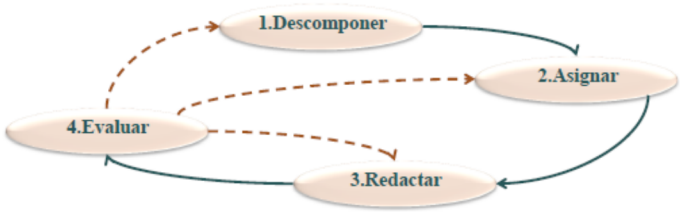
TEMA 2: PROGRAMACION PARALELA

LECCIÓN 2: PROCESO DE PARALELIZACIÓN

Para obtener una versión paralela de una aplicación con una biblioteca de funciones, con directivas o con un lenguaje de programación se podrían seguir los siguientes pasos:

.

1. *Descomponer (descomposition)* en tareas independientes o Localizar paralelismo
2. *Asignar (planificar+mapear)* tareas a procesos y/o threads.
3. *Redactar* código paralelo.
4. *Evaluar* prestaciones.

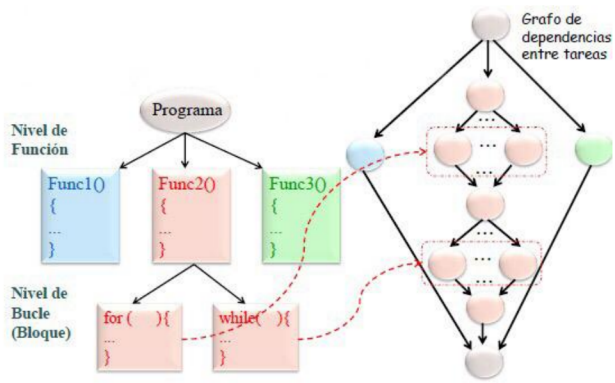


1. Descomponer en tareas independientes

El programador busca en la aplicación unidades de trabajo que puedan realizarse en paralelo, es decir, que sean independientes. Estas unidades junto con los datos que utilizan, formarán las tareas.

Durante esta fase, es conveniente representar la estructura de las tareas (dependencias entre ellas) mediante un grafo dirigido.

En el grafo, los arcos representan el flujo de datos y de control, y los vértices, las tareas que se pueden ejecutar en paralelo en cada momento.



Si partimos de un código secuencial, para extraer el paralelismo nos podemos situar en dos **niveles de abstracción**:

- **Nivel de función**: Analizando las dependencias entre las funciones del código, encontramos las que pueden ser independientes y por tanto se pueden ejecutar en paralelo.

  En el grafo anterior, se han encontrado 3 funciones independientes.

- **Nivel de Bucle:** Analizando las iteraciones de los bucles dentro de una función, podemos encontrar si pueden ser independientes. Con el análisis de los bucles podemos detectar paralelismo de datos.

En el grafo anterior, las iteraciones del bucle *for* son independientes. También los son las del bucle *while*. Además, el grafo muestra que la salida del bucle *for* se usa en el *while*, por lo tanto se ejecutan a distinto nivel.

La extracción del paralelismo de **tareas** se realiza mediante el análisis de dependencias de **funciones**, mientras que el paralelismo de **datos** se extrae mediante el análisis de dependencia **entre iteraciones de bucles**.

2. Asignar tareas a procesos y/o threads

En esta etapa se realiza la **asignación de tareas** del grafo de dependencias **a procesos y hebras.**

- La asignación de tareas incluye:
  - **Agrupación** de tareas en procesos/threads (*scheduling)*
  - **Mapeo** a procesadores/cores (*mapping*)

  Por lo general, **no** es conveniente asignar más de un proceso o hebra por procesador, por lo que la asignación a procesos o hebras está ligada con la asignación a procesadores.
- La granularidad de los procesos y hebras también dependerá de:
  - Número de procesadores, de manera que cuanto mayor sea su número, menos tareas se asignarán a cada proceso.
  - Tiempo de comunicación y sincronización frente al tiempo de cálculo.

Como regla básica, se tiende a asignar las iteraciones de un bucle (paralelismo de datos) a **hebras**, y las funciones a **procesos** (paralelismo de tareas)

La asignación debería repartir la carga de trabajo para optimizar la comunicación y la sincronización de forma que todos los procesadores empiecen y terminen a la vez, y no hacer que unos procesos/threads tengan que esperar a otros. Esto se conoce como *load balancing* (equilibrado)

¿De qué depende el equilibrado?

- **De la arquitectura**
    - **Heterogénea:** Estas arquitecturas tienen componentes con diferentes prestaciones, por lo que para obtener un buen equilibrado, habría que asignar más carga a los nodos más rápidos.
    - **Homogénea:** Una arquitectura homogénea puede ser uniforme o no uniforme:

        Si es **uniforme**, la comunicación de los procesadores con la memoria (y probablemente con otros componentes) supone el mismo tiempo sean cuales sean los dos componentes que intervienen. Por lo tanto, en este tipo de arquitecturas es donde menos influye en las prestaciones no considerar la arquitectura.

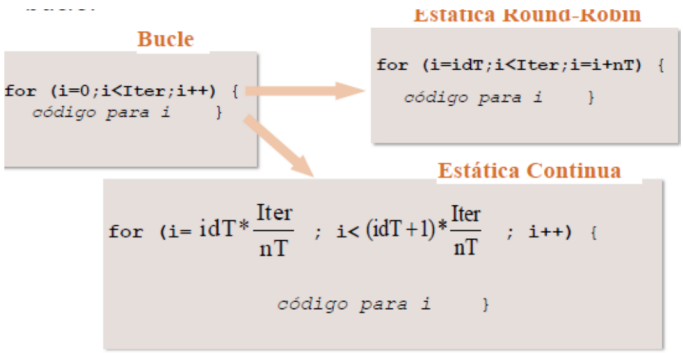
        Sin embargo, si una arquitectura es **no uniforme** la asignación de tareas para minimizar el tiempo de sincronización y comunicación es más complicada.

- **De la aplicación y descomposición en tareas**

Tipos de asignación

- **Estática**: Se realiza en tiempo de compilación. Antes de la ejecución del programa, ya se sabe que tarea va a realizar cada procesador o core. Puede realizarse de manera **explícita** (determinado por el programador), o de manera **implícita** (por la herramienta de programación al generar el ejecutable).

Ejemplo de asignación estática y explícita

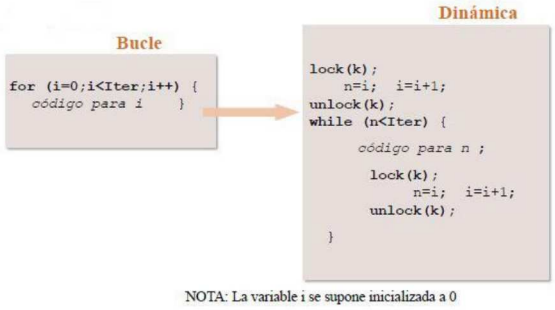


- **Dinámica**: Se realiza en tiempo de ejecución. En cada ejecución del programa, se pueden asignar distintas tareas a un procesador o core. Esto ayuda a evitar asignaciones estáticas complejas, especialmente cuando no se conoce el número de tareas. También permite que una aplicación acabe

aunque falle algún procesador. Sin embargo, este tipo de asignación consume más tiempo de comunicación y sincronización.

La asignación dinámica también se puede realizar de manera **explícita** o **implícita**.

Ejemplo de asignación dinámica y explícita



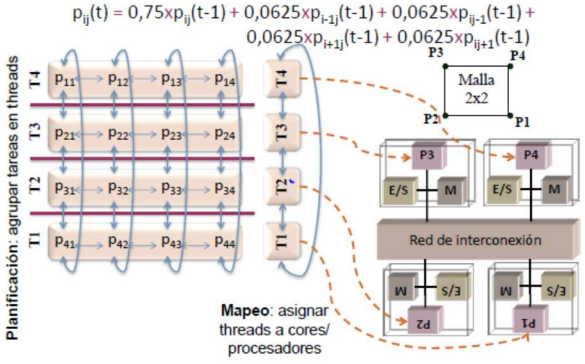
NOTA: La variable i se supone inicializada a 0

Mapeo de procesos/threads a unidades de procesamiento.

Normalmente, el mapeo de threads (light process) es un trabajo que se deja al SO, aunque también lo puede hacer el entorno o el sistema en tiempo de ejecución (runtime system de la herramienta de programación).

Aun así, el programador puede influir en el mapeo a las unidades de procesamiento.

Asignación (planificación + mapeo) Ej : Flitrado de imagen



Códigos del filtrado de imagen

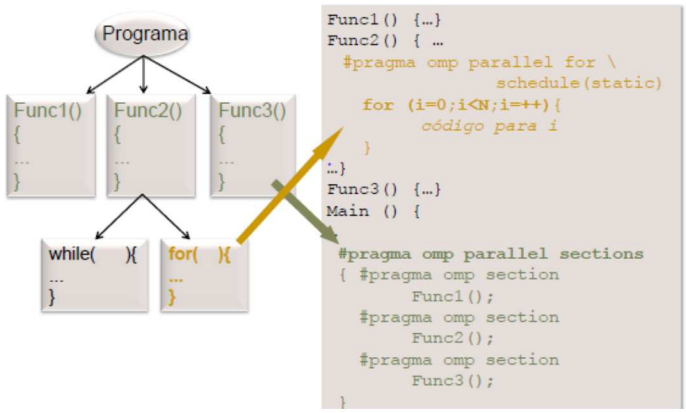
Descomposición por columnas

```
#include <omp.h>
...
omp_set_num_threads(M)
#pragma omp parallel private(i)
{
   for (i=0;i<N;i++) {
      #pragma omp for
      for (j=0;j<M;j++) {
         pS[i,j] = 0,75*p[i,j] +
                   0,0625*(p[i-1,j]+p[i,j-1]+
                           p[i+1,j]+ p[i,j+1]);
      }
   }
}
...
```

Descomposición por filas

```
#include <omp.h>
...
omp_set_num_threads(N)
#pragma omp parallel private(j)
{
   #pragma omp for
   for (i=0;i<N;i++) {
      for (j=0;j<M;j++) {
         pS[i,j] = 0,75*p[i,j] +
                   0,0625*(p[i-1,j]+p[i,j-1]+
                           p[i+1,j]+ p[i,j+1]);
      }
   }
}
...
```

Ejemplo de asignación estática del paralelismo de tareas y datos con OpenMP



3. Redactar el código paralelo

El código va a depender del **estilo** de programación que se utilice (variables compartidas, paso de mensaje o paralelismo de datos), del **modo** de programación (SPMD, MPMD o mixto), del **punto de partida** (versión secuencial, descripción del problema), de la **herramienta** que usemos para hacer el paralelismo explícito (lenguaje de programación, directivas, bibliotecas o compilador) y de la **estructura** (descomposición de datos, granja de tareas, descomposición de datos o divide y vencerás), que depende a su vez de la aplicación.

Habrá que añadir o utilizar en el programa las funciones, directivas o construcciones del lenguaje que hagan falta para:

- **Crear y terminar** procesos o hebras, o en caso de que se creen de forma estática, enrolar y desenrolar procesos o hebras.
- **Localizar** el paralelismo
- **Asignar la carga** conforme al decisión tomada en el paso anterior
- **Comunicar y sincronizar** los procesos o hebras.

4. Evaluar prestaciones

Una vez redactado el programa, se evalúan las prestaciones para comprobar que las prestaciones alcanzadas eran las que se requerían.

Si las prestaciones alcanzadas no son las que se requieren, deberemos volver a etapas anteriores de la implementación para elegir por ejemplo, otra herramienta, realizar un mejor reparto de la carga, o una hacer una mejora en la descomposición de tareas.