www.asesacademia.com

ASES

assessoria d'estudis superiors

PAR

DOSSIER PARCIAL

Tema 1 - Introducción al paralelismo

Limites del Hardware

Llego un punto en la historia donde empiezan a existir limites físicos (minituarización, disipación de calor, etc..) donde el coste de desarollar nuevos procesadores que fueran más rapido era muy caro.

Como las mejoras de arquitectura HW se han agotado, toco que la mejora de rendimiento fuera a nivel software, asi nació el paralelismo y se empezo a construir hardware y software especifico para trabajar con diferentes unidades de proceso.

Paralelismo vs concurrencia

Hay varias aproximaciones/paradigmas para programar:

- Ejecución serie: 1 CPU que procesa en orden las instrucciones de un programa.
- Ejecución concurrente: Varias aplicaciones/tareas **comparten** la CPU.
- Ejecución paralela: Una aplicación se divide en tareas y se ejecuta en diferentes CPU's, **compartiendo memoria**.

Tenemos las siguientes características:

- La ejecución concurrente aporta rendimiento (*throughput*), porque ejecuta diferentes aplicaciones (no relacionadas *a priori*).
- El paralelismo aporta rendimiento a un único programa.
- **Siempre hay que garantizar que el resultado de los tres programas es el mismo.**

Procesos y threads

Para poder ejecutar programas de forma paralela, el HW y los sistemas operativos dan soporte a los **threads.**

- Un thread solo requiere de ProgramCounter y una pila.
- Comparte memoria (variables, código fuente) y estructuras del SO con el padre(PID, tabla de canales, etc).
- Los procesadores tienen HW que ayuda a ejecutar los threads y su contexto.

Los threads se diferencian de los procesos ya que los procesos no comparten memoria.

Cada sistema operativo tiene su interfaz y su peculiaridades con los threads. POSIX threads es la interfaz más extendida.

POSIX Threads

- **P**ortable **O**perating **S**ystem **I**nterface Uni**x**.

Crear un thread **cuesta mucho menos que crear procesos**.

Para crear threads hay que especificar que código, los argumentos del código y los atributos del thread.

➡️`int pthread_create(*thread_id, *atribute, void*(* routine)(void *), void * arg)`

➡️ `int pthread_join(pthread_t thread, void **retval);`

Problemas del paralelismo

Transformar un código para que sea paralelo no es tan fácil, aparecen problemas. El problema viene por **el orden de las escrituras y las lecturas**. Si solo hay lecturas no hay problemas.

Condición de carrera - data race

- Se da sobre las variables compartidas.
- Todos los threads acceden y modifican "a la vez", cualquier resultado es posible.
- La solución pasa por **sincronizar las operaciones de escritura (con locks)** o usar **operaciones atómicas**.

Inanición - starvation

- Se produce cuando un thread **no puede acceder a un recurso compartido**, siempre esta ocupado.
- El thread se bloquea durante mucho tiempo y no puede avanzar.
- No hay solución, se puede mitigar **balanceando la carga de trabajo** entre los threads.

Abrazo mortal - dead lock

- Dos o más threads se esperan mutuamente.
- El programa se queda en una espera infinita.
- El programado debe controlar el orden de las reservas de los recursos.

Live lock

- Variante del dead-lock; los threads no se esperan mutuamente sino que saltan de estado continuamente

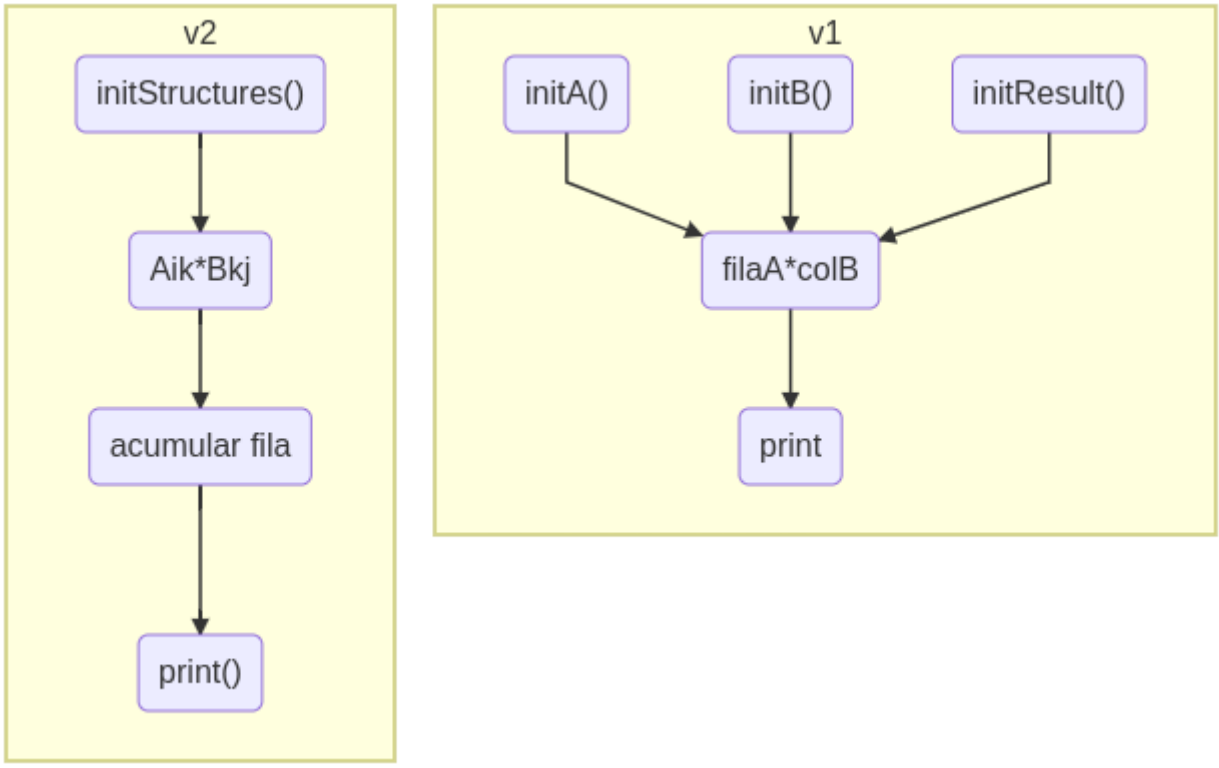
Abstracción de funcionalidades

Es muy útil poder ser capaz de abstraer las funcionalidades del código en un grafo, donde las aristas indican una dependencia de datos y un nodo representa una función.

Por ejemplo, podemos descomponer este código en el siguiente grafo:

```c
//return a pointer to the matrix at i, j
int point_matrix(int* base, int i, int j, int ncols);

int main(){
    init_A();
    init_B();
    init_result();
    for(int i = 0; i < rA; i++){            //for rowsA
        for(int j = 0; j < cB; j++){        //for colsB
            *(point_matrix(res, i, j, c)) = 0;
            for(int k = 0; k < cA; k++){    //for colsA
                //res[i][j]+=A[i][k]*B[k][j];
                int Aik = (*(point_matrix(A, i, k, c)));
                int Bkj = (*(point_matrix(B, k, j, c)));
                int aux =  Aik * Bkj;
                *(point_matrix(res, i, j, c)) += aux;
            }
        }
    }
    print_matrix(res, &r, &c);
}
```



La idea es abstraernos de las funcionalidades que hacen las tareas, solo nos importa:

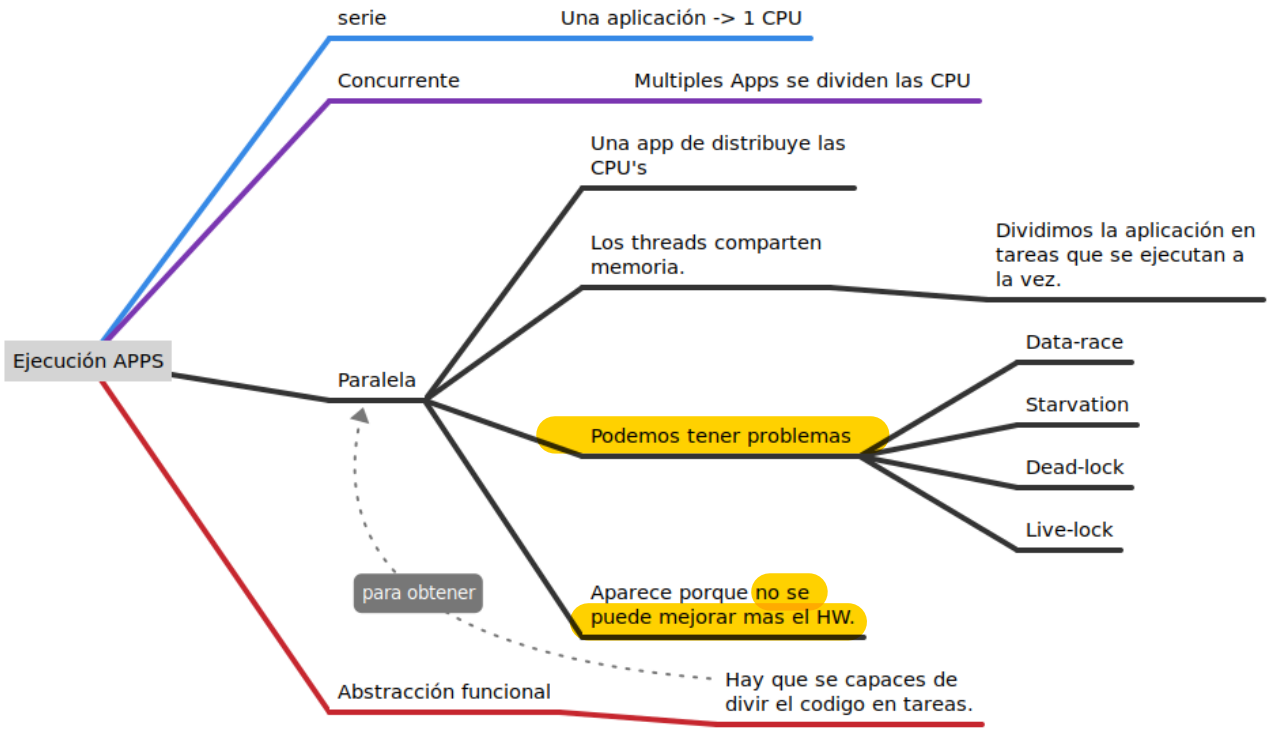
- **Las tareas, no lo que hacen**
- **El tiempo de las tareas**
- **Dependencias de datos y conflictos**

Cada abstracción nos llevara a diferentes estrategias de descomposición.

Que hay que aprender ?

- Diferenciar entre paralelismo, concurrencia y ejecución serie.
- Diferenciar threads y procesos. Definir threads.
- Problemas en ejecuciones paralelas.
- Abstracción de funciones.

Mind-map T1



Tema 2 - Medir el paralelismo

> Version 2.0 - 4/03/2021

La descomposición de tareas y sus dependencias determinan el potencial paralelo de nuestra aplicación, para cuantificarlo existen varias métricas:

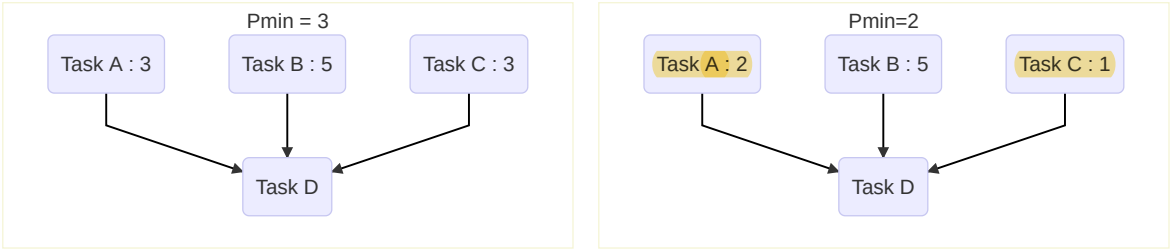
Métricas y definiciones

- T_1 Tiempo de ejecución secuencial (1 procesador).
- T_p Tiempo de ejecución paralela (P procesadores).
- T_∞ Tiempo con recursos infinitos (∞ procesadores).

> Siempre tendremos que T_p es mayor que T_∞ y P. Incluso puede ser mayor que T_1

- $P = \frac{T_1}{T_\infty}$ Ganancia potencial (con infinitos recursos).
- P_{min} := Mínimo de procesadores para tener todo el paralelismo que nos permiten las dependencias.

> En el ejemplo, podemos ver como pasamos de "necesitar" 2 procesadores a 3. La idea es ver como en el caso de la derecha "caben" a la vez la tarea B y las otras dos y en el caso de la izquierda no.



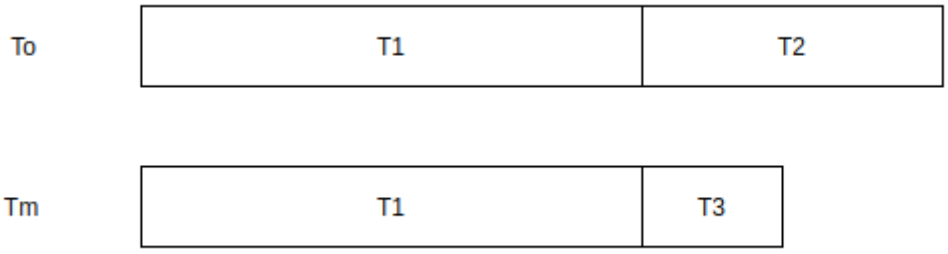
- $SpeedUp(S_p) = \frac{T_1}{T_p}$ Ganancia con p procesadores.
- $Efficiency(Eff_p) = \frac{S_p}{P}$ Eficiencia del paralelismo.

Ley de Amdhal

En general, nos dice donde hay que focalizar los esfuerzos para reducir el tiempo de ejecución de una aplicación.

Demostración ley Amdhal general

Tenemos una aplicación donde el tiempo se divide en dos grandes bloques.



Podemos calcular la ganancia de Tm sobre To, buscando que T_1 y T_3 sean función de T_o para simplificar.

$$G = \frac{T_o}{T_m} = \frac{T_o}{T_1 + T_3}$$

$$F_m = \frac{T_2}{T_0} \text{ (Fracción de tiempo de la mejora) } ; T_1 \rightarrow (1 - F_m) * T_0$$

$$G_m = \frac{T_2}{T_3} \text{ (Mejora) } ; T_3 \rightarrow T_2 / G_m$$

$$G = \frac{T_o}{(1 - F_m) * T_0 + (T_2 / G_m)}$$

$$T_2 = T_0 * F_m$$

$$G = \frac{T_o}{(1 - F_m) * T_0 + (T_0 * F_m / G_m)}$$

$$G = \frac{1}{(1 - F_m) + \frac{F_m}{G_m}}$$

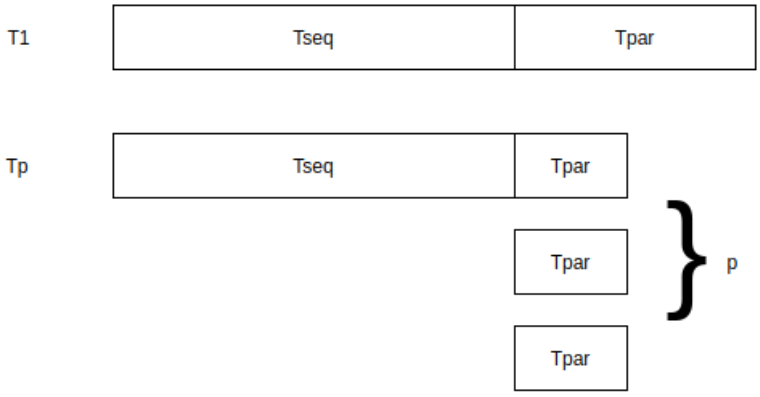
De la última expresión podemos ver:

- Con una gran mejora (alto G_m, aplicamos $\lim_{G_m \rightarrow +\infty}$) la ganancia dependen de de la fracción de tiempo donde se aplique $(1 - F_m)$.

Aplicado al paralelismo, permite decir cuan rápido podemos ir si paralelizamos una aplicación:

Demostración Ley Amdhal

Sobre una aplicación secuencial que tarda T_1, podemos reescribir el tiempo en función de la parte paralela:



$$T_1 = T_{seq} + T_{par} \text{ (tiempo que se puede paralelizar)}$$
$$\varphi := \text{parte que podemos paralelizar}$$
$$1 - \varphi := \text{parte que no podemos paralelizar}$$
$$\varphi = \frac{T_{par}}{T_1}$$
$$T_1 = (1 - \varphi) * T_1 + \varphi * T_1$$

Con esta expresión podemos buscar el tiempo con P procesadores "ideal", donde el tiempo paralelo se distribuye en P procesadores:

$$T_p = T_{seq} + (\frac{T_{par}}{p})$$
$$\text{Substituyendo en funció n de } \varphi \text{ tenemos:}$$
$$T_p = (1 - \varphi) * T_1 + (\frac{\varphi * T_1}{p})$$

Y podemos buscar el *speed-up* en función de φ, donde T_1 desaparece:

$$S_p = \frac{T_1}{T_p} = \frac{T_1}{(1 - \varphi) * T_1 + (\frac{\varphi * T_1}{p})}$$
$$= \frac{1}{(1 - \varphi) + \frac{\varphi}{p}}$$

Podemos extraer:

- Si no hay fracción paralela ($\varphi \to 0$), el S_p tiende a 1 (no hay mejora).
- Si todo el código es paralelo ($\varphi \to 1$), el S_p tiende a P (numero de procesadores).
- Llega un punto donde la mejora ya no depende del número de procesadores que usemos: $\lim_{p \to +\infty} = \frac{1}{1-\varphi}$
- **Las mejoras en una aplicación paralela están limitadas por el tiempo que la aplicación funciona en paralelo (obvio).**

Escalado fuerte vs débil

Dos formas diferentes de evaluar nuestra aplicación. Són modelos de comportamiento que pueden tener nuestra aplicación/programa

- ***Strong scalability*** **: Ir más rápido si añadimos más threads.**
  - El objetivo es reducir el tiempo de ejecución.
  - Augmentando el numero de procesadores/threads, con el mismo tamaño de problema.
- ***Weak scalability*** **: Ir igual de rápido si subimos el tamaño del problema (mantenemos la carga por thread)**
  - El objetivo es resolver problemas más grandes, sin que el tiempo suba mucho.
  - Augmentar el numero de procesadores y el tamaño del problema.

Tema 3 - OpenMP

Version 4.0 -28/03/2021

- Por defecto, las directivas pragma afectan a la siguiente línea de código, podemos usar { } para asignar un bloque.
- Librería para trabajar con múltiples threads, hay que compilar con la flag -fopenmp.
- El compilador define la macro #_OPENMP.
- Por defecto, todas las variables globales son compartidas. Las variables declaradas dentro de bloques OpenMP son privadas.
- Utiliza un modelo fork-join (crear threads y esperar a que acaben).

OpenMP Calls & types

Llamadas y tipos de la librería:

```
//Número de threads de la región actual.
int omp_get_num_threads();
//Id del thread [0 - numThreads-1]
int omp_get_thread_num();
//Indica el número de threads a utilizar
void omp_set_num_threads(int n);
//Retorna el máximo número de threads
int omp_get_max_threads();
//Tiempo actual
double omp_get_wtime();
//Usar normalmente con:
double start = omp_get_wtime();
double end = omp_get_wtime();
printf("Work took %f seconds\n", end-start);

//Locks:
omp_lock_t lock;        //type of a lock
void omp_init_lock(&lock);    //initialize a lock
void omp_set_lock(&lock);
void omp_unset_lock(&lock);
int omp_test_lock(&lock);     //won't block the thread, 0 if fail to
acquire the lock
void omp_destroy_lock(&lock);
```

Constructores básicos

```
#pragma omp parallel [Clausulas]
```

- Crea una región paralela con los threads que podemos indicar con:
  - variable de entorno OMP_NUM_THREADS
  - Llamada a `void omp_set_num_threads(int n);`
  - Clausula num_threads()
- Tiene una barrera implícita al final.
- Clausulas
  - num_threads(N): Ignora el número de threads indicado fuera y la región paralela tendrá N threads.
  - if(exp_bool): Si la expresión evalúa falso, solo se creará un thread en la región.
  - shared(var-list): Todos los threads ven la misma variable
  - private(var-list): Todos los threads tienen una variable privada (**no inicializada!**).
  - firstprivate(var-list): Todos los threads tienen una variable privada inicializada.
  - reduction(<op>:var-list): Define cómo se efectuará (+, -, *, /, ^) el join cuando todos los threads acaben. Las variables de la lista se privatizan automáticamente.

```
#pragma omp single [Clausulas]
```

- La región de código sólo se ejecuta por un thread de la región paralela, los demás esperan.
- Tiene una barrera implícita al final
- Clausulas
  - private, firstprivate, shared
  - nowait : La barrera implícita desaparece y el resto de threads avanzan.

```
#pragma omp parallel
{
      #pragma omp single
      {
      printf("Hi! I'm thread %d \n", omp_get_thread_num());
            foo();
      }
      printf("Hi! I'm thread %d \n", omp_get_thread_num());
}
```

Constructores sincronización

```
#pragma omp barrier
```

- Todos los threads de la región se esperan (sincronizan) en este punto. Una vez llegan todos siguen la ejecución.
- Las barreras implícitas de *single* i *parallell* son de este tipo.

```
//all foo occurrences after all bar occurrences:
#pragma omp parallel num_threads(3)
{
	foo();
	printf("Thread %i end foo()", omp_get_thread_num());
	#pragma omp barrier
	printf("Thread %i starting foo()", omp_get_thread_num());
	bar();
}
```

```
#pragma omp critical [(name)]
```

- Soporte para la exclusión mútua. Solo habrá un thread a la vez en todas las regiones no nombradas.
- Podemos añadir un nombre a la zona de exclusión mutua para esa región en concreto.
- Los critical pueden sustituirse con los locks.

```
for(int i = 0;i < N; i++){
	if(i %2){
		#pragma omp critical(y)
		even++;
	}
	else{
		#pragma omp critical(x)
		odd++;
	}
 }
```

```
#pragma omp atomic [ update | read | write ]
```

- Asegura lecturas, escrituras y actualizaciones son atómicas.
- Más eficiente que el critical usualmente.

Constructores worksharing

```
#pragma omp for [clausules]
```

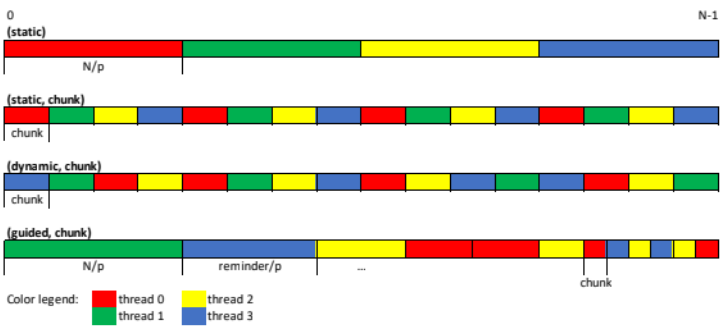
- Distribuye las iteraciones del siguiente bloque entre todos los threads de la región paralela. **El número de iteraciones tiene que ser conocido.**
- Clausulas
  - private, firsprivate, shared, reduction, nowait.
  - schedule[(type, [N])]: indica cómo distribuir las iteraciones
  - collapse(n): En bucles perfectamente anidados (sin nada enmedio y con el número de iteraciones definido en compilación), podemos juntar los N primeros loops.

```
//En el ejemplo, se juntan en un solo bucle las i, j
#pragma omp for collapse(2) schedule(static, 2)
for(int i = 0; i < N; ++i)
      for(int j = 0; j < N; ++j)
            a[i][j] = b[i][j] + c[i][j]
```

  - ordered(n): Indica que hay que ordenar las iteraciones de los n bucles anidados. Para indicar las dependencias dentro del código, hay que usar:
    - depend(sink: expr) : Depende de la iteración resultante de expr (i-1).
    - depend(source) : Depende de toda la iteración actual.

Tipos de scheduling:

- **static:** Las iteraciones se dividen en bloques de tamaño N/num. threads. sha
- **static, N :** Las iteraciones se dividen en bloques de tamaño N.
- **dynamic[, N]:** Las iteraciones son "pedidas" por cada thread. Si no le añadimos la N, N = 1.
- **guided**: Variante de *dynamic* , el tamaño se reduce conforme al número restante.



Tareas explícitas

Podemos generar tareas de manera explícita. Cuando un thread llega a un constructor de tarea se añade a la *pool* de tareas a la espera de que algún thread libre la ejecute.

```
#pragma omp task
{
     bloque codigo
}
```

- Se suele usar dentro de un *single*, normalmente solo queremos crear una tarea una vez.
- Clausulas:
    - shared, private, firstprivate
    - if(exp) Si exp == False, la tarea se crea y se ejecuta por el thread "que encuentra el pragma omp task".
    - final(exp) : Si exp == True, la tarea **y todas sus descendientes estarán marcadas como final** y se ejecutará secuencialmente por el thread que la crea (la estructura de la tarea se genera igualmente), se llama *included task*. Podemos ver si estamos en una tarea final con la llamada `bool omp_in_final()`.
    - mergeable : Aplicado a una tarea final, no crea la estructura de la tarea. (Realmente hace que se ejecute en el mismo espacio que la tarea que la crea).
    - depend : Marca las dependencias para las tareas. Las variables pueden estar dentro de un array.
        - depend(in : list-vars) : La tarea depende de un cálculo de la variable previa.
        - depend(out : list-vars) : El cálculo de las variables en esta tarea será dependiente para otra tarea.
        - depend(inout : list-vars) : Igual que depend(out)

Podemos tener puntos de sincronización en las tareas:

```
#pragma omp taskwait
```

Suspende el thread a la espera que acaben las tareas hijas de la actual tarea.

```
#pragma omp taskgroup
```

Suspende el thread a la espera que acaben todas las tareas descendientes de la actual tarea. En el *taskwait* solo T1, T2 y T4 están garantizados que acaben T1, T2 y T4. En el *taskgroup* T2, T3 y T4 están garantizados a acabar.

```
#pragma omp task {}      // T1
#pragma omp task         // T2
{
    #pragma omp task {}  // T3
}
#pragma omp task {}      // T4

#pragma omp taskwait
```

```
#pragma omp task {}          // T1
#pragma omp taskgroup
{
    #pragma omp task         // T2
    {
        #pragma omp task {}  // T3
    }
    #pragma omp task {}      // T4
}
```

Ac

Podemos controlar cuantas tareas generamos en un bucle:

```
#pragma omp taskloop [clausulas]
```

- Clausulas
    - shared, private, firstprivate, if, final, mergeable
    - grainsize(m):
    - num_tasks(n): Indica el número de tareas totales a generar.
    - collapse(n): Indica el número de loops que queremos juntar.
    - nogrpoup: Elimina el taskgroup implicit.

Barreras memoria - (no aplica al primer parcial)

```
#pragma omp flush (var-list)
```

OpenMP no asegura consistencia todo el tiempo (mantiene un modelo relajado). Para forzar una variable a ser leída de memoria principal hay que usar el flush, de otro modo se puede leer/escribir de forma privada en el thread (thread temporary view).

- Los constructores de sincronización tiene un flush asociado

Ejemplos - OpenMP

Cálculo de media - For worksharing

En el siguiente ejemplo vamos a paralelizar el cálculo de una media aritmética, usando varias estrategias que OpenMP nos proporciona.

```
vector<int> myVector(N);
//First version -> using reduction
#ifdef V1
int sum=0;
int n=0;

#pragma omp parallel
#pragma omp for reduction(+:sum, n)
for(auto it: myVector){
      sum += it;
      n+=1;
}
#endif
//Second version -> shared/priv variables
#ifdef V2
int sumPriv = 0, nPriv = 0;
int sum = 0, n = 0;
#pragma omp parallel
{
      #pragma omp for firstprivate(nPriv, sumPriv)
      for(auto it: myVector){
          sumPriv += it;
          nPriv+=1;
      }
      #pragma omp critical(sum)
      sum += sumPriv;
      #pragma omp critical(n)
      n += nPriv;
}
#endif

cout << "Result: " << sum/n << endl;
```

Cálculo de media - Tasks worksharing

Hay que llamar a la función dentro de una región paralela con un single.

```
void calcula_suma(const vector<int>& v, int& suma, const int N, const
int chunksize){
	int sumaP;

	for(int i = 0; i < N; i+=chunksize){
	#pragma omp task private(sumaP)
	{
		sumaP=0;
		for(int j = i; j < i+chunksize; ++j){
			sumaP += v[j];
		}
		#pragma omp critical
		suma += sumaP;
	}
	}
	return;
}
```

Cálculo de un histograma

Necesitamos un vector de *locks* para “proteger” los accesos a cada entrada del histograma.

```
vector<int> histograma(MAX_VALUE_HIST, 0);
vector<omp_lock_t> hist_locks(MAX_VALUE_HIST);

 //init locks:
for(int i = 0;i< MAX_VALUE_HIST; ++i){
	omp_init_lock(&hist_locks[i]);
}
void tractarNum(int n, int numtask){

	omp_set_lock(&hist_locks[n]);
	histograma[n]++;
	omp_unset_lock(&hist_locks[n]);
	return;
}
```