Universitat d'Alacant
Universidad de Alicante

# Trabajo Optativo

*Paralelismo con OpenMP*

Zuleika María Redondo García
Iñigo Zárate Rico

`Sistemas Empotrados`

Escuela Politécnica Superior,
Ingeniería Robótica

14 de diciembre de 2021

# Índice general

# Parte 1

# Introducción a OpenMP

OpenMP es una interfaz de programación de aplicaciones (API) para la programación paralela en diferentes plataformas. Esta permite añadir procesamiento concurrente a programas codificados en C, C++ y Fortran sobre la base del modelo de ejecución fork/join. Esta API esta disponible en múltiples plataformas como Microsoft Windows o Linux.

El logo de esta, es mostrado en la figura 1.1.

Muchos compiladores soportan OpenMP, para cada uno el flag es diferente, en GCC, que es el usado en este trabajo, el flag es "-fopenmp"


Figura 1.1: Logo de OpenMP.

Antes de entrar en materia, se van a exponer una serie de definiciones y conceptos necesarios para entender como funciona o se maneja la API, OpenMp.

## 1.1. Multiprocessing vs multithreading

Los programas multihilo y los sistemas con multiprocesador son la mejor manera de separar trabajos en acciones y dividir las tareas en varias para la mayoría de procesos de computación. De esta forma, en la mayoría de casos, se reduce el tiempo de ejecución con respecto al proceso original.

El multiprocessing está más relacionado con el hardware, es decir, a ejecutar múltiples procesos o incluso programas al mismo tiempo en las unidades de la CPU. Este método también puede

cambiar el tipo de acceso a memoria.

Mientras que, por su parte, el multithreading se relaciona más con el software. En este caso, por medio de la programación se divide la tarea en diferentes hilos, que se reparten en los cores con los que se cuente o los que se le indique.

Estas diferencias se muestran en la figura 1.2.

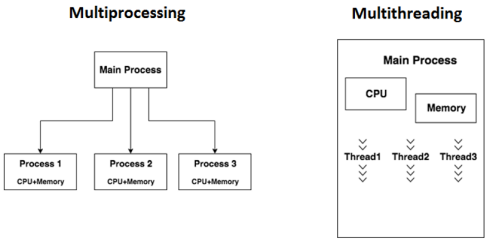


Figura 1.2: Multiprocessing vs Multithreading.

Los hilos son subtareas en las que se divide un proceso. Una tarea se puede dividir en más hilos que cores con los que se cuenta dado que, sencillamente estos se repartirán entre los cores indicados. Finalmente los hilos se tendrán que unir en una única solución del problema de forma correcta, esto se llama sincronización.

Así pues, las características más importantes han sido comparadas en la tabla 1.1.

| **Características** | **Multiprocessing** | **Multithreading** |
|---|---|---|
| **División** | Se dividen procesos o programas en los cores. | Se dividen *hilos* o subtareas del mismo proceso en los cores. |
| **Creación** | La creación de un proceso es lento. | La creación de un *hilo* es rápida. |
| **Memoria** | Cada programa tiene guardado su memoria (código y datos). | Todos los *hilos* comparten la misma memoria. |
| **Peso** | Las tareas son pesadas y tienen su propio espacio de dirección. El proceso de comunicación es muy costoso y pesado por estar en diferentes direcciones de memoria. | Los *hilos* son procesos ligeros y pueden compartir la misma dirección de memoria. La comunicación entre ellos es menos costosa. |
| **Acceso a memoria** | Depende mucho de acceder a la memoria para enviar información a otros procesos. | Evita acceder mucho a la memoria. |

Cuadro 1.1: Comparación de características entre multiprocessing y multithreading.

## 1.2. Paralelismo

Consiste en dividir una gran problema en pequeños subproblemas y resolverlos al mismo tiempo. La diferencia con la concurrencia es precisamente que las tareas se resuelven de forma independiente y en el mismo preciso momento, paralelamente.

Concurrencia: consiste en dividir el problema en pequeñas tareas las cuales se resuelven secuencialmente de forma intercalada dichas tareas. Dichas tareas no tienen por qué resolverse completamente antes de comenzar la siguiente, precisamente ahí reside la idea de la concurrencia, en intercalar breves ejecuciones de cada tarea consecutivamente.

La comparativa entre paralelismo y concurrencia se puede observar en la figura 1.3.

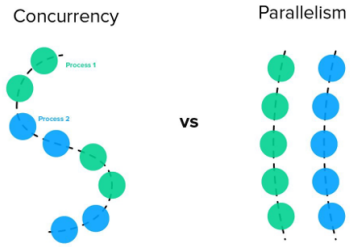


Figura 1.3: Concurrencia vs Paralelismo.

Ejecución:

- Se divide un problema en partes discretas que se puedan resolver simultáneamente.
- Cada parte se divide en instrucciones.
- Las instrucciones se ejecutan en diferentes procesadores.
- Mecanismo de coordinación.

Niveles de paralelismo:

- Trabajo: cada hilo resuelve un problema independientemente.
- Tarea: cada hilo realiza una tarea distinta e independiente del resto. Uno corta el césped y otro poda los setos.
- Proceso: el orden de las instrucciones se cambia para agruparse en conjuntos que serán ejecutados paralelamente sin alterar el resultado. Pipeline.
- Datos: cada procesador realiza la misma tarea sobre un conjunto independiente de datos. Dos o más personas se dividen el espacio de un jardín para cortar el césped.

- Gruesa: gran cantidad de trabajo, alta dependencia, por tanto se requiere de poca sincronización.
- Fina: poca cantidad de trabajo, poca dependencia lo que requiere de una sincronización más precisa.

Scheduling: proceso de asignación de tareas a los hilos. Puede ser en tiempo de compilación o dinámicamente en tiempo de ejecución. Se debe tener en cuenta los datos usados para cada tarea, ya que pueden depender del resultado de otras tareas.

Balanceo de carga: se refiere a repartir las instrucciones de la forma más equitativa posible, ya que de este modo se maximiza la eficiencia del paralelismo. Este es uno de los pasos más importantes, ya que decidirá la sincronización necesaria y el tiempo de hilos en pausa.

Sección crítica: son los segmentos de código que manipulan un recurso (una dirección de memoria, por ejemplo) no debe ser accedida por más de un hilo al mismo tiempo. Si no hay control para gestionar el uso de recursos de cada hilo se le dará acceso según el orden en que lleguen, esto es conocido como condición de carrera.

Pipelining: ruptura o segmentación de una tarea en pasos para realizarlos en diferentes unidades del procesador. Se basa en el concepto de una tubería, la cual no precisa que salga todo el agua de su interior para que se pueda introducir más, sino que él fluye la entrada y salida al mismo tiempo.

En la figura 1.4, se muestra la diferencia entre aplicar Pipelining y no aplicarlo.

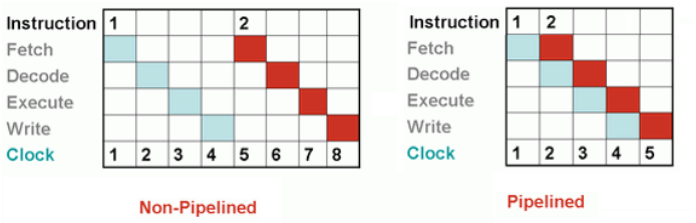


Figura 1.4: No Pipelining vs Pipelining.

## 1.3. Tiempo, Speed-up y Eficiencia

Una vez aplicado el paralelismo, es importante estudiar con qué número de hebras se obtiene el mejor resultado a la hora de ejecutar el código. De esta manera, se estudian los valores de tiempo, speed-up y eficiencia.

$$\boxed{t_p(P) = t_f - t_i} \tag{1.1}$$

El valor del tiempo consiste en la diferencia entre el tiempo al final del programa y al principio. Se puede utilizar, para obtener un resultado con un error pequeño, el constructor "omp_get_wtime(void)", para rutinas de control de tiempo, expuesto a continuación en el apartado 2.2.3.

$$\boxed{S_p(P) = \frac{t_{sec}}{t_p(P)}} \tag{1.2}$$

Por otro lado, el speed-up es la relación entre el tiempo secuencial(1 hebra) y el tiempo en paralelo, es decir, el citado anteriormente en la ecuación 1.1. Cuanto mayor sea el valor de speed-up, mejor, dado que a más valor, mayor será la mejora de velocidad en paralelo con respecto al secuencial

Una forma más extensa del speed-up es el de la ecuación 1.3.

$$\boxed{S_p(P) = \frac{t_{sec}}{\frac{f \cdot t_{sec}}{P} + (1-f) \cdot t_{sec} + t_{overhead}(P)}} \tag{1.3}$$

El resultado se dará en escala unitaria. Un resultado mayor que 1, que tendrá en ejecución secuencial, supondrá satisfactorio en la aplicación del paralelismo.

$$\boxed{\eta_p(P) = \frac{S_p(P)}{P}} \tag{1.4}$$

Finalmente, la eficiencia es la relación entre el speed-up (1.2 y el número de hilos, como se muestra en la ecuación 1.4.

**Siendo **P** el número de hilos, **t_sec**, el tiempo secuencial y **f**, la proporción de la parte paralelizada del programa**

# Parte 2

# Programación

Generalmente, la mayor parte de la programación se hace de forma normal con lenguajes como C o C++, sin embargo, a la hora de aplicar paralelismos a diferentes procesos, se deben añadir elementos característicos de OpenMP, como se va a explicar a lo largo de este apartado.

Para incluir la librería se añade al principio: #include <omp.h>

## 2.1. Variables de entorno

Existe una serie de variables propias de la API, OpenMP, estas se muestran en la tabla 2.1.

| **Variable** | **Descripción** |
|---|---|
| **OMP_SCHEDULE** | Modifica el comportamiento de la cláusula schedule cuando se especifica en una directiva o for parallel for . |
| **OMP_NUM_THREADS** | Establece el número máximo de subprocesos en la región paralela, a menos que se establezca con omp_set_num_threads o num_threads. |
| **OMP_DYNAMIC** | Especifica si el tiempo de ejecución de OpenMP puede ajustar el número de subprocesos en una región paralela. Su valor es FALSE de forma predeterminada. |
| **OMP_NESTED** | Especifica si el paralelismo anidado está habilitado, a menos que el paralelismo anidado esté habilitado o deshabilitado con omp_set_nested . |

Cuadro 2.1: Variables de entorno de OpenMP.

## 2.2. Funciones de librería

### 2.2.1. Para la ejecución del entorno:

- Establece el número de threads a usar en la siguiente región paralela.

```
void omp_set_num_threads(int num_threads);
```

- Devuelve el número de threads que se están usando en una región paralela.

```
int omp_get_num_threads(void);
```

- Obtiene la máxima cantidad posible de threads.

```
int omp_get_max_threads(void);
```

- Devuelve el número del thread en el que se está.

```
int omp_get_thread_num(void);
```

- Devuelve el máximo número de procesadores que se pueden asignar al programa.

```
int omp_get_num_procs(void);
```

- Devuelve un valor distinto de cero si se ejecuta dentro de una región paralela.

```
int omp_in_parallel(void);
```

- Permite ajustar, para las siguientes regiones paralelas, el número de procesos de forma dinámica.

```
omp_set_dynamic(void);
```

- Devuelve un valor que indica si, en las siguientes regiones paralelas, se permite modificar el número de procesos de forma dinámica.

```
omp_get_dynamic(void);
```

- Permite habilitar el paralelismo anidado.

```
omp_set_nested(void);
```

- Devuelve un valor que indica si está habilitado el paralelismo anidado.

```
omp_get_nested(void);
```

### 2.2.2. Para el bloqueo:

- Inicializa un bloqueo simple.

  omp_init_lock

- Inicializa un bloqueo.

  omp_init_nest_lock

- Desinicializa un bloqueo.

  omp_destroy_lock

- Desinicializa un bloqueo anidable.

  omp_destroy_nest_lock

- Bloquea la ejecución de subprocesos hasta que haya un bloqueo disponible.

  omp_set_lock

- Bloquea la ejecución de subprocesos hasta que haya un bloqueo disponible.

  omp_set_nest_lock

- Libera un bloqueo.

  omp_unset_lock

- Libera un bloqueo anidable.

  omp_unset_nest_lock

- Intenta establecer un bloqueo, pero no bloquea la ejecución de subprocesos.

  omp_test_lock

- Intenta establecer un bloqueo anidable, pero no bloquea la ejecución de subprocesos.

  omp_test_nest_lock

### 2.2.3. Para rutinas de control de tiempo:

- Devuelve un valor en segundos del tiempo transcurrido.

```
double omp_get_wtime(void);
```

- Devuelve el número de segundos entre los tics del reloj del procesador.

```
double omp_get_wtick(void);
```

## 2.3. Cláusulas

Se utilizan para definir la forma de tratar y compartir el valor de las variables.

- shared(var): Declara que la variable *var* es compartida por todos los hilos.
- private(var): Crea una copia de la variable para cada hilo de forma independiente. Estas variables se encuentran NO-inicializadas tanto a la entrada como a la salida de la región paralela. De forma que su valor dentro no posee ninguna relación con el exterior de dicha región.
- firstprivate(var): Inicializa una variable con el valor que tenía antes de la región paralela
- lastprivate(var): Sirve para "extraer" la información del interior de la región paralela, de manera que el valor aportado por el último hilo de la ejecución será el que contendrá la variable al finalizar.
- reduction(var): Se utiliza para coordinar a los hilos para variables acumulativas, por ejemplo un contador de repeticiones. Todos los hilos accediendo a la misma variable pueden surgir errores ya que pueden querer modificar su valor simultáneamente, con esta cláusula se evitan dichos errores.
- if (var). Esta clásula se usa para añadir una condición para crear o no la sección paralela. Un valor distinto de cero, crearía los hilos, un valor igual a 0, ejecutaría el código de forma secuencial.
- default(none/shared): Si es none, se debe declarar el ámbito de todas la variables, si es shared (default) las variables son compartidas.
- copyin(var): copia en cada hilo "hijo" el valor de la variable en el hilo maestro del que dependen, el cual puede ser modificado y se modificaría en el principal.
- numthreads(n): establece n hilos a utilizar, igual que omp_set_num_threads().
- nowait: desactiva la sincronización, con esta cláusula los hilos no se esperaran a que todos hayan finalizado, cada uno es independiente.

## 2.4. Constructores

*#pragma* es el método del estándar de C usado para aportar información adicional al compilador, cuando se requiere establecer manualmente un uso más profundo del sistema.

### 2.4.1. CONSTRUCTOR PARALLEL

- Crea un número de hilos para hacer lo que se indica dentro de este.
- Cuando dentro de una región hay otro constructor paralelo se produce un anidamiento.

#pragma omp **parallel** [cláusulas]
bloque

### 2.4.2. CONSTRUCTOR FOR

- Las iteraciones se ejecutan en paralelo por hilos ya existentes.

#pragma omp **for** [cláusulas]
bucle for

### 2.4.3. CONSTRUCTOR SECTIONS

- Cada sección se realiza por un hilo.

#pragma omp **sections** [cláusulas] {
[#pragma omp section]
bloque
[#pragma omp section]
bloque
...
}

### 2.4.4. CONSTRUCTORES COMBINADOS

- Son la combinación de 2 constructores.

#pragma omp **parallel for** [cláusulas]
bucle for

#pragma omp **parallel sections** [cláusulas]

### 2.4.5. CONSTRUCTORES DE EJECUCIÓN SECUENCIAL

- Para que la programación se ejecute por un único hilo (que no tiene porqué ser el hilo maestro).

  #pragma omp **single** [cláusulas]
  bloque

- Para que la programación la ejecute el hilo maestro.

  #pragma omp **master**
  bloque

- Permite ejecutar programación de forma secuencial dentro del bloque en paralelo.

  #pragma omp **ordered**
  bloque

### 2.4.6. CONSTRUCTORES DE SINCRONIZACIÓN

- Se le indica que la programación se ejecute en un subproceso a la vez.

  #pragma omp **critical** [nombre]
  bloque

- Sincroniza todos los threads en el equipo.

  #pragma omp **barrier**

- Se le indica una posición de memoria que se actualiza de forma atómica (la lectura y escritura de la unificación se hace en el mismo bus). En lugar de dejar que múltiples hilos decidan realizar la programación simultáneamente.

  #pragma omp **atomic**
  expresión

### 2.4.7. CONSTRUCTORES DE TASKING

Otra forma de paralelización se hace con los constructores de tasking.

- Crea un task de forma explícita. Y además, se crea un entorno de datos basado en esa región de tarea, en función de las cláusulas de los atributos en el intercambio de datos.

  Se pueden anidar e introducir dentro de otros constructores como los de ejecución secuencial.

  #pragma omp **task**
  bloque estructural

- Especifica que las iteraciones de los bucles asociados serán ejecutados en paralelo empleando tareas explícitas.

  #pragma omp **taskloop**
  bucles for

- Permite especificar que la tarea actual en la que se encuentra puede ser suspendida a favor de una tarea diferente.

  #pragma omp **taskyield**

- Hace esperar al thread hasta que todas las tareas hayan acabado de ejecutarse.

  #pragma omp **taskwait**

- Hace esperar al thread que todas las tareas hayan acabado de ejecutarse, incluyendo, las tareas descendientes.

  #pragma omp **taskgroup**

# Parte 3

# Programas

Con el objetivo de ejemplificar el uso del paralelismo, se han realizado una serie de programas. Éstos, son programas simples haciendo uso de las cláusulas explicadas. Además, se han paralelizado los algoritmos de ordenacion de vectores heapsort y Quicksort.

También se ha realizado un estudio de la evolución del tiempo, speed-up y eficiencia de algunos de dichos programas.

## 3.1. Variables

Se ha desarrollado un programa el cual inicializa un valor en una variable y se le introduce como *private*, se opera en su interior, imprimiendo el thread correspondiente junto con el valor obtenido y, finalmente el valor de dicha variable tras la sección paralela.

```
1  #include <iostream>
2  #include <omp.h>
3
4  int main(){
5
6          int out = 47;
7
8          #pragma omp parallel private(out)
9          {
10                 out = 7 * omp_get_thread_num();
11                 std::cout << "thread " << omp_get_thread_num() << " calculate: " << out << std::endl;
12         }
13
14         std::cout << "out value: " << out << std::endl;
15         return 0;
16 }
```

Figura 3.1: Código haciendo uso de variable privada.

Pasando la variable como private, el resultado se muestra en la figura 3.2. Se puede observar que tras la sección paralela, el valor se mantiene como 47. Por el contrario, pasando la variable como shared (en c++ no es necesario ni especificarlo, ya que por defecto se inicializa de este modo) el

valor de la variable se modifica. Como vemos en la figura 3.3 el valor tras la sección paralela es 7.

```
thread 8 calculate: 56
thread 6 calculate: 42
thread 3 calculate: 21
thread 10 calculate: 70
thread 1 calculate: 7
thread 5 calculate: 35
thread 4 calculate: 28
thread 9 calculate: 63
thread 11 calculate: 77
thread 0 calculate: 0
thread 7 calculate: 49
thread 2 calculate: 14
out value: 47
```

Figura 3.2: Salida del código con variable privada.

```
thread 0 calculate: 7
thread 10 calculate: 7
thread 8 calculate: 7
thread 9 calculate: 7
thread 6 calculate: 7
thread 5 calculate: 7
thread 7 calculate: 7
thread 1 calculate: 7
thread 3 calculate: 7
thread 2 calculate: 7
thread 11 calculate: 7
thread 4 calculate: 7
out value: 7
```

Figura 3.3: Salida del código con variable compartida.

## 3.2. Bucle for en paralelismo

Para hacer uso del paralelismo es un bucle for, se ha desarrollado un programa que con un string dado, cuenta el número de "a" presentes en dicho string. Dicho programa se muestra en la figura 3.4.

El código hace uso de un constructor combinado paralelo y for, indicando el número de hilos creados (esto se añade para posteriormente poder llevar a cabo el estudio de tiempo, speed-up y eficiencia mencionado anteriormente). Además, en esta misma instrucción se incluye una reducción de la variable *counter*.

Dicha reducción es necesaria ya que, de lo contrario la acumulación del número de a no sería correcta debido a que todos los hilos han escrito el "+1" en la misma dirección de memoria con el mismo valor anterior.

```cpp
#include <iostream>
#include <omp.h>
#include <string>

using namespace std;

int main(int argc, char **argv){

        string word = "How many a are in this sentence a a a a a a a a a aa a a a a a a a a a a a";
        int counter = 0;

        #pragma omp parallel for num_threads(6) reduction(+:counter)
                for(auto i = word.begin(); i != word.end(); i++){
                        if(*i == 'a'){
                                counter++;
                                cout << omp_get_thread_num() << " found " << counter << endl;
                        }
                }

        cout << "total a: " << counter << endl;

        return 0;
}
```

Figura 3.4: Código for que cuenta el número de a.

En ciertas ocasiones, se requiere que una parte del bucle se realice de forma ordenada. Para esto, se utiliza la cláusula *ordered*. Para ejemplicar la utilidad y el uso de esta cláusula se ha escrito el código mostrado en la figura 3.5.

En este código se usa la variable i como privada para que no interfiera un hilo con el otro y se ejecute por completo sin problema. El programa imprime el número del hilo actual y añade la etiqueta *before* (para indicar que se encuentra previo a la sección ordenada) o *after* (indica que se encuentra dentro de la sección ordenada).

```cpp
#include <iostream>
#include <omp.h>

using namespace std;

int main(){

        int i;
        #pragma omp parallel for ordered private(i)
                for(i = 0; i < 7; i++){
                        cout << "Thread: " << omp_get_thread_num() << " before " << i << endl;
                        #pragma omp ordered
                        {
                                cout << "Thread: " << omp_get_thread_num() << " after " << i << endl;
                        }
                }

        return 0;
}
```

Figura 3.5: Código for ordenado.

Para este ejemplo se muestra la salida del programa en la figura 3.6. En la figura se observa como la salida con la etiqueta *before* es caótica, lo primero que se muestra es el hilo 3, luego el 2, luego el 0... Incluso varios hilos escriben la salida “Thread” al mismo tiempo mientras se llama a la función para conocer qué hilo se encuentra en ejecución.

Sin emabargo, las salidas con etiqueta *after* se encuentran en orden ascendente, 0, 1, 2... Concretamente, en este caso la salida del hilo 2 ordeandor con la del 6 sin ordenar se acoplan pareciendo que se indica el hilo 26, esto es lo que puede suceder al no encontrarse ordenado.

```
Thread: Thread: Thread: 3 before 3
Thread: 2 before 2
Thread: 0 before 0
Thread: 0 after 0
Thread: 1 before 1
Thread: 1 after 1
Thread: 26 after  before 26
Thread: 4 before 4
5 before 5

Thread: 3 after 3
Thread: 4 after 4
Thread: 5 after 5
Thread: 6 after 6
```

Figura 3.6: Salida del for ordenado.

## 3.3. Sincronización

### 3.3.1. Critical

En primer lugar se ha hecho uso de la sección crítica. Para ello se ha utilizado el programa de la figura 3.7.

En dicho programa, únicamente se imprime el número del thread con la etiqeta *before* y seguidamente otra vez con la etiqueta *after*.

```cpp
#include <iostream>
#include <omp.h>

using namespace std;

int main(){

        #pragma omp parallel
        {
        #pragma omp critical
                {
                        cout << "Thread: " << omp_get_thread_num() << " before" << endl;
                        cout << "Thread: " << omp_get_thread_num() << " after" << endl;
                }


        }


        return 0;
}
```

Figura 3.7: Código de la sección crítica.

De este modo, haciendo uso de la cláusula critical se observa una salida (figura 3.8) en la cuál para cada hilo la salida *before* se encuentra inmediatamente antes de la *after*. Sin embargo, si se comenta la línea 10 (indica la sección critica) se muestra la salida mostrada en la figura 3.9, la cual no se encuentra ordenada.

```
Thread: 1 before
Thread: 1 after
Thread: 6 before
Thread: 6 after
Thread: 2 before
Thread: 2 after
Thread: 10 before
Thread: 10 after
Thread: 12 before
Thread: 12 after
Thread: 16 before
Thread: 16 after
Thread: 8 before
Thread: 8 after
Thread: 5 before
Thread: 5 after
Thread: 9 before
Thread: 9 after
Thread: 11 before
Thread: 11 after
Thread: 3 before
Thread: 3 after
Thread: 13 before
Thread: 13 after
Thread: 4 before
Thread: 4 after
Thread: 14 before
Thread: 14 after
Thread: 15 before
Thread: 15 after
Thread: 17 before
Thread: 17 after
Thread: 0 before
Thread: 0 after
Thread: 7 before
Thread: 7 after
```

Figura 3.8: Salida del código con sección crítica.

```
Thread: Thread: Thread: 41 before before
Thread: 6 before
Thread: 8 before
Thread: 6 after
Thread: 8 after
Thread: Thread: Thread: Thread: 115 before before
Thread: 133Thread:  before1
Thread: 9 before
Thread: 9 after
Thread: 11 after
Thread: 17 before
Thread: 17 after
Thread: 14 before
Thread: 14 after
 before
Thread: 13 after
12 before
Thread: 12 after
 after
Thread: 7 before
Thread: 7 after
Thread: 15 before
Thread: 15 after
Thread: 16 before
Thread: 16 after
Thread: 3 after

Thread: 4 after

Thread: 5 after
Thread: 10 before
Thread: 10 after
2 before
Thread: 2 after
Thread: 0 before
Thread: 0 after
```

Figura 3.9: Salida del código sin sección crítica.

### 3.3.2. Barrier

La siguiente cláusula usada es barrier. El programa usado se muestra en la figura 3.10 el cual imprime un mensaje con la etiqueta *before* antes del uso del barrier y otra con la etiqueta *after*.

```cpp
#include <iostream>
#include <omp.h>

using namespace std;

int main(){

        #pragma omp parallel num_threads(6)
                cout << "Thread n° " << omp_get_thread_num() << " before" << endl;
                #pragma omp barrier
                cout << "Thread n° " << omp_get_thread_num() << " after" << endl;

        return 0;
}
```

Figura 3.10: Código con sincronización barrier.

La salida al programa anterior se muestra en la figura 3.11. Cabe destacar que los hilos imprimen lo que deben imprimir de forma aleatoria, todos con la etiqueta *before* y, hasta que no han finalizado todos de realizar dicha tarea no se imprime el valor con la etiqueta *after*. Esto muestra que el código tras la cláusula únicamente se ejecutará cuando todos los hilos hayan terminado de ejecutar la parte anterior.

```
Thread n° Thread n° Thread n° 0Thread n° 4 before
1 before
 before
5 before
Thread n° 3 before
Thread n° 2 before
Thread n° 0 after
```

Figura 3.11: Salida con sincronización barrier.

### 3.3.3. Sections

Para el uso de las secciones se ha desarrollado el programa mostrado en la figura 3.12. En el cual, cada hilo ejecuta una sección dónde se imprime el número del hilo y el numero de la sección. La salida a dicho programa se muestra en la figura 3.13.

```cpp
#include <iostream>
#include <omp.h>


int main(){
        #pragma omp parallel sections
                {
                        #pragma omp section
                            std::cout << "Thread nº " << omp_get_thread_num() << " saying hello from section 1" << std::endl;

                        #pragma omp section
                            std::cout << "Thread nº " << omp_get_thread_num() << " saying hello from section 2" << std::endl;

                        #pragma omp section
                            std::cout << "Thread nº " << omp_get_thread_num() << " saying hello from section 3" << std::endl;

                        #pragma omp section
                            std::cout << "Thread nº " << omp_get_thread_num() << " saying hello from section 4" << std::endl;

                        #pragma omp section
                            std::cout << "Thread nº " << omp_get_thread_num() << " saying hello from section 5" << std::endl;

                        #pragma omp section
                            std::cout << "Thread nº " << omp_get_thread_num() << " saying hello from section 6" << std::endl;

                        #pragma omp section
                            std::cout << "Thread nº " << omp_get_thread_num() << " saying hello from section 7" << std::endl;
                }
        return 0;
}

```

Figura 3.12: Código con uso de secciones.

```
Thread nº 7 saying hello from section 4
Thread nº 5 saying hello from section 3
Thread nº 1 saying hello from section 7
Thread nº 3 saying hello from section 2
Thread nº 2 saying hello from section 5
Thread nº 9 saying hello from section 1
Thread nº 0 saying hello from section 6
```

Figura 3.13: Salida del código seecionado.

### 3.3.4. Estudio eficiencia

Como se ha comentado previamente, se ha llevado a cabo un estudio para optimizar los recursos del paralelismo, para ello se ha tenido en cuenta el tiempo de ejecución, el speed-up y la eficiencia (expuestos en el apartado 1.3), cada uno representado en una gráfica.

El análisis se centrará principalmente en los valores del speed-up, ya que éste aporta la información de la proporción en que mejora el tiempo de ejecución en paralelo con respecto al secuencial, como se muestra en la ecuación 1.2.

#### for contador

En primer lugar el programa contador de “a” mostrado en la figura 3.4 El resultado del expermiento se muestra en las figuras 3.14 3.15 3.16.

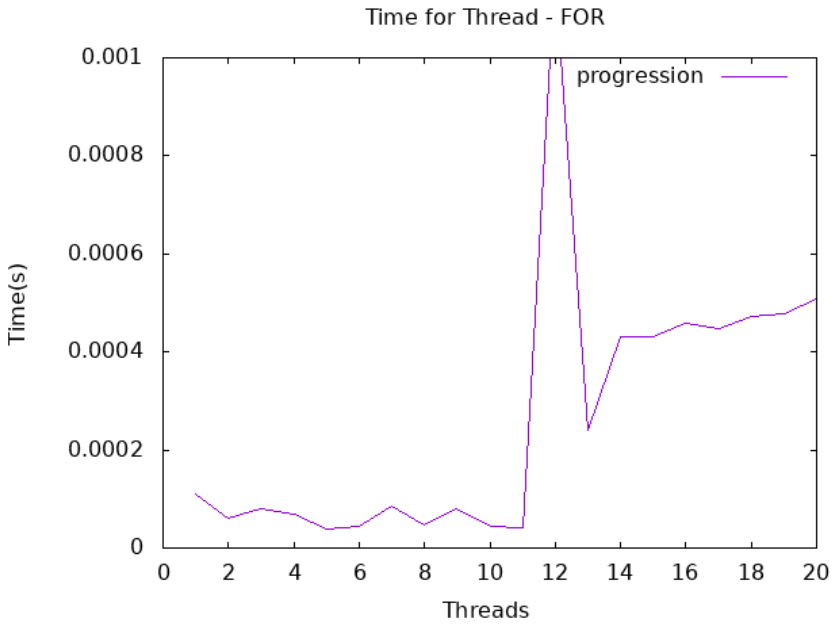


Figura 3.14: Gráfica de tiempo de ejecución según el número de threads.

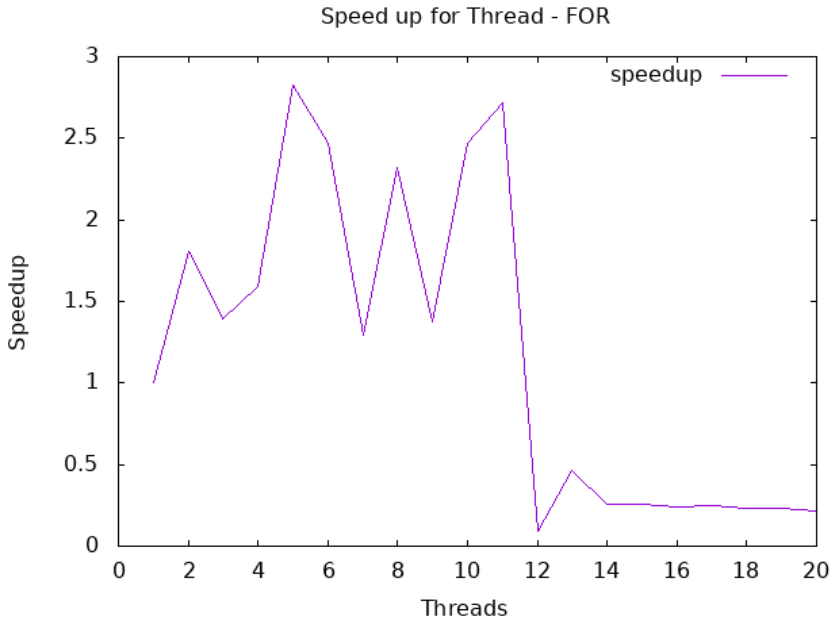


Figura 3.15: Gráfica de speedup según el número de threads.

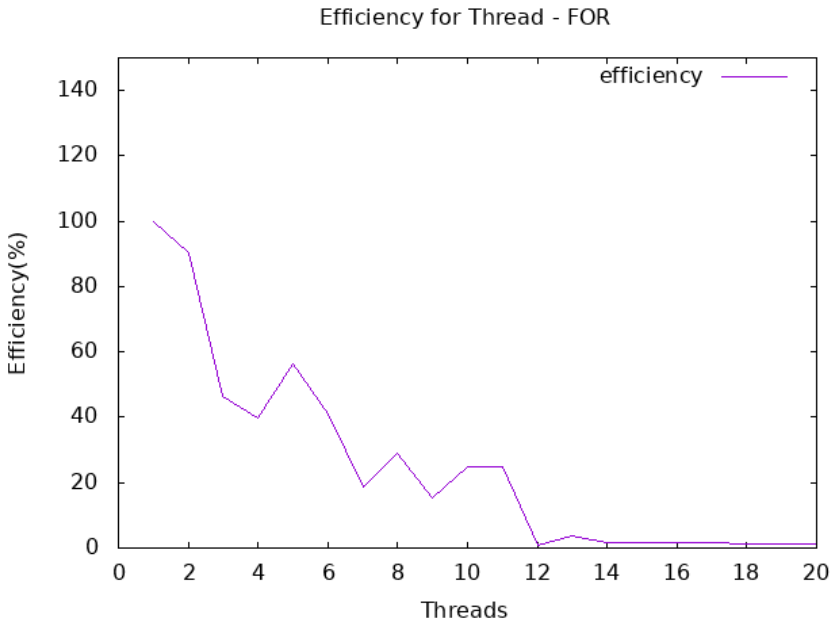


Figura 3.16: Gráfica de eficiencia según el número de threads.

Tras realizar el experimento, se concluye que, el tiempo de ejecución no se ve prácticamente reducido ya que la carga de trabajo es muy baja (recorrer un string corto). Sin emabrgo, en cuanto a speed-up, se aprecian subidas óptimas como el máximo con 5 núcleos.

Cabe destacar el máximo tiempo de ejecución con 12 hilos y la subida con el incremento de los mismos. El tiempo aumenta a la par que se reduce drásticamente el speed-up y la eficiencia ya que, el procesor Intel usado para el experimento soporta como máximo 12 subprocesos. Esto refleja que, hacer uso de un número mayor de hilos que de subprocesos soportados incrementa altamente el tiempo de ejecución (siendo un resultado peor que en monohebra o secuencial) y reduce la eficiencia.

### Algoritmo de ordenación QuickSort en paralelo

El algoritmo QuickSort hace uso de un bucle for que recorre el vector para ordenarlo de manera ascendente. Añadiendo las cláusulas correspondientes se paraleliza dicho bucle y se obtiene el resultado final. En la figura 3.17 se muestran las cláusulas necesarias para dicho cometido.

```
52      for(int j = 1; j <= 20; j++){
53          double one;
54              double begin = omp_get_wtime();
55
56          #pragma omp parallel for num_threads(j)
57              for (int exp = 15; exp <= 20; exp++){
58                  size_t size = size_t( pow(2,exp) );
```

Figura 3.17: Cláusulas para la paralelizacion de QuickSort.

Las gráficas del resultado del experimento se muestran en las figuras. 3.18 3.19 3.20

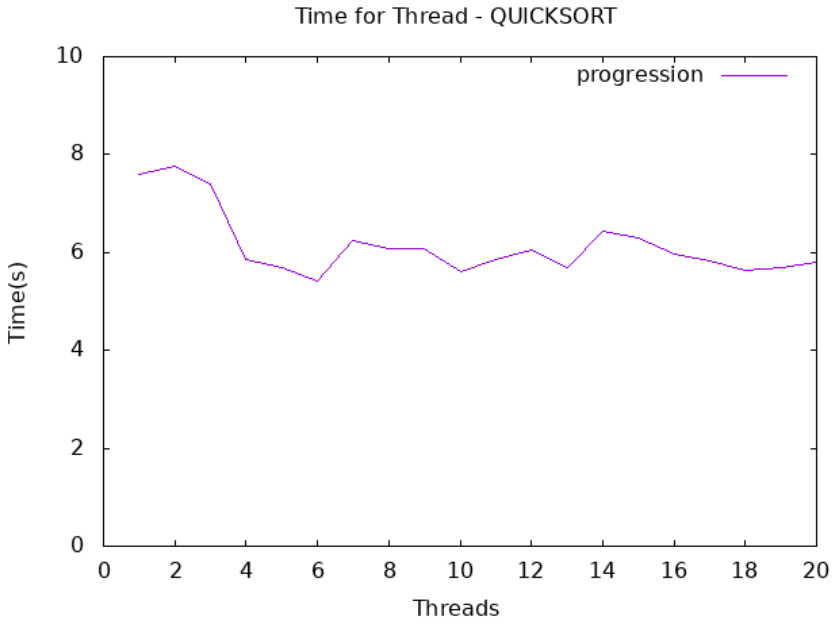


Figura 3.18: Gráfica de tiempo de ejecución según el número de threads.

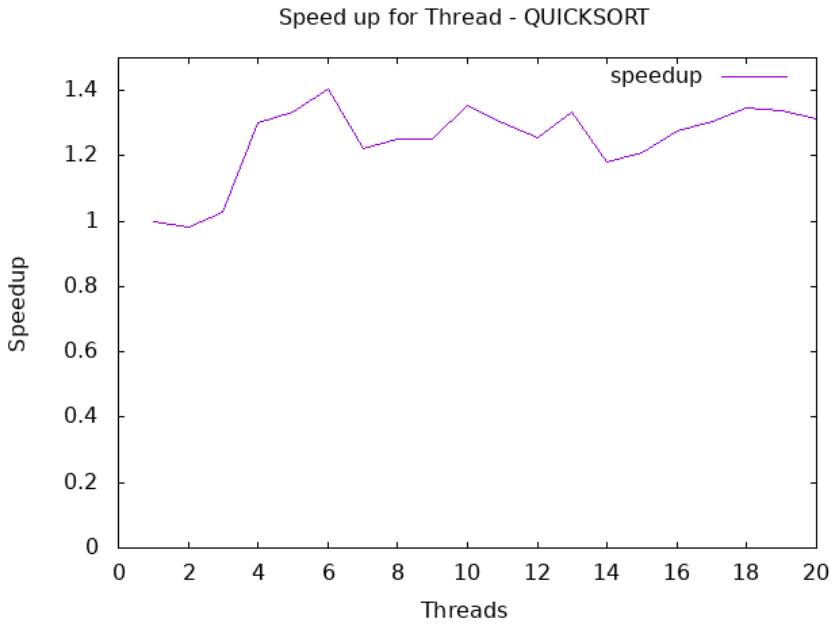


Figura 3.19: Gráfica de speed-up según el número de threads.

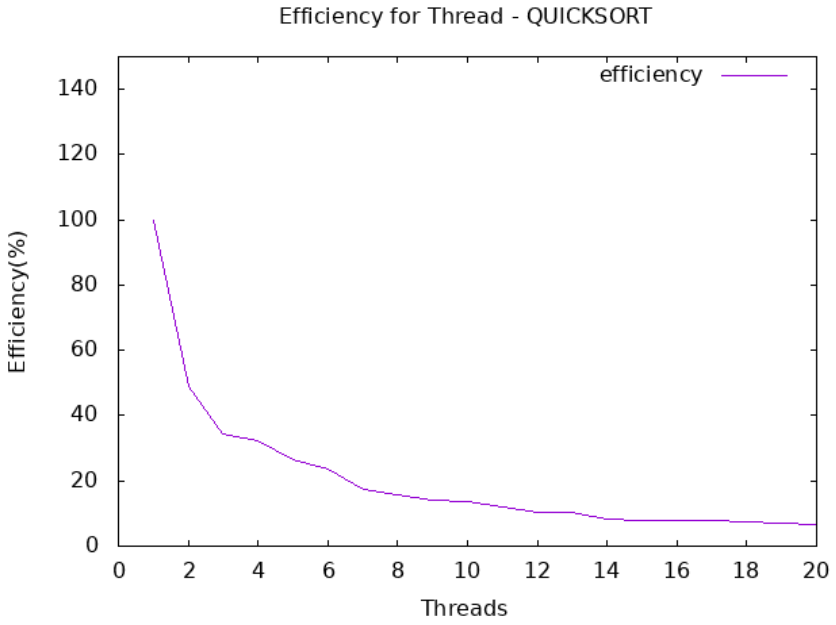


Figura 3.20: Gráfica de eficiencia según el número de threads.

En este caso, se puede comprobar que el tiempo de ejecución se ve reducido aumentando el número de hilos, hasta llegar a un valor en que se mantiene más constante (a partir de 12 hilos).

En cuanto al speed-up, se concluye que al utilizar 2-3 hilos no es óptimo el uso de paralelismo, ya que la carga de trabajo repartida no es rentable en comparación con la inicialización de los procesos. Sin embargo, a partir de los 4 hilos, el speed-up sobrepasa el 1, lo que indica que mejora la ejecución. Especialmente, para esta tarea el número óptimo de hilos es 6, ya que es el valor máximo de speed-up.

Cabe destacara que para este experimento se ha utilizado un mismo vector a ordenar, ya que al ser diferentes, los tiempos de ordenación no sólo dependen del número de hilos sino también de la cantidad de operaciones que se deban hacer para cada vector.

### Algoritmo de ordenación HeapSort en paralelo

Finalmente, se ha paralelizado el algoritmo de ordenación HeapSort de un modo muy similar al QuickSort. Este algoritmo se basa también en un bucle for, el cual se paraleliza.

Las gráficas resultantes se muestran en las gráficas 3.21 3.22 3.23.

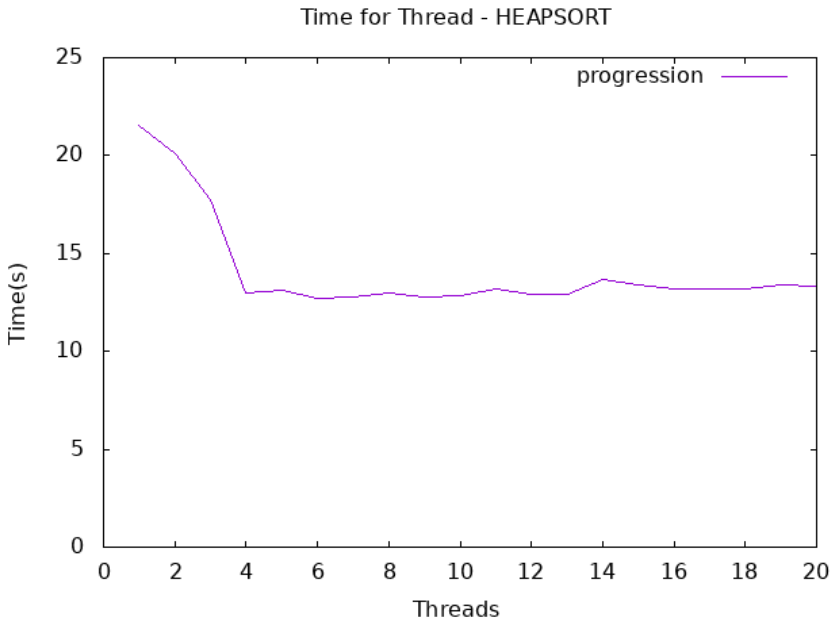


Figura 3.21: Gráfica de tiempo de ejecución según el número de threads.

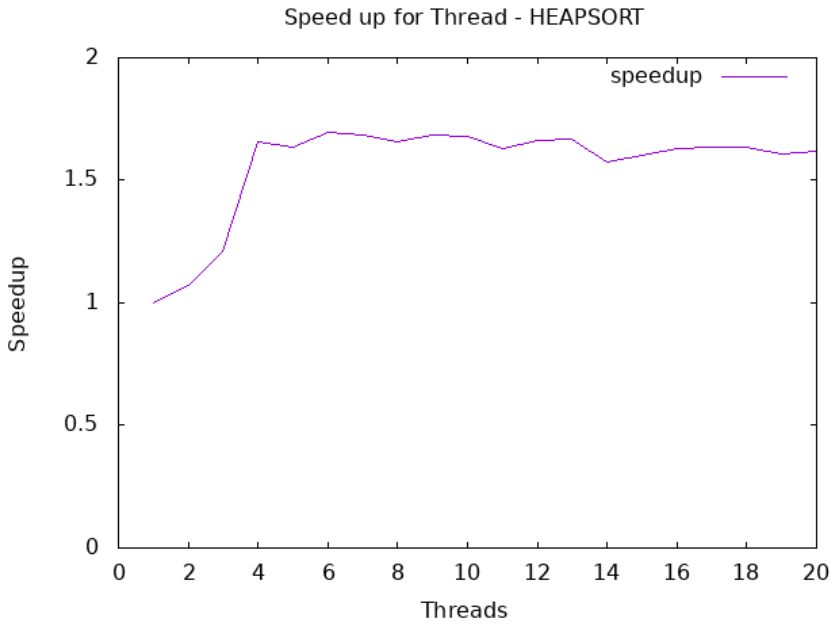


Figura 3.22: Gráfica de speed-up según el número de threads.

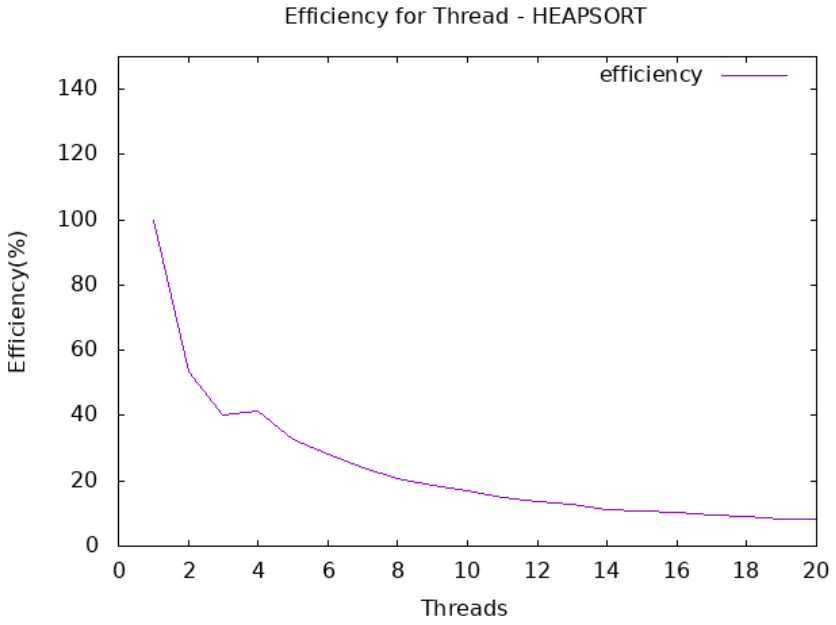


Figura 3.23: Gráfica de eficiencia según el número de threads.

En este caso, se observa que el tiempo se reduce hasta el uso de 4 hilos, a partir del cual, el tiempo de ejecución se puede considerar constante. Del msimo modo, el speed-up aumenta hasta 4 hilos, y se mantiene prácticamente constante.

# Parte 4

# Conclusiones

En este proyecto se ha estudiado la API OpenMP, con esta, es posible aplicar paralelismo en lenguajes como C y C++. Además, se ha observado que OpenMp cuenta con una amplia gama de herramientas para esta función.

Concretamente, se ha aplicado multithreading, probando así, los resultados de este con varios programas, constructores y algoritmos de ordenación como quicksort.

Tal como se ha demostrado en este proyecto, se han llegado a unas claras conclusiones. El multithreading resulta factible para un cierto tipo de programas, sobre todo en aquellos más extensos y que tienen más partes paralelizables, como es lógico.

De esta manera, se ha observado que, en aquellos programas que constan de tareas muy simples, con baja carga, se reduce el tiempo de ejecución levemente. Sin embargo, al ser aplicado en algoritmos con cargas mayores como se realizó con quicksort, se obtuvo una caída de tiempo de ejecución razonable.

De igual modo ocurre con el algoritmo heapsort, en el que la caída es incluso mayor.

Asimismo, cabe destacar la importancia de estudiar el número de hilos más óptimo para cada tarea, ya que, como se ha observado, la relación de mejora con respecto al número de hilos no es lineal para todo su dominio. Aumenta hasta cierto valor, a partir del cual, no se presenta mejora alguna o incluso se reduce la eficacia.

Por tanto, se puede afirmar que existe una limitación física a la hora de reducir el tiempo de ejecución a nivel de hardware, sin embargo, con el uso del paralelismo, se puede continuar con este progreso, sobre todo, en algoritmos que consten de muchas partes con posibilidad de ejecutarse simultáneamente.

# Parte 5

# Bibliografía


- https://www.guru99.com/difference-between-multiprocessing-and-multithreading.html#:~:text=Multiprocessing%20allocates%20separate%20memory%20and,as%20that%20of%20the%20process.&text=Multithreading%20system%20executes%20multiple%20threads%20of%20the%20same%20or%20different%20processes
- https://ferestrepoca.github.io/paradigmas-de-programacion/paralela/paralela_teoria/index.html#two
- https://sites.google.com/site/arquitecturadecomputadoresis/paralelismo-de-procesadores
- https://www.mulesoft.com/resources/api/what-is-an-api
- https://en.wikipedia.org/wiki/Multiprocessing
- https://es.wikipedia.org/wiki/OpenMP
- https://docs.microsoft.com/es-es/cpp/parallel/openmp/reference/openmp-directives?view=msvc-160
- https://www.ditec.um.es/~javiercm/curso_psba/sesion_03_openmp/PSBA_OpenMP.pdf
- https://www.ecured.cu/OpenMP
- https://upcommons.upc.edu/bitstream/handle/2117/117706/130125.pdf?sequence=1&isAllowed=y
- https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf
- https://www.openmp.org/wp-content/uploads/openmp-examples-5-0-1.pdf
- https://docs.microsoft.com/es-es/cpp/parallel/openmp/reference/openmp-directives?view=msvc-170