# Práctica 3
# Paralelismo a nivel de hilos
# Parte I: Introducción a OpenMP

**Objetivos**:

- Aprender a paralelizar una aplicación en una máquina paralela de memoria centralizada mediante hilos (*threads*) usando el estilo de *variables compartidas*.
- Estudiar el API de OpenMP y aplicar distintas estrategias de paralelismo en su aplicación.
- Aplicar métodos y técnicas propios de esta asignatura para estimar las ganancias máximas y la eficiencia del proceso de paralelización.
- Aplicar todo lo anterior a un problema de complejidad y envergadura suficiente.

**Desarrollo**:

En esta práctica los/las estudiantes deben paralelizar, usando OpenMP, la solución a un problema dado para aprovechar los distintos núcleos de los que dispone cada ordenador de prácticas. Se paralelizará por tanto para un sistema multiprocesador (máquina paralela de memoria centralizada), en el que todos los núcleos de un mismo encapsulado ven la misma memoria, es decir, un puntero en un núcleo es el mismo puntero para el resto de los núcleos del microprocesador.

La práctica está dividida en 2 partes. La primera, está pensada para que los alumnos se familiaricen con OpenMP y las posibilidades que ofrece para paralelizar soluciones a problemas que se puedan utilizar en sistemas multiprocesador. En la segunda parte se propondrá un problema concreto para cuya solución y estudio se utilizará lo aprendido en la primera parte.

## Tarea 0: Entrenamiento previo (5% de la nota):

Observe el siguiente programa en C donde se suman dos vectores de *floats* empleando OpenMP para paralelizar el cálculo.

```
#include <omp.h>
#define N 1000
#define CHUNKSIZE 100

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

        int i, chunk;
        float a[N], b[N], c[N];

        /* Inicializamos los vectores */
        for (i=0; i < N; i++)
          a[i] = b[i] = i * 1.0;
        chunk = CHUNKSIZE;

        #pragma omp parallel shared(a,b,c,chunk) private(i)
          {
                  #pragma omp for schedule(dynamic,chunk) nowait
                  for (i=0; i < N; i++)
                    c[i] = a[i] + b[i];

          }   /* end of parallel region */
```

```
}
```

Revise la documentación de OpenMP (API, tutoriales, este enlace: https://computing.llnl.gov/tutorials/openMP/, etc...) y responda:

**0.1** ¿Para qué sirve la variable **`chunk`**?

**0.2** Explique **completamente** el *pragma* :

```
#pragma omp parallel shared(a,b,c,chunk) private(i)
```

¿Por qué y para qué se usa **`shared(a,b,c,chunk)`** en este programa?

¿Por qué la variable `i` está etiquetada como **`private`** en el pragma?

**0.3** ¿Para qué sirve **`schedule`**? ¿Qué otras posibilidades hay?

**0.4** ¿Qué tiempos y otras medidas de rendimiento podemos medir en secciones de código paralelizadas con OpenMP?

## Tarea 1: Estudio del API OpenMP [Parte individual obligatoria (25% de la nota)]

Se deberá estudiar el API de OpenMP y su uso con GNU GCC, comprobando el correcto funcionamiento de algunos de los ejemplos que hay disponibles en Internet (p.ej. ejemplo https://lsi.ugr.es/jmantas/ppr/ayuda/omp_ayuda.php)

**Cada miembro del grupo** deberá realizar un pequeño tutorial a modo de "libro de recetas de paralelismo con OpenMP" con **distintos ejemplos de aplicación** del paralelismo que ofrece OpenMP en función de distintas **estructuras de software**.

## Tarea 2: Ejemplo de problema paralelizable (25% de la nota)

Un problema muy paralelizable que ha tomado relevancia en los últimos años es la resolución del "*proof of work*" de la criptomoneda Bitcoin. El "*proof of work*" es el mecanismo utilizado por Bitcoin para resolver el dilema de los generales bizantinos. Este dilema plantea el problema que se presenta cuando varios generales deben actuar de forma coordinada para asediar una ciudad acatando la orden de uno de ellos, pero no se fían de que todos la acaten o incluso de que la orden sea legítima. En el caso del *blockchain*, este dilema hace referencia al problema de confiar en los distintos nodos que componen la red, ya que alguno de ellos puede intentar añadir bloques falsos para ganar criptomonedas. La forma en la que la mayoría de *blockchains* resuelven este problema es mediante el llamado "*proof of work*". Éste consiste en resolver un problema que sea poco costoso de verificar pero con un coste computacional de resolución elevado, de esta forma, cualquiera que quiera unirse a la red para validar transacciones, debe "pagar" ese coste computacional. El problema usado por Bitcoin consiste en encontrar un *hash* (resumen criptográfico) para el nuevo bloque cuyos primeros n caracteres sean 0. Esto se consigue mediante fuerza bruta modificando el valor de una variable del bloque (*nonce*) y calculando su *hash* hasta que se encuentra uno que cumpla con la condición.

Como podéis observar, es un problema muy sencillo de resolver pero que requiere un alto coste computacional ya que se deben generar millones de *hashes* hasta encontrar uno que cumpla con la condición. Una vez que un nodo encuentra un *hash* válido, éste realiza un *broadcast* al resto de

nodos para que lo validen, si la mayoría lo valida correctamente, el bloque se asigna a la cadena. Para incentivar la colaboración para resolver este problema y asumir el coste que conlleva, al nodo que antes consiga resolverlo se le concede una remuneración en forma de criptomonedas. De ahí que se conozca como "minar" criptomonedas al proceso de obtener un bloque válido para una transacción.

Esta forma de resolver el dilema de los generales bizantinos presenta varios problemas:

- Coste computacional (y energético) elevado y agregado entre todos los mineros que realizan trabajo innecesario y redundante para resolver el problema.
- Lentitud de las transacciones al tener que esperar a que algún minero resuelva el problema.
- Incremento de la potencia de cálculo que hace que el tiempo de computación disminuya y haciendo peligrar la seguridad. Para solventar este problema se suele incrementar periódicamente el número de 0s que debe contener el *hash*.
- Si alguna entidad o grupo malicioso se hace con el control de más del 50% de la red, podrá incorporar bloques fraudulentos.

### Tarea 2.1: paralelizar el problema

Adjunto encontraréis un zip con el código que resuelve este problema de forma secuencial. A lo largo de la primera y segunda sesión de prácticas, con la ayuda del profesor, tendréis que proponer una paralelización con OpenMP.

El código adjunto cuenta con varios archivos, el archivo `miner.h` contiene el código de la clase Miner que se encarga de realizar el minado. Por tanto, este es el fichero que debéis paralelizar, en concreto, el método "`Block* mine(Block *block)`".

Para probar el tiempo que habéis optimizado, tenéis un fichero "`test.cc`" donde ejecuta el minado de un bloque. Para compilarlo ejecutad "`g++ -Wall test.cc -o test -lcrypto -fopenmp`". Esto enlaza con las librerías criptográficas que ya están instaladas en los PCs del laboratorio, sin embargo, en vuestro PC probablemente tengáis que instalarlas usando el comando "`sudo apt-get install libssl-dev`".

### Tarea 2.2: poner a prueba la paralelización

En la siguiente sesión de prácticas, cada grupo ejecutará su programa paralelizado conectado a una aplicación servidor que actuará como solicitante de una transacción y que será ejecutada por el profesor. Al solicitar una transacción, el servidor enviará una copia del nuevo bloque a cada grupo, que tendrá que resolver el hash y enviárselo de nuevo al servidor. El primer grupo que lo consiga recibirá **0.25 puntos extra en la práctica.** Se harán un total de 4 transacciones.

Para compilar el programa cliente "blockchain.cc" debéis ejecutar "`g++ -Wall blockchain.cc -o blockchain -lcrypto -fopenmp`". Para ejecutarlo, poned como primer argumento la ip del servidor y como segundo argumento un identificador del grupo como el nombre de uno de los integrantes ya que ese será el nombre que use el servidor para identificar a cada grupo.

**Notas generales a la práctica. Parte 1:**

- Las respuestas y la documentación generadas tanto en las tareas de la parte I como de la II se integrarán en la memoria conjunta que se entregará **al final de la práctica 3**.
- La implementación realizada tendrá que poder ejecutarse bajo el sistema operativo Linux del laboratorio de prácticas, aunque en casa puede trabajar con otros compiladores y sistemas operativos que soporten OpenMP (icc, Clang, Visual C++, ...)

- Cada grupo entregará, además de las aplicaciones desarrolladas, una memoria, estructurada según indicaciones del profesor, con la información obtenida en las partes I y II, incluyendo los apartados individuales como anexo.
- La información referente a OpenMP se encuentra en www.openmp.org. Para compilar usa g++ con el modificador `-fopenmp`. Para poder compilar con las librerías criptográficas hay que instalar libssl-dev y enlazarlas usando `-lcrypto`.
- La práctica se deberá entregar mediante el método que escoja su profesor de prácticas **antes** de la sesión de prácticas de la semana del 28 de noviembre de 2022.

**Tarea 3: (próximamente se actualizará este guión de prácticas con ésta última tarea (45% de na nota)**

**Nota:**

*Los trabajos teóricos/prácticos realizados han de ser originales. La detección de copia o plagio supondrá la calificación de "0" en la prueba correspondiente. Se informará la dirección de Departamento y de la EPS sobre esta incidencia. La reiteración en la conducta en esta u otra asignatura conllevará la notificación al vicerrectorado correspondiente de las faltas cometidas para que estudien el caso y sancionen según la legislación.*