USC UNIVERSIDADE DE SANTIAGO DE COMPOSTELA
ESCOLA TÉCNICA SUPERIOR DE ENXEÑARÍA

COMPUTACIÓN CONCURRENTE, PARALELA Y DISTRIBUIDA
Grado en Inteligencia Artificial
Práctica 1: Taller de análisis de rendimiento computacional

Objetivo

Profundizar en las distintas métricas y funcionalidades disponibles para analizar el rendimiento computacional de un programa.

Metodología

- Trabajo en grupo durante las sesiones interactivas.

Entrega

- Defensa individual de cada ejercicio por parte de un miembro del grupo durante las sesiones interactivas correspondientes. [Hasta 10 puntos]
- Entrega en el campus virtual en la tarea habilitada antes de la fecha límite. [Hasta 6 puntos]

Enunciado

1. **[1 punto]** El 95% del tiempo de ejecución de un programa está dedicado a un bucle que queremos paralelizar. ¿Cuál es el speedup máximo que podemos esperar de una versión paralela de este programa que se ejecute en una máquina con 6 procesadores?

2. **[1.5 puntos]** Imagina que tienes una aplicación que realiza un conjunto de tareas en secuencia. La aplicación consta de tres etapas: A, B y C. La etapa A representa una carga de trabajo intensiva en CPU, mientras que las etapas B y C son menos intensivas en CPU. Deseas mejorar el rendimiento de la aplicación mediante la paralelización de la etapa A. Actualmente, el tiempo total que tarda la aplicación en ejecutarse es de 100 segundos y la etapa A representa el 60% del tiempo total de ejecución.

   (a) Utiliza la Ley de Amdahl para calcular el speedup máximo que se puede lograr si se paraleliza la etapa A de la aplicación con 4 cores.

   (b) Determina el tiempo de ejecución mejorado de la aplicación después de la paralelización de la etapa A.

3. **[1.5 puntos]** Cuando adaptamos un programa para su ejecución en un sistema multinúcleo, el tiempo de cada procesador se divide entre tiempo de cómputo y tiempo del sobre coste requerido para secciones críticas no paralelizables o movimiento de datos entre procesadores.

   Dado un programa que requiere 100 segundos para ejecutarse en un único procesador, asume que su tiempo de ejecución con p procesadores será el resultado de dividir el tiempo secuencial entre el número de procesadores más 4 segundos adicionales de sobre coste (independientemente del número de procesadores).

   (a) Completa la siguiente tabla, siendo Speedup el obtenido respecto a usar un único procesador.

Nº de procesadores	Tiempo de ejecución	Speedup	Eficiencia
1	100	–	–
2			
4			
8			
16			

(b) ¿Cómo afecta el nº de procesadores a la eficiencia del programa?

4. **[1.5 puntos]** Disponemos de un servidor de computación que puede ser configurado de dos maneras distintas. Se ha ejecutado un benchmark para evaluar el rendimiento de cada configuración, obteniendo los siguientes resultados. En este benchmark, las puntuaciones equivalen a tiempos de ejecución. Por lo tanto las puntuaciones más bajas son preferibles sobre las más altas.

Prueba	Énfasis	Configuración 1	Configuración 2
A	Uso del procesador	100	130
B	Uso de memoria	50	40
C	Uso de E/S	22	12

(a) Calcular el speedup de la configuración 2 con respecto a la 1 para cada una de las diferentes pruebas.

(b) Sabiendo que el tiempo de ejecución de las aplicaciones que se ejecutarán en este sistema se reparte de la siguiente manera:

- 90% uso intensivo del procesador,
- 8% uso intensivo de la memoria,
- 2% uso del sistema de entrada/salida,

¿Qué configuración es preferible para este sistema en este caso de uso concreto?

5. **[2 puntos]** Completa el siguiente fragmento de código con las instrucciones necesarias para medir:

- el tiempo de cómputo de cada tarea paralela.
- el tiempo de cómputo total de todas las tareas paralelas.
- el tiempo transcurrido para el cómputo en paralelo.
- el tiempo del cómputo para ejecutar las tareas secuencialmente.

Añade el código necesario para calcular el speedup entre la ejecución en paralelo y la secuencial y muéstralo por pantalla junto con la eficiencia de la versión paralelizada respecto a la secuencial.

```python
import multiprocessing
import time

def tarea(num, tiempos):
    # Simulación de una tarea que toma 'num' segundos
    print(f"Proceso {num} iniciado")
    time.sleep(num)
    print(f"Proceso {num} terminado. Tiempo transcurrido: {tiempos[num]} segundos")

if __name__ == "__main__":
    # Número de procesos a ejecutar
    num_procesos = 4

    # Crear una lista de procesos
    procesos = []

    # Diccionario compartido para almacenar los tiempos de cada proceso
    tiempos = multiprocessing.Manager().dict()

    for i in range(num_procesos):
        proceso = multiprocessing.Process(target=tarea, args=(i+1, tiempos))
        procesos.append(proceso)
        proceso.start()

    # Esperar a que todos los procesos terminen
    for proceso in procesos:
        proceso.join()


    # Repetir el cálculo con un solo proceso
    for i in range(num_procesos):
        tarea(i+1, tiempos)
```

```
33
34      print(f"Tiempo total transcurrido: {total_time} segundos")
35      print(f"Tiempo total de cómputo de todos los procesos: {total_computing_time} segundos")
36      print(f"Tiempo total de cómputo de un solo proceso: {total_computing_time_single_process} segundos")
37      print(f"Speedup: {speedup:.2f}x")
38      print(f"Eficiencia: {eficiencia:.2f}")
```

6. **[2 puntos]** Utiliza la librería *memory_profiler* para realizar un estudio del uso de memoria del siguiente código. El estudio debe cubrir, como mínimo, los siguientes apartados:

   - Añade al código los decoradores necesarios para hacer el profiling de las funciones *types*, *listas* y *numpy* con 6 dígitos de precisión.
   - Guarda en un archivo de texto plano la salida del comando *mprof run* y comenta línea a línea el archivo obtenido explicando lo que ocurre en cada línea, desde el punto de vista del uso de memoria.
   - Utiliza *mprof* para generar un gráfico con la evolución del uso de memoria a lo largo de la ejecución del programa.
   - Utiliza *mprof* para conocer la cantidad de memoria disponible necesaria para poder ejecutar el programa.

```python
from memory_profiler import profile
import numpy as np

def types():
    a = np.random.rand(10**7).astype(np.float16)
    del a
    a = np.random.rand(10**7).astype(np.float32)
    del a
    a = np.random.rand(10**7).astype(np.float64)
    del a

def listas():
    a = [1] * (10 ** 7)
    b = [2] * (2 * 10 ** 7)
    c = a + b
    d = c
    del b
    a = None
    del c
    del d

def numpy():
    SIZE = int(1e4)
    a = np.random.rand(SIZE, SIZE)
    b = np.random.rand(SIZE, SIZE)
    b2 = b.copy()
    b3 = b
    c = a + b
    del b2
    del b3
    a = None
    d = np.random.rand(SIZE, SIZE)
    sum = np.sum(d)
    tr = d.T
    trcopy = d.T.copy()
    concat = np.concatenate((c, d), axis=0)
    split = np.split(concat, 2, axis=0)
    del c, d, tr, trcopy, concat, split

def main():
    types()
    listas()
    numpy()

if __name__ == '__main__':
    main()
```

7. **[OPCIONAL]** Una buena implementación afecta al rendimiento del código obtenido. Escribe un programa en Python que cree una lista con los cuadrados de los primeros N números naturales pares, siendo N un parámetro del programa. Implementa dos versiones alternativas:

   - Usando un bucle tradicional.
   - Usando comprensión de listas.

   Asegúrate de que las listas obtenidas mediante las dos implementaciones son iguales. Mide el tiempo que lleva hacer el cálculo mediante las dos alternativas para distintos valores de N y compáralas con las medidas de análisis de rendimiento computacional que consideres oportunas.

   ¿Podemos concluir que alguna de las implementaciones sea mejor que la otra?