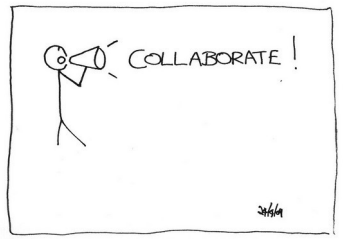
Programación de Arquitecturas Emergentes

Tema 3: Introducción a la programación paralela - OpenMP

Intel Launches 4th Gen Xeon Scalable Processors (formerly codenamed Sapphire Rapids) - Enero 2023

Intel® **Advanced Matrix Extensions** (Intel® AMX) accelerates deep learning (DL) inference and training workloads, such as natural language processing (NLP), recommendation systems, and image recognition.



Pablo Quesada Barriuso <pablo.quesada@usc.es>
Department of Electronics and Computer Engineering
University of Santiago de Compostela

- **Programación de sistemas de memoria compartida**
  - Extensiones del lenguaje de programación
  - Sistemas operativos
  - Librerías

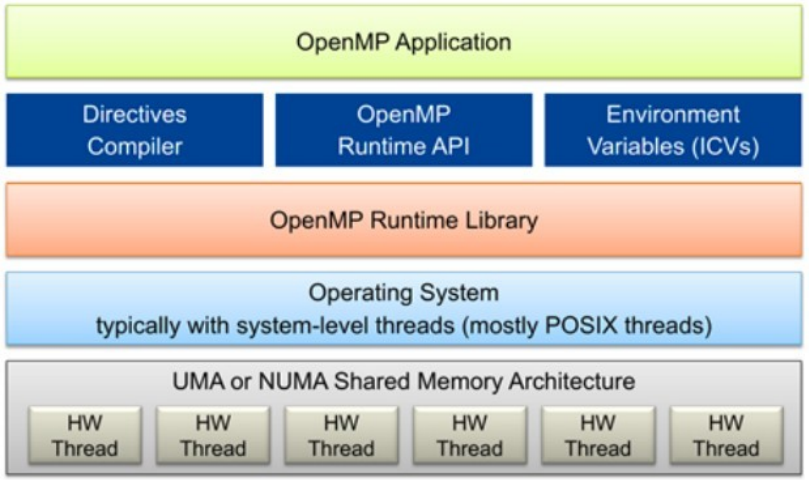
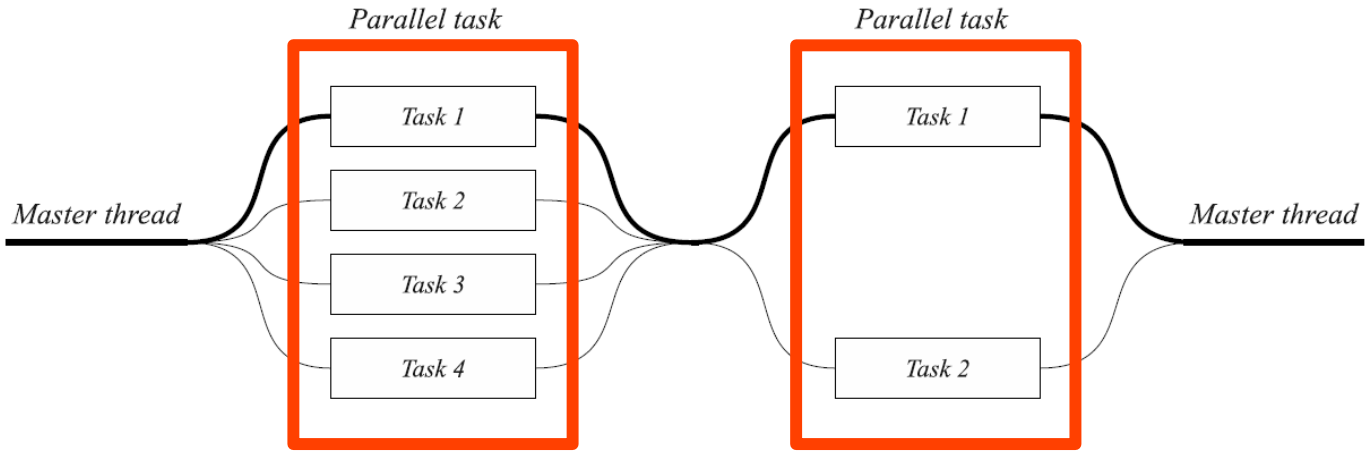


Diagrama by Jih-Wei Liang

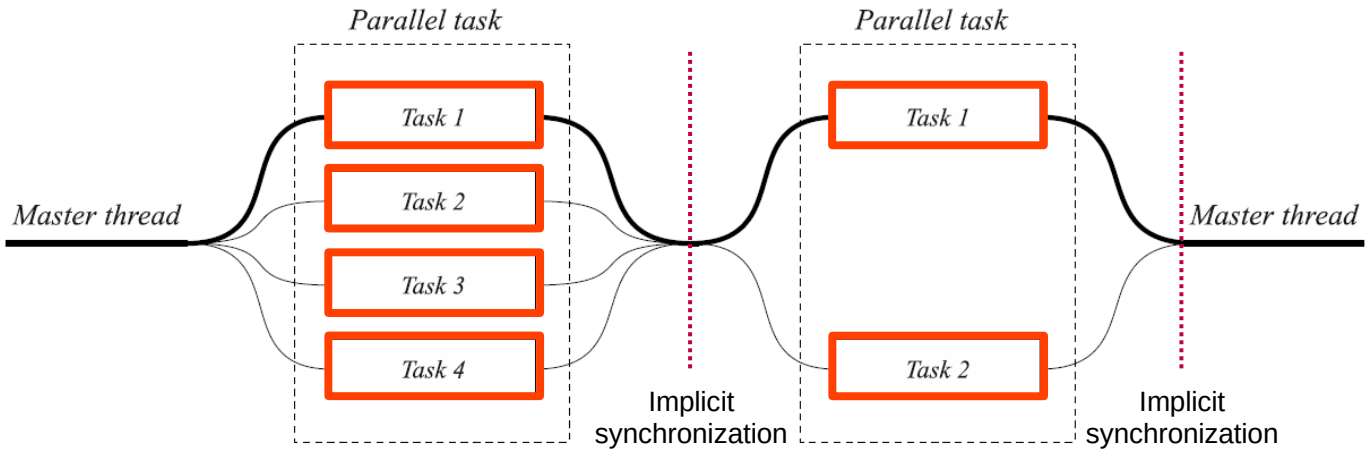
- **OpenMP**
  - **API** de facto para programación paralela en C/C++ y Fortran
  - Fortran, C y C++
  - Incorpora los mecanismos básicos (**creación** de tareas, **comunicación** y **sincronización**)
  - Compuesto de **directivas** de compilación, **funciones** de librería y **variables** de entorno

- **Filosofía de OpenMP**
  - Paralelizar el código secuencial usando directivas del compilador llamadas *pragmas*
  - El compilador genera las instrucciones necesarias
    - Portabilidad (Multiplataforma)
- **Facilidad de uso**
  - Aceleración significativa con pocas directivas
- **Implementación depende del fabricante**
  - No está orientado al hardware
  - No garantiza uso eficiente de la memoria compartida
  - No garantiza 1 thread por core

- Modelo de ejecución ***fork-join***
  - Un programa en OpenMP comienza con un hilo de ejecución (***initial thread***) sobre un dispositivo anfitrión (*host device*)
  - Cuando un thread encuentra una construcción paralela, ese thread crea un equipo (***team***) de uno o más hilos, incluido el mismo, y se convierte en el hilo maestro (***master thread***)
  - Múltiples hilos ejecutan tareas definidas de forma implícita o explícita mediante directivas



- **Construcción paralela (parallel regions)**
  - Se **crea un conjunto de tareas** de forma implícita, ***una por thread***, donde el código de cada tarea está definido por el código dentro de la región paralela – *mismo código para cada thread*
  - Cada tarea **se asigna a un thread diferente** en el equipo y queda atada (***tied***) a ese thread
  - Hay una **barrera implícita** al final de la región paralela

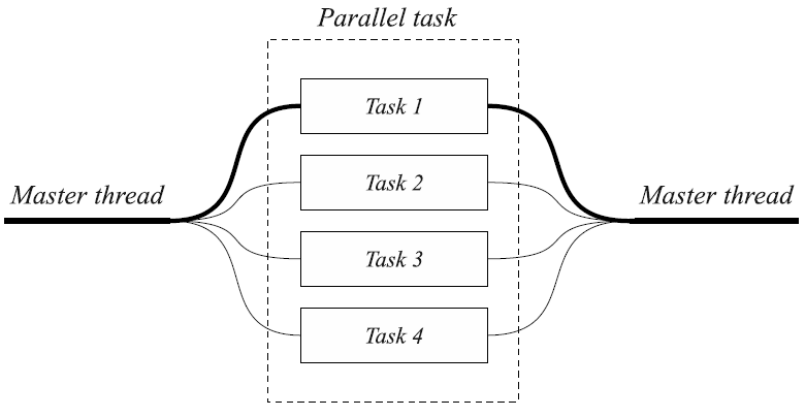


- **Modelo de memoria en OpenMP**
  - Todos los threads en OpenMP tienen **acceso a un espacio** para almacenar y recuperar variables, llamado '**memoria**'
  - Cada hilo puede tener su propia **vista temporal de la memoria**
    - Estructura intermedia como registros de la máquina, caché, o almacenamiento local
  - La vista temporal permite a un hilo almacenar variables y evitar el acceso a memoria principal
  - Cada hilo tiene un tipo de memoria a la que no pueden acceder otros threads, llamada **threadprivate memory**

- **Modelo de memoria en OpenMP**
  - Los **accesos** a las variables **no** se garantiza que sean **atómicos**
  - Si múltiples threads escriben en la misma posición de memoria (o un thread lee de una posición de memoria y otro escribe en la misma posición) ***sin sincronización***, se produce una ***condición de carrera*** (race condition)
  - El modelo de memoria tiene una ***consistencia relajada*** porque la vista temporal de la memoria no se requiere que sea consistente con la memoria en todo momento.
  - Las escrituras y las lecturas a una posición de memoria pueden adelantar a otras lecturas y escrituras

- **Creación de tareas a través de pragmas**
  - Formato de las directivas de compilación
    - siguen estándares para directivas de compilación en C/C++
    - distingue mayúsculas y minúsculas
    - se aplica a la línea de código siguiente (o bloque {})
    - directivas largas pueden expresarse en varias líneas → \
  - Para interpretar los pragmas debemos **compilar** nuestro código con la opción **-fopenmp** (en el caso del copilador gcc)
  - Si no usamos esa opción el código se ejecuta de forma secuencial
    - Permite evaluar por ejemplo la versión secuencial y paralela

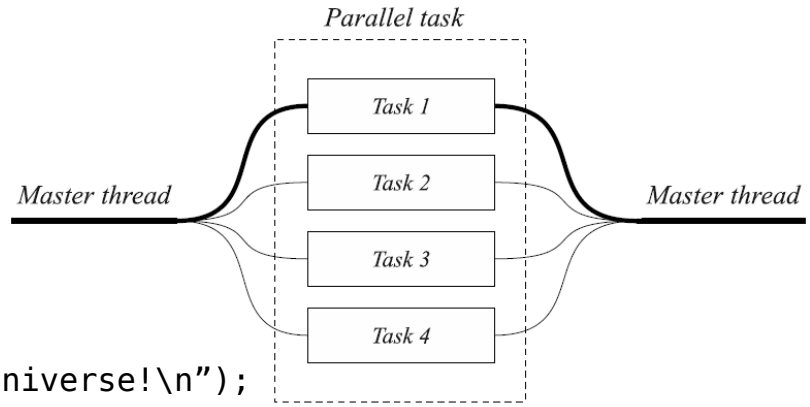
- Creación de tareas a través de **pragmas**
  - `#pragma omp parallel private(var1, var2) shared(var3)`
  - `#pragma omp` → **requerido**
  - `nombre-de-directiva` → **parallel**
  - `[clausulas, …]` → **private** y **shared**
  - Las cláusulas dirigen el comportamiento de la directiva
    - paralelización condicional
    - grado de concurrencia
    - gestión de datos
    - . . .




```
#include <omp.h>
main ()
{
   int var1, var2, var3;

   // código secuencial ...

   // Comienzo de sección paralela.
   // El thread maestro expande (fork) un conjunto de threads
   // Especifica el ámbito de todas las variables
   #pragma omp parallel private(var1, var2) shared(var3) num_threads(4)
   {
       // sección paralela ejecutada por todos los threads
       . . .
   }   // todos los threads se unen al thread maestro (join)

   // continúa código secuencial
}
```

- **Cláusulas**
  - Paralelización condicional, Grado de concurrencia, Gestión de datos y cooperación
- **Funciones de la librería**
  - Gestión del entorno, cerrojos y timing
- **Variables de entorno**
  - OMP_NUM_THREADS, OMP_SCHEDULE, OMP_DYNAMIC, …

Hello World!

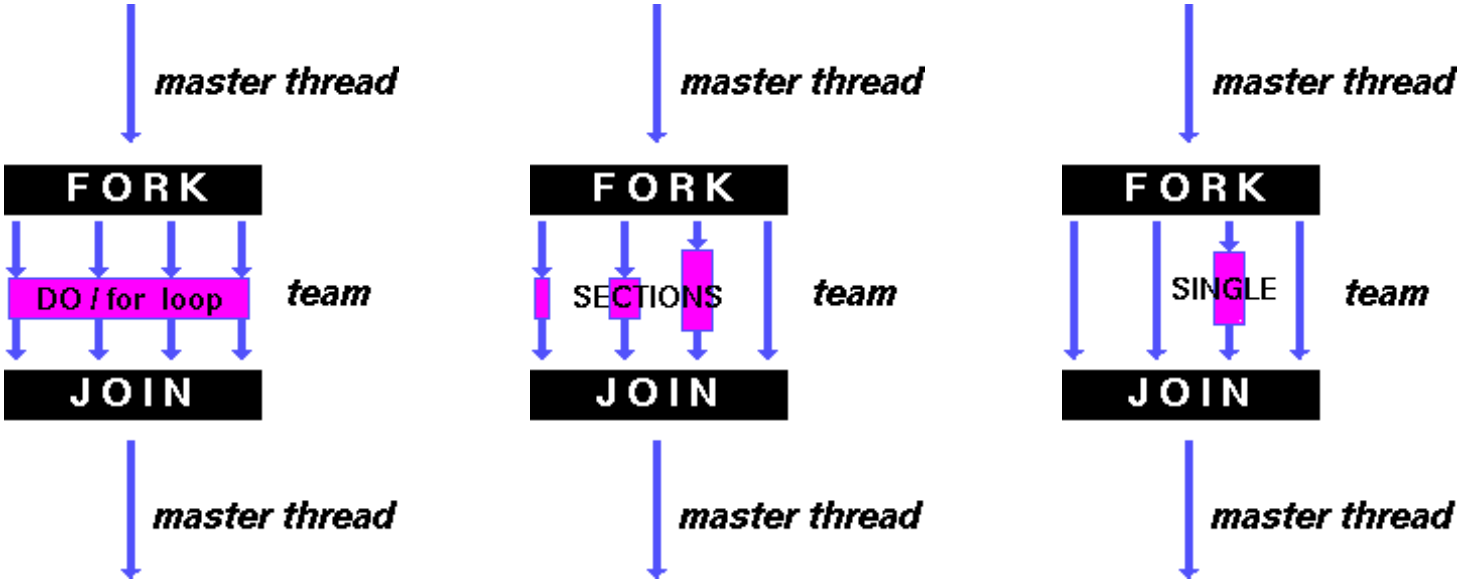



```
#include <omp.h>
main ()
{
int nthreads, tid;
printf("Hello there!, I'm the master of the universe!\n");
#pragma omp parallel private(nthreads, tid)
{
    tid = omp_get_thread_num();
    printf("Hello from thread %d\n", tid);
    /* code executed only by master thread (tid = 0) */
    if (tid == 0) {
         nthreads = omp_get_num_threads();
         printf("Number of threads = %d\n", nthreads);
    }
 } // end of parallel region (join)
printf("Good bye from the the master of the universe!\n");
}
```

```
export OMP_NUM_THREADS=4

Hello there!, I'm the master of
the universe!
Hello from thread 2
Hello from thread 0
Number of threads = 4
Hello from thread 3
Hello from thread 1
Good bye from the the master of
the universe!
```

Construcciones de tipo ***worksharing***

El trabajo dentro de la construcción paralela se divide entre el equipo y se ejecuta de forma cooperativa



- Definición de regiones paralelas → **bucles for**

Opción 1

```
#pragma omp parallel [clausulas, …]
#pragma omp for [clausulas, …]
```

Opción 2

```
#pragma omp parallel for [clausulas, …]
for (i = 0; i < n-1; i++)
{
    a[i] = a[i+1] + a[i];
}
```

- Presupone independencia de datos

  El análisis de dependencias es responsabilidad del programador

- ¿Cómo se reparten las iteraciones? → cláusula schedule

`schedule (static, chunk)` → número de iteraciones / chunk se asignan a los threads mediante técnica **round-robin** empezando por el thread 0.

`schedule (dynamic, chunk)` → número de iteraciones / chunk se asignan a los threads **tan pronto puedan** ejecutar un nuevo bloque.

`schedule (guided, chunk)` → reparto **exponencial decreciente** del número de iteraciones se asignan a los threads **tan pronto puedan** ejecutar un nuevo bloque.

`schedule (runtime)` → usa variable de entorno **OMP_SCHEDULE**

- ¿Cómo se reparten las iteraciones? → cláusula schedule

```
#pragma omp parallel for schedule (static, 2)
for (i = 0; i < 16; i++)

// ¿cómo se reparten las iteraciones entre 4 threads?
```

```
Thread 0, iteration: 0
Thread 0, iteration: 1
Thread 1, iteration: 2
Thread 1, iteration: 3
Thread 3, iteration: 6
Thread 3, iteration: 7
Thread 2, iteration: 4
Thread 2, iteration: 5
```

```
Thread 0, iteration: 8
Thread 0, iteration: 9
Thread 2, iteration: 12
Thread 2, iteration: 13
Thread 1, iteration: 10
Thread 1, iteration: 11
Thread 3, iteration: 14
Thread 3, iteration: 15
```

- ¿Cómo se reparten las iteraciones? → cláusula schedule

```
#pragma omp parallel for schedule (dynamic, 2)
for (i = 0; i < 16; i++)

// ¿cómo se reparten las iteraciones entre 4 threads?
```

```
Thread 0, iteration: 0
Thread 0, iteration: 1
Thread 1, iteration: 2
Thread 1, iteration: 3
Thread 3, iteration: 4
Thread 3, iteration: 5
Thread 2, iteration: 6
Thread 2, iteration: 7
```

```
Thread 0, iteration: 10
Thread 0, iteration: 11
Thread 2, iteration: 12
Thread 2, iteration: 13
Thread 1, iteration: 9
Thread 1, iteration: 10
Thread 3, iteration: 14
Thread 3, iteration: 15
```

- ¿Cómo se reparten las iteraciones? → cláusula schedule

```
#pragma omp parallel for schedule (guided, 2)
for (i = 0; i < 16; i++)

// ¿cómo se reparten las iteraciones entre 4 threads?
```

```
Thread 0, iteration: 7
Thread 0, iteration: 8
Thread 0, iteration: 9
Thread 2, iteration: 0
Thread 2, iteration: 1
Thread 2, iteration: 2
Thread 2, iteration: 3
Thread 3, iteration: 4
Thread 3, iteration: 5
Thread 3, iteration: 6
```

First assignment of iterations

```
Thread 2, iteration: 14
Thread 2, iteration: 15
Thread 1, iteration: 10
Thread 1, iteration: 11
Thread 0, iteration: 12
Thread 0, iteration: 13
```

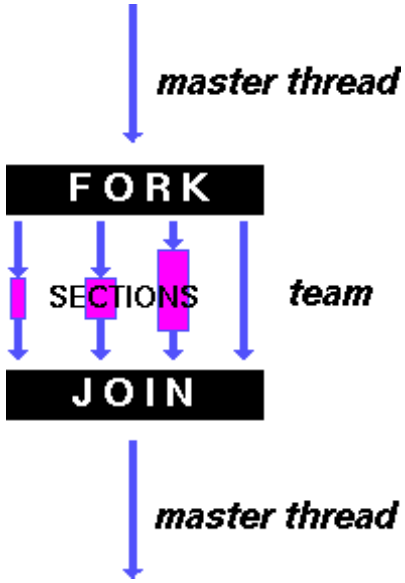
Second assignment of iterations

- El **ámbito de las variables** depende del tipo de construcción paralela y de las cláusulas usadas

```
int i, j
#pragma omp parallel for
for (i = 0; i < 16; i++)
  // la variable i es privada dentro de esta región (índice de bucle for)
  // la variable j es compartida dentro de esta región

#pragma omp parallel for private(j)
for (i = 0; i < 16; i++)
  // la variable i es privada dentro de esta región (índice de bucle for)
  // la variable j es privada dentro de esta región

#pragma omp parallel for default(private)
for (i = 0; i < 16; i++)
  // la variable i es privada dentro de esta región (índice de bucle for)
  // la variable j es privada dentro de esta región
```

- Definición de regiones paralelas → **secciones**

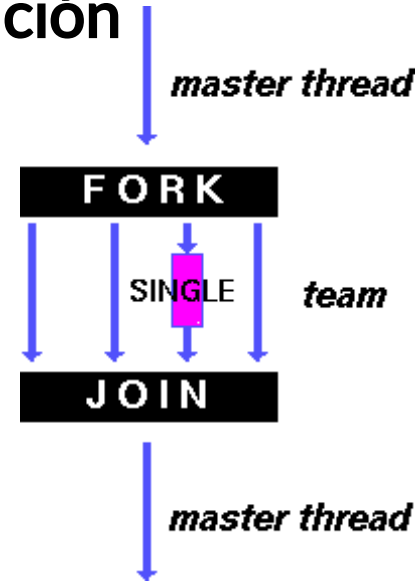
```
#pragma omp parallel sections [clausulas, …]
{
    #pragma omp section
    // código A
    #pragma omp section
    // código B
    #pragma omp section
    // código C
}
```



- Reparto de bloques de código diferente entre threads
  El análisis de dependencias es siempre responsabilidad del programador

- Q. ¿Qué threads ejecuta cada section?
- Q. ¿Qué pasa si num. sections > núm. threads?

Definición de regiones paralelas → **sincronización**



- `#pragma omp single [clausulas, …]`
  { bloque ejecutado sólo por un thread }
  // implicit barrier

- `#pragma omp master`
  { bloque ejecutado sólo por el thread 0 }
  // no barrier

- `#pragma omp critical [nombre-region]`
  { bloque ejecutado en **exclusión mutua** } // no barrier
  El nombre permite tener varias secciones críticas

- `#pragma omp atomic`
  { Modificación **atómica** en la misma posición de memoria }
  // no barrier
  Implementación eficiente de una sección crítica para asignaciones simples

Sincronización **explícita**

- `#pragma omp` **`barrier`**
  { sincroniza todos los threads dentro de una región paralela }

- `#pragma omp` **`flush`**
  { Asegura consistencia en la memoria compartida }
  Consistencia: ¿Cuándo son visibles las escrituras?

- `#pragma omp` **`ordered`**
  { ejecución ordenada de iteraciones dentro de un bucle }
  Permite ejecutar otro código en paralelo

Sincronización **implícita**

- **Al final de cada región paralela:** Implica también una directiva **flush**

- `#pragma omp` **`single`**

- **Construcciones de tipo *task***

  Un sólo thread genera todas las tareas, y se ejecutan entre todos los threads

```
typedef struct node node;
struct node { int data; node * next; };
void process (int data); // do something here
void increment_list_items(node *head)
{
      #pragma omp parallel private(head)
      {
           #pragma omp single
           {
                node *p = head;
                while (p) {
                     #pragma omp task         // p es firstprivate por defecto
                         process(p->data);
                     p = p→next;
                }
           }
      } // en este punto todas las tareas se han completado
 }
```

Estas regiones paralelas se asignan a una **cola de tareas**

Todos los **threads** **extraen** tareas de la cola

Permite paralelizar **algoritmos recursivos** y **balanceo de carga**

Construcciones de tipo *task*

- `#pragma omp` **`taskwait`**

  Barrera para sincronizar tareas hijas generadas por otras tareas

```
int fib(int n) {
   int i, j;
   if (n < 2 ) return n;
   else {
     #pragma omp task shared(i) firstprivate(n)
       i = fib(n-1);
     #pragma omp task shared(j) firstprivate(n)
       j = fib(n-2);
     #pragma omp taskwait     // asegura que i, j se ha completado
       return i + j;
   }
}
```

Nota: La función fib() debe invocarse dentro de una región paralela para que las tareas definidas en el ejemplo se ejecuten en paralelo

Resumen de sintaxis OpenMP 4.5 C/C++

https://www.openmp.org/wp-content/uploads/OpenMP-4.5-1115-CPP-web.pdf

OpenMP Application Program Interface 4.5

https://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf

directivas, funciones, variables de entorno, memoria compartida...