#2303A51271

Batch:- 8

```
import numpy as np
import time
from numba import njit, prange

def serial_vector_op(A,B,alpha):
C=np.empty_like(A)   for i in
range(len(A)):
C[i]=alpha*A[i]+B[i]   return C

@njit(parallel=True) def
parallel_vector_op(A,B,alpha):
C=np.empty_like(A)   for i in
prange(len(A)):
C[i]=alpha*A[i]+B[i]   return C

def main():   sizes=[10_000, 100_000,
1_000_000, 2_000_000]   alpha=2.5

  print(f"{'Size':>10} | {'Serial(s)':>10} | {'Parallel(s)':>12} |
Speedup")   print("-"*55)

  for N in sizes:     A
=np.random.rand(N)
B= np.random.rand(N)

    # Run once to compile Numba functions
parallel_vector_op(A,B,alpha)

    #serial timing
start=time.time()
serial_vector_op(A,B,alpha)
t_serial=time.time()-start

    #parallel timing
start=time.time()
    parallel_vector_op(A,B,alpha)     t_parallel=time.time()-
start

    speedup=t_serial/t_parallel
    print(f"{N:10d} | {t_serial:10.4f} | {t_parallel:12.4f} |
{speedup:6.2f}x")
```

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

      Size |  Serial(s) |  Parallel(s) | Speedup
-----------------------------------------------------
     10000 |     0.0044 |       0.0001 |  81.47x
    100000 |     0.0463 |       0.0003 | 146.23x
   1000000 |     0.7449 |       0.0024 | 315.64x
2000000 |     1.7024 |       0.0047 | 366.00x

from itertools import starmap
import numpy as np
from numba import njit,prange
import time # Moved import time to the top


def serial_matmul(A,B):
n = A.shape[0]   C =
np.zeros((n,n))   for i
in range(n):     for j
in range(n):       for k
in range(n):
        C[i,j] += A[i,k] * B[k,j]
return C

@njit(parallel=True) # Corrected indentation
def parallel_outer_matmul(A,B):   n =
A.shape[0]   C = np.zeros((n,n))   for i in
range (n):     for j in range (n):       for
k in range(n):         C[i,j] +=A[i,k]
*B[k,j]   return C # Corrected indentation

@njit(parallel=True) # Corrected indentation
def parallel_inner_matmul(A,B):   n =
A.shape[0]   C = np.zeros((n,n))   for idx
in prange(n*n):     i = idx // n     j = idx
% n     for k in range(n):
      C[i,j] += A[i,k] * B [k,j]
return C # Corrected indentation

def main():
N = 300
A = np.random.rand(N,N)
B = np.random.rand(N,N)
```

```
     # Warm-up calls for Numba JIT
parallel_outer_matmul(A,B)
parallel_inner_matmul(A,B)

     #serial timing
start = time.time()
serial_matmul(A,B)
     t_serial = time.time() - start

     #parallel outer      start
= time.time()
parallel_outer_matmul(A,B)
     t_parallel_outer = time.time() - start

     #parallel inner      start
= time.time()
parallel_inner_matmul(A,B)
     t_parallel_inner = time.time() - start

     print(f"Serial time        : {t_serial:.3f} s")
print(f"Parallel outer loop: {t_parallel_outer:.3f} s")
print(f"Collaped loop      : {t_parallel_inner:.3f} s")
print(f"Speedup(outer)     : {t_serial/t_parallel_outer:.2f} x")
print(f"Speedup(inner)     : {t_serial/t_parallel_inner:.2f} x") if
__name__ == "__main__":   main()
Serial time        : 19.577 s
Parallel outer loop: 0.041 s
Collaped loop      : 0.024 s
Speedup(outer)     : 478.53 x
Speedup(inner)     : 813.28 x
from numba.np.ufunc import parallel
import numpy as np import time
import os
from numba import njit, prange, get_num_threads

def heavy_work(size):
  total=0.0   for i in
range(size):
total+=(i%7)*0.5
return total

def serial_processing(workloads):
results=np.zeros(len(workloads))   for
i in range(len(workloads)):
results[i]=heavy_work(workloads[i])
```

```
  return results

@njit(parallel=True) def
parallel_processing(workloads):
results=np.zeros(len(workloads))
for i in prange(len(workloads)):
    total=0.0     for j in
range(workloads[i]):
total+=(j%7)*0.5
results[i]=total   return results

  def main():     print("Loard Balancing With Irregular
Worklaods")     print("-----------------------------------
---")     print(f"Logical CPU cores    : {os.cpu_count()}")
print(f"Numba threads in use   : {get_num_threads()}\n")
workloads=np.random.randint(10_000,500_000,size=40)
parallel_processing(workloads)

    #serial timing
start=time.time()
    serial_processing(workloads)     t_serial=time.time()-
start

    #parallel timing
start=time.time()
    parallel_processing(workloads)     t_parallel=time.time()-
start

    print(f"Serial execution time    : {t_serial:.3f} s")
print(f"Parallel execution time  : {t_parallel:.3f} s")
print(f"Speedup                  : {t_setial/t_parallel:.2f}x")

if __name__ == "__main__":
main()
Serial time         : 21.128 s
Parallel outer loop: 0.042 s
Collaped loop       : 0.024 s
Speedup(outer)      : 508.36 x
Speedup(inner)      : 877.28 x
from numba.np.ufunc import parallel
import numpy as np import time
import os
from numba import njit,prange,get_num_threads
```

```
def heavy_work(size):
total = 0.0   for i in
range (size):     total +=
(i % 7) * 0.5   return
total

def serial_processing(workloads):
results = np.zeros(len(workloads))   for
i in range (len(workloads)):
results[i] = heavy_work(workloads[i])
return results


@njit (parallel=True) # Corrected 'true' to 'True'
def parallel_processing(workloads):   results =
np.zeros(len(workloads))   for i in
prange(len(workloads)):
    total= 0.0     for j in
range(workloads[i]):
      total += (j % 7) * 0.5
results[i] = total   return
results

def main(): # Added colon here
    print("load balancing with irregular workloads")     print("------
-----------------")
    print(f"logical CPU cores : {os.cpu_count()}")
print(f"NUmba threads : {get_num_threads()} \n")

    #sim
    workloads = np.random.randint(10_000,500_000,size=40) # Corrected
'randit' to 'randint'

    #warm up
    parallel_processing(workloads)

    #serial timing     start =
time.time()
serial_processing(workloads)
t_serial = time.time() - start

    #parallel     start
= time.time()
    parallel_processing(workloads)
t_parallel = time.time() - start

    print(f"Serial exe time: {t_serial:.3f} s")
print(f"Parallel exe time: {t_parallel:.3f} s")
print(f"Speedup: {t_serial/t_parallel:.2f} x")
```

```
if __name__ == "__main__": # Corrected indentation
main()

load balancing with irregular workloads
-----------------------
logical CPU cores : 2
NUmba threads : 2

Serial exe time: 0.878 s
Parallel exe time: 0.019 s
Speedup: 46.47 x

import random
import time
import multiprocessing as mp

def chunk_sum(chunk):
return sum(chunk)

def chunk_max(chunk):
return max(chunk)

def parallel_reduce(data, func):
cpu_count = mp.cpu_count()
chunk_size = len(data) // cpu_count
    chunks = [data[i:i + chunk_size] for i in range(0, len(data),
chunk_size)]

    with mp.Pool(cpu_count) as pool:

results = pool.map(func, chunks)

return func(results) if __name__ ==

"__main__":

    N = 10_000_000
    data = [random.randint(1, 100) for _ in range(N)]

    start = time.time()
serial_sum = sum(data)
serial_max = max(data)
    serial_time = time.time() - start

    start = time.time()
    parallel_sum = parallel_reduce(data, chunk_sum)
parallel_max = parallel_reduce(data, chunk_max)
parallel_time = time.time() - start

    print("Serial Sum:", serial_sum)
print("Serial Max:", serial_max)
print("Serial Time:", serial_time)
```

```
    print("Parallel Sum:", parallel_sum)
print("Parallel Max:", parallel_max)
print("Parallel Time:", parallel_time)

    print("Sum Correct:", serial_sum == parallel_sum)
print("Max Correct:", serial_max == parallel_max)
print("Speedup:", serial_time / parallel_time)
```

```
Serial Sum: 504913358
Serial Max: 100
Serial Time: 0.2057969570159912
Parallel Sum: 504913358
Parallel Max: 100
Parallel Time: 1.3743224143981934
Sum Correct: True
Max Correct: True
Speedup: 0.14974430661971583
```

```
import time import
numpy as np
from numba import njit, prange

# ---------------- Serial Version ----------------
@njit def
monte_carlo_pi_serial(n):
inside = 0     for i in
range(n):          x =
np.random.random()          y =
np.random.random()          if
x*x + y*y <= 1.0:
inside += 1     return 4.0 *
inside / n

# ---------------- Parallel Version ----------------
@njit(parallel=True) def monte_carlo_pi_parallel(n):     inside =
0     for i in prange(n):          x = np.random.random()          y
= np.random.random()          if x*x + y*y <= 1.0:
inside += 1   # Numba handles this as a safe reduction     return
4.0 * inside / n

# ---------------- Main Experiment ----------------def
main():     samples_list = [5_000_000, 10_000_000,
50_000_000]
```

```
    # Warm-up for JIT compilation
monte_carlo_pi_serial(10_000)
monte_carlo_pi_parallel(10_000)

    print(f"{'Samples':>12} | {'Serial(s)':>10} | {'Parallel(s)':>12}
| {'Speedup':>8} | {'π (parallel)':>12}")
print("-" * 70)

    for n in samples_list:
        # Serial
t0 = time.time()
        pi_s = monte_carlo_pi_serial(n)
t1 = time.time()          serial_time =
t1 - t0

        # Parallel
t0 = time.time()
        pi_p = monte_carlo_pi_parallel(n)
t1 = time.time()          parallel_time =
t1 - t0

        speedup = serial_time / parallel_time

        print(f"{n:12d} | {serial_time:10.3f} | {parallel_time:12.3f}
| {speedup:8.2f} | {pi_p:12.6f}")

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

```
     Samples |  Serial(s) |  Parallel(s) |  Speedup | π (parallel)
----------------------------------------------------------------------
     5000000 |      0.062 |        0.066 |     0.94 |     3.141834
    10000000 |      0.221 |        0.251 |     0.88 |     3.142626
    50000000 |      1.135 |        0.595 |     1.91 |     3.141647
```