

# PEP 1 - Paralelos

## → SIMD

- Instrucción única, múltiples datos.
- Ideal para tareas que requieren operaciones repetitivas en conjuntos extensos de datos.

## \* Instrucciones SSE

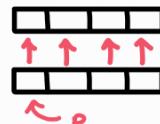
- Registros MMX de 128 bits alojan distintos tipos de datos.

-- m128 d →  $2 \times 64$  bits doubles  
-- m128 →  $4 \times 32$  bits floats  
-- m128i →  $4 \times 32$  bits int  
↳ 8x 16 bits uint



## \* Operaciones

- Carga -- m128 A = -mm-load-ps (float \*p)  
-- m128 A = -mm-store-ps (float \*p)



- Asignación -- m128 A = -mm-set-ss (float p)

- Aritmética -- mm-add-ps (--m128 a, --m128 b)

-- mm-sub-ps (--m128 a, --m128 b)

-- mm-mul-ps (--m128 a, --m128 b)

-- mm-min-ps (--m128 a, --m128 b)

-- mm-max-ps (--m128 a, --m128 b)

Cada posición ai opera con posición bi.

min o max del registro.

## → SISTOPE ↴

- \* Concurrencia → Capacidad de un sistema para manejar múltiples tareas simultáneamente.

↳ Mejor rendimiento, aumento throughput (I/O solo blo-  
quea una hebra) y aumenta eficiencia.

- \* Sincronización → Coordinación de tareas paralelas que comparten recursos comunes.

① Semáforos: Contador para controlar el acceso a recursos compartidos.  
`wait(s)`      `signal(s)`

② Mutex: Mecanismo de sincronización que garantiza el acceso a una sola hebra en el recurso compartido.

`mutex-lock()`    `mutex-tryunlock()`    `mutex-unlock()`

③ Monitor: Encapsulan datos compartidos y sus métodos y solo permite el acceso a una hebra.

④ Variables de Condición: Conjunto de mutexs para permitir que las hebras esperen o señalicen.

## ⑤ Barreras: Indica a las hebras que esperen a que todas lleguen para seguir.

- \* Condición de carrera → El resultado de una operación depende del orden de ejecución de las hebras.
- \* Sección crítica → Trozo de código ejecutado por múltiples hebras, en la que se accede a datos compartidos y al menos una hebra escribe.
- \* Exclusión mutua → Requerimiento sobre una sección crítica.

① E.M: Solo una hebra está a la vez en la SC.

② Sin deadlock: Que las hebras no se queden bloqueadas indefinidamente.

③ Sin inanición: Que todas las hebras logren entrar a la SC.

④ Progreso: Si la SC está libre, debe entrar una hebra.

## → Diseño de Algoritmos Paralelos

### ① Particionamiento

- Divide la computación en tareas más pequeñas.
- Granularidad fina (agresivo al asignar tareas).

datos  
cómputo  
ambos

ej: números primos. La granularidad depende del número asignado (si es mayor es más cómputo). Lo más agresivo es 1 tarea → 1 número.

- No considera el número de procesadores.

### ② Comunicación

- Comunicación coordinada de tareas
  - \* Altamente acopladas: cada 10-100 instrucciones.
  - \* Medianamente acopladas: cada 1000 - 10.000 instrucciones.
  - \* Débilmente acopladas: casi nada.

- Un canal de comunicación implica un costo de tiempo de computación, se debe evitar si es innecesario.

### ③ Aglomeración

- Volver atrás y agrupar en tareas más grandes para mejorar el rendimiento.
- Replicar datos para evitar comunicación.

### ④ Mapeo

- Asignar al recurso computacional.
- Busca minimizar el costo de comunicación y maximizar la utilización de procesadores.

## → $\mu$ C++

- Extensión a C++ con características de concurrencia.

### \* Mutex

- Sus métodos se ejecutan con exclusión mutua.
- Cuando la ejecución de un mutex comienza, el objeto mutex se cierra (lock) y cuando termina se abre.
- Si una tarea invoca un miembro mutex cuando está cerrado, la tarea se bloquea.
- Mutex no tiene hebra, es la hebra de la tarea quien lo ejecuta.

no es necesario  
↓  
miembros privados y protegidos → por defecto nomutex

\_ Mutex M {  
Private:  
char z(...);  
Public:  
M(); → siempre no mutex  
~M(); → siempre mutex  
int f(); → por defecto mutex

### \* Corutina

- Objeto con su propio estado de ejecución.  
**Puede suspend() y resume()**
- No posee hebra de ejecución, la hebra que ejecuta sus miembros hace cambio de contexto al entrar en la corutina. Abandona su estado y toma el último estado de la corutina.
- Sus miembros no poseen exclusión mutua.
- Semi-corutina** → Reactiva la corutina que previamente había activado a la corutina.
- Full-corutina** → Activa explícitamente un miembro de otra corutina, no necesariamente la que había activado.

### \_ Coroutine M {

Private:  
void main();  
Public  
M();  
~M();  
int f(); ↗  
main() {  
M c(); ↗ accede a método  
c.f(); ↗  
} ↗ se activa con un método público al hacer resume().

### \* Monitor

- Todos los miembros públicos son por defecto mutex.  
Sino se debe indicar con -nomutex
- Cada miembro mutex tiene una cola ordenada donde se bloquean las tareas cuando lo invocan en lock.
- Para que las tareas se bloquen dentro del monitor, se declaran variables de condición.
- Scheduling interno** → Planificación de tareas bloqueadas dentro del monitor. Utiliza variables de condición.

**wait()**: Bloquea a la tarea invocadora en la cola de la V.C., se libera el mutex del monitor y se realiza scheduling interno.

**signal()**: muere próxima tarea bloqueada en la cola de la V.C. al tope del stack de aceptación.

### \_ Monitor M {

Private:  
int i;  
Public:  
M();  
~M();  
void x(...);  
- Nomutex int y(...);  
main() {  
M \*m = new M(); ↗

- Scheduling externo → Planificación de tareas bloqueadas fuera, es decir, que solicitaron ingresar al monitor y se bloquearon.
- Cuando el condicional de - When es verdadero, se acepta la próxima tarea en FIFO que está bloqueada en la cola de mutex de - Acept.

### \* Tarea

- Tiene su propia hebra y estado de ejecución.
- Sus miembros públicos poseen EM.
- Al momento de su inicialización se ejecuta main()
- La tarea termina cuando termina la ejecución de main(), la hebra se elimina y la tarea se convierte en un monitor, por lo que se puede recuperar la información a través de sus métodos públicos.

```
- Task T:
  Private:
  Protected:
    void main();
  Public:
    void r();
?
```

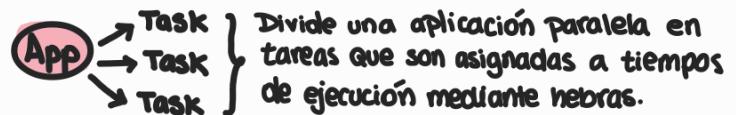
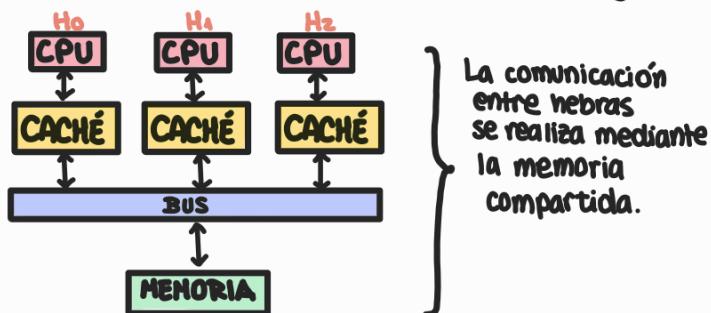
	Hebra	Estado de ejecución	EM
Objeto clase			
Monitor			■
Corutina			
Corutina Monitor	■		■
Tarea	■		■

### \* Otros :

- u Semaphore
- u Lock
- u Barrier

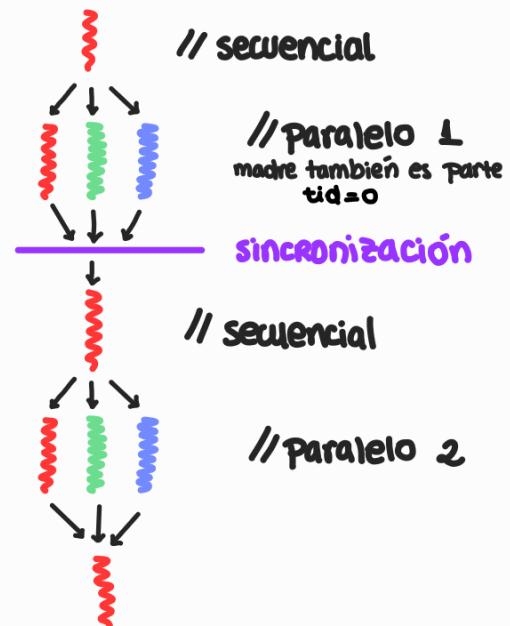
# → OPEN MP

- Estándar para programación paralela en sistemas de memoria compartida.
- Beneficios → Aplicaciones paralelas eficientes y multihebreadas.  
Manipulación de arreglos y matrices grandes.



- Directivas → Código que especifica cómo un compilador debe procesar un bloque de código. \*Pragma `omp nombre_directiva <cláusulas>`
- Cláusula → Modifica el funcionamiento de las directivas. `num_threads()`  
Especifica número de hebras

- La ejecución comienza con la hebra madre
- Inicia la construcción paralela para crear a las hebras hijas
- Al término de la construcción paralela las hebras se sincronizan por la existencia implícita de una barrera.
- La hebra madre es la única que continúa. Se destruye memoria local.



## \* Memoria

```
int val1; → variable global
```

```
void main() {
```

```
    int ntask = 3;      val3 = 2;
    int val3;
```

```
*pragma omp parallel num-threads(ntask) {
    val1 = omp-get-thread-num(); }
```

(...)  $val1 = 2$  { como es una variable global, se sobrescribe el último valor de la última hebra ejecutada.

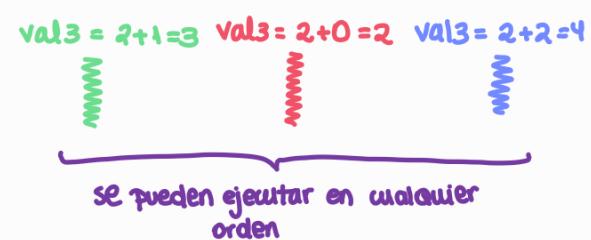
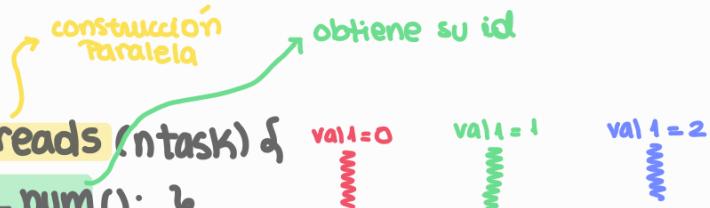
```
*pragma omp parallel num-threads(ntask) {
```

```
    int val2 = omp-get-thread-num();
    val3 = val1 + val2; }
```

(...)

$val3 = 4$  → último valor

MEMORIA PRIVADA  
Propia memoria no accesible de la hebra



- La visión temporal de la memoria que tiene una hebra puede ser distinto al estado actual de la memoria.

\* Pragma `omp flush(var)` → Garantiza que los cambios realizados en las variables compartidas se sincronicen para todas las hebras.

\* Pragma `omp parallel num-threads(n task)`

`a = 10`

\* Pragma `omp flush(a)` → Consistencia entre visión temporal de la memoria y memoria de la variable.  
`(...)`

- ① Primera hebra modifica una variable.
- ② Primera hebra realiza un flush.
- ③ Segunda hebra realiza un flush.
- ④ Segunda hebra lee el valor de la variable.

} Se debe seguir ese orden, ya que por si solo flush no garantiza que lea el valor actualizado.

Cláusulas:

`private(list)` → variable list privadas a cada hebra.

↳ variables locales invocadas por funciones son privadas.

`for(i=0; ...)` var → i

`shared(list)` → variable list es compartida por todas las hebras.

↳ Por defecto todas las variables son compartidas.

## \* Variables de Medio Ambiente

- Seteadas antes de la ejecución del programa.

`OMP_NUM_THREADS` → Define el número de hebras durante la ejecución del programa.

`setenv OMP_NUM_THREADS 16`

`export OMP_NUM_THREADS = 16`

} Para todas las regiones

} Paralelas del programa.

`OMP_SCHEDULE` → Define cómo se dividen las iteraciones de los bucles entre las hebras.

`setenv OMP_SCHEDULE "STATIC, 4"`

`export OMP_SCHEDULE = "STATIC, 4"`

## \* Funciones Runtimes

- Controlar y consultar el medio ambiente de la ejecución paralela.
- Sincronizar acceso a datos compartidos.

`int = omp_get_num_threads();` → Número de hebras totales

`int = omp_get_thread_num();` → obtiene el tid

## \* Directiva Parallel

- Cláusulas:

① `if (expresión regular)`: El bloque se ejecutará solo si la expresión es verdadera.

② `private (lista variables)`: Declara variables privadas.

- ③ `firstprivate` (lista variables): Asigna a las variables una copia del valor que tenían antes de la región paralela.
- ④ `shared` (lista variables): Declara variables compartidas.
- ⑤ `default (none)`: Obliga que todas las variables sean definidas como `private` o `shared`.
- ⑥ `copyin` (lista variables): Copia valores en variables privadas desde la memoria compartida.
- ⑦ `reduction (operador)` : Cada hebra realiza operaciones sobre su copia de la variable y al final se combina en una variable con el operador.
- ⑧ `num-threads (expresión)` : Define el número de hebras a ejecutar.

• Para determinar el número de hebras, revisa las siguientes re las en orden:



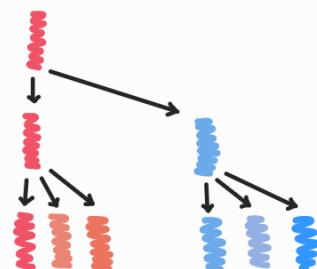
### \* Paralelismo Anidado

- Si una hebra de un equipo 1 encuentra una construcción `parallel`, crea otro equipo 2 y pasa a ser la hebra maestra del equipo 2.
- Se activa con `OMP_NESTED = TRUE` o `omp-set-nested()`

`omp-set-nested(1);`

\* Pragma `omp parallel num-threads(2) {  
(...)  
}`

\* Pragma `omp parallel num threads(3) {  
(...)  
}`



\* `hebras > CPU` => Peor rendimiento

### \* Compartición

#### ① Predeterminada → Antes de comenzar construcción (explícito)

- `theadprivate` (persiste luego de \*Pragma).
- declaración dentro de scope (`private`) \*
- almacenamiento dinámico (`shared`) \*
- Variables en `for` o `parallel for` (`private`) \*
- almacenamiento estático declarado dentro de scope (`shared`) \*

Siempre es compartido porque está en el heap.

```

→ char *p = malloc(10);
*pragma omp parallel num-threads(5) {
    → int tid;
    → static int j;
    for (int i)
        ↑
    }
    
```

② **Explícita** → Aparece en la lista de una cláusula de compartición.

\*pragma omp parallel private (i)

③ **Implícita** → No aparece en la lista de una cláusula de compartición, no tiene atributos predeterminados.

- si no aparece la cláusula default es shared.
- La variable hereda el atributo que tiene en el scope mayor.

default (shared) → Variables con atributos implícitos son shared.

default (none) → NO asume referencias para variables. Se debe:

- declarar explícitamente en cláusula.
- declarar fuera de \*pragma.
- objeto const.
- variable que controla loop.
- thread private.

• Hay que tener cuidado con variables shared por la consistencia relajada.

int x;

\*Pragma omp parallel shared(x) {

    if (...): x = 5; → 1 hebra cambia valor. No todas lo ven

\*Pragma omp barrier {

    if (!...): print(x);

↳ sincroniza la visión local de cada hebra => flush  
con la memoria global.

## \* Repartición

- Utiliza hebras ya existentes. Se usa dentro de parallel.
- Barrera implícita por defecto.

### ① Directiva for

- Iteraciones ejecutadas en paralelo.
- Distribuye iteraciones entre hebras.
- Para definir qué iteraciones le tocan a cada hebra, existe la cláusula:

schedule (kind, size)

\*pragma omp parallel private (j)

\*pragma omp for

for (j=0 ; j < N ; j++)

Divide largo arreglo entre hebras.

static: iteraciones / size y se asigna estéticamente con Round Robin.

Si no se indica size, se hacen porciones iguales en tamaño.

Asigna antes de ejecución.

dynamic: a cada hebra se le asigna su porción size en forma dinámica a la que esté disponible, cuando termina solicita otra porción.

guided: dynamic, pero las porciones se reducen exponencialmente hasta llegar a size.

nowait → Elimina la barrera implícita. Las hebras no esperan que las otras terminen el for para continuar.

## ② Directiva section

- Cada bloque es ejecutado por una hebra.

\* Pragma omp parallel

~~E~~ ~~S~~

\* pragma omp sections

\* pragma omp section

~~S~~

\* pragma omp section

~~S~~

## ③ Directiva single

- Bloque se ejecuta solo por una hebra.
- Posee una barrera implícita

\* pragma omp parallel

~~E~~ ~~F~~ ~~S~~

\* Pragma omp single

~~S~~

## ④ Construcciones de sincronización

\* Pragma omp master → Bloque ejecutado solo por hebra maestra.  
↳ No tiene barrera implícita.

\* pragma omp critical → Bloque que permite el acceso a solo una hebra.

\* pragma omp barrier → Barrera.

\* Tareas \* pragma omp task → *no necesariamente dentro de parallel*

- Cuando una hebra encuentra un task, crea la tarea y planifica.
- Cuando una hebra comienza la ejecución de la tarea, la ejecutará hasta el final (no necesariamente es atómica).

## → Rendimiento

- Costo computacional.
- Tiempo de ejecución

$$T = I_c \cdot CPI \cdot \gamma$$

- Tiempo procesador  $\gamma$

- Frecuencia del reloj  $f = \frac{1}{\gamma}$  [GHz]

- Número de instrucciones  $I_c$

- Promedio de ciclos por instrucción CPI

## \* Speedup

- Medir el rendimiento de un programa paralelo.

$$S_{\text{asintótico Real}} = \frac{T_{\text{mejor s(n)}}}{T_p(n)}$$

$$S_{\text{absoluto}} = \frac{T_{\text{mejor s y Preloz}}}{T_p}$$

$$S(n) = \frac{T_s}{T_p}$$

- $S_{\text{real}} = \frac{T_{\text{metor}}}{T_p}$

- $S_{\text{relativo}} = \frac{T_s}{T_p}$

↳ no necesariamente el mejor algoritmo

## \* Escalabilidad

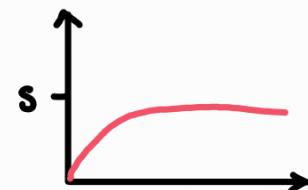
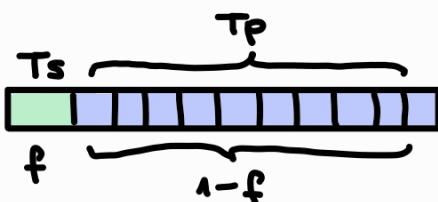
- Speedup límite → Máximo alcanzable

↳ Llega un momento que añadir más P no mejora en nada. (tiempo de sincronización, comunicación, ...)  
→ No siempre mejora indefinidamente.

- Un programa es escalable cuando su rendimiento mejora a medida que el tamaño del sistema crece (procesadores o tamaño del problema).

## • Ley de Amdahl

$$S(n) = \frac{T_s + T_p/n}{T_s + T_p}$$



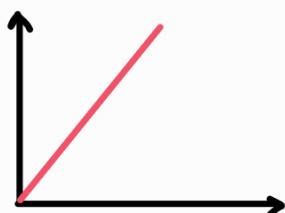
$$f = \frac{T_s}{T_s + T_p}$$

fracción secuencial

- El Speedup está limitado por la fracción secuencial  
 $S(n) \leq 1/f$

## • Ley de Gustafson

$$S(n) = \frac{T_s + n T_p}{T_s + T_p}$$



## • Eficiencia

$$E(n) = \frac{S(n)}{n}$$

→ carga fija  $E(n) \leq 1$

→ carga variable puede  $E(n) > 1$