Implementación de grafos de sincronización

SCD (UGR) 21-22

Carlos Ureña - Nov 2020

Identificar los nodos iniciales y finales, crear nodos iniciales y finales únicos.

Definición de tarea inicial y final

- Una **tarea inicial** es una tarea sin ningún arco entrante (no debe esperar a ninguna otra tarea).
- Un **tarea final** es una tarea sin ningún arco saliente (ninguna otra tarea debe esperarla).

Antes de considerar la implementación del grafo:

- Si hay más de una tarea inicial, se crea un único nodo inicial vacío, previo a todos los nodos iniciales originales.
- Si hay más de una tarea final, se crea un único nodo final vacío, posterior a todos los nodos finales originales.

Después de eso, **habrá siempre una única tarea inicial y una única tarea final**, y todos las tareas del grafo tendrán al menos un arco entrante o al menos un arco saliente (excepto si el grafo consiste de una única tarea, en cuyo caso no hay que sincronizar nada).

(Una tarea vacío se implementa con una sentencia que no hace nada, no aparece en el listado final como ninguna sentencia)

Implementación con cobegin/coend. Identificación de partes secuenciales

Una **parte secuencial** es un conjunto (ordenado) de 2 o más tareas A,B,C,D,…,X tales que:

- Todas las tareas (menos la primera, **A**) tienen exactamente un único arco entrante (desde la anterior tarea en la secuencia)
- Todas las tareas (menos la última, **X**) tienen exactamente un único arco saliente (hacia la siguiente tarea de la secuencia)

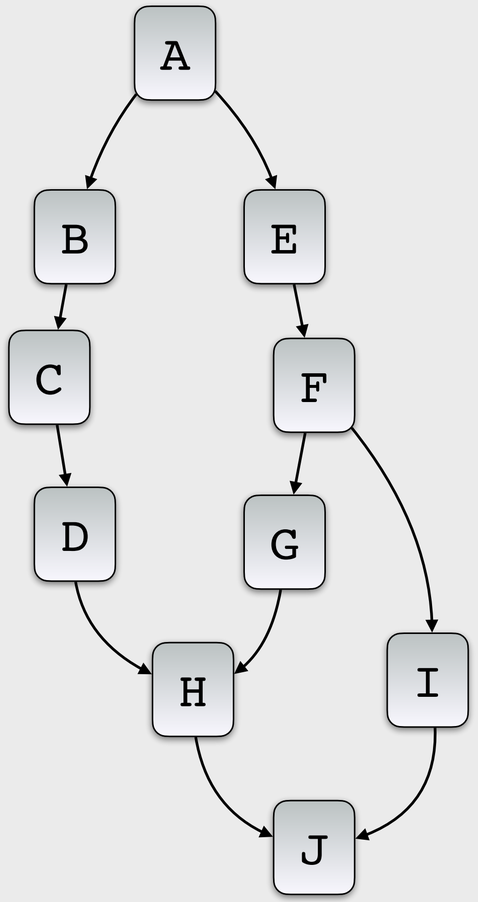
En estas condiciones, el conjunto de tareas A,B,C,D,… puede convertirse en una única tarea que consiste en una sentencia compuesta **begin-end:**

```
begin
    A; B; C; ….; X;
end
```

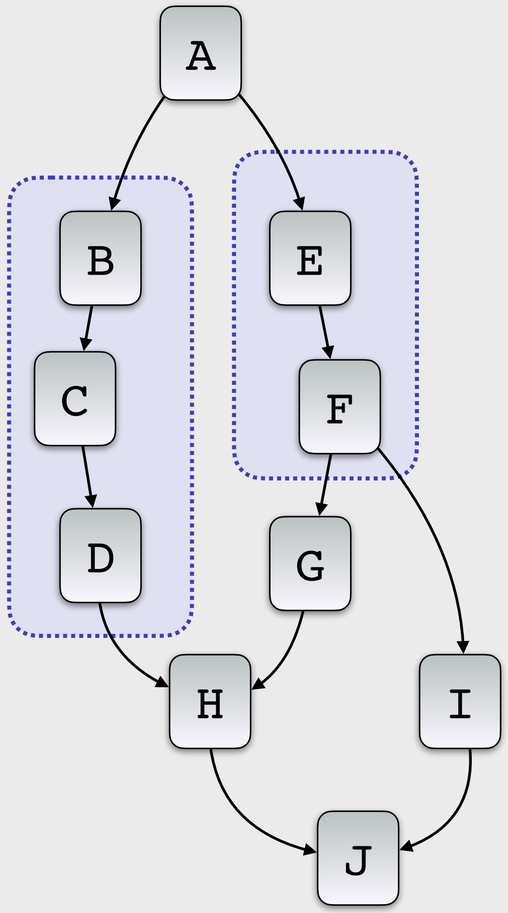
El grafo tiene una única entrada por A y una única salida por X

Implementación con cobegin/coend. Identificación de partes secuenciales

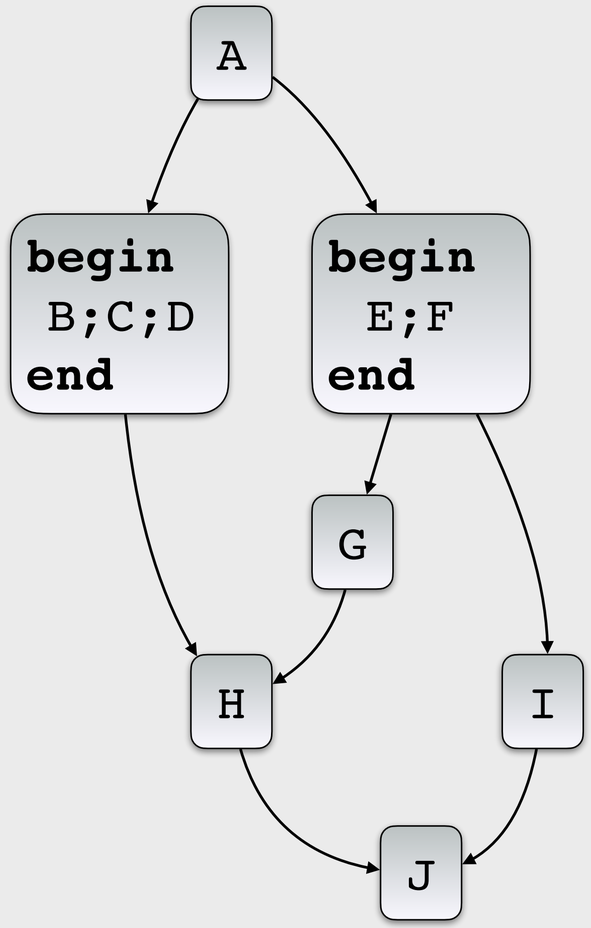
Grafo original



Identificación de partes **secuenciales**



Grafo equivalente



Implementación con *cobegin-coend*. Identificación de partes concurrentes:

Una **parte concurrente** es un conjunto de 2 o más tareas A, B, C… tales que:

- No hay arcos entre ellas (ni directa, ni indirectamente).
- A cada una de esas tareas entra exactamente un arco, todos procedentes de una misma tarea previa **E**
- De cada una de ellas sale exactamente un arco, todos dirigidos a una misma tarea posterior **S**

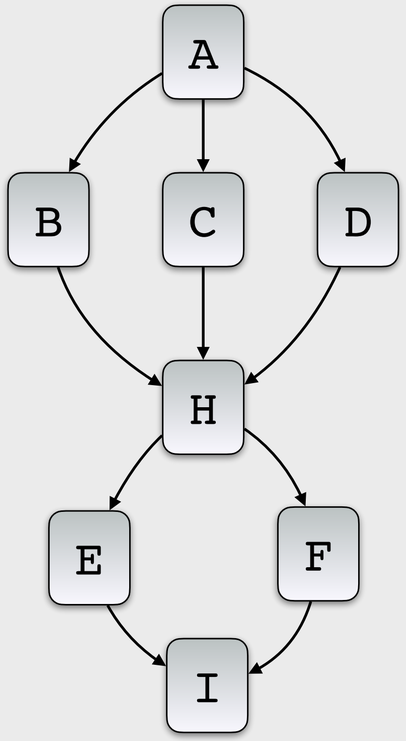
En estas condiciones, el conjunto de tareas A,B,C… se puede convertir (contraer) en una única tarea que consiste en una sentencia compuesta **cobegin-coend:**

```
cobegin
   A; B; C; …
coend
```

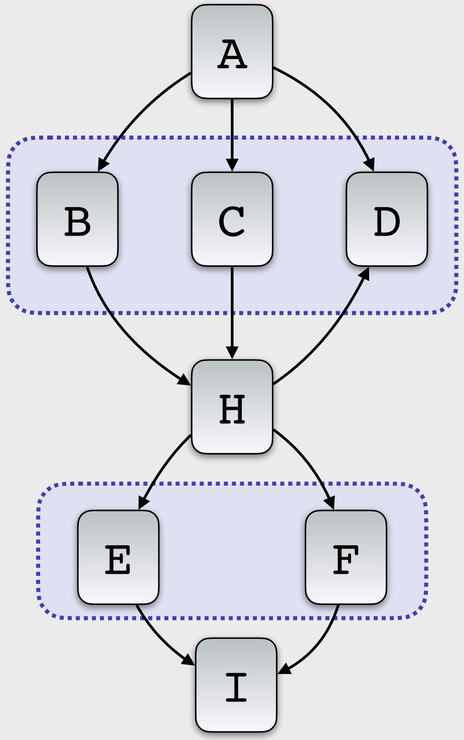
El sub-grafo tiene una única entrada por E y una única salida por S.

Implementación con cobegin/coend. Identificación de partes concurrentes

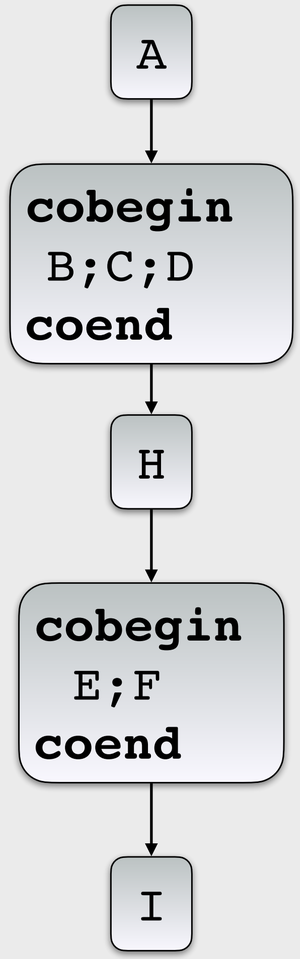
Grafo original



Identificación de partes **concurrentes**

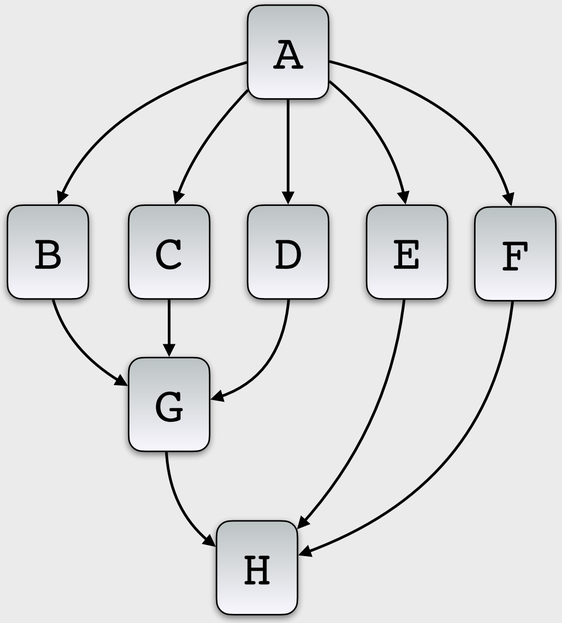


Grafo equivalente

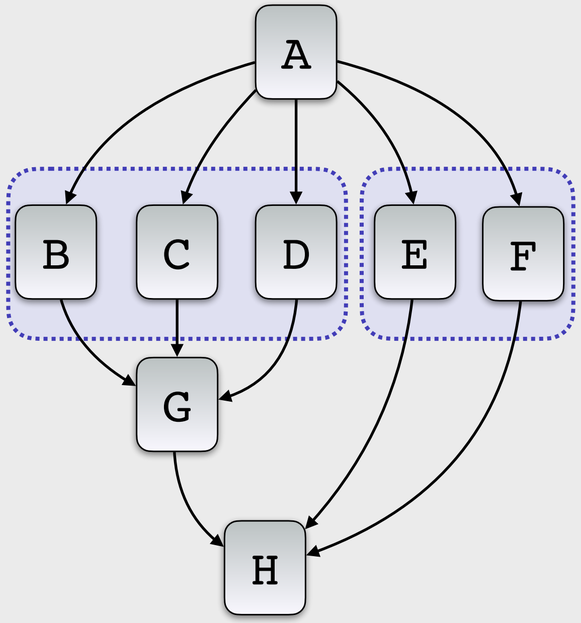


Implementación con cobegin/coend. Identificación de partes concurrentes

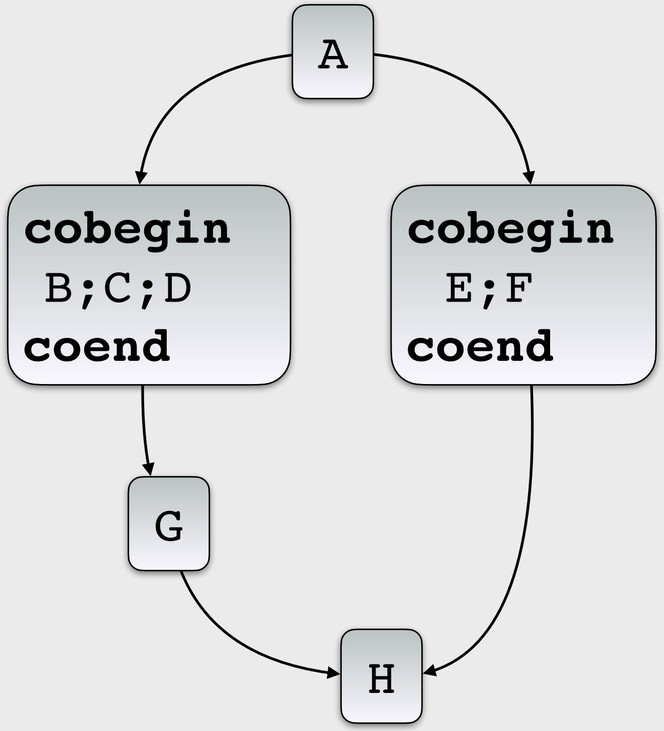
Grafo original



Identificación de partes **concurrentes**



Grafo equivalente



Procedimento general para cobegin-coend:

1. Identificar (y crear si no existen) el nodo inicial y el nodo final.
2. Identificar partes (subgrafos) secuenciales y concurrentes, los más grandes posibles, y una vez identificados:
   - Convertir cada parte secuencial en una única tarea con una sentencia **begin-end**
   - Convertir cada parte concurrente en una única tarea con una sentencia **cobegin-coend**

El paso 2 se debe repetir varias veces hasta que no se pueda identificar ninguna parte secuencial ni ninguna concurrente.

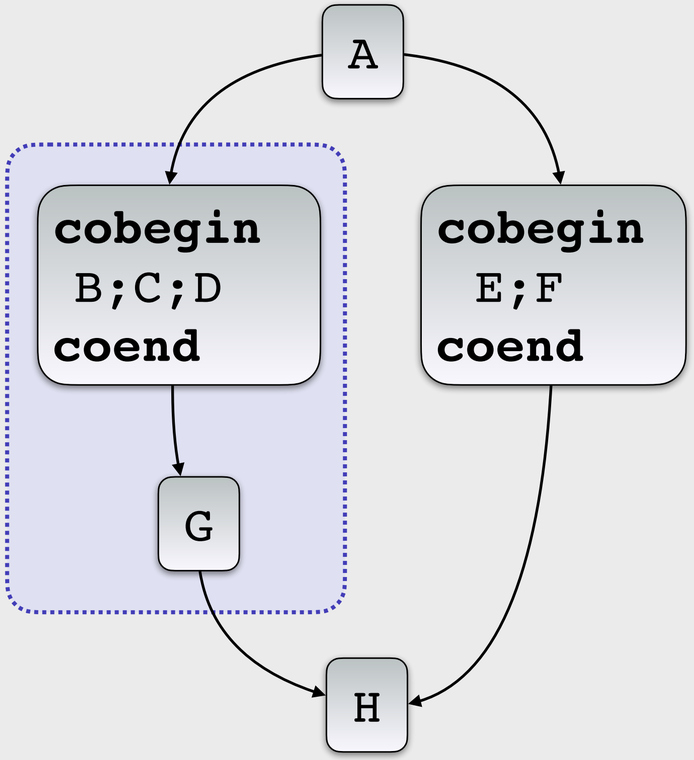
Como resultado se puede obtener:

- Un grafo con una única tarea, que será una sentencia compuesta que implementa el grafo original.
- Un grafo con más de una tarea, que no puede simplificarse más de esta forma.

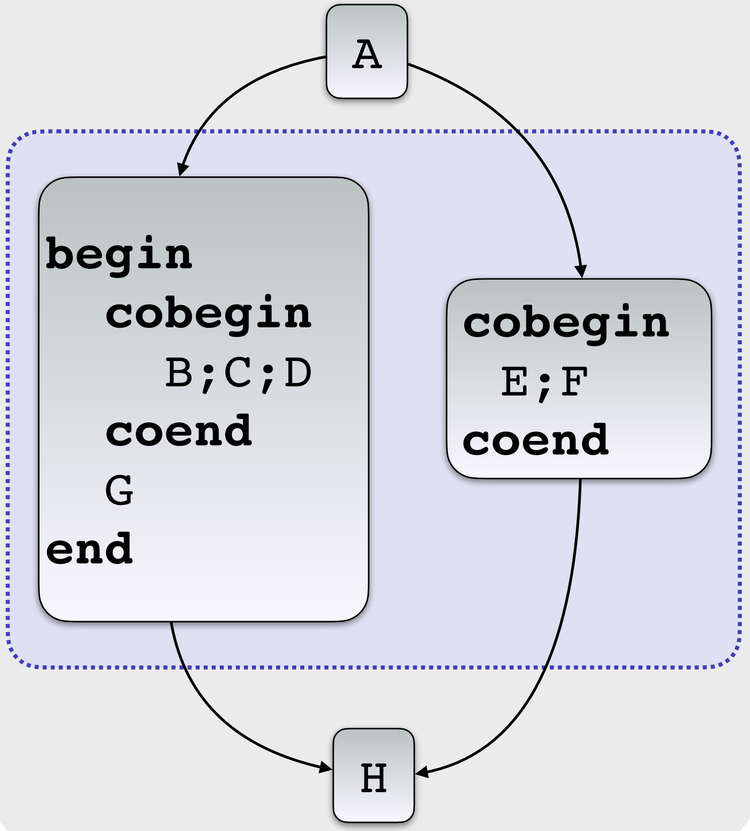
En el segundo caso, no se puede implementar la tarea con **cobegin-coend** aprovechando todo el paralelismo potencial del grafo. Habrá que seleccionar que paralelismo potencial hay que *sacrificar* o prohibir (se añaden arcos entre nodos que se podrían ejecutar potencialmente en paralelo, para intentar de nuevo el paso 2 hasta reducir el grafo a una única tarea).

Procedimento general para cobegin-coend: Ejemplo de iteraciones sucesivas

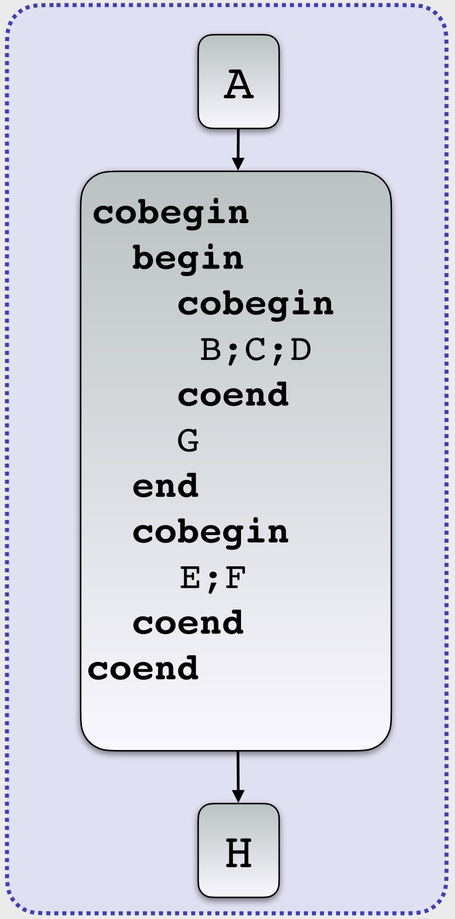
Grafo original, con identificación de parte **secuencial.**



Grafo equivalente, con identificación de parte **concurrente.**



Grafo equivalente, con identificación de parte **secuencial.**



Procedimento general para cobegin-coend: Ejemplo de iteraciones sucesivas

Grafo equivalente de una única tarea

```
begin
  A
  cobegin
    begin
      cobegin
        B;C;D
      coend
      G
    end
    cobegin
      E;F
    coend
  coend
  H
end
```

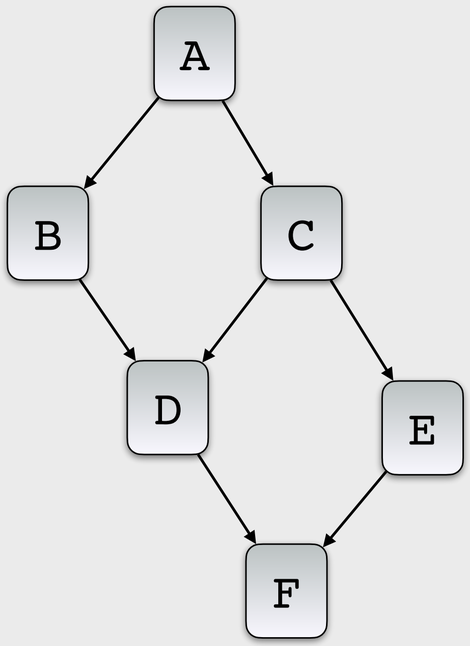
Programa final (se simplifican **cobegin-coend** anidados)

```
begin
  A
  cobegin
    begin
      cobegin
        B;C;D
      coend
      G
    end
    E;F
  coend
  H
end
```

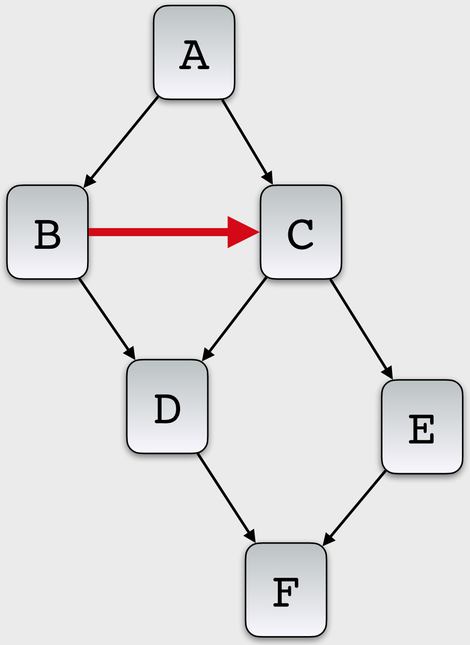
Procedimento general para *cobegin-coend*: grafo irreducible.

En este grafo (irreducible) insertamos un arco nuevo para poder implementarlo con **begin-end**

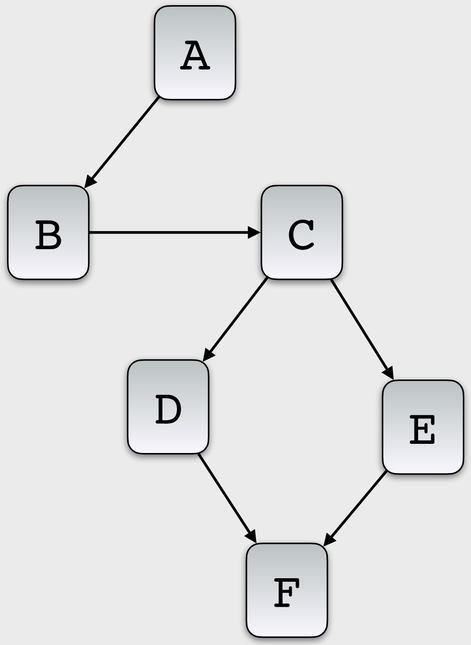
Ejemplo de grafo **irreducible** (no hay partes secuenciales ni concurrentes)



Añadimos un arco B —> C nuevo (en rojo) (se reduce el paralelismo potencial)



Eliminamos arcos A —> C y B —> D (ahora son redundantes) El grafo es reducible iterativamente.



Programa final Implementa el grafo pero B precede a C.

```
begin
   A;
   B;
   C;
   cobegin
      D; E
   coend
   F;
end
```

Procedimento general para fork-join.

En el caso de **fork-join**, se pueden dar estos pasos:

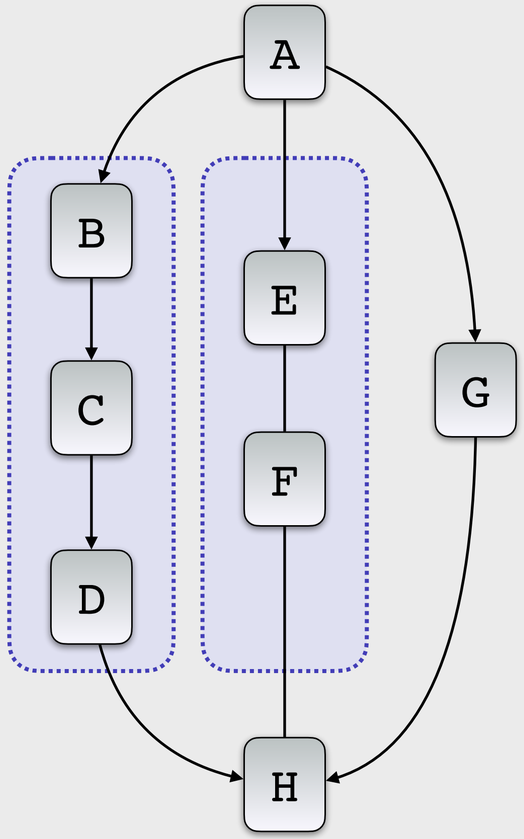
1. Identificar partes secuenciales, lo más largas posibles, y tantas como sea posible. Cada parte secuencial se convierte en un **procedure** en pseudo-código (un función o subprograma), al cual debemos dar un nombre. Contiene la secuencia de tareas entre **begin-end**. El grafo equivalente tiene una única tarea para cada parte secuencial. Esa tarea consiste en una llamada al procedure.
2. Identificamos partes concurrentes, lo más largas posible y tantas como sea posible. Cada parte concurrente se implementa con varios **forks**, seguidos de varios **join** (uno por cada fork).
3. Los dos pasos anteriores se deben repetir hasta que no se puedan encontrar más partes secuenciales ni concurrentes.

Si el grafo resultante (que no se puede reducir más) tiene varias tareas , se identifica un camino secuencial *principal* desde la tarea inicial hasta la final y se completan el resto de tareas con **fork-join** (esto se ve en otra transparencia posterior).

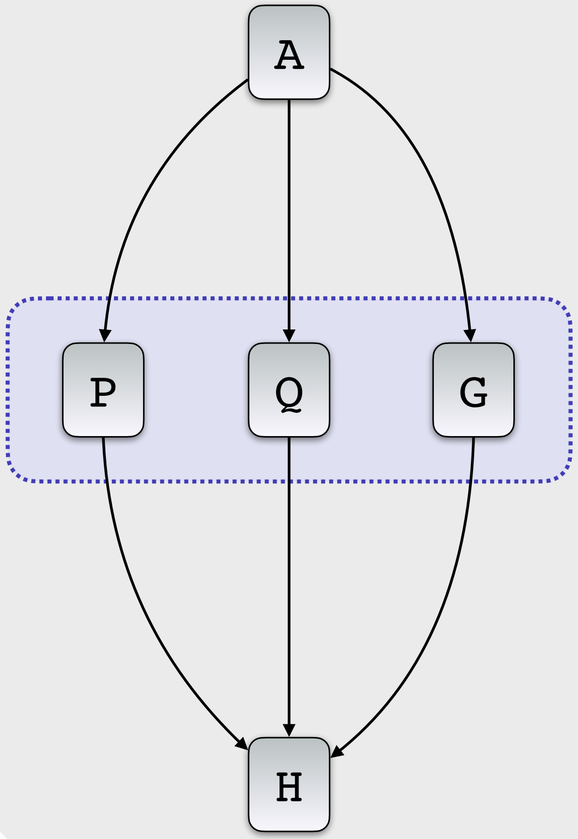
A veces resulta más sencillo usar directamente la identificación del camino principal para llegar a una solución más simple sin procedures auxiliares, aunque se puedan identificar partes secuenciales o concurrentes.

Procedimento general para fork-join. Ejemplo 1 (1/2).

Grafo original, con partes secuenciales (en azul)

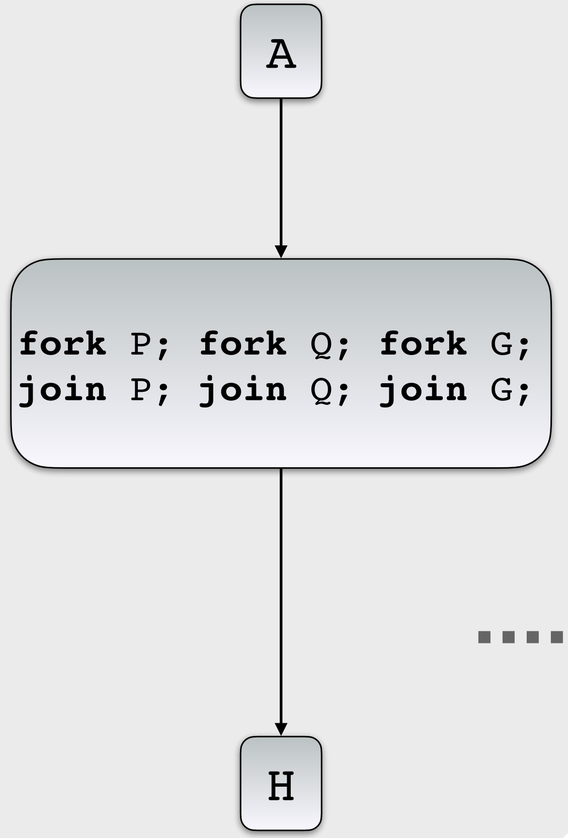


Grafo equivalente, con partes concurrentes (en azul) y los nuevos **procedures** auxiliares (**P** y **Q**)



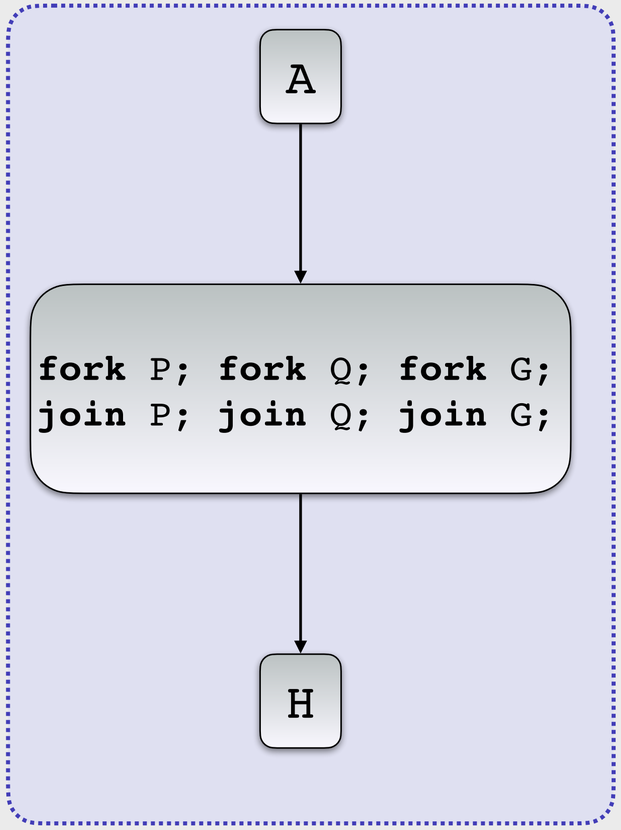

```
procedure P;
begin
  B; C; D;
end

procedure Q;
begin
  E; F;
end
```

Grafo equivalente, con tarea con **fork-join**



Procedimento general para fork-join. Ejemplo 1 (2/2).

Identificación de una (única) parte secuencial



Grafo equivalente de una única tarea (no es necesario definir un **procedure** auxiliar, pues es la única tarea del grafo)

```
begin
  A;
  fork P; fork Q; fork G;
  join P; join Q; join G;
  H;
end
```

Implementación final (con definición de procedures **P** y **Q**)

```
procedure P;
begin
  B; C; D;
end
procedure Q;
begin
  E; F;
end

begin
  A;
  fork P; fork Q; fork G;
  join P; join Q; join G;
  H;
end
```

Procedimento general para fork-join. Grafos sin partes concurrentes ni secuenciales.

Es posible que lleguemos a (o partamos de) un grafo en el cual no se pueden identificar ya más partes concurrentes ni secuenciales (lo llamamos irreducible), en ese caso se dan estos pasos:

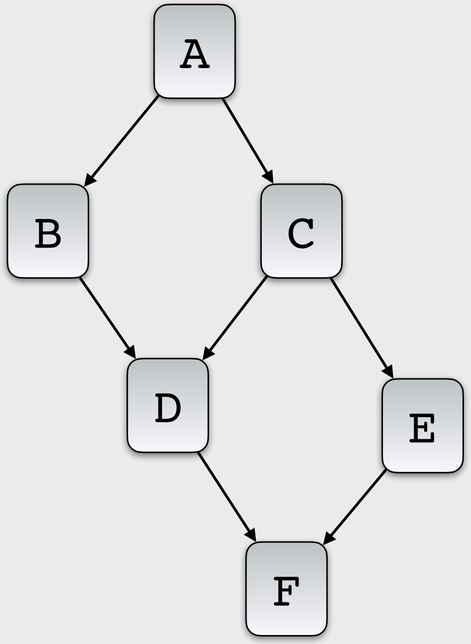
1. Identificar un camino (una secuencia ordenada de tareas) desde el nodo inicial hasta el final, de forma que haya un arco de cada tarea al siguiente (da igual que haya otros arcos). Lo llamamos *camino principal.* (Siempre existe)
2. Crear una *tarea principal* con la ejecución secuencial de tareas del camino principal (entre **begin/end)**
3. Buscamos todas y cada una de las tareas **X,** fuera de ese camino, tales que cumplen cada una de estas dos condiciones:
    - **X** tiene un único arco entrante, desde un nodo **A** que ya está en el camino principal, y
    - **X** tiene un único arco saliente, hacía un nodo **B** que ya está en el camino principal.
4. Para cada una de esas tareas **X** hacemos
    - Incluir **fork X** inmediatamente después de **A** en la tarea principal
    - Incluir **join X** inmediatamente antes de **B** en la tarea principal.

Al final, la tarea principal es la implementación del grafo.

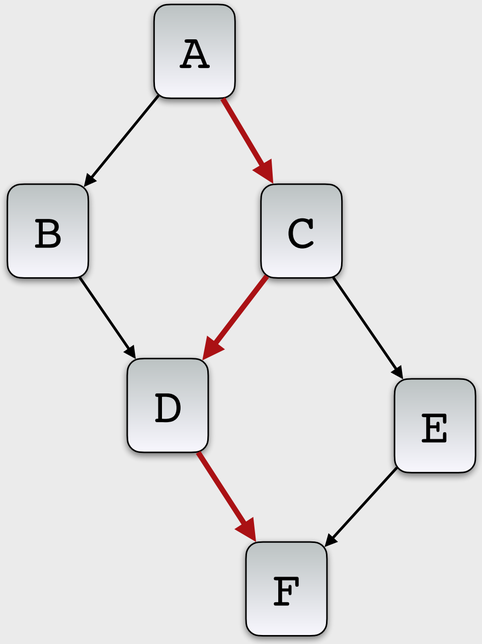
Procedimento general para *fork-join:* Ejemplo 2 (grafo original irreducible).

En este grafo (irreducible) usamos la identificación del camino principal. **fork-join** aprovecha todo el paralelismo potencial (con **begin/end** no se puede aprovechar)

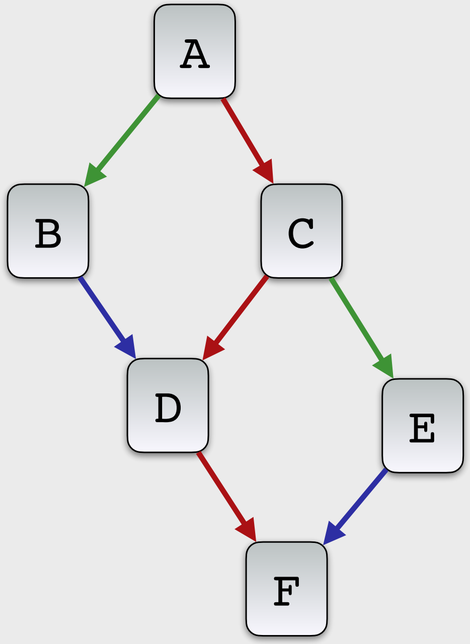
Grafo original.
No tiene partes secuenciales
ni concurrentes.



Arcos del camino principal (rojo)



Arcos implementados con **fork** (verde) y **join** (azul)



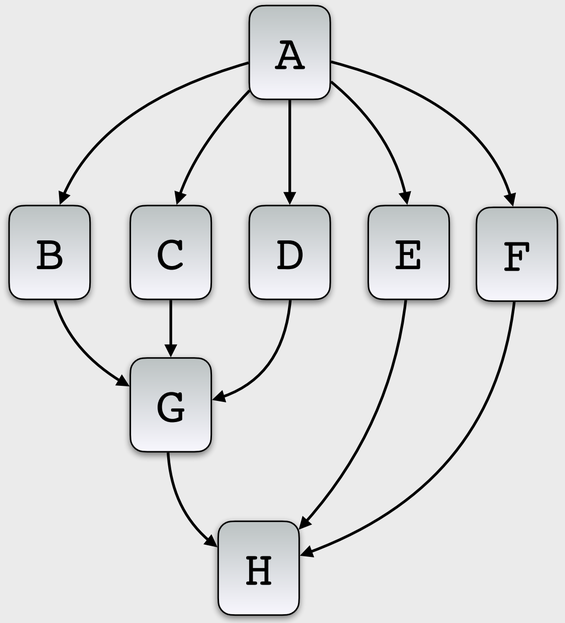
Tarea principal, completada con las sentencias **fork** y **join.**
Es la implementación final.

```
begin

  A;
  fork B;

  C;
  fork E;

  join B;
  D;

  join E;
  F;

end
```

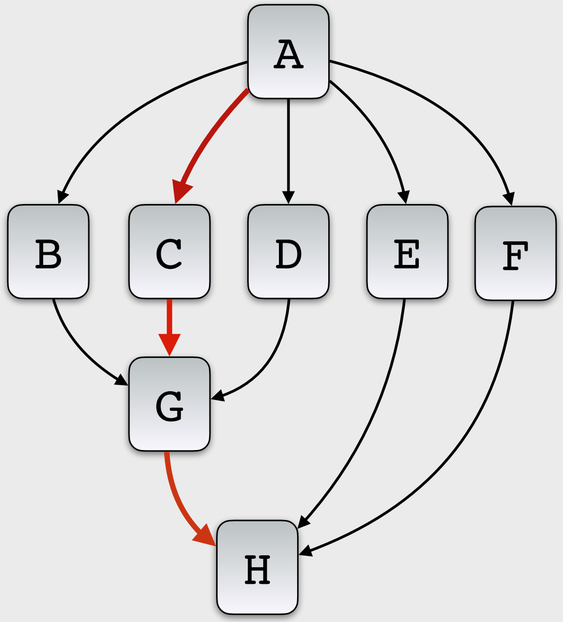
Procedimento general para *fork-join:* Ejemplo 3 - grafo reducible implementado con camino (1/2)

El camino principal **a veces** se puede usar también en grados reducibles, para obtener más directamente una implementación más sencilla (sin procedures auxiliares)

Grafo original



Secuencia ejecutada por el proceso principal (A —> C —> G —> H)



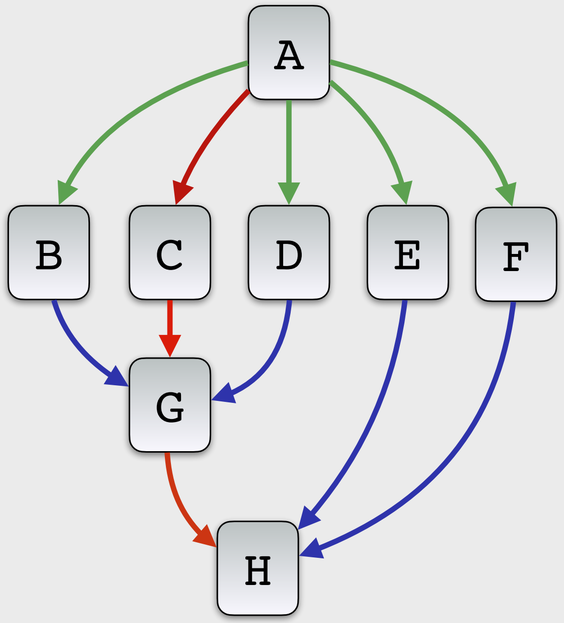
Tarea principal (ejecuta el camino principal)

```
begin
   A;
   C;
   G;
   H;
end
```

Procedimento general para. fork-join. Ejemplo 3 - grafo reducible implementado con camino (1/2).

Al igual que con **begin-end**, la implementación aprovecha todo el paralelismo potencial

Arcos implementados con **fork** (en verde) y con **join** (en azul)



Programa completo con las sentencias **fork** y **join**

```
begin
   A;
   fork B; fork D;
   fork E; fork F;
   C;
   join B ; join D;
   G;
   join E; join F;
   H;
end
```