Actividad 5: Ejercicios de MPI y Map-reduce.

Cómputo de Alto Rendimiento

Luis Fernando Izquierdo Berdugo

23 de marzo de 2025

Ejercicio 1

Código:

```
from mpi4py import MPI
import numpy as np
import time

def calcular_pi(n):
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    # El proceso 0 pide el número de subdivisiones
    if rank == 0:
        n = int(input("Ingrese el número de subdivisiones: "))
    else:
        n = None

    # Se transmite el valor de n a todos los procesos
    n = comm.bcast(n, root=0)

    # Medición del tiempo de inicio en paralelo
    comm.Barrier()
    start_time = MPI.Wtime()

    # Definir la suma de Riemann
    h = 1.0 / n  # Ancho de cada subintervalo
    local_sum = 0.0
    for i in range(rank, n, size):
        x = (i + 0.5) * h
```

```
        local_sum += 4.0 / (1.0 + x * x)
    local_sum *= h

    # Reducir los resultados parciales al proceso 0
    pi_total = comm.reduce(local_sum, op=MPI.SUM, root=0)

    # Medición del tiempo de finalización en paralelo
    comm.Barrier()
    end_time = MPI.Wtime()
    elapsed_time = end_time - start_time

    # Medición de tiempo en versión secuencial (solo en el proceso 0)
    if rank == 0:
        start_seq = time.time()
        pi_seq = sum(4.0 / (1.0 + ((i + 0.5) * h) ** 2) for i in range(n)) * h
        end_seq = time.time()
        elapsed_seq = end_seq - start_seq

        print(f"Aproximación de π con {n} subdivisiones: {pi_total}")
        print(f"Tiempo transcurrido (paralelo): {elapsed_time:.6f} segundos")
        print(f"Tiempo transcurrido (secuencial): {elapsed_seq:.6f} segundos")

        # Cálculo del Speed-up y eficiencia
        speed_up = elapsed_seq / elapsed_time
        efficiency = speed_up / size
        print(f"Speed-up: {speed_up:.2f}")
        print(f"Eficiencia: {efficiency:.2f}")

if __name__ == "__main__":
    calcular_pi(0)
```

Captura de pantalla de ejecución del código.

```
(Maestria) izluis@MacBook-Pro-de-Campeon Unidad 5 % mpiexec -n 2 python3 pi.py
Ingrese el número de subdivisiones: 10
Aproximación de π con 10 subdivisiones: 3.1424259850010983
Tiempo transcurrido (paralelo): 0.000139 segundos
Tiempo transcurrido (secuencial): 0.000013 segundos
Speed-up: 0.09
Eficiencia: 0.05
(Maestria) izluis@MacBook-Pro-de-Campeon Unidad 5 % mpiexec -n 2 python3 pi.py
Ingrese el número de subdivisiones: 1000
Aproximación de π con 1000 subdivisiones: 3.1415927369231254
Tiempo transcurrido (paralelo): 0.000200 segundos
Tiempo transcurrido (secuencial): 0.000205 segundos
Speed-up: 1.03
Eficiencia: 0.51
```

```
(Maestria) izluis@MacBook-Pro-de-Campeon Unidad 5 % mpiexec -np 2 python3 pi.py
Ingrese el número de subdivisiones: 1000000000
Aproximación de π con 1000000000 subdivisiones: 3.141592653590007
Tiempo transcurrido (paralelo): 66.428232 segundos
Tiempo transcurrido (secuencial): 218.932353 segundos
Speed-up: 3.30
Eficiencia: 1.65
(Maestria) izluis@MacBook-Pro-de-Campeon Unidad 5 %
```

En las métricas se puede observar que en numeros pequeños de subdivisiones como 10 y 1000, la implementación secuencial es mejor en cuento a tiempo que el paralelo, sin embargo, al incrementar a 1,000,000,000 de subdivisiones se observa un tiempo mejor, lo cual incrementa el valor de speed-up y de eficiencia. De igual manera, se observa que el valor de pi se hace más exacto conforme a más subdivisiones se ejecuten.

Ejercicio 2

Código

```python
from mpi4py import MPI


comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()


# Tiempo de inicio
start_time = MPI.Wtime()


if rank == 0:
    mensaje = "test paralelo"  # Mensaje inicial
    print(f"Soy el proceso {rank} y envío el mensaje {mensaje} al proceso {rank + 1}")
```

```
    comm.send(mensaje, dest=rank + 1)
else:
    mensaje = comm.recv(source=rank - 1)
    print(f"Soy el proceso {rank} y he recibido {mensaje}")
    if rank < size - 1:
        print(f"Soy el proceso {rank} y envío el mensaje {mensaje} al proceso {rank + 1}")
        comm.send(mensaje, dest=rank + 1)

# Tiempo final y cálculo de métricas
end_time = MPI.Wtime()
elapsed_time = end_time - start_time

total_time = comm.reduce(elapsed_time, op=MPI.MAX, root=0)
if rank == 0:
    secuential_time = total_time * size  # Tiempo secuencial supuesto
    print(f"Tiempo paralelo: {total_time:.6f} segundos")
    print(f"Tiempo secuencial supuesto: {secuential_time:.6f} segundos")
    speedup = secuential_time / total_time
    efficiency = speedup / size
    print(f"Speedup: {speedup:.6f}")
    print(f"Eficiencia: {efficiency:.6f}")

MPI.Finalize()
```

Ejecución

```
(Maestria) izluis@MacBook-Pro-de-Campeon Unidad 5 % mpiexec -n 10 python3 mensajes.py
Soy el proceso 9 y he recibido test paralelo
Soy el proceso 8 y he recibido test paralelo
Soy el proceso 8 y envío el mensaje test paralelo al proceso 9
Soy el proceso 6 y he recibido test paralelo
Soy el proceso 6 y envío el mensaje test paralelo al proceso 7
Soy el proceso 0 y envío el mensaje test paralelo al proceso 1
Tiempo paralelo: 0.055102 segundos
Tiempo secuencial supuesto: 0.551020 segundos
Speedup: 10.000000
Eficiencia: 1.000000
Soy el proceso 1 y he recibido test paralelo
Soy el proceso 1 y envío el mensaje test paralelo al proceso 2
Soy el proceso 7 y he recibido test paralelo
Soy el proceso 7 y envío el mensaje test paralelo al proceso 8
Soy el proceso 2 y he recibido test paralelo
Soy el proceso 2 y envío el mensaje test paralelo al proceso 3
Soy el proceso 3 y he recibido test paralelo
Soy el proceso 3 y envío el mensaje test paralelo al proceso 4
Soy el proceso 4 y he recibido test paralelo
Soy el proceso 4 y envío el mensaje test paralelo al proceso 5
Soy el proceso 5 y he recibido test paralelo
Soy el proceso 5 y envío el mensaje test paralelo al proceso 6
```

```
(Maestria) izluis@MacBook-Pro-de-Campeon Unidad 5 % mpiexec -n 2 python3 mensajes.py
Soy el proceso 1 y he recibido test paralelo
Soy el proceso 0 y envío el mensaje test paralelo al proceso 1
Tiempo paralelo: 0.001592 segundos
Tiempo secuencial supuesto: 0.003184 segundos
Speedup: 2.000000
Eficiencia: 1.000000
```

Se estima el tiempo secuencial usando el tiempo total multiplicado por el número de procesos utilizados., siendo el speedup siempre el número de procesos y una eficiencia constante de 1. Algo que llamó mi atención en este ejercicio fue que cuando aumentaba el número de procesos a más que el número de núcleos de mi PC (2), el programa empezaba a imprimir los envíos y recibos de mensajes después de arrojarme el resultado.

Ejercicio 3

Código

```python
from mpi4py import MPI
import numpy as np
import time


def producto_escalar():
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()
```

```
N = 10 # Tamaño del vector

if rank == 0:
    A = np.random.rand(N)
    B = np.random.rand(N)
else:
    A = None
    B = None

# Tamaño del fragmento que recibe cada proceso
local_N = N // size
local_A = np.zeros(local_N)
local_B = np.zeros(local_N)

# Distribuir los vectores entre los procesos
comm.Scatter(A, local_A, root=0)
comm.Scatter(B, local_B, root=0)

# Medición del tiempo en paralelo
comm.Barrier()
start_time = MPI.Wtime()

# Producto escalar parcial
local_dot = np.dot(local_A, local_B)

# Reunir resultados en el proceso 0
total_dot = comm.reduce(local_dot, op=MPI.SUM, root=0)

comm.Barrier()
end_time = MPI.Wtime()
elapsed_time = end_time - start_time

if rank == 0:
    # Versión secuencial
    start_seq = time.time()
    dot_seq = np.dot(A, B)
    end_seq = time.time()
```

```
    elapsed_seq = end_seq - start_seq

    print(f"Producto escalar paralelo: {total_dot}")
    print(f"Producto escalar secuencial: {dot_seq}")
    print(f"Tiempo transcurrido (paralelo): {elapsed_time:.6f} segundos")
    print(f"Tiempo transcurrido (secuencial): {elapsed_seq:.6f} segundos")

    # Cálculo del Speed-up y eficiencia
    speed_up = elapsed_seq / elapsed_time
    efficiency = speed_up / size
    print(f"Speed-up: {speed_up:.2f}")
    print(f"Eficiencia: {efficiency:.2f}")

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

```
(Maestria) izluis@MacBook-Pro-de-Campeon Unidad 5 % mpiexec -np 2 python3 produc
to.py
Producto escalar paralelo: 250000875.5193238
Producto escalar secuencial: 250000875.5193253
Tiempo transcurrido (paralelo): 59.521230 segundos
Tiempo transcurrido (secuencial): 58.994781 segundos
Speed-up: 0.99
Eficiencia: 0.50
(Maestria) izluis@MacBook-Pro-de-Campeon Unidad 5 % mpiexec -np 2 python3 produc
to.py
Producto escalar paralelo: 24.19825851412274
Producto escalar secuencial: 24.19825851412274
Tiempo transcurrido (paralelo): 0.000447 segundos
Tiempo transcurrido (secuencial): 0.000009 segundos
Speed-up: 0.02
Eficiencia: 0.01
(Maestria) izluis@MacBook-Pro-de-Campeon Unidad 5 % mpiexec -np 2 python3 produc
to.py
Producto escalar paralelo: 2.024838542203067
Producto escalar secuencial: 2.024838542203067
Tiempo transcurrido (paralelo): 0.000399 segundos
Tiempo transcurrido (secuencial): 0.000012 segundos
Speed-up: 0.03
Eficiencia: 0.02
```

Se ejecutó el código con vectores tamaño 1,000,000,000, 100 y 10 donde se observó de nuevo que en tiempo de procesamiento no hay una gran diferencia incluso en vectores muy grandes, siendo el Speed-Up más grande de 0.99, eficiencia de 0.5 y diferencia de tiempo de poco menos de un segundo.

Ejercicio 4

Código

```
from mpi4py import MPI
import numpy as np
import time

def multiplicacion_matriz_vector():
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    N = size  # La matriz debe ser NxN, con N igual al número de procesos

    if rank == 0:
        A = np.random.rand(N, N)
        x = np.random.rand(N)
    else:
        A = None
        x = np.zeros(N)

    # Cada proceso recibe una fila de A
    local_A = np.zeros(N)
    comm.Scatter(A, local_A, root=0)

    # Se transmite el vector x a todos los procesos
    comm.Bcast(x, root=0)

    # Medición del tiempo en paralelo
    comm.Barrier()
    start_time = MPI.Wtime()

    # Cada proceso calcula su producto fila-vector
    local_result = np.dot(local_A, x)

    # Reunir resultados en el proceso 0
```

```
    result = None
    if rank == 0:
        result = np.zeros(N)
    comm.Gather(local_result, result, root=0)

    comm.Barrier()
    end_time = MPI.Wtime()
    elapsed_time = end_time - start_time

    if rank == 0:
        # Versión secuencial
        start_seq = time.time()
        result_seq = np.dot(A, x)
        end_seq = time.time()
        elapsed_seq = end_seq - start_seq

        print(f"Prueba con un tamaño de {N}x{N}")
        print(f"Resultado paralelo: {result}")
        print(f"Resultado secuencial: {result_seq}")
        print(f"Tiempo transcurrido (paralelo): {elapsed_time:.6f} segundos")
        print(f"Tiempo transcurrido (secuencial): {elapsed_seq:.6f} segundos")

        # Cálculo del Speed-up y eficiencia
        speed_up = elapsed_seq / elapsed_time
        efficiency = speed_up / size
        print(f"Speed-up: {speed_up:.2f}")
        print(f"Eficiencia: {efficiency:.2f}")

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

Ejecución

```
(Maestria) izluis@MacBook-Pro-de-Campeon Unidad 5 % mpiexec -n 4 python3 matriz_vector
.py
Prueba con un tamaño de 4x4
Resultado paralelo: [1.0508684  1.07589984 1.53469987 1.14028288]
Resultado secuencial: [1.0508684  1.07589984 1.53469987 1.14028288]
Tiempo transcurrido (paralelo): 0.000099 segundos
Tiempo transcurrido (secuencial): 0.000010 segundos
Speed-up: 0.10
Eficiencia: 0.03
(Maestria) izluis@MacBook-Pro-de-Campeon Unidad 5 % mpiexec -n 2 python3 matriz_vector
.py
Prueba con un tamaño de 2x2
Resultado paralelo: [0.68143615 0.66606245]
Resultado secuencial: [0.68143615 0.66606245]
Tiempo transcurrido (paralelo): 0.000052 segundos
Tiempo transcurrido (secuencial): 0.000008 segundos
Speed-up: 0.15
Eficiencia: 0.08
```

Se hizo la prueba con vectores matrices de tamaño 4x4 y 2x2, en lo cual se observó que los procesos secuenciales son mejores que los paralelos, teniendo speed-up de tamaños 0.1 y 0.15, así como eficiencias de 0.03 y 0.08.

Ejercicio 5

Código

Map.py

```python
import sys
import time
import re


def map_words():
    """
    Fase Map: Lee líneas de texto desde la entrada estándar (sys.stdin),
    elimina caracteres especiales excepto ñ y letras con acentos,
    convierte a minúsculas, divide cada línea en palabras y emite pares palabra\t1.
    """
    # Inicia un temporizador para medir el tiempo de ejecución
    start_time = time.time()

    # Lee líneas de la entrada estándar
    for line in sys.stdin:
        # Elimina caracteres especiales excepto letras con acentos y ñ
        clean_line = re.sub(r'[^a-zA-ZáéíóúÁÉÍÓÚñÑ0-9\s]', '', line).lower()
```

```
        # Divide la línea en palabras
        words = clean_line.strip().split()
        # Emite cada palabra con un conteo inicial de 1
        for word in words:
            print(f"{word}\t1")

    # Calcula el tiempo total de ejecución
    end_time = time.time()
    elapsed_time = end_time - start_time

    # Imprime el tiempo transcurrido en la salida de error estándar (sys.stderr)
    print(f"Tiempo transcurrido (Map): {elapsed_time:.6f} segundos", file=sys.stderr)

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

reduce.py

```
import sys
from collections import defaultdict
import time

def reduce_words():
    """
    Fase Reduce: Lee pares palabra\tconteo desde la entrada estándar (sys.stdin),
    agrupa las palabras y suma sus conteos totales.
    """
    # Inicia un temporizador para medir el tiempo de ejecución
    start_time = time.time()

    # Diccionario para almacenar la cuenta de cada palabra
    # defaultdict(int) inicializa automáticamente los valores en 0
    word_counts = defaultdict(int)

    # Lee líneas de la entrada estándar
    # Cada línea contiene una palabra y su cuenta separadas por un tabulador
    for line in sys.stdin:
```

```
        word, count = line.strip().split("\t")  # Divide la línea en palabra y cuenta
        word_counts[word] += int(count)        # Suma la cuenta al total de la palabra

    # Itera sobre las palabras y sus cuentas, ordenadas alfabéticamente
    for word, count in sorted(word_counts.items()):
        # Imprime cada palabra y su cuenta separadas por un tabulador
        print(f"{word}\t{count}")

    # Calcula el tiempo total de ejecución
    end_time = time.time()
    elapsed_time = end_time - start_time

    # Imprime el tiempo transcurrido en la salida de error estándar (sys.stderr)
    print(f"Tiempo transcurrido (Reduce): {elapsed_time:.6f} segundos", file=sys.stderr)

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

Ejecución

```
(Maestria) izluis@MacBook-Pro-de-Campeon Unidad 5 % cat *.txt | python map.py | python
 reduce.py > res.txt
Tiempo transcurrido (Map): 0.001329 segundos
Tiempo transcurrido (Reduce): 0.015254 segundos
```

res.txt

```
0000    2
1   184
10  1
11  1
12  1
13  2
15  1
18  1
180 2
180c    2
2   44
20  2
```

200 2
25 2
250 2
26 1
3 21
350 3
380 2
4 12
40 1
5 4
6 4
7 5
70 2
8 3
9 4
90 2
a 14
aadir 2
acudi 1
acudió 1
advenimiento 2
agregar 3
ahora 3
aislamiento 2
al 4
alados 2
alc 1
alcé 1
alegra 1
alegría 9
all 1
allí 1
almendras 3
amable 2
amables 2
ambamos 1
amorosa 2

amábamos 2
arenosa 2
azcar 5
azotea 3
bajo 2
barnizar 3
besaban 2
bol 2
bondadoso 2
brazos 2
calle 2
caminbamos 1
camino 2
caminábamos 1
canciones 2
canela 3
cantbamos 1
canto 2
cantos 2
cantábamos 1
casa 3
cerezas 3
charola 2
colocar 2
como 4
compadecidas 2
con 13
contemplar 3
corazn 1
corazón 1
cortar 2
creci 1
creció 1
cuando 12
cubos 2
cuchicheaba 2
cuidados 2

cuidando 2
dar 2
das 1
de 19
desde 2
deshidratadas 2
dicen 2
diferentes 2
dulzura 2
durante 2
días 2
el 10
ella 2
elocuente 3
en 7
enferm 1
enfermó 1
enfriar 2
engalanadas 2
envidiara 2
era 6
eran 3
es 2
escuchan 2
escucharme 2
escucharnos 2
espolvoreada 2
espolvorear 3
estaba 2
estaban 3
estbamos 1
este 2
estábamos 1
extender 2
extraos 1
extraños 1
extremada 2

forma 2
fuera 2
fuerte 2
fuerza 2
g 11
gente 2
gozaba 2
gracias 2
grande 2
gritar 2
ha 2
haba 1
hablbamos 1
hablo 2
hablábamos 1
había 1
harina 4
harinas 2
hasta 3
hasto 1
hastío 1
hermosa 2
hermosura 2
hoja 2
hombre 2
hornear 5
hoy 2
huevo 3
huevos 3
impregnadas 2
instante 2
juntos 3
la 14
labios 3
las 4
le 3
leche 3

lengua 2
llena 2
lleno 2
los 5
luego 4
lunas 2
mantequilla 3
mar 2
maravillosas 2
masa 2
me 4
melodas 1
melodías 1
mesa 2
mezcla 2
mezclado 2
mi 27
mil 2
minutos 2
mira 2
miraba 2
mirad 2
mis 7
ml 2
mo 1
molida 3
ms 1
muerta 3
mundo 2
muri 1
murió 3
más 1
mío 2
naci 1
nacido 2
nació 4
nadie 4

ningn 1
ningún 1
no 4
noble 2
noches 2
nos 5
nuestras 3
nuestros 4
obtener 2
odos 1
oigo 2
ojos 2
orgulloso 2
orse 1
otoo 1
otoño 1
oídos 1
oírse 1
palabras 2
palideci 1
palideció 1
para 5
pasar 2
pero 5
pesadas 2
placentero 2
polvo 3
por 3
porque 4
proclam 1
proclamé 1
prodigu 1
prodigué 1
profundos 2
pues 5
qu 1
que 8

qued 1
quedé 1
quien 2
quiso 2
qué 1
raspa 2
re 1
recordar 2
recuerdo 3
recuerdos 2
reflexiones 2
rejilla 2
retirar 2
rodeaba 2
rodillo 2
ríe 1
sal 3
scones 2
se 2
seguidas 2
senta 1
sentbanse 1
sentábanse 1
sentía 1
ser 4
seres 2
siete 2
sin 2
slo 1
sobre 4
sol 2
soledad 2
solo 4
solos 2
sorpresa 2
su 3
sub 1

subí 1
suenan 2
sueos 1
sueños 2
suficiente 2
sus 2
susurra 2
sólo 4
tambin 1
también 3
ternura 2
todas 2
todo 2
todos 3
tostadas 2
trabajar 4
tringulos 2
triste 2
tristeza 16
un 5
una 6
unirse 2
vaciar 2
vecino 2
vecinos 4
venid 4
ventana 2
ver 2
viento 2
vigil 1
vigilé 1
visitarnos 2
viviente 2
vivientes 2
voces 2
vueltas 2
vuelve 2

```
y  41
ya  3
yace  2
yo  8
za  2
```

En este código se utilizó cat en vez de ls en el comando de ejecución, ya que, al ejecutar con ls, solamente se leía el nombre de los archivos y no el contenido de estos. De igual se observan tiempos muy pequeños al ejecutar el map y el reduce.

Referencias:

Arellano Vázquez, M. (s.f.). *Paradigam map-reduce*. Ciudad de México, México: INFOTEC.

Arellano Vázquez, M. (s.f.). *Lenguaje MPI.* Ciudad de México, México: INFOTEC.

Arellano Vázquez, M. (2025). *Manual de Funciones de MPI en C++ y Python 3.* Ciudad de México, México: INFOTEC.