Universidad Europea

Memoria realización ejercicio 3 Programación Concurrente y Distribuida

Arturo Alba Sánchez-Mayoral

Nº expediente: 21838473

Repo GitHub: https://github.com/ArtySaurio/Act3PCDExt

INVESTIGACION PREVIA

Antes de empezar a codificar tengo que buscar una librería que me permita paralelizar el trabajo entre los cores de mi procesador. De esta manera encuentro la librería de multiprocessing de Python. Gracias a esta librería pude realizar el código de una manera bastante intuitiva.

RESOLUCION DE MERGE SORT

Para realizar la parte del merge sort deberemos crear 3 funciones. La primera, llamada “merge”, la segunda llamada “mergesort” y la tercera llamada “mergesortparl”.

```
15  def merge(*args):
16      left, right = args[0] if len(args) == 1 else args
17      left_length, right_length = len(left), len(right)
18      left_index, right_index = 0, 0
19      merged = []
20
21      while left_index < left_length and right_index < right_length:
22          if left[left_index] <= right[right_index]:
23              merged.append(left[left_index])
24              left_index += 1
25          else:
26              merged.append(right[right_index])
27              right_index += 1
28      if left_index == left_length:
29          merged.extend(right[right_index:])
30      else:
31          merged.extend(left[left_index:])
32      return merged
```

En la primera función, como su propio nombre indica, será la manera tradicional de realizar el merge. Necesitaremos esta función para poder utilizar la siguiente “mergesort”, que hará uso de la primera.

```
35  def mergesort(data):
36      length = len(data)
37
38      if length <= 1:
39          return data
40
41      middle = length // 2
42      left = mergesort(data[:middle])
43      right = mergesort(data[middle:])
44      return merge(left, right)
```

En la segunda función utilizaremos el merge codificado anteriormente y lo ordenaremos según el algoritmo mergesort.

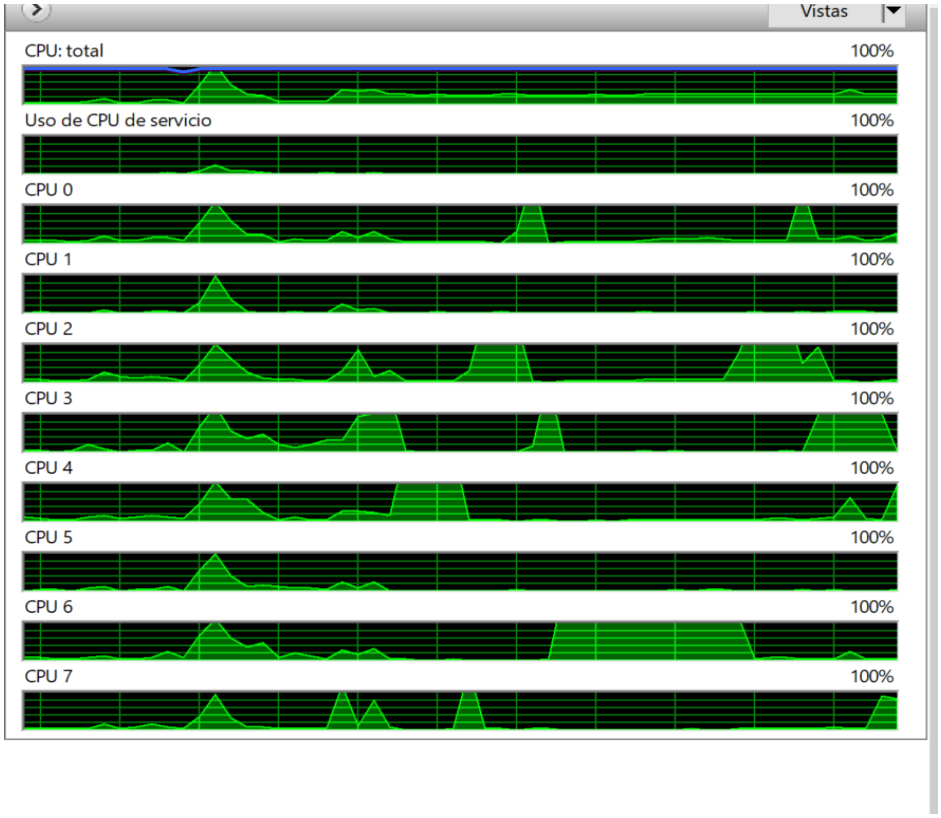
```python
def mergesortparl(data):

    #Haciendo uso del metodo pool, almacenamos los procesos en ejecucion. ividi
    processes = multiprocessing.cpu_count()
    pool = multiprocessing.Pool(processes=processes)
    tam = int(math.ceil(float(len(data)) / processes))
    data = [data[i * tam:(i + 1) * tam] for i in range(processes)]
    data = pool.map(mergesort, data)

    #Ordenamos todas las particiones en una sola, obteniendo asi el arrayfinal
    while len(data) > 1:
        #Cuando las aprticiones son impares realizamos "pop" de la ultima y la
        extra = data.pop() if len(data) % 2 == 1 else None
        data = [(data[i], data[i + 1]) for i in range(0, len(data), 2)]
        data = pool.map(merge, data) + ([extra] if extra else [])
    return data[0]
```

En la tercera función crearemos una “pool” de procesos. Habrá uno por cada core del procesador, una vez hecho eso, dividiremos el array creado en partes de igual tamaño por cada proceso. Usando la función map hacemos un mapeo completo de la pool de procesos, indicándole además el algoritmo que usamos y el array de datos que gestionamos. De esta manera al acabar de fragmentar nuestro array en partes y gestionar la pool entre los procesos, se debe aplicar el mergesort a las distintas partes para obtener el array finalmente ordenado.

RESULTADO DE PRIMERA PARTE



Como podemos ver en la imagen del resultado mi maquina consta de 8 cores.

Dichos cores de mi ordenador se gestionan haciendo uso de la antes nombrada pool de la librería multiprocessing para aligerar la carga y realizar la paralelización correcta del algoritmo.

RESOLUCION DE FIBONACCI

De la misma manera que realicé el merge, para realizar la paralelizaicon del fibonacci deberemos crear 3 funciones. La primera, llamada “fibo”, la segunda llamada “coredistrib” y la tercera llamada “fiboparl”.

```
67  def fibo(n):
68      #Inicializamos los dos primeros valores a 0 y 1 para iniciar la secuencia
69      a = 0
70      b = 1
71
72      if n < 0:
73          print("No has introducido un valor válido...")
74      elif n == 0:
75          return a
76      elif n == 1:
77          return b
78      else:
79          for i in range(1, n):
80              c = a + b
81              a = b
82              b = c
83          return b
```

En la función fibo se codifica el algoritmo original de Fibonacci, indicando y partiendo de que los primeros dos números de la secuencia son 0 y 1, en nuestro caso a y b.

```
114    def coredistrib(indice, tarea, arrayfinal):
115        Fib = fibo(tarea)
116        arrayfinal[indice] = Fib
```

En la segunda función, utilizaremos la primera creada anteriormente y distribuiremos las distintas tareas o también llamadas semisumas para realizar nuestro enfoque del algoritmo correctamente.

```
86  def fiboparl(n):
87
88      #Miramos los cores de nuestra maquina
89      n_cores = multiprocessing.cpu_count()
90      print('Numero de cores del pc: ', n_cores)
91
92      #Ya que fibonacci es secuencial divido la carga de trabajo haciendo semi-sumas de las partes que asigno
93      #Decomponemos las secuencias en los distintos cores del ordenador
94      distribuciont = [n - 3, n - 4, n - 4, n - 5, n - 4, n - 5, n - 5, n - 6]
95      arrayfinal = multiprocessing.RawArray('d', n_cores)
96      cores = []
97
98      for core in range(n_cores):
99          cores.append(
100             multiprocessing.Process(target=coredistrib, args=(core, distribuciont[core], arrayfinal)))
101     for core in cores:
102         #Haciendo uso del metodo star inicio la carga de los trabajos para cada uno de los cores de mi cpu
103         core.start()
104     for core in cores:
105         #Mediante el uso del metodo join se bloquean los hilos hasta que terminan su tarea todos los cores
106         core.join()
107     #Inicializo la secuencia a 0 para poder iniciar el algoritmo
108     fibo = 0
109     for sum in arrayfinal:
110         fibo += sum
111     return fibo
```

En la tercera función tenemos finalmente, la función paralela de Fibonacci. Como Fibonacci es secuencial, habrá que dividir la carga de trabajo mediante unas semi sumas de las partes asignadas, ya que si no, seria bastante complicada la paralelización puesto que como venimos recordando desde el comienzo del ejercicio Fibonacci es un algoritmo secuencial. Después descomponemos las secuencias en los distintos cores del ordenador y finalmente obtenemos el resultado del array final.