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

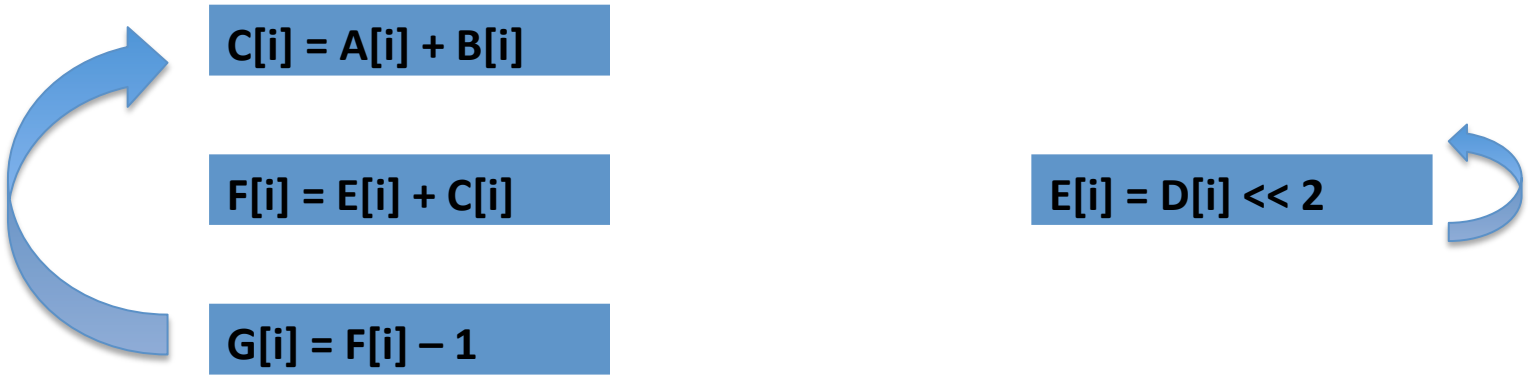
```
for (i = 0; i < N; i++) {
   C[i] = A[i] + B[i];
   F[i] = E[i] + C[i];
   G[i] = F[i] – 1;
}
```

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

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;
}
```

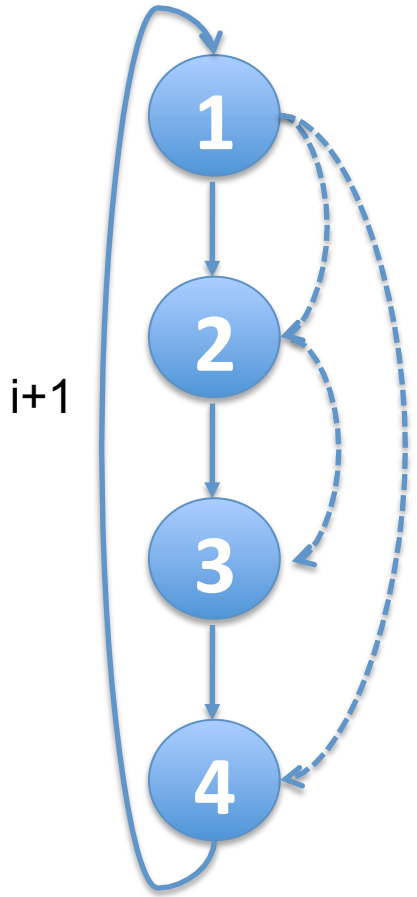


Paralelismo em Arquiteturas MIMT

- Doall
- Doacross
- Software pipelining
- Decoupled Software Pipelining

Grafo de Dependências

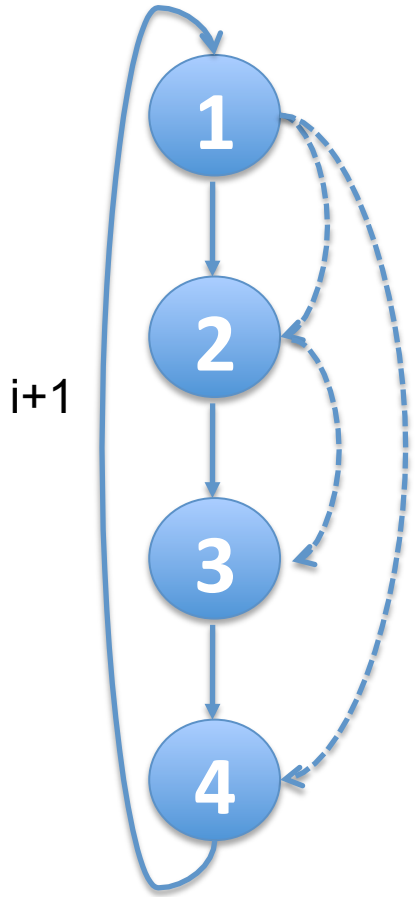
- Arestas cheias
  - Fluxo de controle
- Arestas pontilhadas
  - Dependência de dados



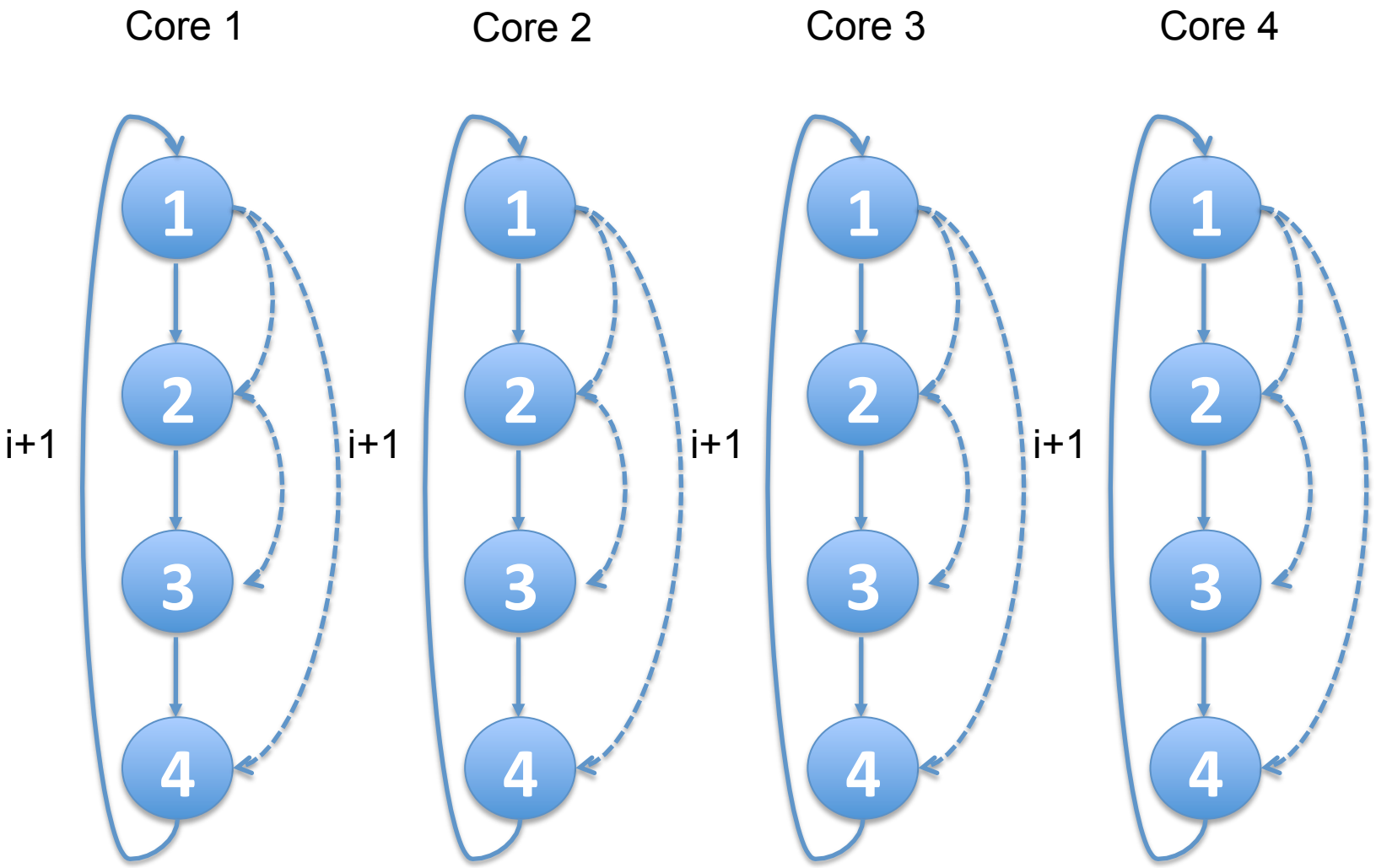
Doall

- 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

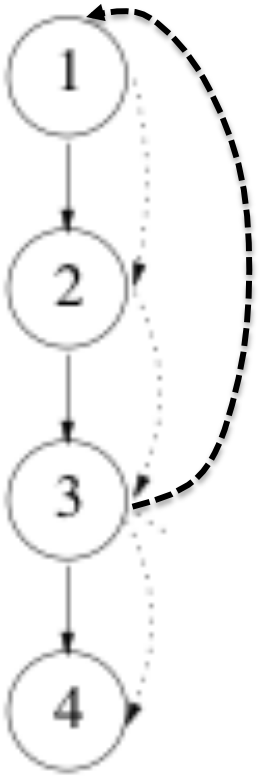
```
for (i = N/4; i < N/2; i++) {
   (1) C[i] = A[i] + B[i];
   (2) D[i] = C[i] << 2;
   (3) E[i] = D[i] + 1;
   (4) G[i] = C[i] - 1;
 }
```

……

Doacross

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

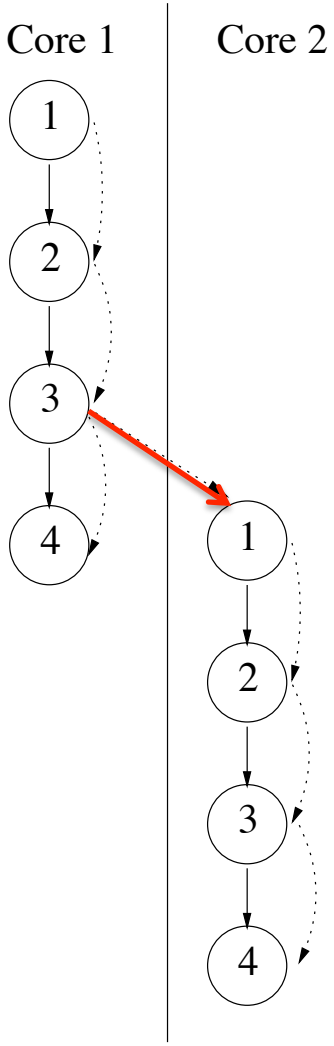
```
for (i = 0; i < N; i++) {
   (1) C[i] = A[i] + B[i];
   (2) D[i] = C[i] << 2;
   (3) A[i+1] = D[i] + 1;
   (4) E[i] = A[i+1] - 1;
 }
```



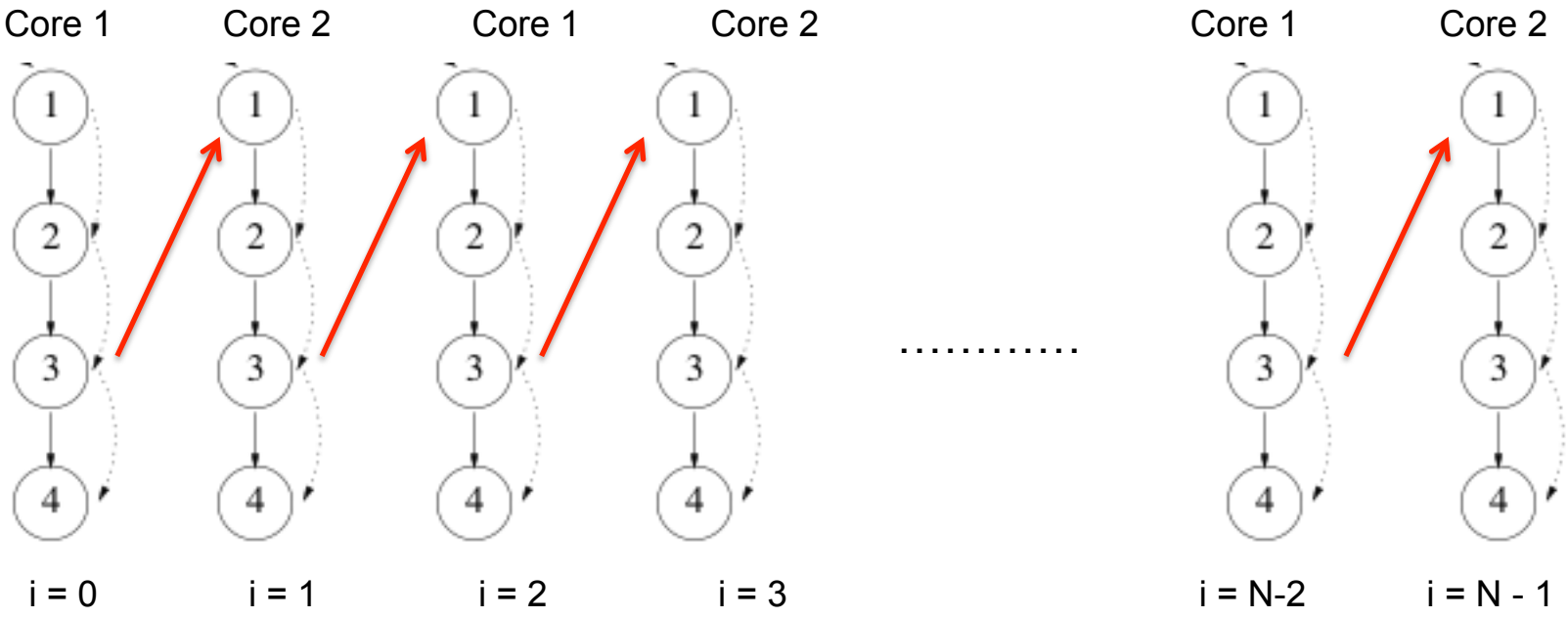
Doacross

- E se dependência for “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] = BIG(A[i+1]);
 }
```

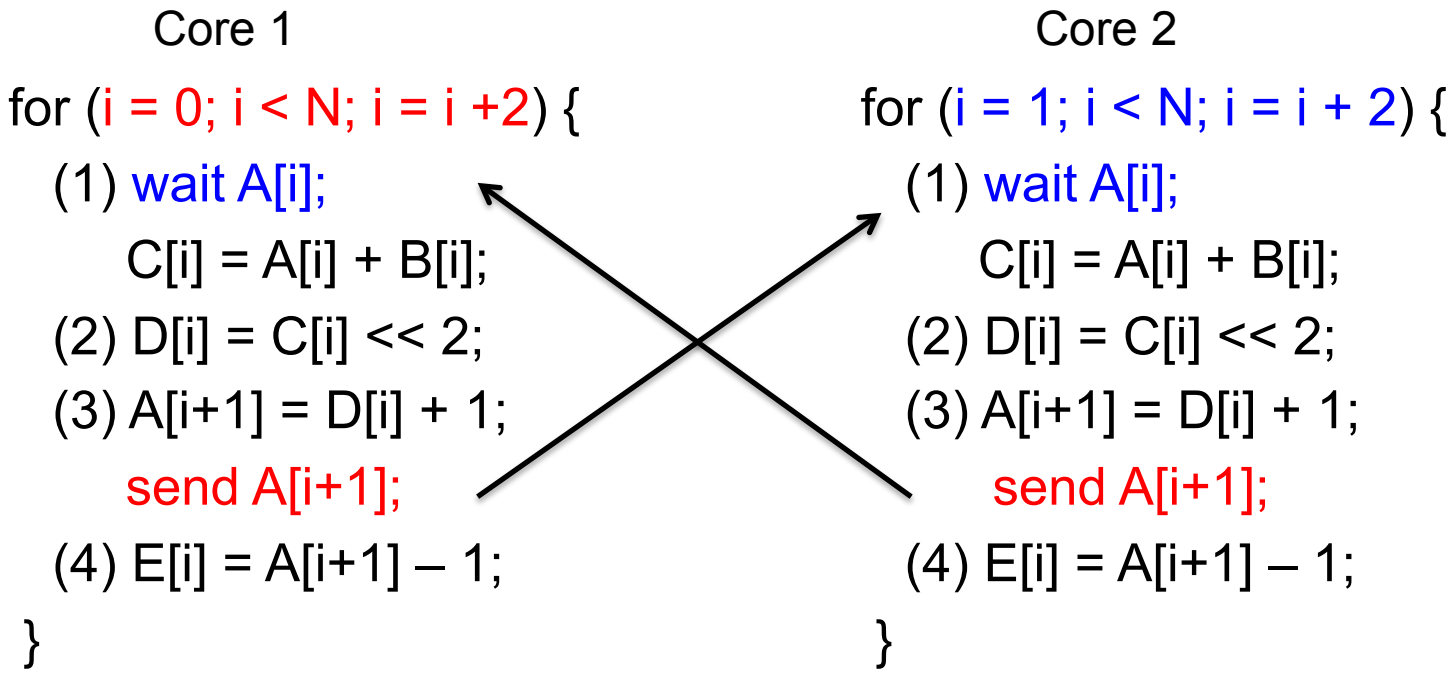


Doacross (Software Pipelining)



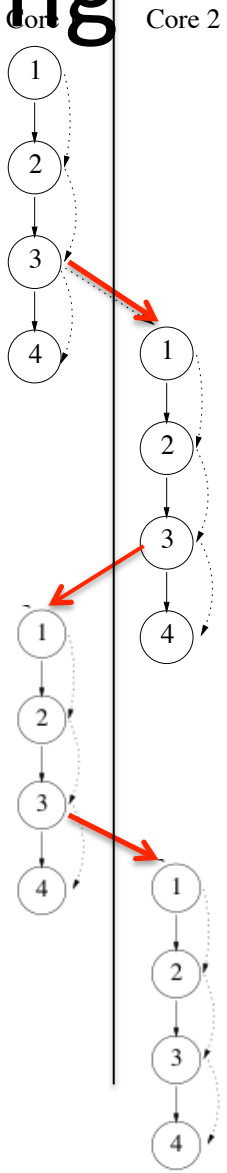
Doacross

- 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.



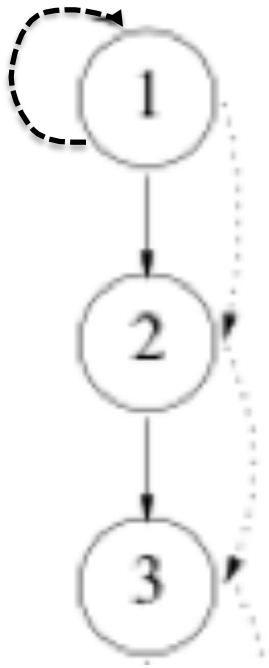
Software Pipelining

- 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



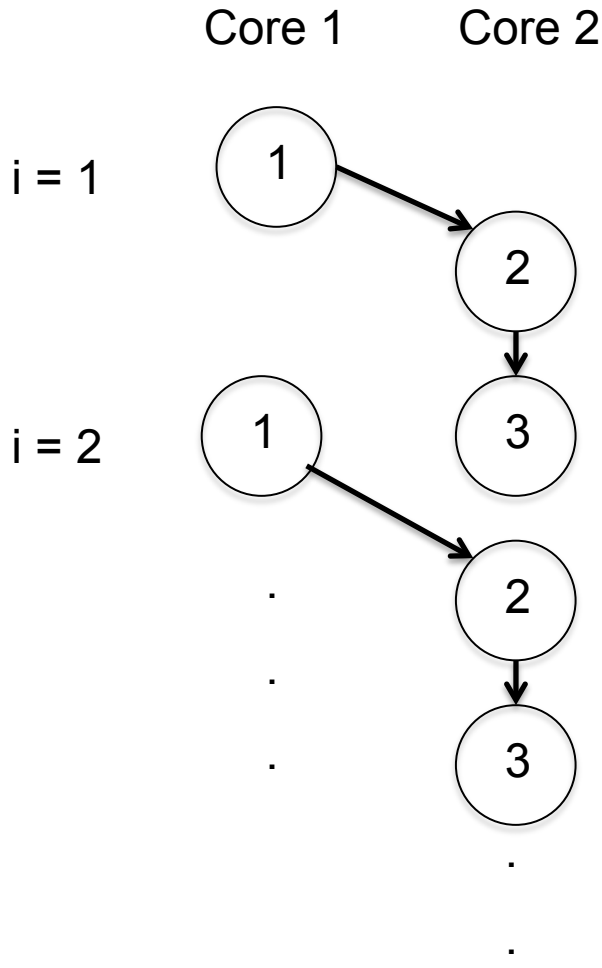
Um outro exemplo

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



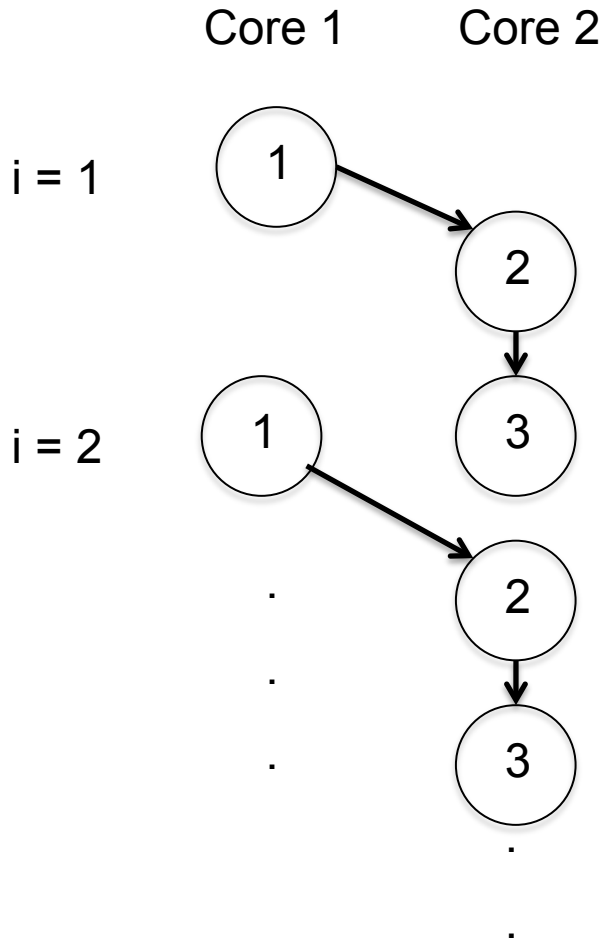
Como ficaria com Software Pipelining?

- Speed-up:
  - 3N/(2N+1) ~ 50%



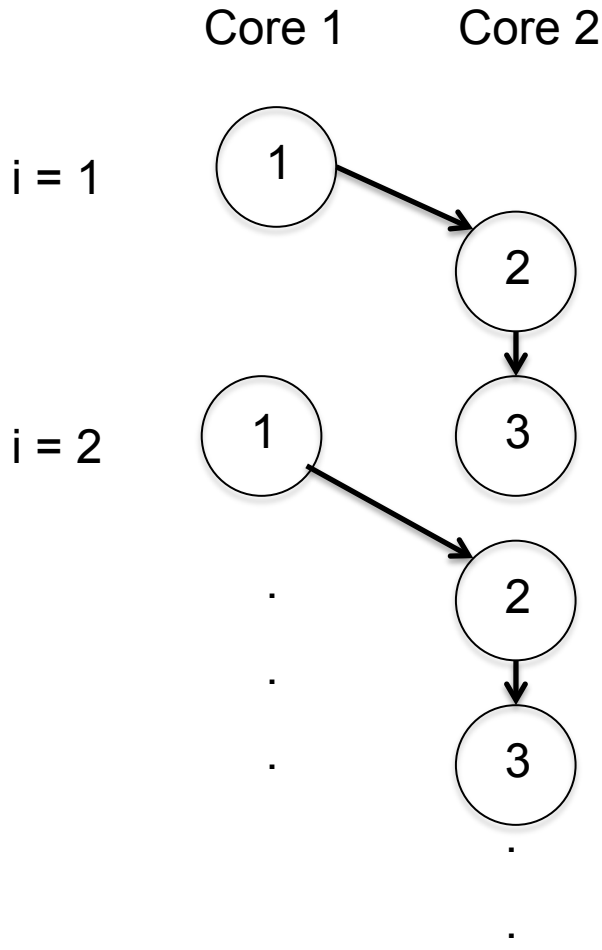
Será que dá para melhorar?

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



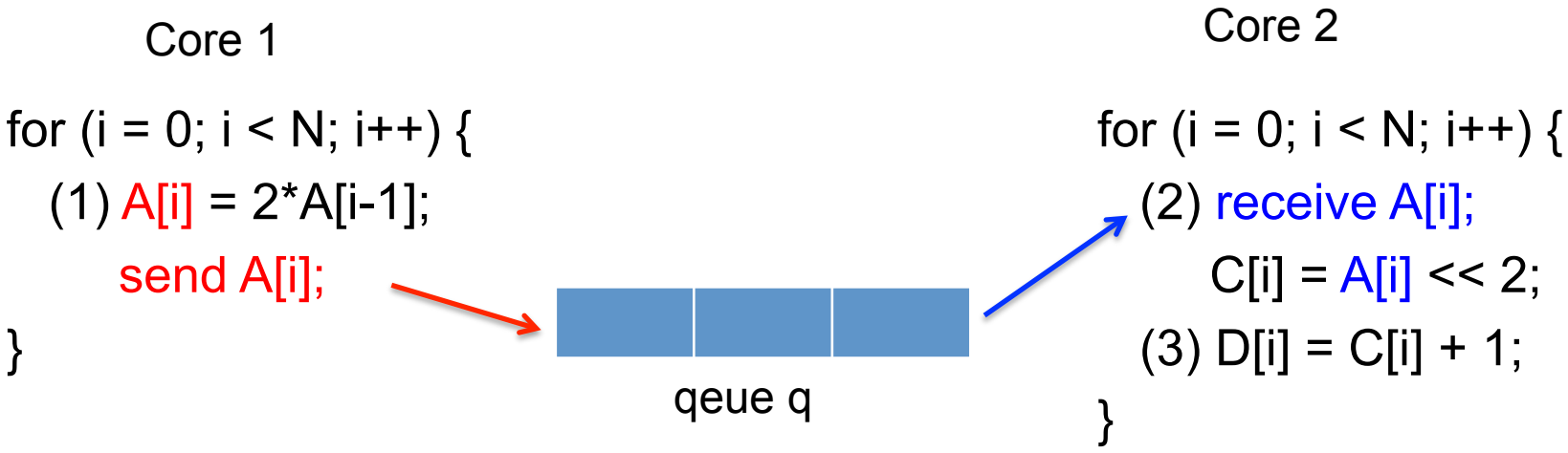
Será que dá para melhorar?

- 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?

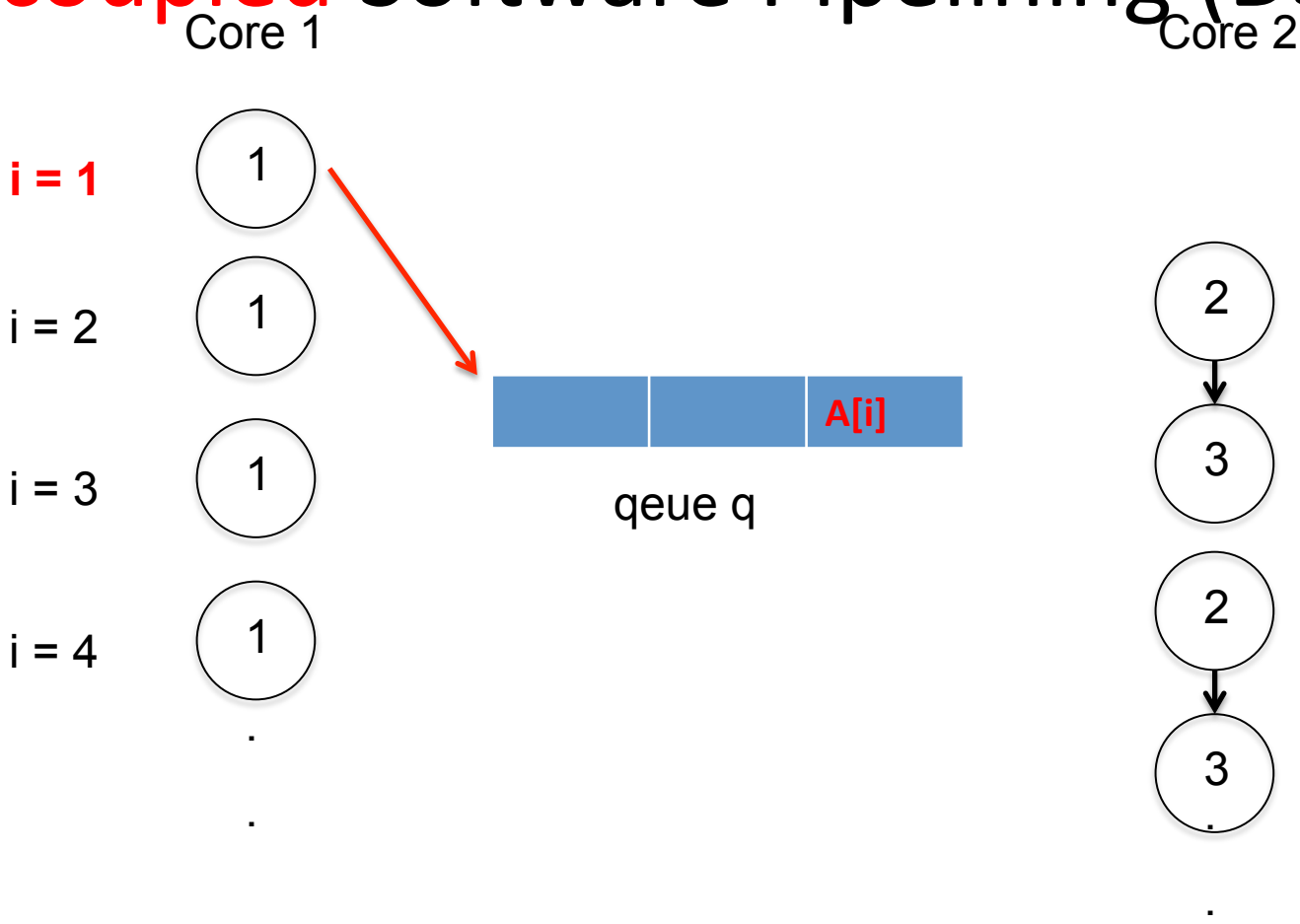


Decoupled Software Pipelining (DSWP)

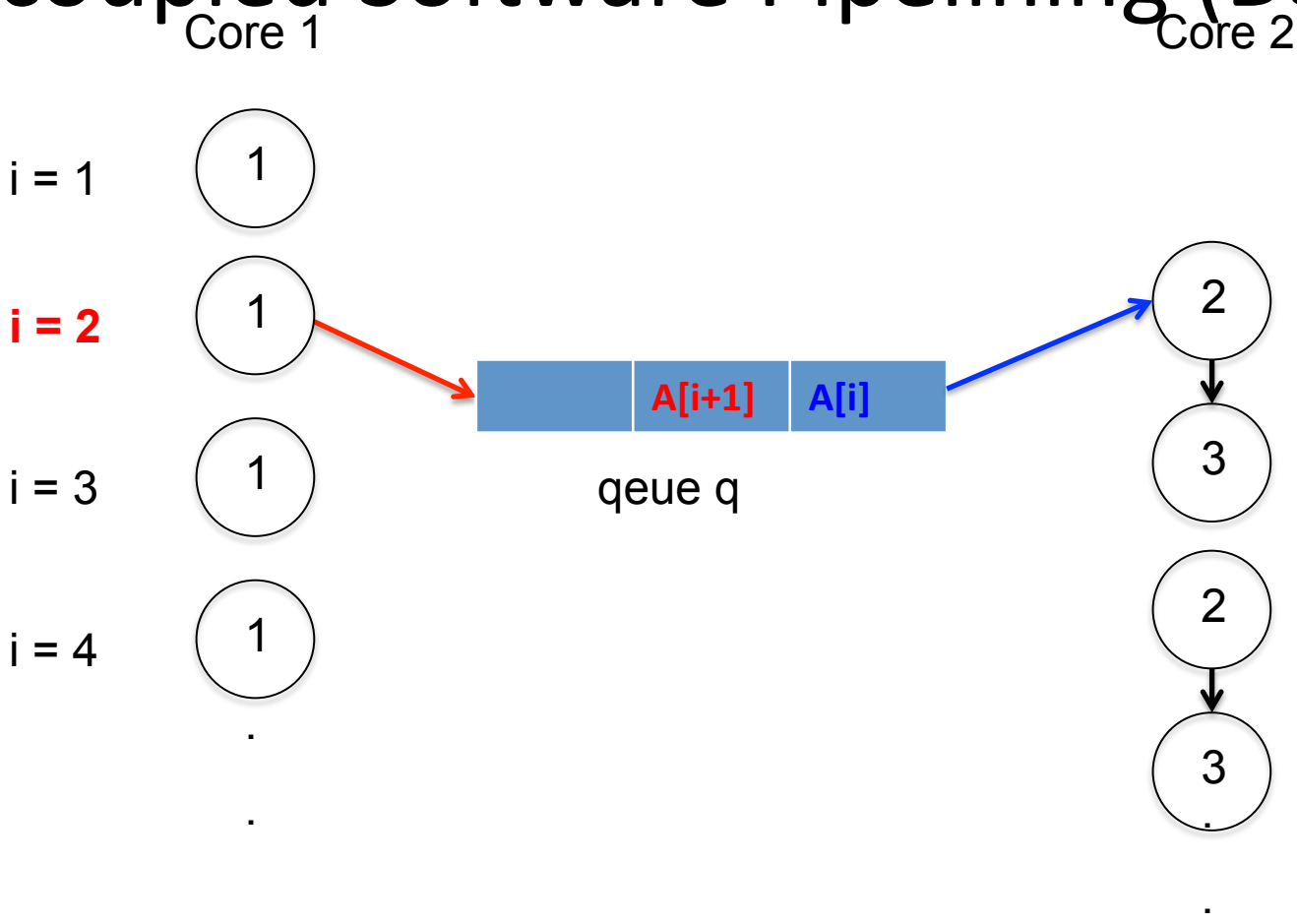
- Usando uma fila para comunicar dados
  - A[i] calculado no core 1, enviado para core 2
  - Fila desacopla execução de ambos cores!



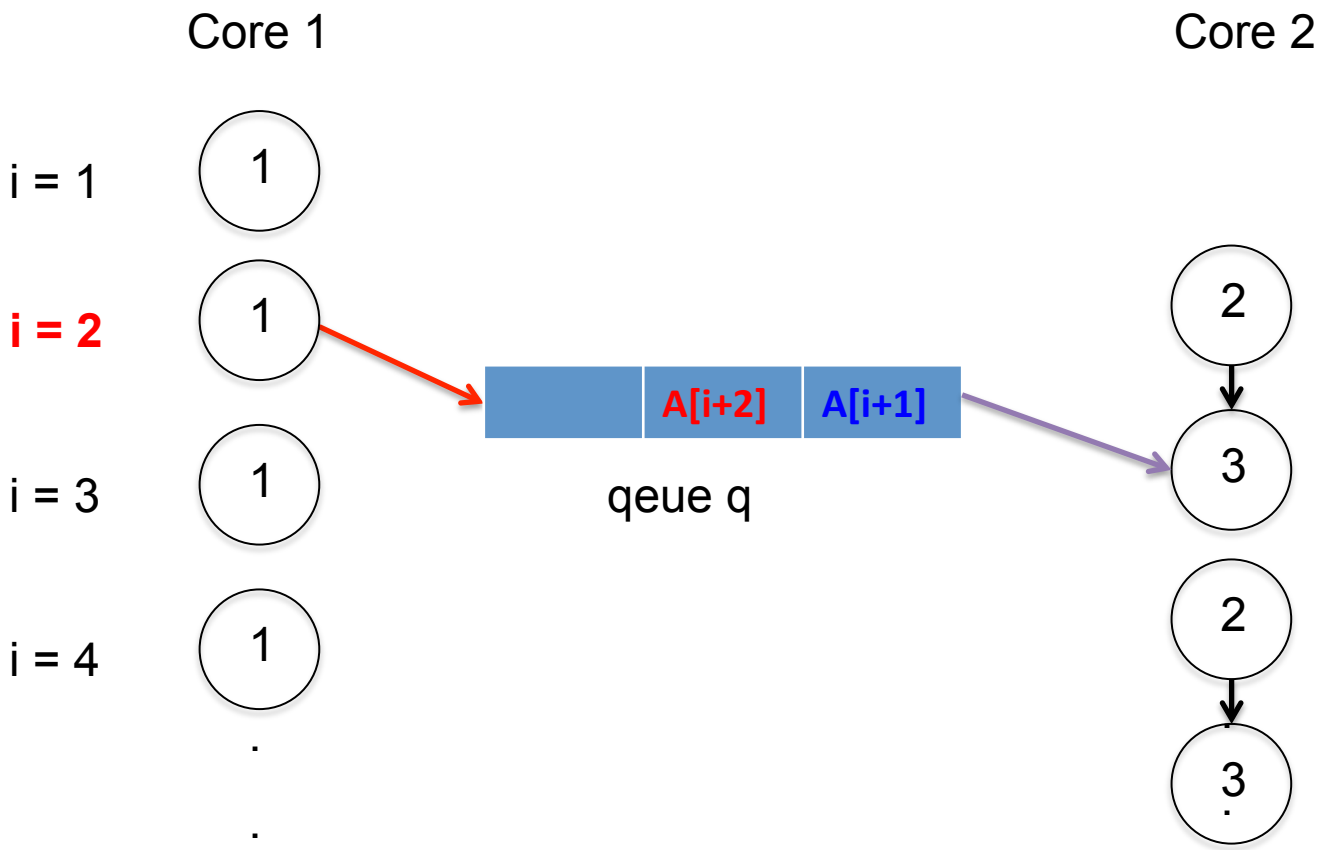
Decoupled Software Pipelining (DSWP)



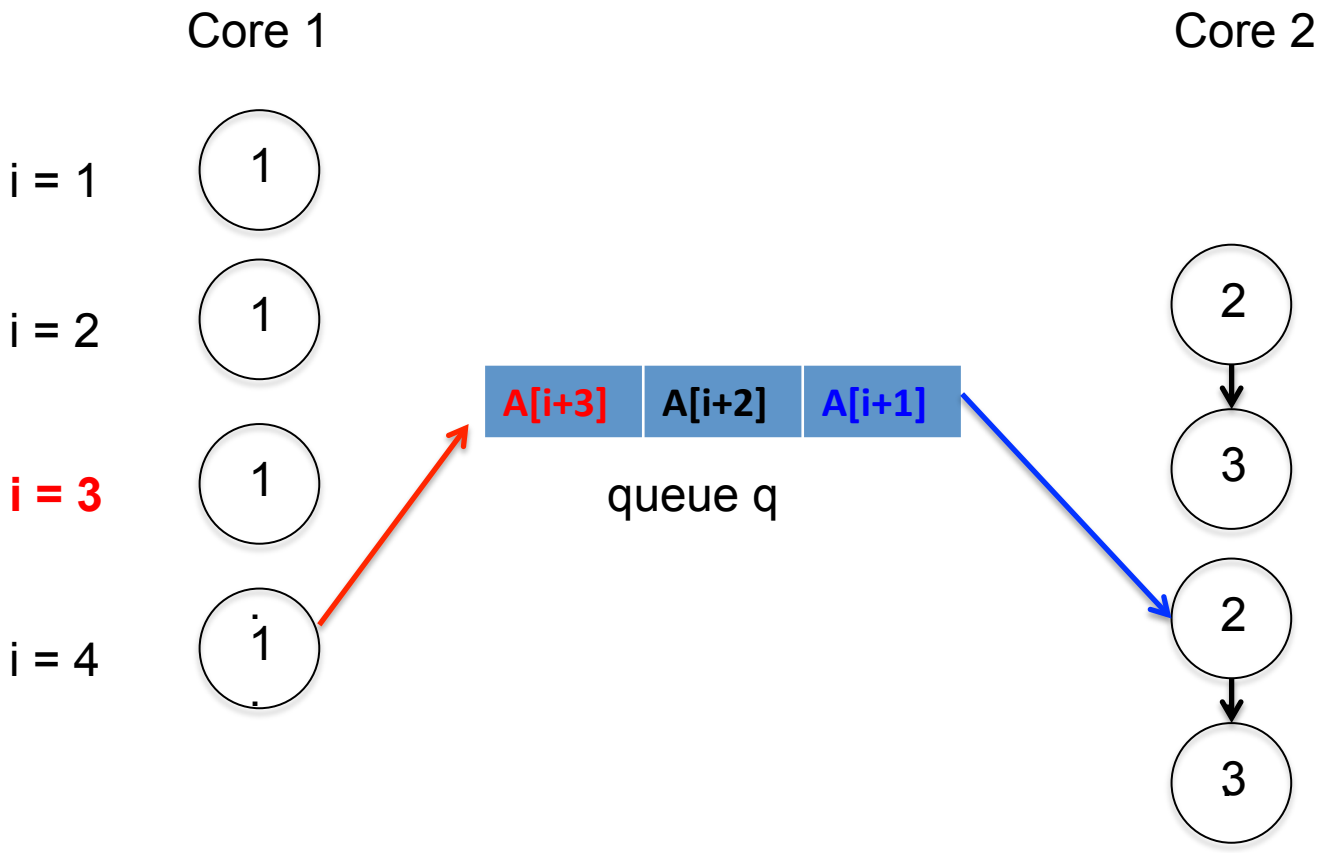
Decoupled Software Pipelining (DSWP)



Decoupled Software Pipelining (DSWP)

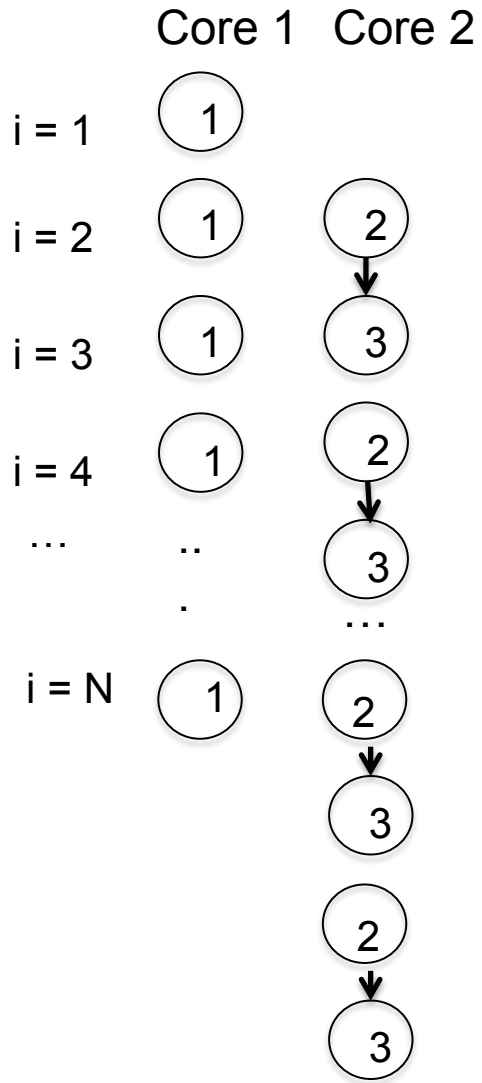


Decoupled Software Pipelining (DSWP)



Como ficaria com DSWP?

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



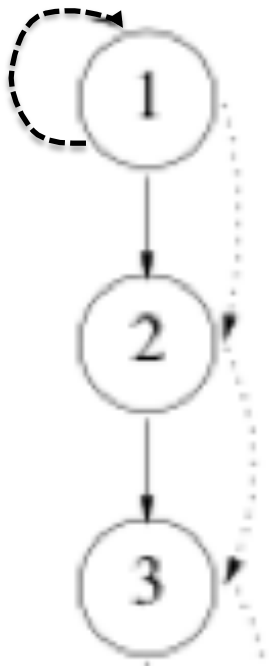
Sempre Funciona?

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

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

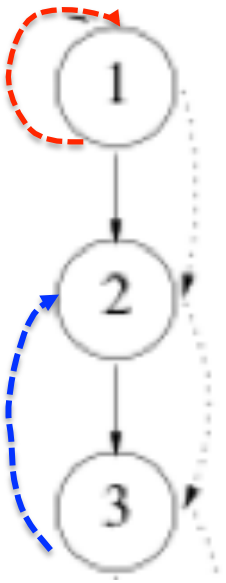
Como assim?

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



Um outro Exemplo?

```
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;
}
```



Core 1

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

Core 2

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

Bibliografia Suplementar

- G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. IEEE/ACM MICRO, 0:105–118, 2005.
- Randy Allen & Ken Kennedy. Optimizing Compilers for Modern Architectures – A Dependence Based Approach