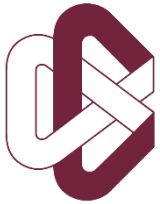
CONAHCYT

CONSEJO NACIONAL DE HUMANIDADES
CIENCIAS Y TECNOLOGÍAS



CIMAT

MULTIPROCESSING EN PYTHON

Dr. Francisco Javier Hernández López

CONAHCYT, CIMAT-Mérida

fcoj23@cimat.mx

- Lenguaje de programación interpretado.
- El código se ejecuta directamente, hay un interprete (CPython) que lee la instrucción en tiempo real y la ejecuta.
- A diferencia de C/C++, el código no necesita ser compilado.
- Sintaxis clara y legible.
- Librería estándar extensa, donde a través de módulos de software adicionales podemos agregar tipos de datos, funciones y objetos.
- Riqueza en documentación y comunidades de software.

https://www.python.org/downloads/
Zaccone, G. (2015). Python parallel programming cookbook. Packt Publishing Ltd.

CÓMPUTO PARALELO USANDO PYTHON

- Python es rápido, sin embargo, si se requiere de mayor velocidad, algunas opciones son:
  - Tener módulos escritos en C/C++ que puedan ser importados en Python.
  - Utilizar un compilador *Just-In-Time* (JIT) o *Ahead-Of-Time* (AOT) (Cython).
  - Usar los módulos de cómputo paralelo de Python.

Zaccone, G. (2015). Python parallel programming cookbook. Packt Publishing Ltd.

HILOS EN PYTHON

- Paralelismo basado en hilos es el camino estándar para escribir programas paralelos.
- El interprete de Python no es seguro para manejo de hilos.
- Para soportar programas Python multi-hilos se utiliza un bloqueo global llamado *Global Interpreter Lock* (GIL):
  - Solo un hilo puede ejecutar código Python a la vez.
  - Python cambia automáticamente al siguiente hilo después de un corto período de tiempo o cuando un hilo hace algo que puede llevar un tiempo más largo.
  - Si varios hilos intentan acceder al mismo objeto de datos, este puede terminar en un estado inconsistente.

Zaccone, G. (2015). Python parallel programming cookbook. Packt Publishing Ltd.

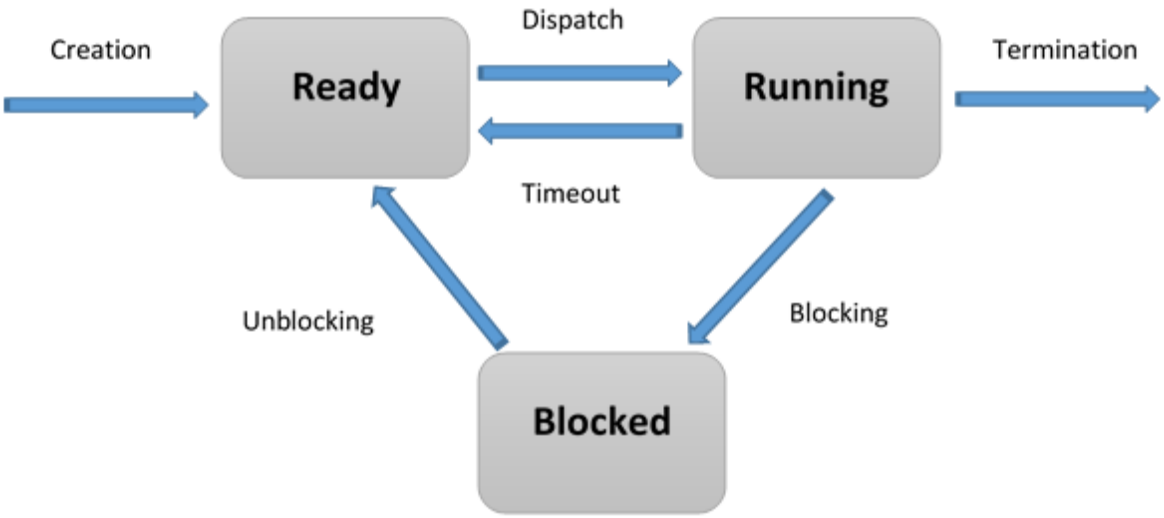
DIFERENCIAS ENTRE HILOS Y PROCESOS

Hilos	Procesos
Comparten memoria.	No comparten memoria.
Inicio/Cambio son menos costosos computacionalmente.	Inicio/Cambio son costosos computacionalmente.
Requiere pocos recursos (Proceso ligero).	Requiere más recursos computacionales.
Necesitan mecanismos de sincronización para el manejo correcto de los datos.	No requieren sincronización de la memoria.

Zaccone, G. (2019). Python Parallel Programming Cookbook: Over 70 recipes to solve challenges in multithreading and distributed system with Python 3. Packt Publishing Ltd.

PARALELISMO BASADO EN HILOS

- Generalmente, una aplicación se realiza mediante un único proceso, el cual se divide en múltiples hilos independientes, los cuales representan actividades de diferentes tipos que se ejecutan en paralelo y compiten entre sí.



Ciclo de vida de los hilos. Tomado de [Zaccone, 2019]

Zaccone, G. (2019). Python Parallel Programming Cookbook: Over 70 recipes to solve challenges in multithreading and distributed system with Python 3. Packt Publishing Ltd.

EJEMPLO: DEFINIENDO UN HILO

```python
import threading

def my_func(thread_number):
    return print('my_func called by thread N°{}'.format(thread_number))

def main():
    threads = []
    for i in range(10):
        t = threading.Thread(target=my_func, args=(i,))
        threads.append(t)
        t.start()
        t.join()

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

```
my_func called by thread N°0
my_func called by thread N°1
my_func called by thread N°2
my_func called by thread N°3
my_func called by thread N°4
my_func called by thread N°5
my_func called by thread N°6
my_func called by thread N°7
my_func called by thread N°8
my_func called by thread N°9
```

EJEMPLO: DETERMINAR QUE HILO ESTÁ EN EJECUCIÓN (PARTE 1)

```python
import threading
import time

def function_A():
    print (threading.currentThread().getName()+str('--> starting \n'))
    time.sleep(2)
    print (threading.currentThread().getName()+str( '--> exiting \n'))
    return

def function_B():
    print (threading.currentThread().getName()+str('--> starting \n'))
    time.sleep(2)
    print (threading.currentThread().getName()+str( '--> exiting \n'))
    return

def function_C():
    print (threading.currentThread().getName()+str('--> starting \n'))
    time.sleep(2)
    print (threading.currentThread().getName()+str( '--> exiting \n'))
    return
```

EJEMPLO: DETERMINAR QUE HILO ESTÁ EN EJECUCIÓN (PARTE 2)

```
23  if __name__ == "__main__":
24
25      t1 = threading.Thread(name='function_A', target=function_A)
26      t2 = threading.Thread(name='function_B', target=function_B)
27      t3 = threading.Thread(name='function_C',target=function_C)
28
29      t1.start()
30      t2.start()
31      t3.start()
32
33      t1.join()
34      t2.join()
35      t3.join()
```

```
function_A--> starting
function_B--> starting
function_C--> starting
function_B--> exiting
function_C--> exiting
function_A--> exiting
```

SINCRONIZACIÓN DE HILOS

- Esto se lleva a cabo mediante bloqueos con el método *Lock()* del módulo *threading*.
- *Lock()* es un objeto que típicamente es accesible por múltiples hilos. Es lo que un hilo debe tener para poder proceder a la ejecución de una sección protegida de un programa.
- Ya que se ha creado el bloqueo, hay dos métodos que permiten la sincronización de dos o más hilos:
  - acquire() → Recibe un parámetro (opcional). Cuando no se especifica o es *True*, fuerza al hilo a suspender su ejecución hasta que se libere el bloqueo y luego pueda adquirirse. Cuando es *False* entonces regresa un *Boolean,* el cual es *True* si el bloqueo se ha adquirido o *False* en otro caso.
  - release() → Libera el bloqueo.

Zaccone, G. (2019). Python Parallel Programming Cookbook: Over 70 recipes to solve challenges in multithreading and distributed system with Python 3. Packt Publishing Ltd.

EJEMPLO: SINCRONIZACIÓN CON LOCK() P1

```python
import threading
import time
import os
from threading import Thread
from random import randint

# Lock Definition
threadLock = threading.Lock()

class MyThreadClass (Thread):
    def __init__(self, name, duration):
        Thread.__init__(self)
        self.name = name
        self.duration = duration
    def run(self):
        #Acquire the Lock
        threadLock.acquire()
        print ("---> " + self.name + \
                " running, belonging to process ID "\
                + str(os.getpid()) + "\n")
        time.sleep(self.duration)
        print ("---> " + self.name + " over\n")
        #Release the Lock
        threadLock.release()
```

Tenemos una clase *MyThreadClass* que hereda atributos de la clase *Thread*.
Clase hija: *MyThreadClass*
Clase padre: *Thread*

Se adquiere el bloqueo y se realiza la tarea, mientras que el resto de los hilos permanecen en espera.

Cuando se encuentra el *release()*, el siguiente hilo obtiene el bloqueo y el resto espera hasta el final de la ejecución.

EJEMPLO: SINCRONIZACIÓN CON LOCK() P2

```
27  def main():
28      start_time = time.time()
29      # Thread Creation
30      thread1 = MyThreadClass("Thread#1 ", randint(1,10))
31      thread2 = MyThreadClass("Thread#2 ", randint(1,10))
32      thread3 = MyThreadClass("Thread#3 ", randint(1,10))
33      thread4 = MyThreadClass("Thread#4 ", randint(1,10))
34      thread5 = MyThreadClass("Thread#5 ", randint(1,10))
35      thread6 = MyThreadClass("Thread#6 ", randint(1,10))
36      thread7 = MyThreadClass("Thread#7 ", randint(1,10))
37      thread8 = MyThreadClass("Thread#8 ", randint(1,10))
38      thread9 = MyThreadClass("Thread#9 ", randint(1,10))
40      # Thread Running
41      thread1.start()
42      thread2.start()
43      thread3.start()
44      thread4.start()
45      thread5.start()
46      thread6.start()
47      thread7.start()
48      thread8.start()
49      thread9.start()
51      # Thread joining
52      thread1.join()
53      thread2.join()
54      thread3.join()
55      thread4.join()
56      thread5.join()
57      thread6.join()
58      thread7.join()
59      thread8.join()
60      thread9.join()
```

Se crean 9 hilos, cada uno con su propio nombre y duración.

EJEMPLO: SINCRONIZACIÓN CON LOCK() P3

```
62      # End
63      print("End")
64
65      #Execution Time
66      print("--- %s seconds ---" % (time.time() - start_time))
67
68
69  if __name__ == "__main__":
70      main()
```

```
---> Thread#1  running, belonging to process ID 25352
---> Thread#1  over
---> Thread#2  running, belonging to process ID 25352
---> Thread#2  over
---> Thread#3  running, belonging to process ID 25352
---> Thread#3  over
---> Thread#4  running, belonging to process ID 25352
---> Thread#4  over
---> Thread#5  running, belonging to process ID 25352
---> Thread#5  over
---> Thread#6  running, belonging to process ID 25352
---> Thread#6  over
---> Thread#7  running, belonging to process ID 25352
---> Thread#7  over
---> Thread#8  running, belonging to process ID 25352
---> Thread#8  over
---> Thread#9  running, belonging to process ID 25352
---> Thread#9  over
End
--- 32.06511926651001 seconds ---
```

Tanto la creación del hilo como su ejecución suceden de forma secuencial.

EJEMPLO: SINCRONIZACIÓN CON LOCK() V2 P1

```
7   # Lock Definition
8   threadLock = threading.Lock()
9
10  class MyThreadClass (Thread):
11      def __init__(self, name, duration):
12          Thread.__init__(self)
13          self.name = name
14          self.duration = duration
15      def run(self):
16          #Acquire the Lock
17          threadLock.acquire()
18          print ("---> " + self.name + \
19                  " running, belonging to process ID "\
20                  + str(os.getpid()) + "\n")
21          threadLock.release()
22          time.sleep(self.duration)
23          print ("---> " + self.name + " over\n")
24          #Release the Lock
```

Movemos el método *release()* antes del *sleep()*

EJEMPLO: SINCRONIZACIÓN CON LOCK() V2 P2

```
---> Thread#1  running, belonging to process ID 8428
---> Thread#2  running, belonging to process ID 8428
---> Thread#3  running, belonging to process ID 8428
---> Thread#4  running, belonging to process ID 8428
---> Thread#5  running, belonging to process ID 8428
---> Thread#6  running, belonging to process ID 8428
---> Thread#7  running, belonging to process ID 8428
---> Thread#8  running, belonging to process ID 8428
---> Thread#9  running, belonging to process ID 8428
---> Thread#1  over
---> Thread#3  over
---> Thread#2  over
---> Thread#5  over
---> Thread#6  over
---> Thread#8  over
---> Thread#9  over
---> Thread#7  over
---> Thread#4  over
End
--- 10.010955095291138 seconds ---
```

Solo la creación del hilo sucede de forma secuencial

Luego de la creación del hilo, el nuevo hilo adquiere el bloqueo, mientras que el anterior continua con la ejecución de su tarea.

PARALELISMO BASADO EN PROCESOS

- El módulo multiprocessing de Python implementa el paradigma de programación de memoria compartida.

```
2  import multiprocessing
3
4  def myFunc(i):
5      print ('calling myFunc from process n°: %s' %i)
6      for j in range (0,i):
7          print('output from myFunc is :%s' %j)
8      return
9
10 if __name__ == '__main__':
11     for i in range(6):
12         process = multiprocessing.Process\
13                   (target=myFunc, args=(i,))
14         process.start()
15         process.join()
```

```
calling myFunc from process n°: 0
calling myFunc from process n°: 1
output from myFunc is :0
calling myFunc from process n°: 2
output from myFunc is :0
output from myFunc is :1
calling myFunc from process n°: 3
output from myFunc is :0
output from myFunc is :1
output from myFunc is :2
calling myFunc from process n°: 4
output from myFunc is :0
output from myFunc is :1
output from myFunc is :2
output from myFunc is :3
calling myFunc from process n°: 5
output from myFunc is :0
output from myFunc is :1
output from myFunc is :2
output from myFunc is :3
output from myFunc is :4
```

Zaccone, G. (2019). Python Parallel Programming Cookbook: Over 70 recipes to solve challenges in multithreading and distributed system with Python 3. Packt Publishing Ltd.

EJECUTANDO PROCESOS EN EL SEGUNDO PLANO *(BACKGROUND)* P1

- El módulo *multiprocessing* permite la ejecución de procesos en el *background* usando la bandera "*daemon*" (*True or False*).

```
1  import multiprocessing
2  import time
3
4  def foo():
5      name = multiprocessing.current_process().name
6      print ("Starting %s \n" %name)
7      if name == 'background_process':
8          for i in range(0,5):
9              print('---> %d \n' %i)
10         time.sleep(1)
11     else:
12         for i in range(5,10):
13             print('---> %d \n' %i)
14         time.sleep(1)
15     print ("Exiting %s \n" %name)
```

Zaccone, G. (2019). Python Parallel Programming Cookbook: Over 70 recipes to solve challenges in multithreading and distributed system with Python 3. Packt Publishing Ltd.

EJECUTANDO PROCESOS EN EL SEGUNDO PLANO *(BACKGROUND)* P2

```
18  if __name__ == '__main__':
19      background_process = multiprocessing.Process\
20                           (name='background_process',\
21                            target=foo)
22      background_process.daemon = True
23
24      NO_background_process = multiprocessing.Process\
25                              (name='NO_background_process',\
26                               target=foo)
27
28      NO_background_process.daemon = False
29
30      background_process.start()
31      NO_background_process.start()
32
33      background_process.join()
34      NO_background_process.join()
35      #time.sleep(1)
```

```
Starting background_process
---> 0
---> 1
---> 2
---> 3
---> 4
Starting NO_background_process
---> 5
---> 6
---> 7
---> 8
---> 9
Exiting background_process
Exiting NO_background_process
```

EJEMPLO: PRODUCTOR-CONSUMIDOR P1

- Intercambio de datos usando una COLA

```python
import multiprocessing
import random
import time

class producer(multiprocessing.Process):
    def __init__(self, queue):
        multiprocessing.Process.__init__(self)
        self.queue = queue

    def run(self) :
        for i in range(10):
            item = random.randint(0, 256)
            self.queue.put(item)
            print ("Process Producer : item %d appended to queue %s"\
                   % (item,self.name))
            time.sleep(1)
            print ("The size of queue is %s"\
                   % self.queue.qsize())
```

Zaccone, G. (2019). Python Parallel Programming Cookbook: Over 70 recipes to solve challenges in multithreading and distributed system with Python 3. Packt Publishing Ltd.

EJEMPLO: PRODUCTOR-CONSUMIDOR P2

```
20  class consumer(multiprocessing.Process):
21      def __init__(self, queue):
22          multiprocessing.Process.__init__(self)
23          self.queue = queue
24
25      def run(self):
26          while True:
27              time.sleep(1)
28              if (self.queue.empty()):
29                  print("the queue is empty")
30                  break
31              else :
32                  time.sleep(2)
33                  item = self.queue.get()
34                  print ('Process Consumer : item %d popped from by %s '\
35                         % (item, self.name))
36                  #time.sleep(1)
```

EJEMPLO: PRODUCTOR-CONSUMIDOR P3

```
39  if __name__ == '__main__':
40          queue = multiprocessing.Queue()
41          process_producer = producer(queue)
42          process_consumer = consumer(queue)
43          process_producer.start()
44          process_consumer.start()
45          process_producer.join()
46          process_consumer.join()
```

```
Process Producer : item 237 appended to queue producer-1
The size of queue is 1
Process Producer : item 202 appended to queue producer-1
The size of queue is 2
Process Producer : item 16 appended to queue producer-1
The size of queue is 3
Process Consumer : item 237 popped from by consumer-2
Process Producer : item 29 appended to queue producer-1
The size of queue is 3
Process Producer : item 26 appended to queue producer-1
The size of queue is 4
Process Producer : item 78 appended to queue producer-1
The size of queue is 5
Process Consumer : item 202 popped from by consumer-2
Process Producer : item 111 appended to queue producer-1
The size of queue is 5
Process Producer : item 186 appended to queue producer-1
The size of queue is 6
Process Producer : item 226 appended to queue producer-1
The size of queue is 7
Process Consumer : item 16 popped from by consumer-2
Process Producer : item 137 appended to queue producer-1
The size of queue is 7
Process Consumer : item 29 popped from by consumer-2
Process Consumer : item 26 popped from by consumer-2
Process Consumer : item 78 popped from by consumer-2
Process Consumer : item 111 popped from by consumer-2
Process Consumer : item 186 popped from by consumer-2
Process Consumer : item 226 popped from by consumer-2
Process Consumer : item 137 popped from by consumer-2
the queue is empty
```

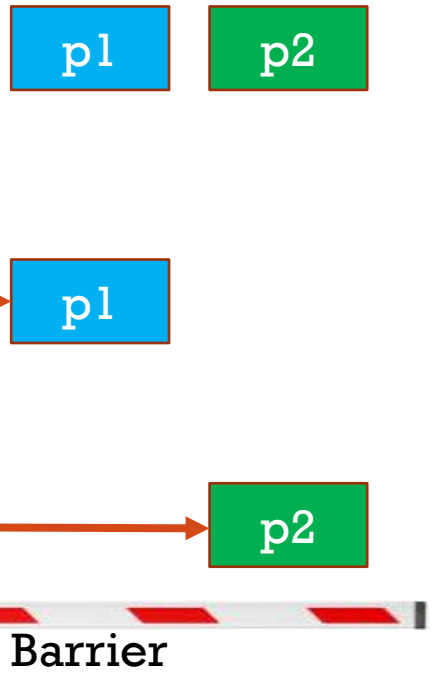
EJEMPLO: SINCRONIZACIÓN DE PROCESOS P1

```python
import multiprocessing
from multiprocessing import Barrier, Lock, Process
import time
from datetime import datetime

def test_with_barrier(synchronizer, serializer):
    name = multiprocessing.current_process().name
    if(name=='p1 - test_with_barrier'):
        now_p1 = time.time()
        print("process %s arrives at ----> %s" \
            %(name,datetime.fromtimestamp(now_p1)))
        time.sleep(2)
    else:
        now_p2 = time.time()
        print("process %s arrives at ----> %s" \
            %(name,datetime.fromtimestamp(now_p2)))
        time.sleep(1)

    synchronizer.wait()

    now = time.time()
    with serializer:
        print("process %s ----> %s" \
            %(name,datetime.fromtimestamp(now)))
```



P2 espera a que P1 alcance la barrera.

Zaccone, G. (2019). Python Parallel Programming Cookbook: Over 70 recipes to solve challenges in multithreading and distributed system with Python 3. Packt Publishing Ltd.

EJEMPLO: SINCRONIZACIÓN DE PROCESOS P2

```
27  def test_without_barrier():
28      name = multiprocessing.current_process().name
29      now = time.time()
30      print("process %s ----> %s" \
31            %(name ,datetime.fromtimestamp(now)))
```

```
33  if __name__ == '__main__':
34      synchronizer = Barrier(2)
35      serializer = Lock()
36      p1=Process(name='p1 - test_with_barrier'\
37              ,target=test_with_barrier,\
38              args=(synchronizer,serializer))
39      p2=Process(name='p2 - test_with_barrier'\
40              ,target=test_with_barrier,\
41              args=(synchronizer,serializer))
42      p3=Process(name='p3 - test_without_barrier'\
43              ,target=test_without_barrier)
44      p4=Process(name='p4 - test_without_barrier'\
45              ,target=test_without_barrier)
```

p3 p4

El número 2 indica la cantidad de procesos que debe esperar la barrera.

```
47      p2.start()
48      p1.start()
49      p3.start()
50      p4.start()
51
52      p1.join()
53      p2.join()
54      p3.join()
55      p4.join()
```

```
process p3 - test_without_barrier ----> 2024-01-30 15:11:03.234105
process p4 - test_without_barrier ----> 2024-01-30 15:11:03.237098
process p2 - test_with_barrier arrives at ----> 2024-01-30 15:11:03.246072
process p1 - test_with_barrier arrives at ----> 2024-01-30 15:11:03.247070
process p1 - test_with_barrier ----> 2024-01-30 15:11:05.249716
process p2 - test_with_barrier ----> 2024-01-30 15:11:05.249716
```

PARALELISMO USANDO UN GRUPO DE PROCESOS *(PROCESS POOL)*

```python
import multiprocessing

def f(x):
    return x+10

if __name__ == '__main__':
    pool = multiprocessing.Pool(processes=2)
    pool_outputs = pool.map(f,[1,2,3,4,5,6,7,8,9,10])

    pool.close()
    pool.join()
    print ('Pool    :', pool_outputs)
```

```
Pool    : [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

Zaccone, G. (2019). Python Parallel Programming Cookbook: Over 70 recipes to solve challenges in multithreading and distributed system with Python 3. Packt Publishing Ltd.

EJEMPLO: MULTIPLICAR ELEMENTO A ELEMENTO UNA LISTA DE NÚMEROS

```python
import multiprocessing

def function_square(data):
    result = data*data
    return result

if __name__ == '__main__':
    inputs = list(range(0,100))
    pool = multiprocessing.Pool(processes=4)
    pool_outputs = pool.map(function_square, inputs)

    pool.close()
    pool.join()
    print ('Pool    :', pool_outputs)
```

```
Pool    : [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400, 441, 484, 529, 57
6, 625, 676, 729, 784, 841, 900, 961, 1024, 1089, 1156, 1225, 1296, 1369, 1444, 1521, 1600, 1681, 1764, 1849, 1936, 202
5, 2116, 2209, 2304, 2401, 2500, 2601, 2704, 2809, 2916, 3025, 3136, 3249, 3364, 3481, 3600, 3721, 3844, 3969, 4096, 42
25, 4356, 4489, 4624, 4761, 4900, 5041, 5184, 5329, 5476, 5625, 5776, 5929, 6084, 6241, 6400, 6561, 6724, 6889, 7056, 7
225, 7396, 7569, 7744, 7921, 8100, 8281, 8464, 8649, 8836, 9025, 9216, 9409, 9604, 9801]
```

EJEMPLO: CALCULANDO π EN PARALELO P1

```
2  import multiprocessing as mp
3  import timeit
4
5  #parallel code
6  def calc_partial_pi(rank,nproc,nsteps,dx):
7      partial_pi=0.0
8      #print(range(rank,nsteps,nproc))
9      for i in range(rank,nsteps,nproc):
10         #print(i)
11         x=(i+0.5)*dx
12         partial_pi+=4.0/(1.0+x*x)
13     partial_pi*=dx
14     return partial_pi
```

```
16 def pi_parallel(nsteps,dx,nproc):
17     inputs=[(rank,nproc,nsteps,dx) for rank in range(nproc)]
18
19     pool=mp.Pool(processes=nproc)
20     result=pool.starmap(calc_partial_pi,inputs)
21     pi=sum(result)
22     return(pi)
```

EJEMPLO: CALCULANDO π EN PARALELO P2

```
24  if __name__=='__main__':
25      #serial code
26      t1=timeit.default_timer()
27      nsteps=10000000
28      dx=1.0/nsteps
29      pi=0.0
30      for i in range(nsteps):
31          x=(i+0.5)*dx
32          pi+=4.0/(1.0+x*x)
33      pi*=dx
34      t2=timeit.default_timer()
35      print('pi: ',pi)
36      print('time serial: ',t2-t1, 'seconds')
```

```
38      #parallel code
39      #nproc=mp.cpu_count()
40      nproc=4
41      print('number of CPU cores: ',nproc)
42      t1=timeit.default_timer()
43      pi=pi_parallel(nsteps,dx,nproc)
44      t2=timeit.default_timer()
45      print('pi: ',pi)
46      print('time parallel: ',t2-t1, 'seconds')
```

```
pi:  3.141592653589731
time serial:  2.3783033999999996 seconds
number of CPU cores:  4
pi:  3.141592653589686
time parallel:  0.5818465000000002 seconds
```

GRACIAS POR SU ATENCIÓN

Francisco J. Hernández-López

fcoj23@cimat.mx

WebPage:

www.cimat.mx/~fcoj23