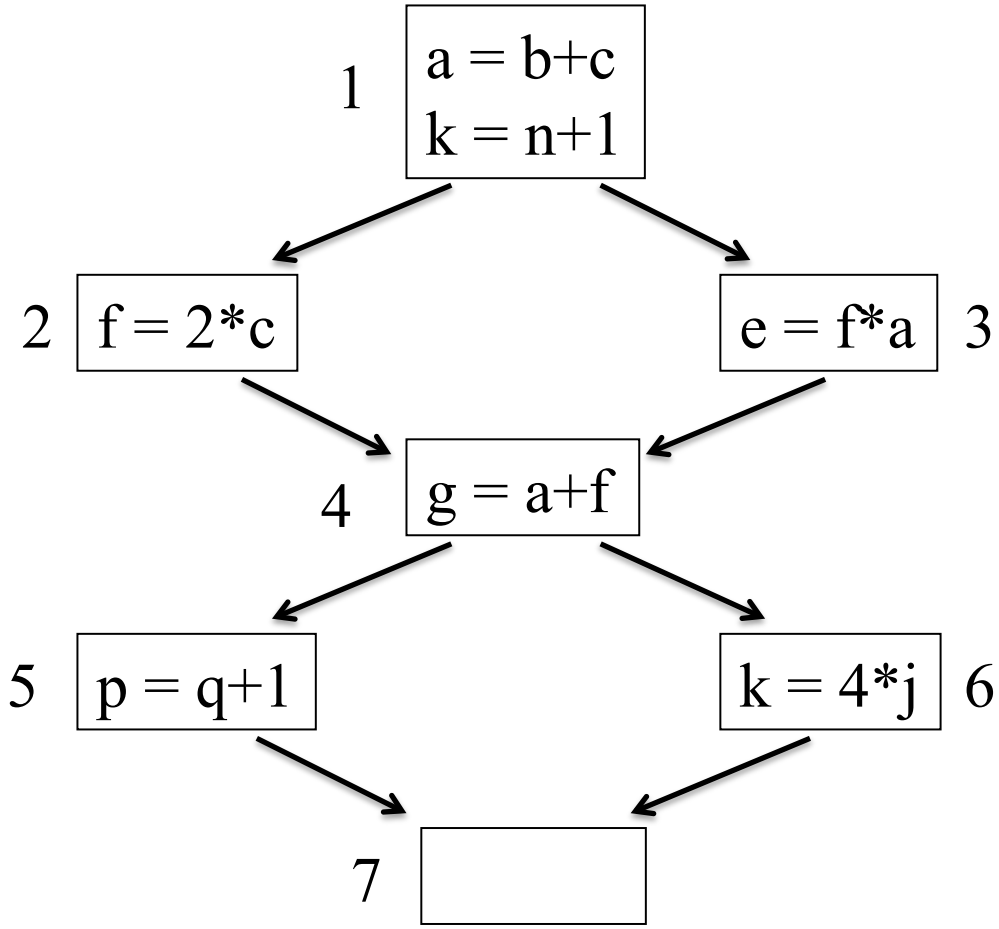
Traço C: 1, 2, 4, 5 e 7 Prob(C): 3% Tempo(C):

Traço D



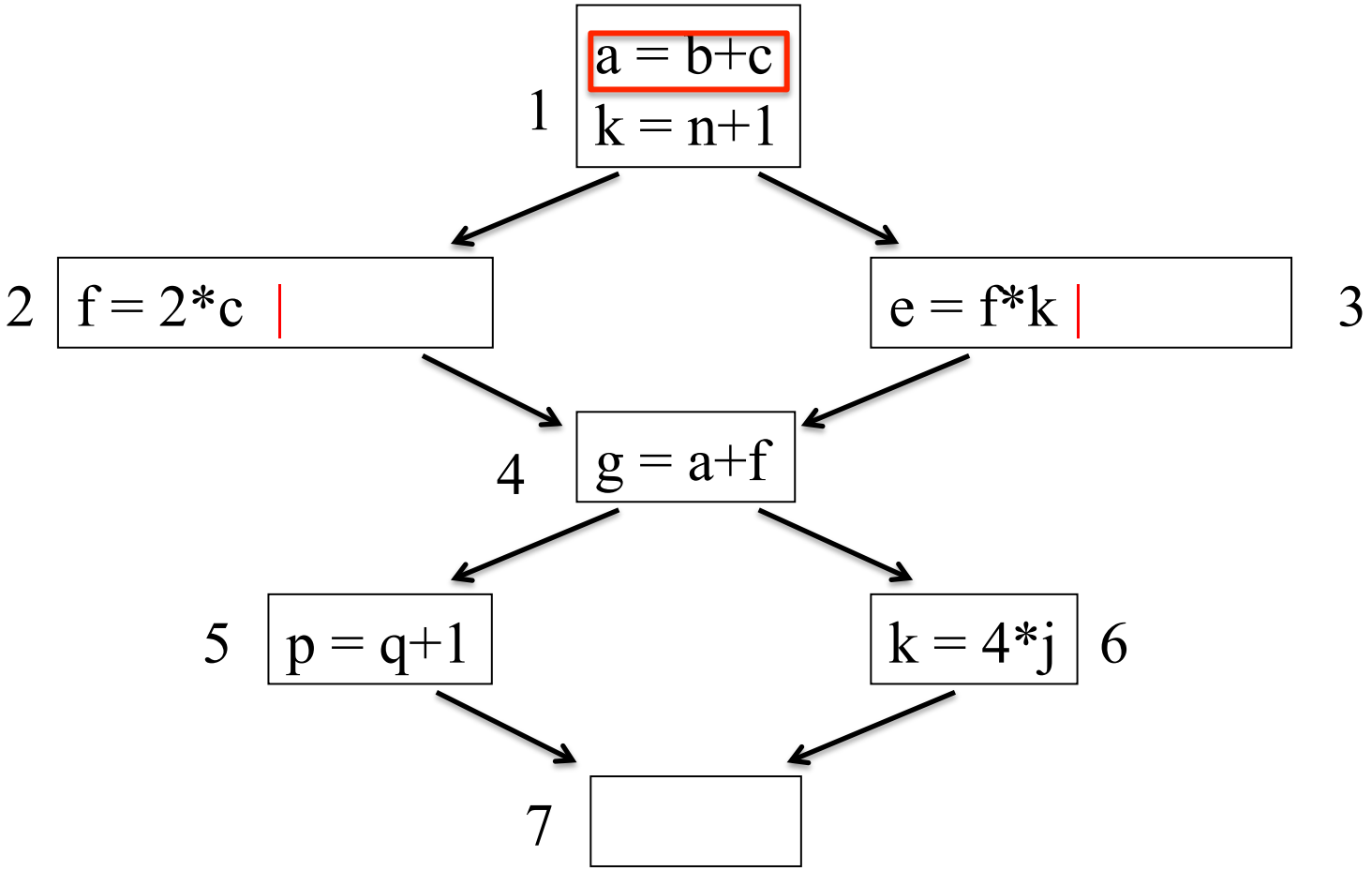
Traço D: 1, 3, 4, 6 e 7 Prob(D): 3% Tempo(D):

Tempo Total

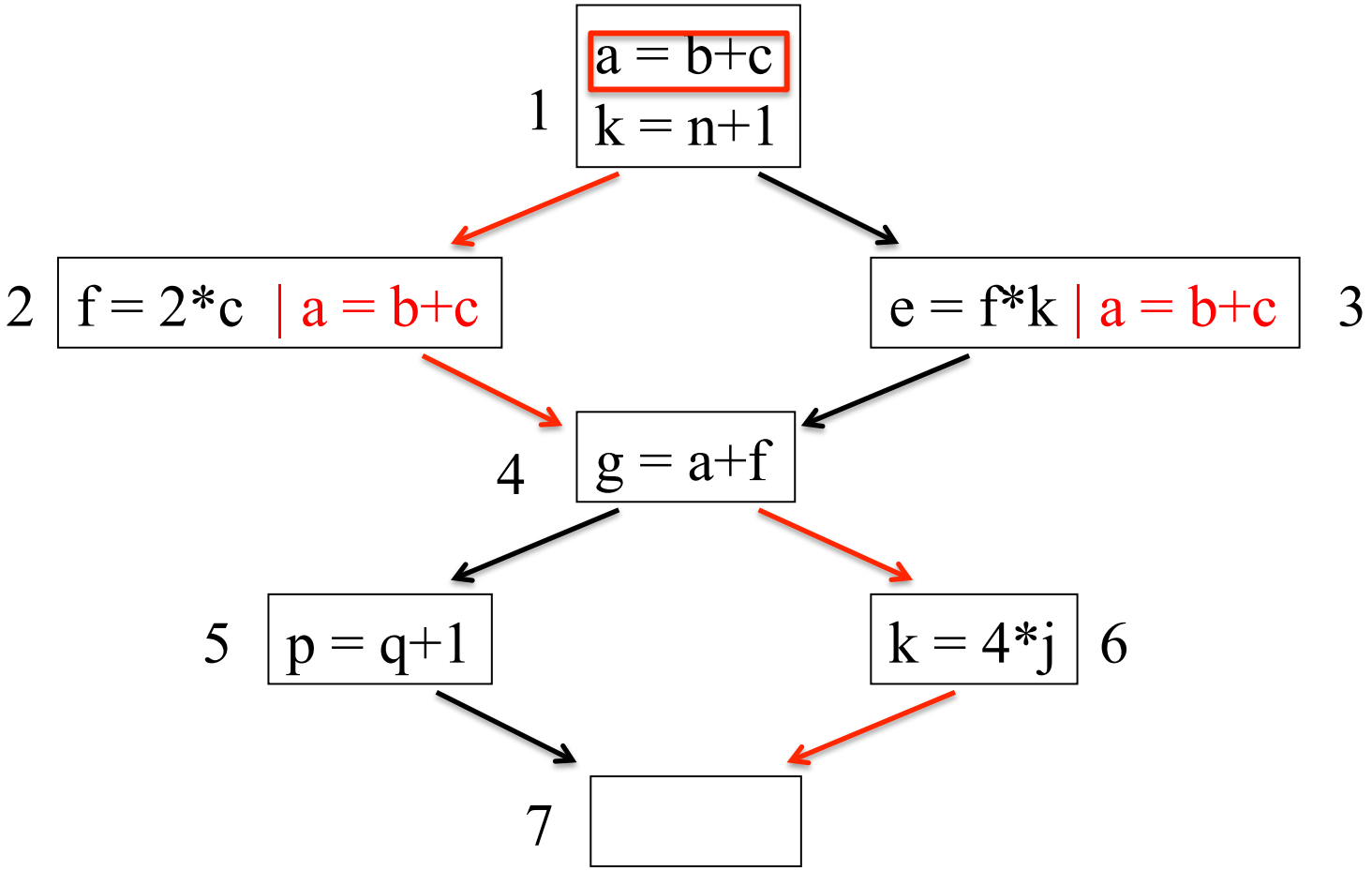


Prob(Total): Time(Total):

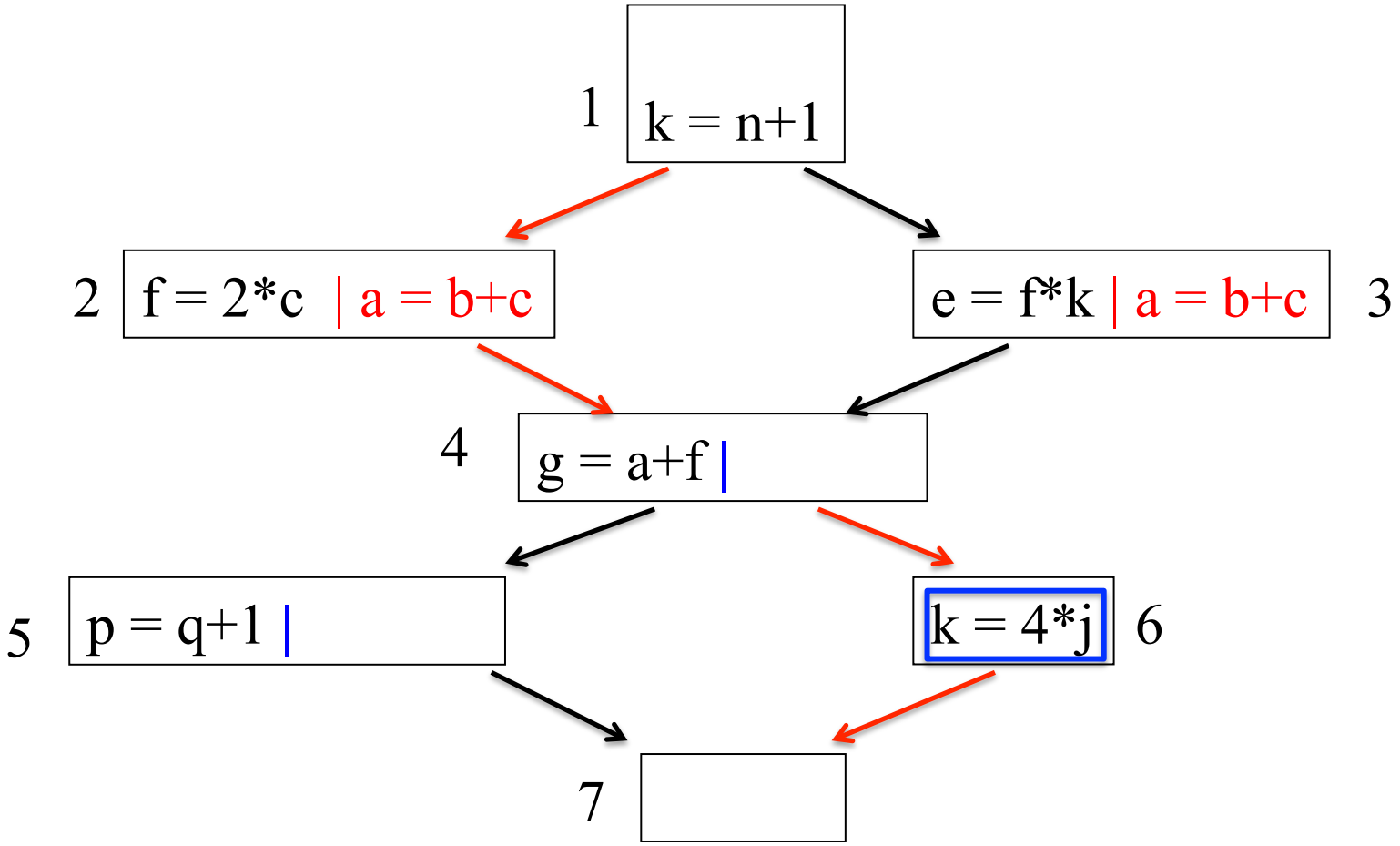
Movendo para Baixo



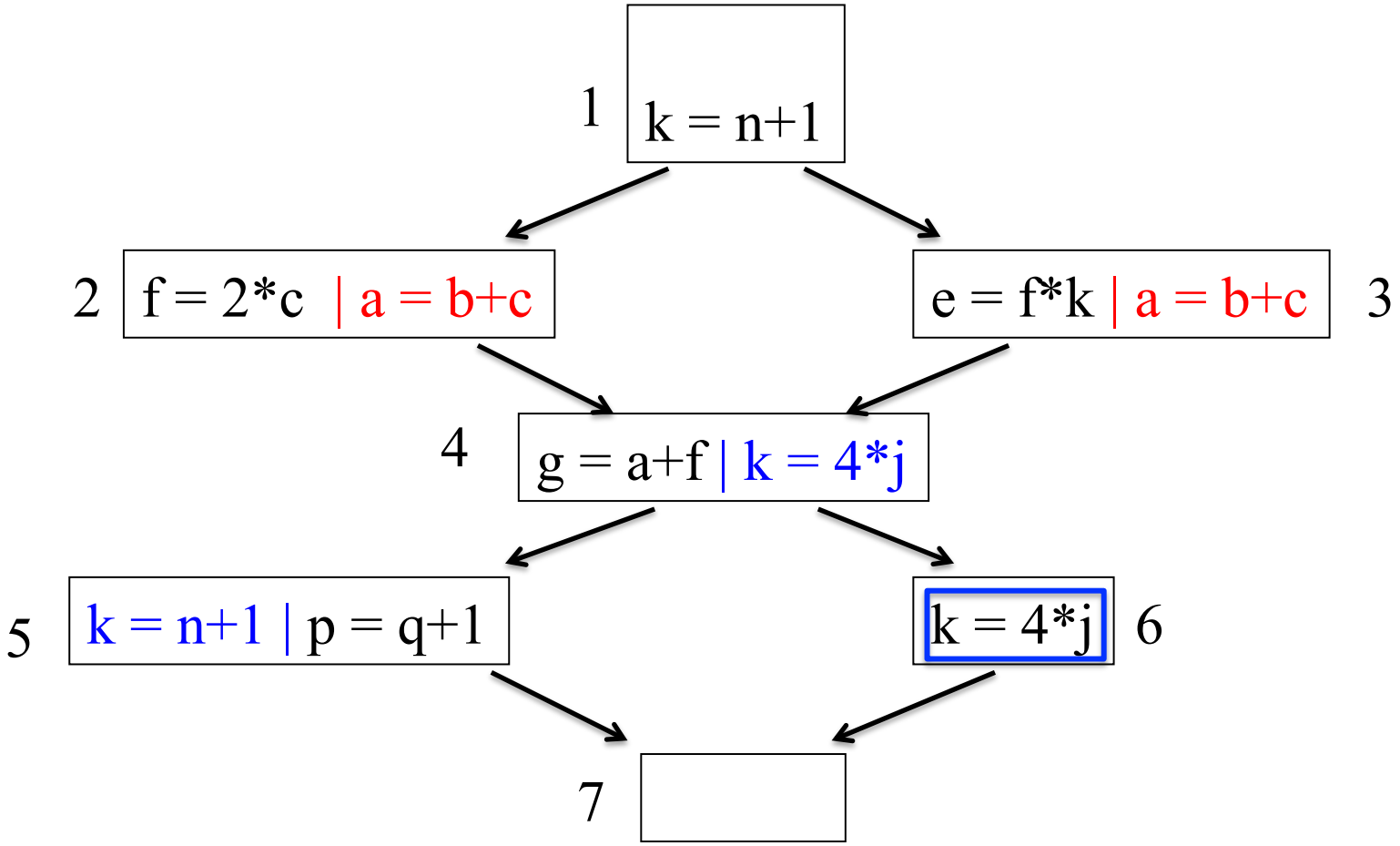
Movendo para baixo



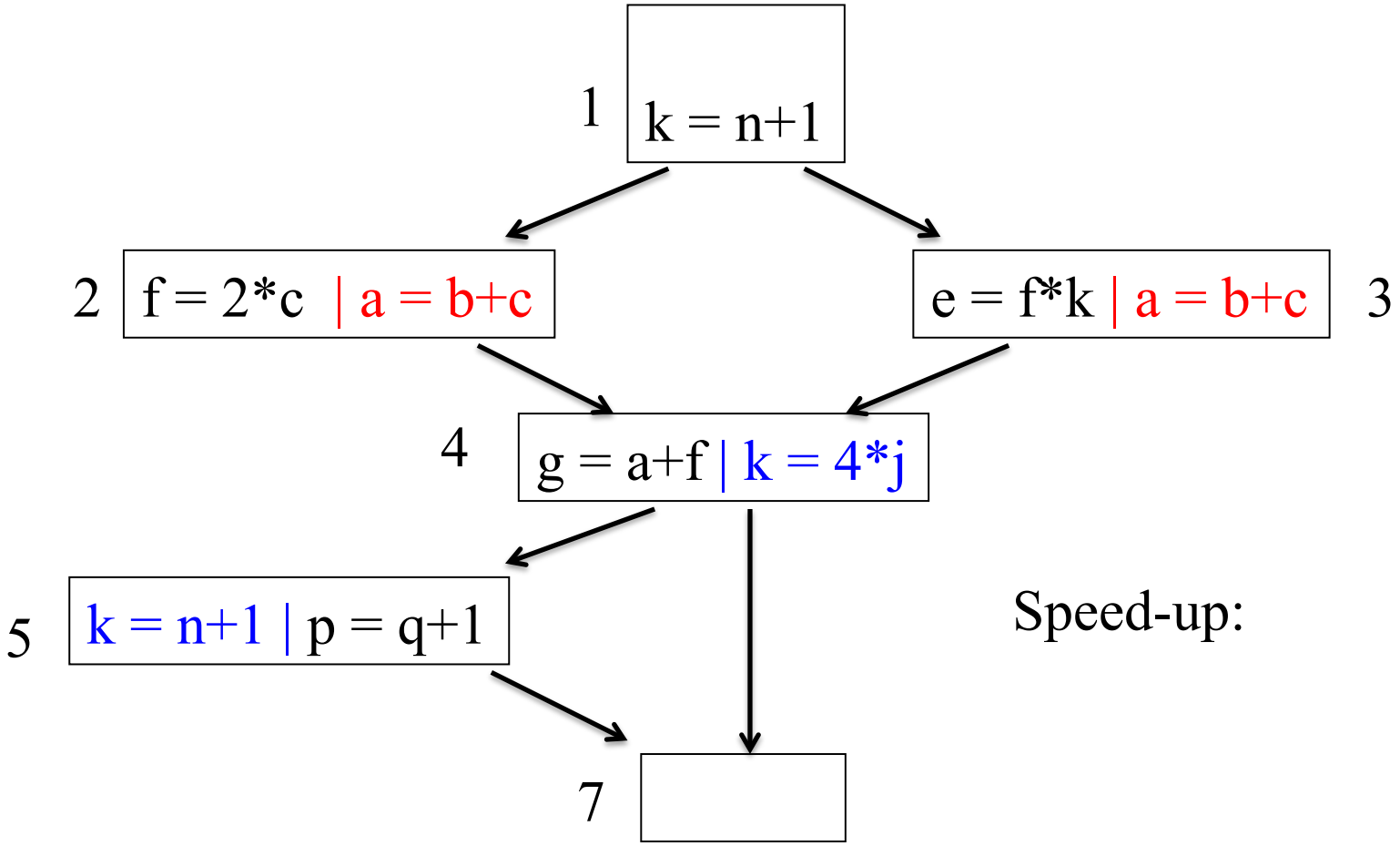
Movendo para cima



Movendo para cima



Qual será o novo tempo?



Speed-up:

Tempo(Novo):

Roteiro

- Arquiteturas Paralelas
- Paralelismo em MIMD
- Paralelismo em Multicores

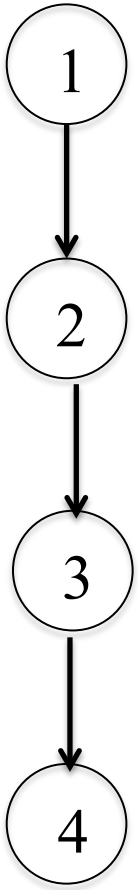
Paralelimo em Arquiteturas MIMT

- Doall
- Doacross
- Software pipelining
- Decoupled Software Pipelining

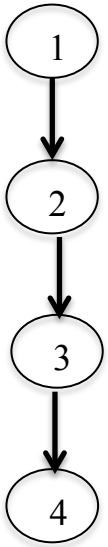
Doall

- Não existe ciclo de dependência
  - Podemos alocar um conjunto de iterações em cada núcleo

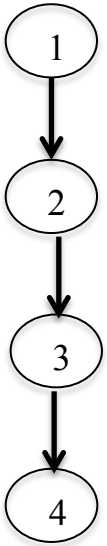
```
for (i = 0; i <= N; i++) {
   (1) C[i] = A[i] + B[i];
   (2) D[i] = C[i] << 2;
   (3) E[i] = D[i] + 1;
   (4) G[i] = C[i] - 1;
 }
```



Doall



Core 1



Core 2

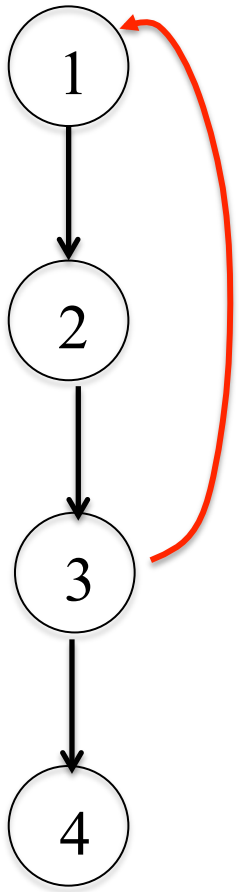
```
for (i = 0; i < N/2; i++) {
   (1) C[i] = A[i] + B[i];
   (2) D[i] = C[i] << 2;
   (3) E[i] = D[i] + 1;
   (4) G[i] = C[i] - 1;
 }
```

```
for (i = N/2; 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] = E[i] - 1;
 }
```

Doacross

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

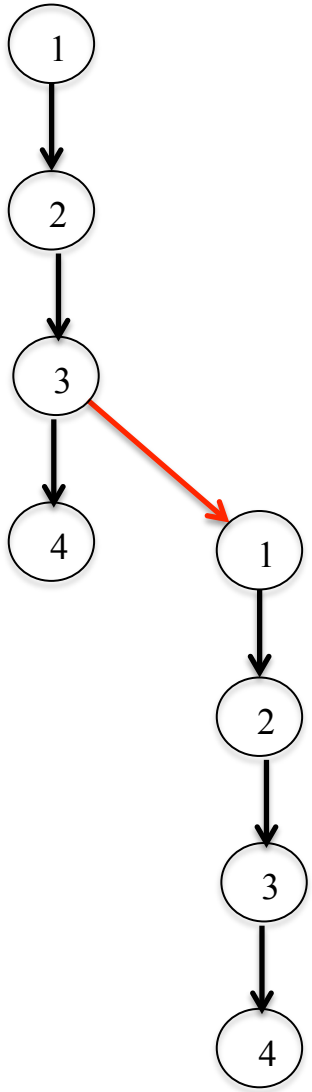
```
for (i = 0; i < N; i++) {
   (1) C[i] = A[i] + B[i];
   (2) D[i] = C[i] << 2;
   (3) A[i+1] = D[i] + 1;
   (4) E[i] = A[i+1] - 1;
 }
```



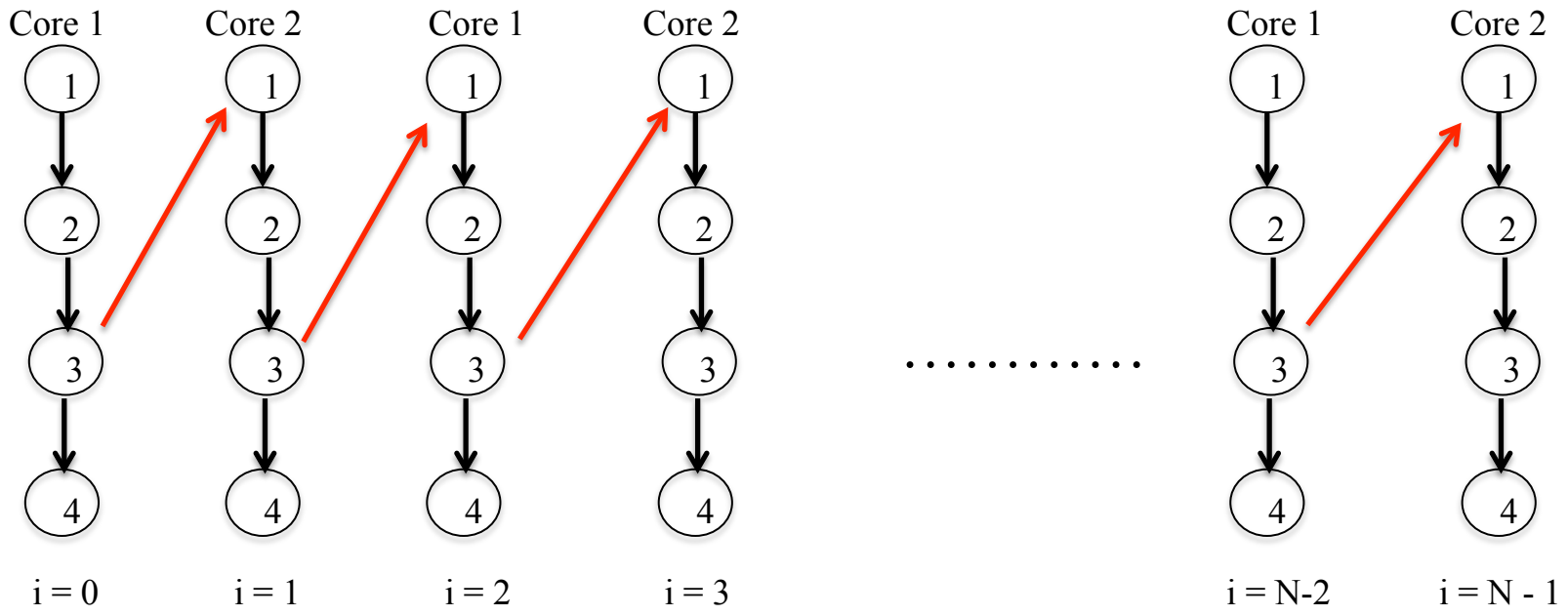
Doacross

- E se dependência “loop-carried”?
  - Replicar o laço em núcleo 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] = BIG(A[i+1]);
 }
```

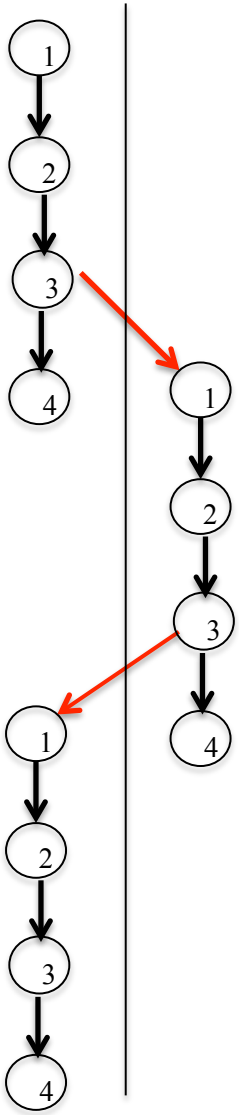


Doacross (Software Pipelining)



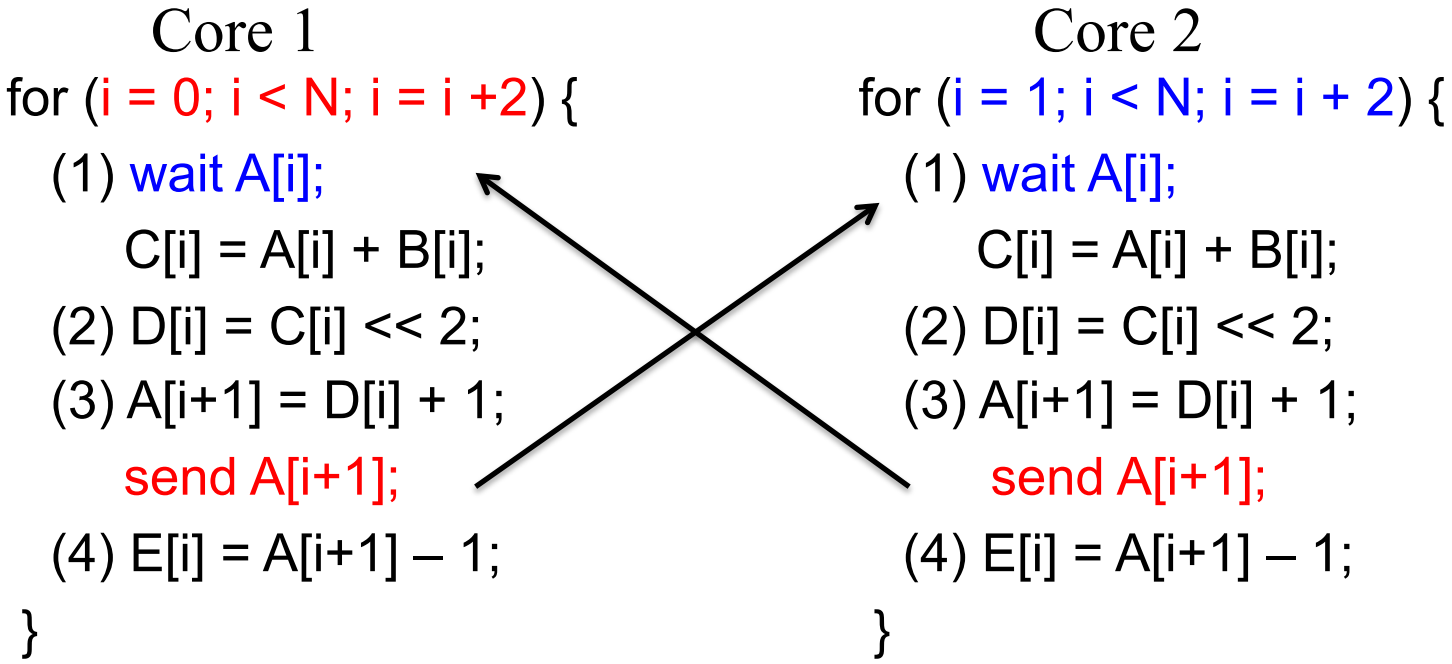
Software Pipelinig

- Paralelismo
  - 4 (em i) 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



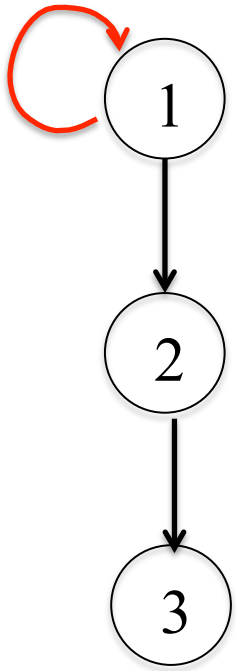
Doacross

- 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.



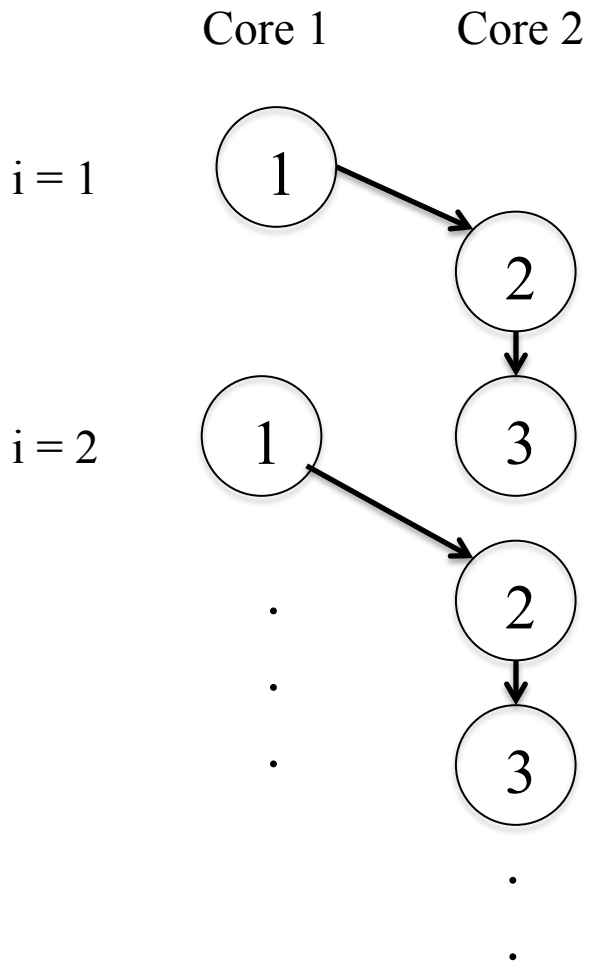
Dividindo o Laço em Dois

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



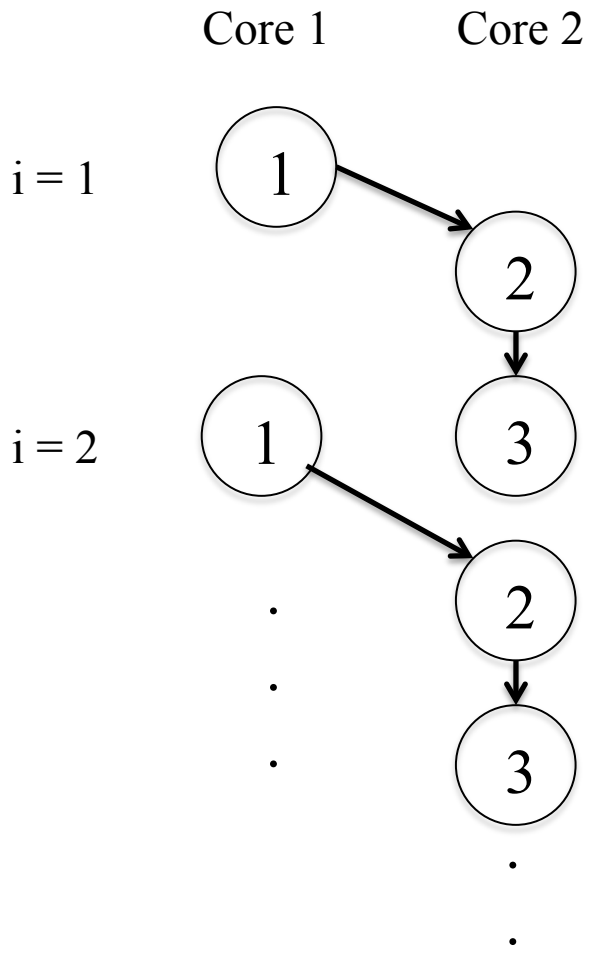
Como ficaria com Software Pipelining?

- Speed-up:
  - 3N/(2N+1) ~ 50%
- Comunicação
  - Via memória ou
  - Via hardware



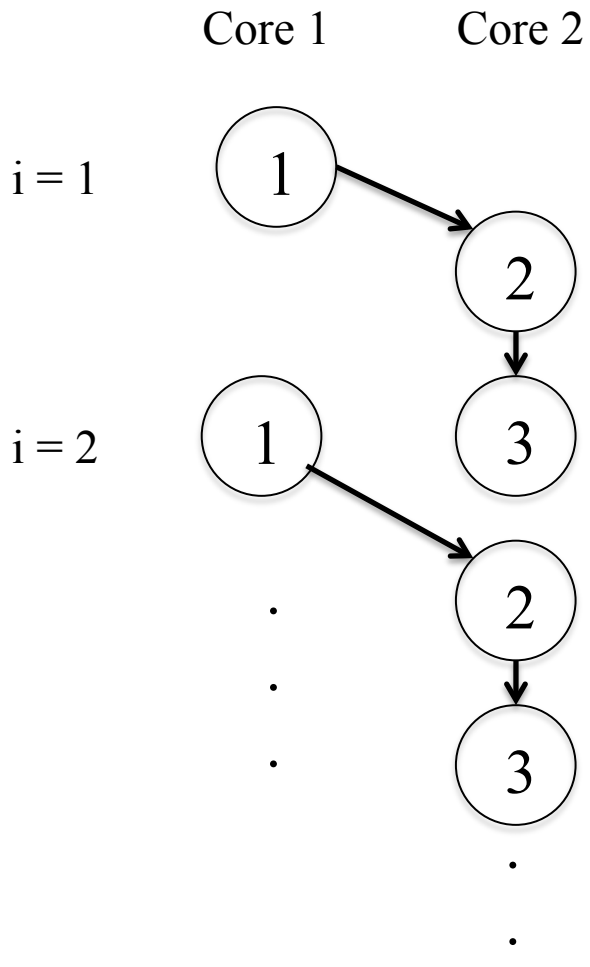
Será que dá para melhorar?

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



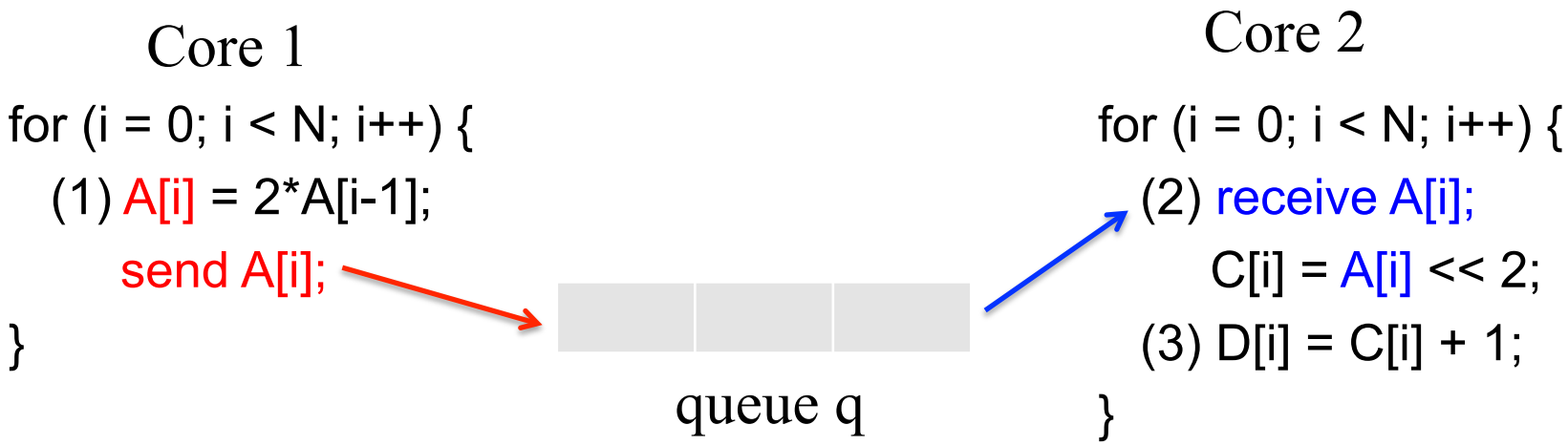
Será que dá para melhorar?

- 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?

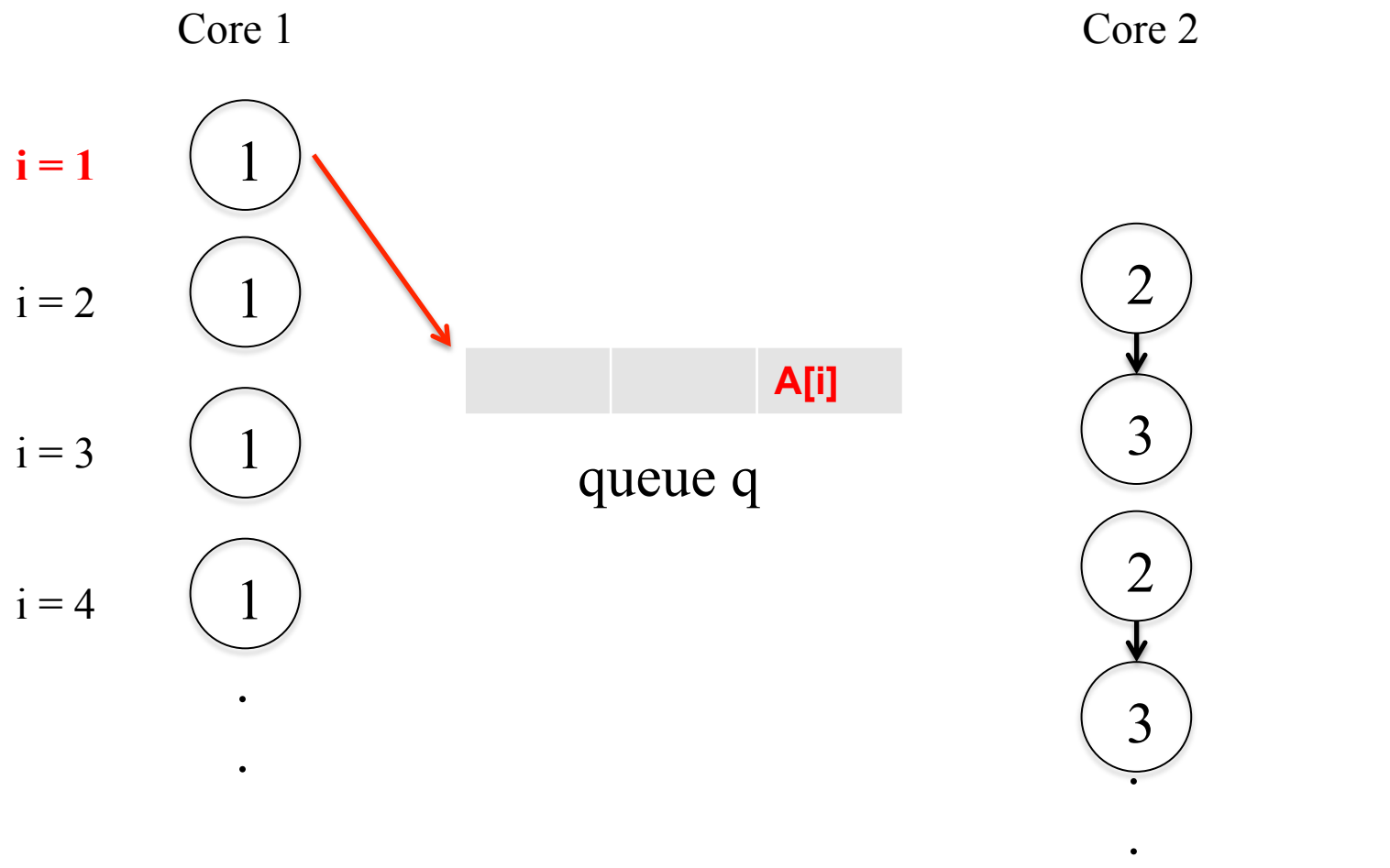


Decoupled Software Pipelining (DSWP)

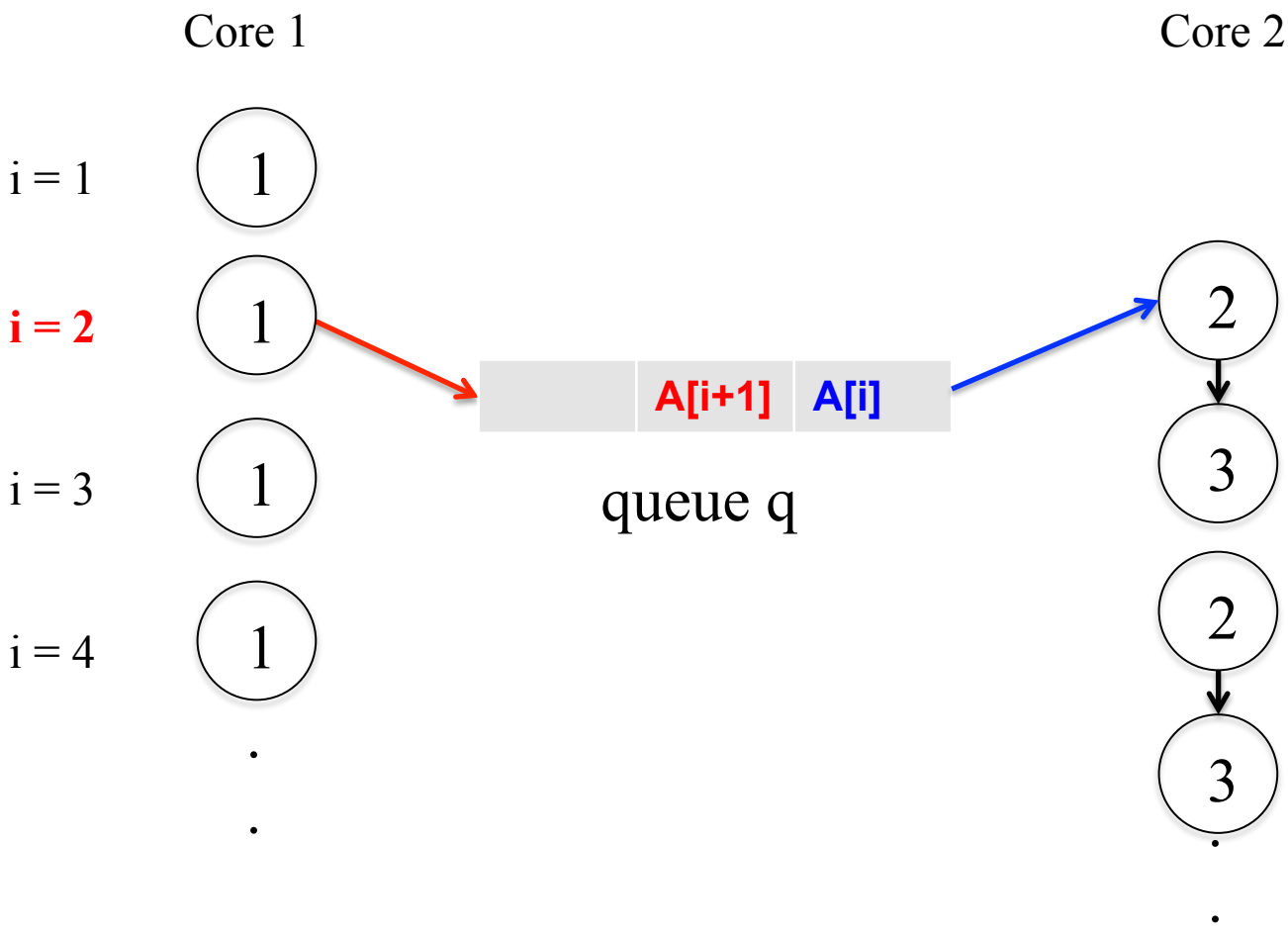
- Usando uma fila para comunicar dados
  - A[i] calculado no core 1, enviado para core 2
  - Fila desacopla execução de ambos cores!



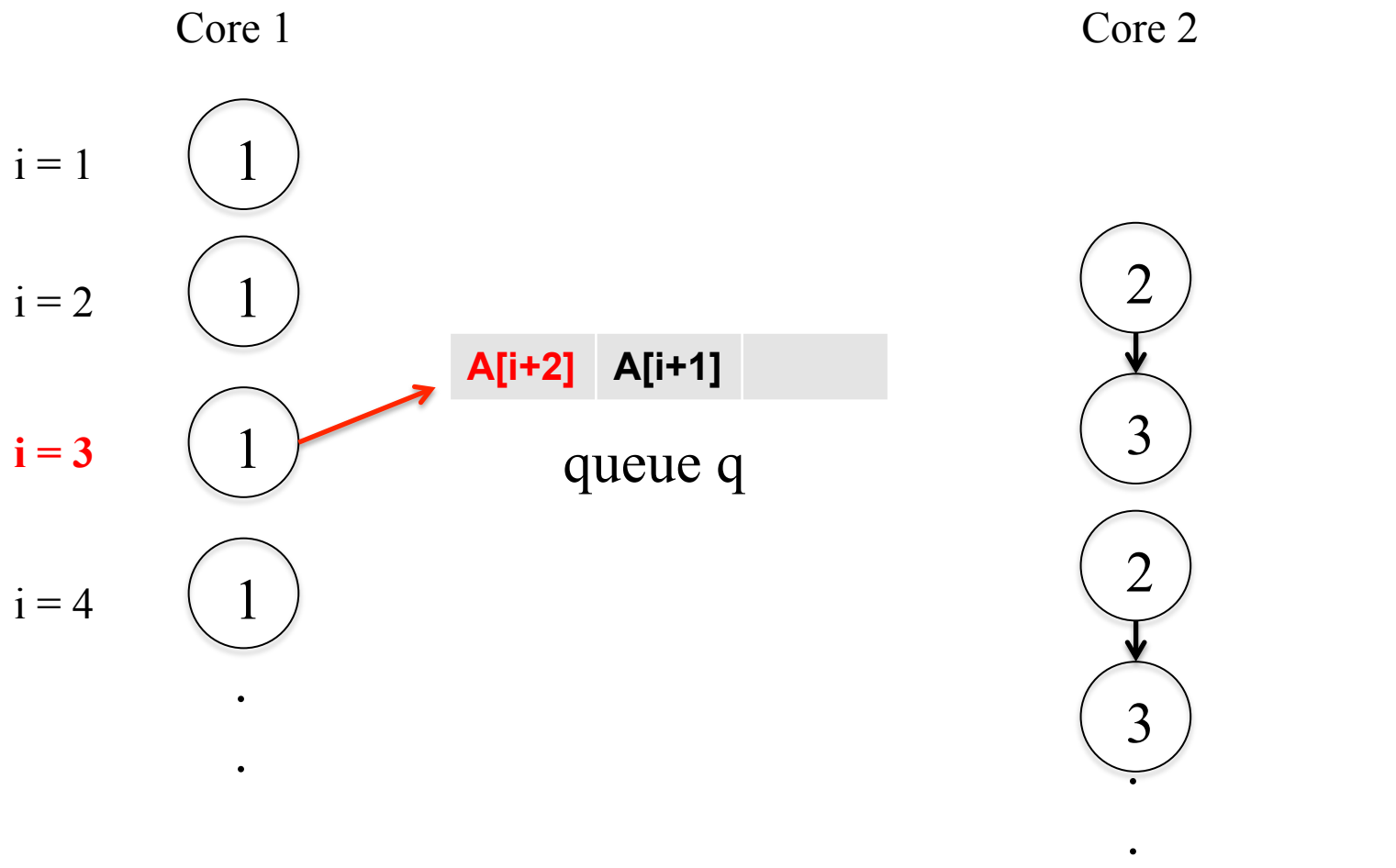
Decoupled Software Pipelining (DSWP)



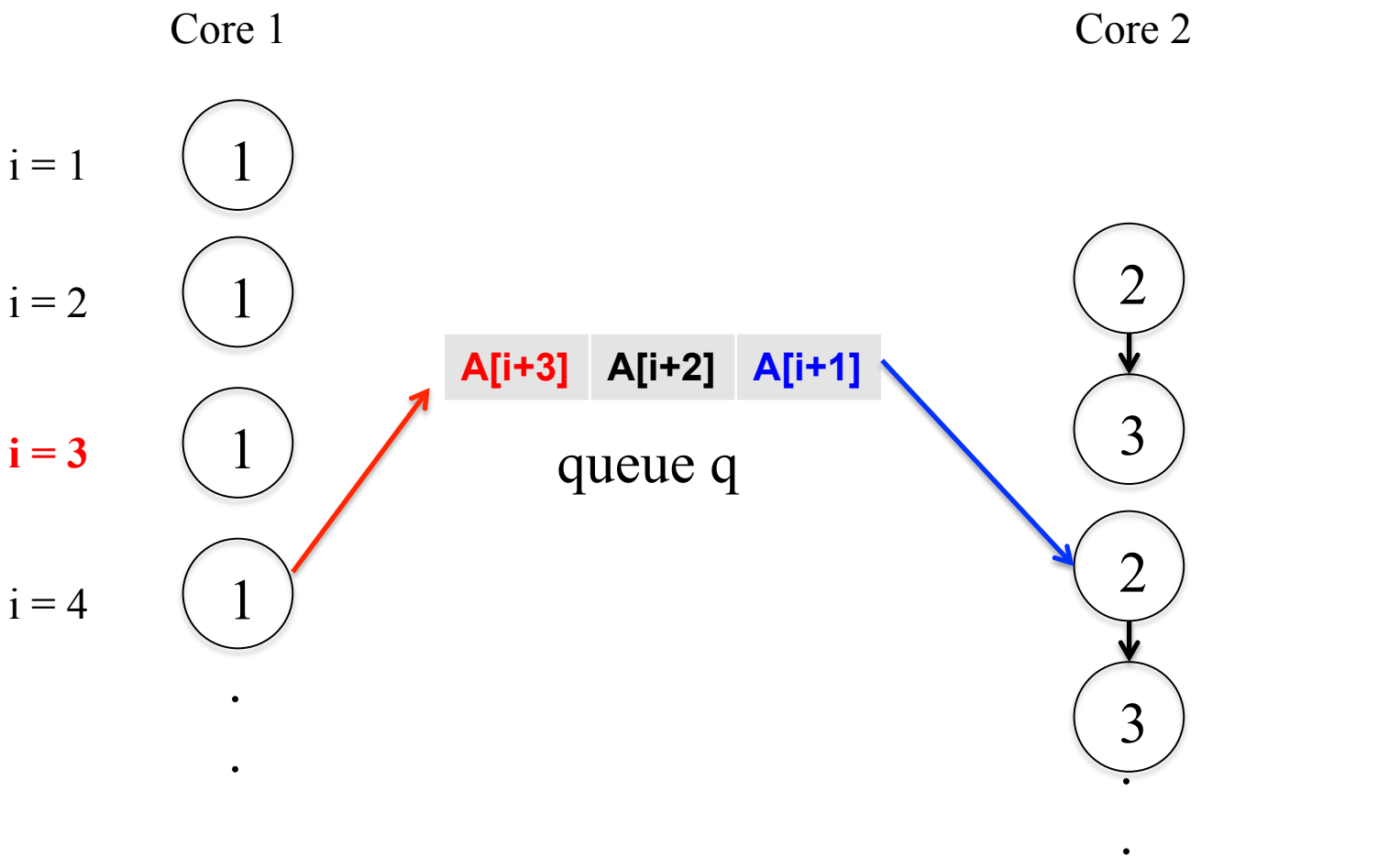
Decoupled Software Pipelining (DSWP)



Decoupled Software Pipelining (DSWP)

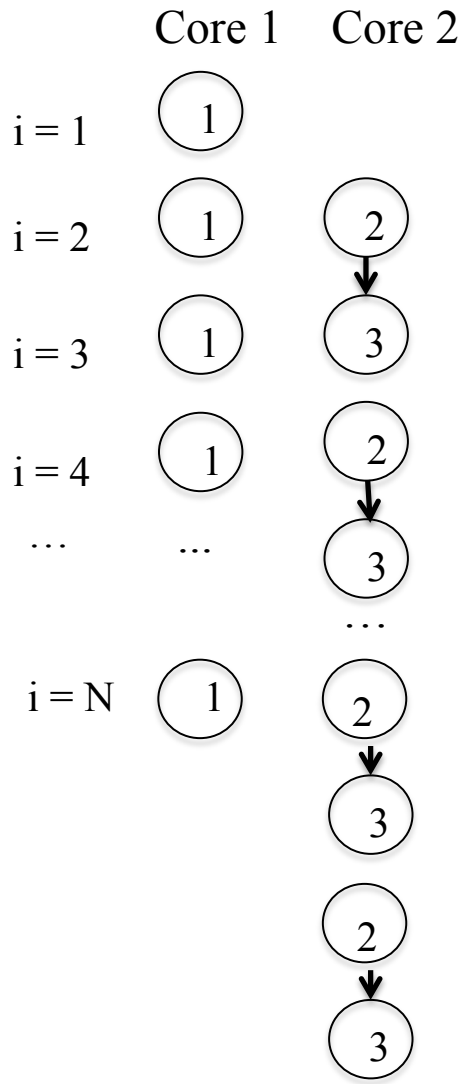


Decoupled Software Pipelining (DSWP)



Como ficaria com DSWP?

- Speed-up:
  - 3N/ ????? ~ Sua vez!!



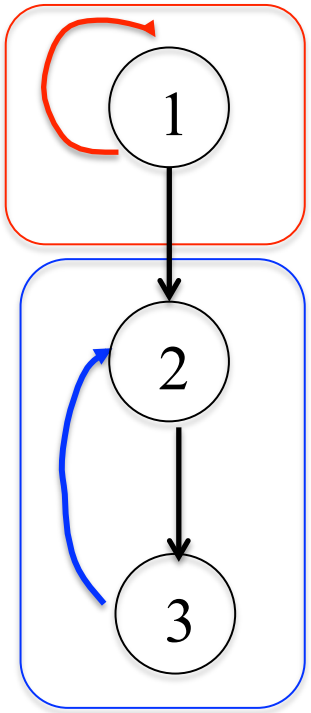
Sempre Funciona?

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

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

Como Assim?

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

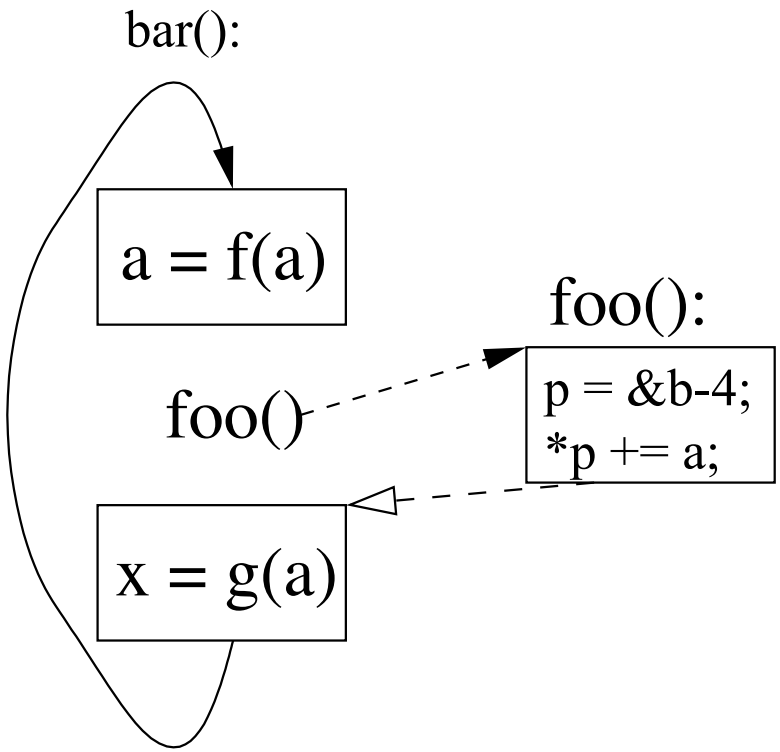


É sempre possível detectar ciclos?

```
static int a;
static int b;

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

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



Referências

- Software Pipelining. Allan V.H, Jones R.B, Lee R. M, Allan S. J. ACM Comput. Surv. 27 3 1995 367-432.

- Global Multi-Threaded Instruction Scheduling, Guilherme Ottoni and David I. August, Proceedings of the 40th IEEE/ACM International Symposium on Microarchitecture (MICRO), December 2007.

- Parallel-Stage Decoupled Software Pipelining, Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew Bridges, and DavidI. August, Proceedings of the 2008 International Symposium on Code Generation and Optimization (CGO), April 2008.

- Programming Multicores: Do Applications Programmers Need to Write Explicitly Parallel Programs? Arvind, David I. August, Keshav Pingali, Derek Chiou, Resit Sendag, and Joshua J. Yi, IEEE Micro, Volume 30, Number 3, May 2010.

- Software Pipeplining: http://en.wikipedia.org/wiki/Software_pipelining

Detecção de Paralelismo em Laços de Arquiteturas MIMD e Multicore

Guido Araujo

WAMCA/WSCAD 2011

Vitória, ES

LSC COMPUTER SYSTEMS LABORATORY

IC UNICAMP