Práctica 2.- Resolución de problemas y análisis de rendimiento con MPI

Arquitectura y Computación de Altas Prestaciones. Universidad de Granada. Curso 2023/24

Objetivos: En esta práctica se pretende que los estudiantes aborden problemas más complejos que en la anterior, y hagan además mediciones rigurosas de rendimiento. Esto último se culminará con la realización de una gráfica de aceleración.

Bases: Esta práctica se realizará individualmente y los programas desarrollados estarán pensados para funcionar en un entorno Linux. Habrá que entregar los archivos de código fuente desarrollados y una memoria (en formato PDF) que explique el trabajo realizado. La entrega se realizará mediante la actividad que se creará en SWAD para tal efecto. Las entregas tardías supondrán una penalización en la nota (0.5 puntos por día hasta alcanzar el límite de 5).

Entorno: La mayor parte del trabajo se puede realizar en equipos personales. Sin embargo, según la actividad, habrá que acceder al cluster de prácticas ATCGrid. Cada estudiante tendrá un usuario y contraseña en ATCGrid. Serán asignados por el profesor de forma individual e intransferible. Tampoco se podrán cambiar las claves recibidas. El acceso al cluster de prácticas puede requerir estar conectado por VPN a la red de la universidad.

Ejercicio 1 (2 puntos)

En la práctica anterior se abordó la paralelización del cálculo de π mediante una de las aproximaciones numéricas que existen. Ahora se pide que trabajes con otra opción de resolución del mismo problema, la Serie de Leibniz. Dicha serie tiene la siguiente forma:

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \frac{\pi}{4}$$

Y se puede generalizar así:

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \frac{\pi}{4}$$

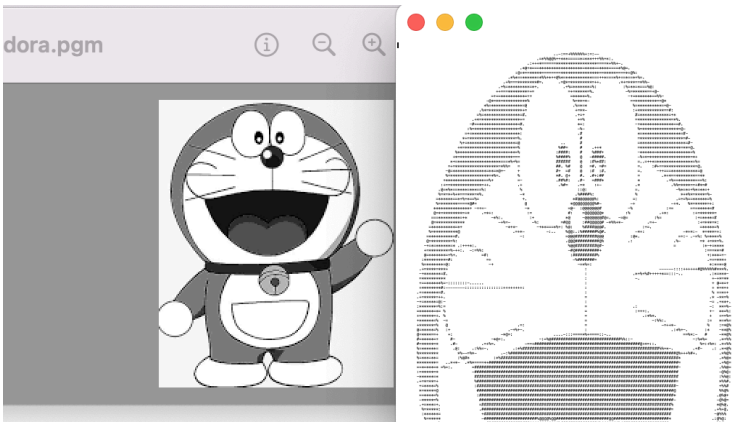
Se proporciona entonces una implementación secuencial de este cálculo (archivo "*pi_Leibniz.c*"). Paraleliza dicho código usando MPI y buscando un balanceo de la carga lo más equilibrado posible. Como siempre, asume que sólo el proceso 0 tiene acceso a los

parámetros iniciales de ejecución. Incluye además una medición del tiempo empleado en el cálculo. ¿Qué opinas de esta versión respecto a la vista en la práctica 1?

Ejercicio 2 (3 puntos)

La forma más simple de imagen digital tiene una representación matricial bi-dimensional en la que cada píxel contiene un número entero. Cada uno de esos valores está acotado, normalmente entre 0 y 255, según la resolución en niveles de gris, a consecuencia de la cuantización y discretización de la intensidad luminosa recogida (por un sensor, si hablamos de una fotografía). Se tiene entonces lo que se conoce como una imagen en "blanco y negro". Uno de los formatos más sencillos para almacenar este tipo de imagen es el PGM ("*Portable Gray Map*") (consulta https://netpbm.sourceforge.net/doc/pgm.html para más información).

En este contexto, se proporciona una librería simplificada para la lectura de ese tipo de imágenes ("pgm.c" y "pgm.h"). Con ellas, se pide implementar un programa MPI capaz de convertir imágenes PGM a una representación ASCII:



Sigue la siguiente tabla de conversión:

Rango de gris	Carácter asociado
[0-25]	#
[26-51]	@
[52-77]	%
[78-103]	+
[104-129]	*
[130-155]	=
[156-181]	:
[182-207]	-
[208-233]	.
[234-255]	' ' (espacio)

Aquí se muestra un ejemplo de la lógica secuencial indicando la parte a paralelizar:

```
unsigned char** Original = pgmread(argv[1], &rows, &cols);
char** Salida = (char**) GetMem2D(rows, cols, sizeof(char))

toASCII(Original, rows, cols, Salida);//MPI

toDisk(Salida, rows, cols);
```

El proceso 0 deberá centralizar toda la entrada/salida. Será por tanto responsable de leer la imagen indicada y volcar finalmente el resultado a disco. No obstante, tendrá también que hacer una parte de la conversión, y el reparto de de la misma será lo más equitativo posible.

Echa un vistazo a la función GetMem2D, ¿qué nos aporta en contraste con el siguiente enfoque? Piensa además si nos facilita o dificulta la labor para usar MPI.

```
void ** GetMem2D(int rows, int columns, int sizeofTipo) {
  int i;
  void ** h;
  h = malloc(sizeof(void*)*rows);

  for(int i = 0; i<rows; i++){
    h[i] = GetMem(columns, sizeofTipo);
  }

  return h;
}
```

Finalmente, ¿qué opinas del nivel de paralelismo que tiene este problema? Relaciónalo con la Ley de Amdahl.

Ejercicio 3 (5 puntos)

Se pide que implementes, primero en secuencial y luego en MPI, y de la forma más eficiente posible (balanceo de carga equilibrado y con todos los procesos disponibles trabajando), el cálculo del coeficiente de Tanimoto. Éste, también conocido como coeficiente o índice de Jaccard, mide la similitud entre dos conjuntos. Concretamente, este coeficiente se define de la siguiente forma, dados dos conjunto A y B:

$$Tanimoto(A,\ B) = \frac{|A \cap B|}{|A \cup B|}$$

Recuerda que $|\cdot|$ hace referencia a la cardinalidad o número de elementos en un conjunto, así como que $\cap$ y $\cup$ hacen referencia a la intersección y unión de conjuntos, respectivamente. A nivel conceptual, estos enlaces te serán de ayuda:

- https://es.planetcalc.com/1664/
- https://es.wikipedia.org/wiki/%C3%8Dndice_de_Jaccard

Un ejemplo de llamada (obviando la parte MPI) sería: ./programa tamA tamB

El proceso 0 será el que tenga acceso inicialmente a los conjuntos en su totalidad, así como el encargado de mostrar resultados. Incluirá también una medición del tiempo empleado en el cálculo propiamente dicho (sin contar la inicialización de los conjuntos).

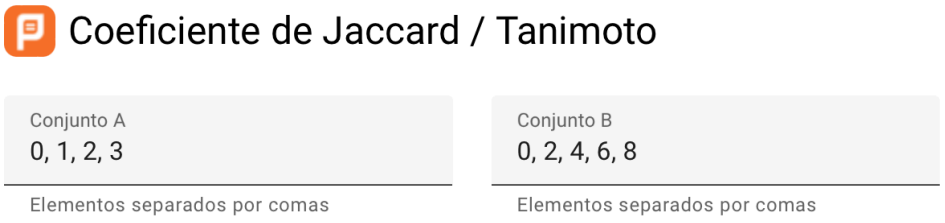
Para no tener que acceder a datos reales, que el proceso 0 inicialice ambos conjuntos con valores enteros (*int*) siguiendo el siguiente esquema:

A = {0, 1, 2, 3…, (|A|-1)} y B = {0, 2, 4, 6, …, 2*(|B|-1)}

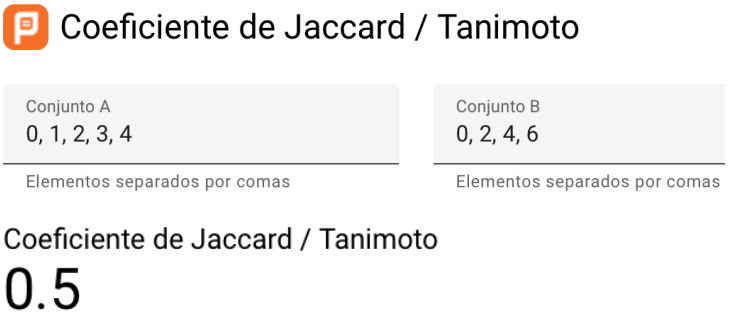
No obstante, asume que se trata de conjuntos de elementos sin ningún tipo de patrón numérico entre ellos.

Como ejemplo de comprobación, se presentan los siguientes casos:

1) |A| =4 y |B| = 5



2) |A| =5 y |B| = 4



Una vez implementado y comprobado el cálculo, fija una instancia del problema que tarde, en secuencial, al menos 20 segundos o más (en ATCGrid). Haz entonces una gráfica de aceleración o *Speedup* de tu versión secuencial con respecto a la MPI. Cambia el número de unidades de ejecución como estimes oportuno pero con, al menos, 3 casos con más de un proceso. Para los cálculos utiliza tiempos promediados tras 5 ejecuciones (a incluir en una tabla antes de la gráfica en sí). Incluye además, en la gráfica de aceleración, el caso ideal. Finalmente, comenta los resultados. Si la hubiera, ¿a qué atribuyes la diferencia entre la aceleración idea y la obtenida?