Práctica 3
Paralelismo a nivel de hilos
Parte II: Paralelización con OpenMp del entrenamiento y evaluación de un clasificador K-Nearest Neighbors utilizando Cross-Validation y Fine Tuning.

Objetivos:

- Obtener una versión paralela con OpenMP de la solución secuencial proporcionada.
- Analizar y explotar en todos los casos los diferentes tipos posibles de paralelismo que se perciban en la solución secuencial dada.
- Realizar pruebas suficientes que permitan estimar valores de rendimiento de la versión paralela frente a la secuencial.
- Visualizar los datos de rendimiento de la forma más adecuada para poder argumentar los resultados obtenidos y relacionarlos con las decisiones de paralelización tomadas.

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 2: Aspectos que impactan en un diseño paralelo
A continuación, se presentan 2 códigos que tratan de ejemplificar aspectos a habituales a tener cuenta a la hora de paralelizar un algoritmo. Su comprensión puede ser de utilidad para las siguientes tareas.

Tarea 2.1 Analice el siguiente código anterior en el cual se inicializa un **vector stl** de gran tamaño de dos formas diferentes. Ejecútelo y responda a las preguntas planteadas.

```
#include <iostream>
#include <vector>
#include <time.h>
#include <ctime>

#define SIZE 1000000

using namespace std;

int main()
{
    // A:
    clock_t begin = clock();
```

```
    // declaramos un vector sin inicializar.
    std::vector<int> a;
    // usamos push_back para añadir cada elemento.
    for(int i=0;i<SIZE;i++)
    {
        a.push_back(i);
    }
    double aTime = double(clock() - begin) / CLOCKS_PER_SEC;
    cout<<"Elapsed time: "<<aTime<<"s"<<endl;

    // B:
    begin = clock();
    // declaramos un vector inicializado al tamaño requerido.
    std::vector<int> b(SIZE);
    for(int i=0;i<SIZE;i++)
    {
        a[i] = i;
    }
    double bTime = double(clock() - begin) / CLOCKS_PER_SEC;
    cout<<"Elapsed time: "<<bTime<<"s"<<endl;
    cout<<"Gain: "<<aTime/bTime<<endl;
}
```

- ¿Qué diferencias observas entre ambas versiones?
- ¿Qué versión es más rápida?
- ¿A qué se debe la diferencia de tiempo entre la primera y la segunda versión?
- **¿Se podrían paralelizar ambas versiones?** Razone la respuesta.

Ojo con las posiciones en memoria de las matrices

Tarea 2.2 Analice el siguiente código en el cual se recorre una matriz de gran tamaño de dos formas diferentes. Ejecútelo y responda a las preguntas planteadas.

```
#include <iostream>
#include <vector>
#include <time.h>
#include <ctime>

const int SIZE = 1000;

using namespace std;

int main()
{
    // A:
    clock_t begin = clock();
    // declarar una matriz de tamaño SIZExSIZE.
    int a[SIZE][SIZE];
    // una matriz in c se almacena en memoria como un array de una única
    // dimensión por filas.
    // Para una matriz de 3x3:
    // 1 2 3
    // 4 5 6
    // 7 8 9
    // se almacena en memoria como el array: 1 2 3 4 5 6 7 8 9
    // si recorremos la matriz por columnas, visitamos las celdas en el siguiente
    // orden: 1 4 7 2 5 8 3 6 9
    for(int i=0;i<SIZE;i++)
    {
        for(int j=0;j<SIZE;j++)
            a[j][i] = i+j;
    }
```

```
    double aTime = double(clock() - begin) / CLOCKS_PER_SEC;
    cout<<"Elapsed time: "<<aTime<<"s"<<endl;

    // B:
    begin = clock();
    int b[SIZE][SIZE];
    // si recorremos la matriz por filas, visitamos las celdas en el siguiente
    // orden: 1 2 3 4 5 6 7 8 9
    for(int i=0;i<SIZE;i++)
    {
        for(int j=0;j<SIZE;j++)
            b[i][j] = i+j;
    }
    double bTime = double(clock() - begin) / CLOCKS_PER_SEC;
    cout<<"Elapsed time: "<<bTime<<"s"<<endl;
    cout<<"Gain: "<<aTime/bTime<<endl;
}
```

- ¿Qué diferencias hay entre ambas versiones?
- Pruebe diferentes tamaños de matriz hasta que observes diferencias notables de tiempo.
  - ¿Qué versión se ejecuta más rápido?
  - ¿A partir de qué tamaño comienza a notarse la diferencia de tiempo?
  - ¿A qué se debe la diferencia de tiempo entre ambas versiones?

Tarea 3: Paralelización del entrenamiento y evaluación de un clasificador K-Nearest Neighbors utilizando Cross-Validation y Fine Tuning mediante OpenMp

Todos los grupos de cada turno de prácticas deben acometer la paralelización del problema planteado. Para ello se analizará la solución secuencial facilitada siguiendo las indicaciones del enunciado y los profesores. Posteriormente, se transformará esta, utilizando OpenMP, para que incorpore paralelismo a nivel de hilos.

Contexto del Problema:

El problema que vamos a tratar en esta práctica es el entrenamiento y evaluación de un clasificador K-Nearest Neighbors utilizando técnicas de validación Cross-Validation y Fine Tuning. En concreto, vamos a usar el dataset MNIST que consiste en imágenes de dígitos manuscritos utilizado para problemas de reconocimiento de caracteres (OCR). Este dataset cuenta con un total de 60000 muestras de entrenamiento y 10000 muestras de test.

Machine learning:

El machine learning es una rama de la inteligencia artificial cuyo objetivo consiste en hacer algoritmos que permitan el aprendizaje (semi)automático para realizar cierta tarea. El caso más común son los algoritmos clasificadores que, como su nombre indica, realizan operaciones de clasificación de muestras en un conjunto de categorías dado.

La forma en la que estos algoritmos aprenden se puede dividir en 3 tipos:

I. Aprendizaje supervisado: el algoritmo requiere de un conjunto de muestras previamente etiquetadas y su labor consiste en intentar encontrar las peculiaridades que hacen que las muestras de una categoría se diferencien de las de otra categoría.

II. Aprendizaje no supervisado: el algoritmo recibe como entrada un conjunto de muestras sin etiquetar y la labor del algoritmo consiste en encontrar patrones de similitud entre las muestras y etiquetarlas.
III. Aprendizaje semisupervisado: combina los dos casos anteriores.

Evaluación de clasificadores

Existen diferentes métricas para evaluar el rendimiento de un clasificador. Entre estas métricas tenemos *Accuracy*, *Precision* y *Recall*. Estas métricas miden cómo de bien es capaz de clasificar muestras no presentes en el conjunto de entrenamiento. Una evaluación correcta de un clasificador es fundamental para obtener un clasificador capaz de generalizar un problema. Un clasificador mal evaluado, puede ser capaz de clasificar muy bien el dataset utilizado para entrenarlo, pero no ser capaz de funcionar correctamente con muestras que no haya visto antes. Esto es lo que se conoce como *overfitting*.

Para garantizar que un clasificador es capaz de generalizar un problema, se suele dividir el dataset en dos subconjuntos, un subconjunto de entrenamiento y otro de test. El primero sirve para entrenar el clasificador y el segundo para evaluar su rendimiento. De esta forma, el rendimiento se evalúa con muestras que no se han tenido en cuenta para el entrenamiento. Además, para entrenar el clasificador, se suele realizar una segunda división al conjunto de entrenamiento dejando un subconjunto conocido como conjunto de validación para evaluar el rendimiento del clasificador dentro del proceso de entrenamiento y ajustar parámetros para mejorar el rendimiento.

N-Fold Cross-Validation

Atendiendo a estas métricas, existen técnicas de evaluación exhaustiva de un clasificador dado un dataset de muestras. La técnica de dividir el conjunto de entrenamiento para obtener un conjunto de validación se puede repetir sistemáticamente para evaluar el rendimiento del clasificador utilizando diferentes subconjuntos de entrenamiento y de validación y haciendo la media de las métricas obtenidas. Esto es lo que se conoce como *Cross-Validation*. Dependiendo de la cantidad de divisiones que hagamos del dataset, se definen principalmente dos tipos de Cross-Validation:

I. N-Fold: consiste en dividir el dataset en N subconjuntos y realizando N validaciones del clasificador utilizando un subconjunto diferente como validación y el resto como entrenamiento.
II. Leave-one-out: esta es la técnica Cross-Validation más exhaustiva y consiste en realizar un N-Fold donde N es igual al tamaño del dataset, es decir, realizar tantas validaciones como muestras haya en el dataset utilizando en cada validación una muestra como validación y el resto como entrenamiento.

Clasificador KNN:

El clasificador K-Nearest Neighbors es uno de los clasificadores más sencillos que existen. Funciona calculando todas las distancias entre la muestra de test y las muestras de entrenamiento, ordenándolas de menor a mayor y eligiendo la etiqueta de las muestras de entrenamiento más repetidas entre las K más cercanas. KNN, por tanto, funciona muy bien en problemas de clasificación donde las muestras de la misma etiqueta están agrupadas en el espacio y son separables linealmente.

Este clasificador utiliza aprendizaje supervisado y, dada su naturaleza, el proceso de entrenamiento de este clasificador consiste simplemente en añadir muestras al conjunto de entrenamiento. Esta característica hace que este clasificador tan sencillo sea realmente potente en tareas de clasificación

en las que las condiciones del problema son variantes ya que, por ejemplo, se puede añadir una nueva categoría simplemente añadiendo una nueva muestra etiquetada con esa categoría. Un ejemplo en el que esta característica es muy útil es en el reconocimiento facial, ya que permite añadir nuevos usuarios sin necesidad de reentrenar el clasificador, simplemente añadiendo nuevas caras al conjunto de entrenamiento.

Fine tuning

Normalmente, los clasificadores tienen multitud de parámetros que permiten ajustarlo a particularidades del problema a tratar. En el caso del KNN, únicamente dispone del parámetro K. Una forma de encontrar el mejor valor de los parámetros de un clasificador consiste en hacer lo que se denomina proceso de **fine tuning** que consiste en realizar múltiples validaciones del rendimiento del clasificador utilizando diferentes valores para los posibles parámetros. Este proceso se conoce también como *grid search* ya que el resultado final es una matriz N dimensional con el rendimiento del clasificador para cada combinación de parámetros. En el caso del KNN, el resultado será un vector.

Caso especial (opcional para subir nota)

Para poder explotar el paralelismo funcional y forzar sincronización entre procesos, vamos a suponer un caso especial de ejecución de nuestro programa de fine tuning con Cross-Validation y evaluación final del clasificador. Este caso especial consiste en tratar de ejecutar en paralelo la parte de fine tuning y la de evaluación final. Tal y como se puede observar, la evaluación final del clasificador depende del resultado del fine tuning ya que hasta que éste no determina cual es el mejor parámetro K, la evaluación final del clasificador no se puede realizar. Pues bien, vamos a forzar el paralelismo funcional de estas dos partes del problema realizando en paralelo el fine tuning y la evaluación final. Para ello, cada vez que el fine tuning encuentre un nuevo mejor K, la función de evaluación deberá invalidar el procesamiento actual y reiniciarlo con el nuevo mejor K. De esta forma, el proceso de evaluación funcionará bajo un criterio especulativo en el cual asume que el mejor valor final de K será el valor que se acaba de obtener. Si esta especulación es correcta, habremos conseguido optimizar el tiempo final de ejecución de ambos casos.

Si se consigue implementar este caso de ejecución especulativa para explotar el paralelismo funcional, será muy útil comparar métricas obtenidas frente a la versión que no incluya esta característica (Tcpu, Speed-Up o Ganancia, etc.).

Pasos recomendados:

a. **IMPORTANTE**: Para poder ejecutar el código secuencial, leer y seguir los pasos del archivo *README* incluido en el archivo comprimido con los fuentes que se entrega junto a este enunciado. Especialmente, i**nstalar OpenCV**, siguiendo las instrucciones de instalación de OpenCV en ubuntu (basta con instalar dependencias, cmake y make install, normalmente aparecen pasos extras para poder compilar con g++ fácilmente, pero se pueden obviar ya que incluimos un cmake para la práctica que nos ahorra ese problema)

Para crear el make y obtener el dataset:

```
chmod a+x get_dataset.sh
./get_dataset.sh
```

Para compilar:

```
cd build
make
```

Para ejecutar:

```
./cv
```

b. **Analizar la solución secuencial facilitada para asegurar la comprensión del problema**. Prestar atención a detalles vistos en prácticas anteriores que tengan importancia para diseñar la solución paralela: variables, talla del problema, etc.
c. Realice una representación gráfica en forma de grafo de control de flujo del funcionamiento de la solución secuencial dada.
d. Utilizando OpenMP y a la vista de los detalles anteriores, diseñe una solución paralela a nivel de hilo a partir de la versión secuencial
e. Realice pruebas de su versión paralela para diferentes casos y configuraciones tomando medidas de tiempos y ganancias.
f. Realice una representación gráfica en forma de grafo de control de flujo del funcionamiento de la solución paralela basada en OpenMP.
g. Comente los siguientes aspectos tanto de la solución secuencial como de la paralela según se indique:

Sobre el código implementado

- Comentar las porciones de código de más interés tanto de la solución secuencial como de la paralela implementadas:
  - Aspectos que definan la talla del problema.
  - Estructuras de control del código de especial interés en la solución al problema.
  - Es importante que se justifique **lo más detalladamente** posible los cambios que se hayan realizado con respecto a la versión secuencial para paralelizar la solución con OpenMP.
  - Instrucciones y bloques de OpenMp utilizados para la paralelización del código.

Sobre el paralelismo explotado

- **Revisando la documentación de la *"Unidad 3. Computación paralela"***. Indique lo siguiente con respecto a su práctica:
  - Tipos de paralelismo usado.
  - Modo de programación paralela.
  - Qué alternativas de comunicación (explícitas o implícitas emplea su programa).
  - Estilo de la programación paralela empleado.
  - Tipo de estructura paralela del programa

Sobre los resultados de las pruebas realizadas y su contexto

- Caracterización de la máquina paralela en la que se ejecuta el programa. (p.ej. Número de nodos de cómputo, sistema de caché, tipo de memoria, etc.).
  En Linux use la orden: `cat /proc/cpuinfo` para acceder a esta información.
- ¿Qué significa la palabra `ht` en la salida de la invocación de la orden anterior? ¿Aparece en su ordenador de prácticas?
- Calcular lo siguiente (con gráficas asociadas y su explicación breve):
  - Ganancia en velocidad (*speed-up*) en función del número de unidades de cómputo (*threads* en este caso) y en función de los parámetros que modifiquen el **tamaño del problema** (p.ej. dimensiones de una matriz, número de iteraciones considerado, etc.…).

- Comente los resultados de forma razonada. ¿Cuál es la ganancia en velocidad máxima teórica?
  Nota: Puede entregar gráficas en 2D o 3D según resulte más ilustrativo.
- **Responda de forma justificada:** ¿Cuál es la implementación más eficiente de las 2?

Tarea 4: Redacción de una memoria en la que se analice el diseño utilizando OpenMP y los resultados obtenidos.
Se redactará una memoria que incluya, de forma clara, las respuestas a cada una de las tareas, con especial atención al análisis de la solución paralela y los resultados obtenidos probándola.

Notas generales a la práctica:

- Las respuestas y la documentación generadas tanto en las tareas de la parte I como de la II se integrarán en la memoria conjunta y estructurada, incluyendo los apartados individuales como anexo, que se entregará al final de la práctica 3.
- Para el trabajo en la parte II y en la memoria se utilizará la plataforma de desarrollo GitHub. Esta permitirá una gestión de un **repositorio común de trabajo** que deberá crear cada grupo incluyendo a sus miembros y al profesor. Este dará a conocer su usuario en dicha plataforma para ser añadido.
- 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++, ...)
- La información referente a OpenMP se encuentra en www.openmp.org. Para compilar use g++ con los siguientes modificadores `-fopenmp` y `-std=c++11` para que use las librerías del estándar c++11.
- **Es obligatorio** entregar un *Makefile* con las reglas oportunas para compilar y limpiar cada programa solicitado de forma rápida y sencilla.
- La práctica se deberá entregar mediante el método que escoja su profesor de prácticas antes de la sesión de prácticas de la semana del 22 de noviembre de 2021.

***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 (Reglamento de disciplina académica de los Centros oficiales de Enseñanza Superior y de Enseñanza Técnica dependientes del Ministerio de Educación Nacional BOE 12/10/1954).