

# Computacion paralela Trabajo 02

MACEDO PINTO LUIS MIGUEL

17 de Mayo de 2024

## 1 SUMA PARALELA

### 1.1 Modificar el programa para generar arrays aleatorios en (a) y (b)

```
1 #CODIGO ORIGINAL
2 import multiprocessing
3 def worker(tid, a, b, c):
4     c[tid] = a[tid] + b[tid]
5     print(f"c[{tid}]=c[{tid}]")
6
7
8 if __name__ == "__main__":
9     a = [1, 2, 3, 20, 5, 10]
10    b = [6, 7, 8, 9, 10, 20]
11    c = multiprocessing.Array('i', 6) # Shared array
12
13    processes = []
14    for tid in range(6):
15        process = multiprocessing.Process(target=worker, args=(tid, a, b, c))
16        processes.append(process)
17        process.start()
18
19    for process in processes:
20        process.join()
21
```

Figure 1: Instrucciones a modificar en la parte del array

Listing 1: Código modificado en la parte del array de manera aleatoria.

```
#generando numeros aleatorios
import multiprocessing
import numpy as np

def worker(tid, a, b, c):
    c[tid] = a[tid] + b[tid]
    print(f"c[{tid}]=c[{tid}]")

if __name__ == "__main__":
    # Generar arrays aleatorios usando NumPy
    a = np.random.randint(10, size=5)
    b = np.random.randint(10, size=5)
    c = multiprocessing.Array('i', 5) # Shared array

    processes = []
```

```

for tid in range(5):
    process = multiprocessing.Process(target=worker,
    args=(tid, a, b, c))
    processes.append(process)
    process.start()

for process in processes:
    process.join()

```

## DESCRIPCIÓN

Para general arrays aleatorios, use el módulo numpy. arriba está el código modificado para generar arrays aleatorios en lugar de utilizar los arrays predefinidos como se ve en la imagen.

1. Generamos dos arrays aleatorios a y b con números enteros aleatorios entre 0 y 9, cada uno de longitud 5, utilizando la función randint de NumPy. Luego, crea un array compartido c de tamaño 5 utilizando multiprocessing.Array.

```

# Generar arrays aleatorios usando NumPy
a = np.random.randint(10, size=5)
b = np.random.randint(10, size=5)
c = multiprocessing.Array('i', 5) # Shared array

```

Figure 2: Instrucciones a modificar en la parte del array

### 1.2 Realizar la modificación para el calculode una suma ordinaria y una suma paralela

### 1.3 SUMA ORDINARIA

```

1 #SUMA ORDINARIA O SECUENCIAL
2 import numpy as np
3
4 def worker(tid, a, b, c):
5     c[tid] = a[tid] + b[tid]
6     print(f'c[{tid}]-{c[tid]}')
7
8 if __name__ == "__main__":
9     # Generar arrays aleatorios usando NumPy
10    a = np.random.randint(10, size=5)
11    b = np.random.randint(10, size=5)
12    c = np.zeros(5, dtype=int) # Array para almacenar resultados
13
14    # Ejecutar la función worker secuencialmente
15    for tid in range(5):
16        worker(tid, a, b, c)
17
18    print("Resultado final:")
19    print(c)
20

```

Figure 3: Suma Ordinaria

Listing 2: Modificación para el cálculo a suma Paralela

```
#SUMA ORDINARIA O SECUENCIAL
import numpy as np

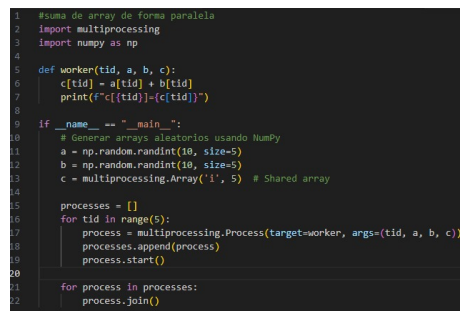
def worker(tid, a, b, c):
    c[tid] = a[tid] + b[tid]
    print(f"c[{tid}]=c[tid]")

if __name__ == "__main__":
    # Generar arrays aleatorios usando NumPy
    a = np.random.randint(10, size=5)
    b = np.random.randint(10, size=5)
    c = np.zeros(5, dtype=int) # Array para almacenar resultados

    # Ejecutar la función worker secuencialmente
    for tid in range(5):
        worker(tid, a, b, c)

    print("Resultado final:")
    print(c)
```

## 1.4 SUMA PARALELA



```
1 #suma de array de forma paralela
2 import multiprocessing
3 import numpy as np
4
5 def worker(tid, a, b, c):
6     c[tid] = a[tid] + b[tid]
7     print(f"c[{tid}]=c[tid]")
8
9 if __name__ == "__main__":
10     # Generar arrays aleatorios usando NumPy
11     a = np.random.randint(10, size=5)
12     b = np.random.randint(10, size=5)
13     c = multiprocessing.Array('i', 5) # Shared array
14
15     processes = []
16     for tid in range(5):
17         process = multiprocessing.Process(target=worker, args=(tid, a, b, c))
18         processes.append(process)
19         process.start()
20
21     for process in processes:
22         process.join()
```

Figure 4: Suma Paralela

Listing 3: Modificación para el cálculo a suma Ordinaria

```
#SUMA PARALELA
import multiprocessing
import numpy as np

def worker(tid, a, b, c):
    c[tid] = a[tid] + b[tid]
    print(f"c[{tid}]=c[tid]")
```

```

if __name__ == "__main__":
    # Generar arrays aleatorios usando NumPy
    a = np.random.randint(10, size=5)
    b = np.random.randint(10, size=5)
    c = multiprocessing.Array('i', 5) # Shared array

    processes = []
    for tid in range(5):
        process = multiprocessing.Process(target=worker, args=(tid, a, b, c))
        processes.append(process)
        process.start()

    for process in processes:
        process.join()

```

## DESCRIPCIÓN

1. Utilizamos el módulo multiprocessing para realizar una suma paralela de dos matrices. Aquí tienes una descripción línea por línea:

2. `import multiprocessing`: Importa el módulo multiprocessing, que proporciona soporte para la ejecución de procesos utilizando una interfaz similar a la de threading pero con capacidad de utilizar múltiples núcleos de CPU.

3. `import numpy as np`: Importa el módulo NumPy bajo el alias np. NumPy es una biblioteca popular en Python utilizada para realizar operaciones matemáticas en arreglos multidimensionales.

4. `def worker(tid, a, b, c)`:: Define una función llamada worker que toma cuatro argumentos: tid (identificador de hilo), a, b y c. Esta función se encarga de sumar los elementos correspondientes de los arreglos a y b y guardar el resultado en c[tid].

### 1.5 Evidenciar la optimización de tiempo entre ambos algoritmos.

### 1.6 OPTIMIZACION DEL TIEMPO DE MODO ORDINARIO

Listing 4: Optimizacion de tiempo modo ordinario

```

#OPTIMIZACION DE TIEMPO ALGORITMO ORDINARIO O SECUENCIAL
import numpy as np
import time

```

```

def worker(tid, a, b, c):
    c[tid] = a[tid] + b[tid]
    print(f"c[{tid}]=c[tid]")
if __name__ == "__main__":
    # Generar arrays aleatorios usando NumPy
    a = np.random.randint(10, size=5)
    b = np.random.randint(10, size=5)
    c = np.zeros(5, dtype=int) # Array para almacenar resultados

    start_time = time.time()

    # Ejecutar la funci n worker secuencialmente
    for tid in range(5):
        worker(tid, a, b, c)

    end_time = time.time()

    print("Resultado final:")
    print(c)
    print("Tiempo de ejecuci n:", end_time - start_time, "segundos")

```

## 1.7 OPTIMIZACION DE SUMA PARALELA

Figure 5: Suma Paralela

Listing 5: Optimizacion de tiempo de suma paralela

```

#OPTIMIZACION DE TIEMPO ALGORITMO PARALELO
import multiprocessing
import numpy as np
import time

def worker(tid, a, b, c):
    c[tid] = a[tid] + b[tid]

if __name__ == "__main__":
    # Generar arrays aleatorios usando NumPy
    a = np.random.randint(10, size=5)
    b = np.random.randint(10, size=5)
    c = multiprocessing.Array('i', 5) # Shared array

    start_time = time.time()

```

```

processes = []
for tid in range(5):
    process = multiprocessing.Process(target=worker, args=(tid, a, b, c))
    processes.append(process)
    process.start()

for process in processes:
    process.join()

end_time = time.time()

print("Resultado final:")
print(c[:])
print("Tiempo de ejecuci n:", end_time - start_time, "segundos")

```

## DESCRIPCIÓN

1. Para optimizar este código y aprovechar al máximo el paralelismo, podrías considerar el uso de `multiprocessing.Pool`, que administra automáticamente el número de procesos disponibles y distribuye las tareas entre ellos de manera eficiente. Aquí te muestro cómo hacerlo

2. Para optimizar este código, puedes utilizar la biblioteca `NumPy` para realizar operaciones vectorizadas en lugar de iterar sobre los elementos de los arrays. Esto aprovecha la capacidad de `NumPy` para realizar cálculos de manera eficiente en matrices completas en lugar de elementos individuales. Aquí te muestro cómo hacerlo.

link: <https://github.com/LUISMIGUELMACEDOPINTO/PROG.PARALELA>