Computación Avanzada

Memoria distribuida / Pasaje de mensajes

4° Licenciatura En Sistemas de Información

2020

Universidad: UADER FCyT Concepción del Uruguay

Profesores: María Fabiana Piccoli, Carlos Casanova

Alumnos: Cepeda Leandro, Gonzalez Exequiel,

Ejercicio 1: Hello World.

a) El siguiente código resuelve un saludo en paralelo de todos los procesos que trabajan en paralelo

```
#include<stdio.h>
#include<mpi.h>

main(intargc, char **argv){
          int node;
          MPI_Init(&argc, &argv);
          MPI_Comm_rank(MPI_COMM_WORLD, &node);
          printf("Hello World fromNode %d\n",node);
          MPI_Finalize();
}
```

ej1a.py

```
#!/usr/bin/env python3

from mpi4py import MPI

comm = MPI.COMM_WORLD
node = comm.Get_rank()

print(f'Hello world fromNode {node}')
```

Ejecutar el programa para distinta cantidad de procesos. Analizar la salida de la ejecución y compararla entre ellas.

```
lean@lean:~/MEGA/MEGAsync/Lean/UADER/4° Lic. en Sistemas de Informacion/Computacion Avanzada/CA [tps]/CA [tp2]$ mpiexec -n 4 python3 ej1a.py
Hello world fromNode 1
Hello world fromNode 2
Hello world fromNode 0
Hello world fromNode 3
```

b) Modificar el programa anterior de manera que los procesos paralelos envíen el mensaje de saludo "Saludos desde el proceso #" al Proceso 0, quien será el responsable de imprimirlo en la salida estándar. La salida tendrá las siguientes características:

Hola, soy el proceso 0 (hay "n" procesos) y recibo:
(Hola desde el proceso #)
(Hola desde el proceso #)
...

ej1b.py

```
#!/usr/bin/env python3

from mpi4py import MPI

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

if rank != 0:
    data = f'Hola desde el proceso {rank}'
    req = comm.isend(data, dest=0, tag=rank)
    req.wait()
else:
    for i in range(1, size):
        req = comm.irecv(source=i, tag=i)
        data = req.wait()
        print(f'Hola soy el proceso {rank} (hay {size} procesos) y recibo: {data}')
```

```
lean@lean:~/MEGA/MEGAsync/Lean/UADER/4° Lic. en Sistemas de Informacion/Computacion Avanzada/CA [tps]/CA [tp2]$ mpiexec -n 4 python3 ej1b.py
Hola soy el proceso 0 (hay 4 procesos) y recibo:
Hola desde el proceso 1
Hola desde el proceso 2
Hola desde el proceso 3
```

c) Modificar el programa anterior, de manera tal que sea posible recibir e imprimir los mensajes enviados al Proceso 0 en orden ascendente. Es decir, para cuatro procesos la salida debería ser la siguiente:

Hola, soy el proceso 0 (hay 4 procesos) y recibo:
(Hola desde el proceso 1)
(Hola desde el proceso 2)
(Hola desde el proceso 3)
(Hola desde el proceso 4)

ej1c.py

```
#!/usr/bin/env python3

from mpi4py import MPI

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

if rank != 0:
    data = f'Hola desde el proceso {rank}'
    comm.send(data, dest=0, tag=rank)
else:
    print(f'Hola soy el proceso {rank} (hay {size} procesos) y recibo:')
    for i in range(1, size):
        data = comm.recv(source=i, tag=i)
        print(f'{data}')
```

```
lean@lean:~/MEGA/MEGAsync/Lean/UADER/4° Lic. en Sistemas de Informacion/Computacion Avanzada/CA [tps]/CA [tp2]$ mpiexec -n 4 python3 ej1c.py
Hola soy el proceso 0 (hay 4 procesos) y recibo:
Hola desde el proceso 1
Hola desde el proceso 2
Hola desde el proceso 3
```

d) ¿Qué modificaciones debe hacer al programa anterior para que el proceso responsable de imprimir sea otro proceso distinto del Proceso 0? Realícelas.

ej1d.py

```
#!/usr/bin/env python3

from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
receptor = 3

if rank != receptor:
    data = f'Hola desde el proceso {rank}'
    comm.send(data, dest=receptor, tag=rank)
else:
    print(f'Hola soy el proceso {receptor} (hay {size} procesos) y recibo:')
    for i in range(0, size):
        if i != receptor:
            data = comm.recv(source=i, tag=i)
            print(f'{data}')
```

```
lean@lean:~/MEGA/MEGAsync/Lean/UADER/4° Lic. en Sistemas de Informacion/Computacion Avanzada/CA [tps]/CA [tp2]$ mpiexec -n 4 python3 ej1d.py
Hola soy el proceso 3 (hay 4 procesos) y recibo:
Hola desde el proceso 0
Hola desde el proceso 1
Hola desde el proceso 2
```

Ejercicio 2: Comunicaciones Colectivas.

Para los ejemplos de algoritmos dados en teoría de comunicaciones colectivas, se pide:

a) Modificarlo de manera tal que sólo se usen comunicaciones punto a punto bloqueante.

ej2a_mpi_bcast.py

```
#!/usr/bin/env python3

from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
transmitter = 0

if rank == transmitter:
    buffer = input(f'Hola, soy el proceso {rank}, ingrese un número: ')

    for i in range(1, size):
        comm.send(buffer, dest=i, tag=i)

    print(f'Hola, soy el proceso {rank} y el número es el {buffer}')
else:
    data = comm.recv(source=transmitter, tag=rank)
    print(f'Hola, soy el proceso {rank} y el número es el {data}')
```

```
lean@lean:~/MEGA/MEGAsync/Lean/UADER/4° Lic. en Sistemas de Informacion/Computacion Avanzada/CA [tps]/CA [tp2]/ej2$ mpiexec -n 4 python3 ej2a_mpi_bcast.py
Hola, soy el proceso 0, ingrese un número: 2
Hola, soy el proceso 0 y el número es el 2
Hola, soy el proceso 2 y el número es el 2
Hola, soy el proceso 1 y el número es el 2
Hola, soy el proceso 3 y el número es el 2
```

ej2a_mpi_reduce.py

```
#!/usr/bin/env python3

from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
receiver = 0

if rank != 0:
    comm.send(rank+1, dest=receiver, tag=rank)
else:
    factorial = 1
    for i in range(1, size):
        data = comm.recv(source=i, tag=i)
        factorial = factorial * data
    print(f'Hola soy el proceso {rank} el factorial de {size} es {factorial}')
```

```
lean@lean:~/MEGA/MEGAsync/Lean/UADER/4° Lic. en Sistemas de Informacion/Computacion Avanzada/CA [tps]/CA [tp2]/ej2$ mpiexec -n 4 python3 ej2a_mpi_reduce.py
Hola soy el proceso 0 el factorial de 4 es 24
```

b) Modificarlo de manera tal que sólo se usen comunicaciones punto a punto no-bloqueante.

ej2b_mpi_bcast.py

```
#!/usr/bin/env python3

from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
transmitter = 0

if rank == transmitter:
    buffer = input(f'Hola, soy el proceso {rank}, ingrese un número: ')

    for i in range(size):
        req = comm.isend(buffer, dest=i, tag=i)
    req = comm.irecv(source=transmitter, tag=rank)
    data = req.wait()
    print(f'Hola, soy el proceso {rank} y el número es el {data}')
else:
    req = comm.irecv(source=transmitter, tag=rank)
    data = req.wait()
    print(f'Hola, soy el proceso {rank} y el número es el {data}')
```

```
lean@lean:~/MEGA/MEGAsync/Lean/UADER/4° Lic. en Sistemas de Informacion/Computacion Avanzada/CA [tps]/CA [tp2]/ej2$ mpiexec -n 4 python3 ej2b_mpi_bcast.py
Hola, soy el proceso 0, ingrese un número: 123
Hola, soy el proceso 0 y el número es el 123
Hola, soy el proceso 2 y el número es el 123
Hola, soy el proceso 3 y el número es el 123
Hola, soy el proceso 1 y el número es el 123
```

ej2b_mpi_reduce.py

```
#!/usr/bin/env python3

from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
receiver = 0

if rank != 0:
    req = comm.isend(rank+1, dest=receiver, tag=rank)
    req.wait()
else:
    factorial = 1
    for i in range(1, size):
        req = comm.irecv(source=i, tag=i)
        data = req.wait()
        factorial = factorial * data
    print(f'Hola soy el proceso {rank} el factorial de {size} es {factorial}')
```

```
lean@lean:~/MEGA/MEGAsync/Lean/UADER/4° Lic. en Sistemas de Informacion/Computacion Avanzada/CA [tps]/CA [tp2]/ej2$ mpiexec -n 4 python3 ej2b_mpi_reduce.py
Hola soy el proceso 0 el factorial de 4 es 24
```

c) ¿Existe alguna diferencia entre ambas soluciones?¿Fue necesario en alguna de ellas el uso de sincronización por barreras?

La diferencia entre ambas soluciones es la comunicación entre procesos según sean bloqueante o no bloqueante.

Un send/receive bloqueante suspende la ejecución del programa hasta que el buffer que se está enviando/recibiendo es seguro de usar. En el caso de un send bloqueante, significa que los datos a ser enviados han sido copiados en el buffer del sistema (envío) y no necesariamente han sido recibidos por la tarea que recibe.

Las llamadas no bloqueantes retornan inmediatamente después de ser iniciada la comunicación. El programador no sabe si los datos han sido enviados o copiados en el buffer del sistema (envío) o si los datos ha ser recibidos han llegado.

En ninguno de los casos fue necesario la sincronización por barrera para la resolución de los ejercicios.

d) Implemente cada comunicación colectiva descripta en teoría con comunicaciones punto a punto unicamente.

ej2d_mpi_allgater.py

```
#!/usr/bin/env python3

from mpi4py import MPI
import numpy as np

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

# Creo un vector con la logitud de la cantidad de procesos
x = np.arange(size, dtype=np.int) + rank

print(f'Soy el Proceso {rank}, mi vector es {x}')

# Creo la matriz donde voy a almacenar el resultado del all gather
y = np.zeros((size, len(x)), dtype=np.int)

# Comunicacion colectiva
comm.Allgather([x, MPI.INT], [y, MPI.INT])

print(f'Soy el Proceso {rank}, y recibo {y}')
```

```
lean@lean:~/MEGA/MEGAsync/Lean/UADER/4° Lic. en Sistemas de Informacion/Computacion Avanzada/CA [tps]/CA [tp2]/ej2$ mpiexec -n 4 python3 ej2d_mpi_allgater.py
Soy el Proceso 0, mi vector es [0 1 2 3]
Soy el Proceso 0, y recibo [[0 1 2 3]
 [1 2 3 4]
 [2 3 4 5]
 [3 4 5 6]]
Soy el Proceso 1, mi vector es [1 2 3 4]
Soy el Proceso 1, y recibo [[0 1 2 3]
 [1 2 3 4]
 [2 3 4 5]
 [3 4 5 6]]
Soy el Proceso 3, mi vector es [3 4 5 6]
Soy el Proceso 3, y recibo [[0 1 2 3]
 [1 2 3 4]
 [2 3 4 5]
 [3 4 5 6]]
Soy el Proceso 2, mi vector es [2 3 4 5]
Soy el Proceso 2, y recibo [[0 1 2 3]
 [1 2 3 4]
 [2 3 4 5]
 [3 4 5 6]]
```

ej2d_mpi_alltoall.py

```
#!/usr/bin/env python3

from mpi4py import MPI
import numpy as np

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

# Cargo una fila random
m = np.random.randint(0, 10, size=(size, 1), dtype=np.int)

# Imprimo mi fila
print(f'\nSoy el proceso {rank} y GENERE: \n \n {m}\n')

# Vector donde se almacena la fila luego del alltoall
r = np.zeros((size, 1), dtype=np.int)

# Comunicacion colectiva
comm.Alltoall(m, r)

# Imprimo el resultado
for i in range(len(r)):
    print(f'Soy el proceso {rank} y RECIBI la fila {r[i]} del proceso {i}')
```

```
lean@lean:~/MEGA/MEGAsync/Lean/UADER/4° Lic. en Sistemas de Informacion/Computacion Avanzada/CA [tps]/CA [tp2]/ej2$ mpiexec -n 4 python3 ej2d_mpi_alltoall.py

Soy el proceso 0 y GENERE:

 [[4]
 [2]
 [5]
 [7]]

Soy el proceso 0 y RECIBI la fila [4] del proceso 0
Soy el proceso 0 y RECIBI la fila [8] del proceso 1
Soy el proceso 0 y RECIBI la fila [2] del proceso 2
Soy el proceso 0 y RECIBI la fila [6] del proceso 3

Soy el proceso 3 y GENERE:

 [[6]
 [4]
 [5]
 [5]]

Soy el proceso 3 y RECIBI la fila [7] del proceso 0
Soy el proceso 3 y RECIBI la fila [0] del proceso 1
Soy el proceso 3 y RECIBI la fila [4] del proceso 2
Soy el proceso 3 y RECIBI la fila [5] del proceso 3
```

ej2d_mpi_bcast.py

```
#!/usr/bin/env python3

from mpi4py import MPI
import numpy as np

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

# El proceso 0 genera los data
if(rank == 0):

    data = np.random.randint(0, 10, size)
    print(f'Soy el proceso 0 y genere los datos: {data}')

else:
    # Los demas procesos tambien tienen que tener definida la variable
    data = None

# Comunicacion colectiva
data = comm.bcast(data, root=0)

# Cada proceso imprime los data recibidos
print(f'Soy el proceso, {rank} y recibi {data}')
```

```
lean@lean:~/MEGA/MEGAsync/Lean/UADER/4° Lic. en Sistemas de Informacion/Computacion Avanzada/CA [tps]/CA [tp2]/ej2$ mpiexec -n 4 python3 ej2d_mpi_bcast.py
Soy el proceso 0 y genere los datos: [9 6 5 3]
Soy el proceso, 0 y recibi [9 6 5 3]
Soy el proceso, 1 y recibi [9 6 5 3]
Soy el proceso, 2 y recibi [9 6 5 3]
Soy el proceso, 3 y recibi [9 6 5 3]
```

ej2d_mpi_gather.py

```
#!/usr/bin/env python3

from mpi4py import MPI
import random

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

# Creo un numero random
data = random.randint(0, 10)
print(f'Soy el proceso, {rank}, y mi numero es, {data}')

# Comunicacion colectiva
buffer = comm.gather(data, root=0)

if(rank == 0):
    # El proceso 0 es el que recibe los datas
    print(f'Soy el proceso {rank} y mis datos son, {buffer}')
```

```
lean@lean:~/MEGA/MEGAsync/Lean/UADER/4° Lic. en Sistemas de Informacion/Computacion Avanzada/CA [tps]/CA [tp2]/ej2$ mpiexec -n 4 python3 ej2d_mpi_gather.py
Soy el proceso, 1, y mi numero es, 0
Soy el proceso, 3, y mi numero es, 4
Soy el proceso, 0, y mi numero es, 8
Soy el proceso 0 y mis datos son, [8, 0, 8, 4]
Soy el proceso, 2, y mi numero es, 8
```

ej2d_mpi_reduction.py

```
#!/usr/bin/env python3

from mpi4py import MPI
import random

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

# Creo un numero random
data = random.randint(0, 100)
print(f'Soy el proceso, {rank}, y mi numero es, {data}')

# Comunicacion colectiva
res = comm.reduce(data, op=MPI.MAX, root=0)
# MPI_MAX        Máximo entre los elementos
# MPI_MIN        Mínimo entre los elementos
# MPI_SUM        Suma
# MPI_PROD       Producto
# MPI_LAND       AND lógico (devuelve 1 o 0, verdadero o falso)
# MPI_BAND       AND a nivel de bits
# MPI_LOR        OR lógico
# MPI_BOR        OR a nivel de bits
# MPI_LXOR       XOR lógico
# MPI_BXOR       XOR a nivel de bits
# MPI_MAXLOC     Valor máximo entre los elementos y el rango del proceso que lo tenía
# MPI_MINLOC     Valor mínimo entre los elementos y el rango del proceso que lo tenía

if(rank == 0):
    # El proceso 0 es el que recibe el resultado de la operacion
    print(f'Soy el proceso 0 y el resultado de la operacion es, {res}')
```

```
lean@lean:~/MEGA/MEGAsync/Lean/UADER/4° Lic. en Sistemas de Informacion/Computacion Avanzada/CA [tps]/CA [tp2]/ej2$ mpiexec -n 4 python3 ej2d_mpi_reduction.py
Soy el proceso, 2, y mi numero es, 73
Soy el proceso, 3, y mi numero es, 11
Soy el proceso, 0, y mi numero es, 82
Soy el proceso 0 y el resultado de la operacion es, 82
Soy el proceso, 1, y mi numero es, 28
```

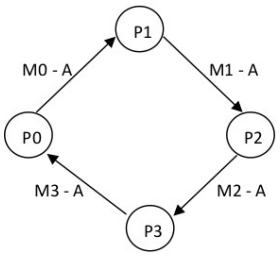
ej2d_mpi_scater.py

```
#!/usr/bin/env python3

from mpi4py import MPI
import numpy as np
import random

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

# Tamanio del sub vector de cada proceso
miN = size // size

# Inicializo los arreglos vacios
dataSent = np.empty(size, dtype=np.int)       # SendBuff
dataReceived = np.empty(miN, dtype=np.int)  # RecvBuff

if(rank == 0):
    # El proceso 0 se encarga de cargar el arreglo random
    dataSent = np.random.randint(0, 10, size=(1, size))

    print(f'Soy el proceso 0 y los datos son {dataSent}')

# Comunicacion colectiva
comm.Scatter([dataSent, MPI.INT], [dataReceived, MPI.INT])

print(f'Soy el proceso, {rank}, y mis datos son, {dataReceived}')
```

```
lean@lean:~/MEGA/MEGAsync/Lean/UADER/4° Lic. en Sistemas de Informacion/Computacion Avanzada/CA [tps]/CA [tp2]/ej2$ mpiexec -n 4 python3 ej2d_mpi_scater.py
Soy el proceso, 2, y mis datos son, [3]
Soy el proceso, 1, y mis datos son, [8]
Soy el proceso 0 y los datos son [[1 8 3 7]]
Soy el proceso, 0, y mis datos son, [1]
Soy el proceso, 3, y mis datos son, [7]
```

Ejercicio 3: Anillo.

Desarrollar un programa MPI denominado “anillo”, en donde los procesos deben hacer circular un mensaje a través de un “anillo lógico”. El programa deberá recibir como parámetro un entero n que indicará la cantidad de vueltas que el mensaje debe dar al anillo. Ejemplo con cuatro procesos:



El proceso 0 envía el M0 con el dato “A” al proceso 1
El proceso 1 envía el M1 con el dato “A” al proceso 2
El proceso 2 envía el M2 con el dato “A” al proceso 3
El proceso 3 envía el M3 con el dato “A” al proceso 0

Este proceso será repetido tantas veces como lo indique El parámetro “n”.

REAMDE.md

> Para correr el programa recibe un argumento n (cantidad de vueltas) al Anillo: ej3_mpi_anillo.py [n]
> Ejemplo: mpiexec -n 4 python3 ej3_mpi_anillo.py 2

ej3_mpi_anillo.py

```
#!/usr/bin/env python3

import sys
from mpi4py import MPI

def main(n):

    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()
    message = 'A'

    for i in range(n):
        print(f'\nVuelta al anillo N°: {i} \n')

        if rank == 0:

            fromProc = size-1
            res = comm.irecv(None, source=fromProc)
            print(f'El proceso {rank} recibio el mensaje {message} del proceso {fromProc}', flush=True)

            nextProc = rank + 1
            comm.send(message, dest=nextProc)
            print(f'El proceso {rank} envia el mensaje {message} al proceso {nextProc}', flush=True)
            res.wait()

        else:

            fromProc = rank-1
            data = comm.recv(None, source=fromProc)
            print(f'El proceso {rank} recibio el mensaje {data} del proceso {fromProc}', flush=True)

            nextProc = 0 if (rank + 1) ⩾ size else rank + 1
            comm.send(message, dest=nextProc)
            print(f'El proceso {rank} envia el mensaje {message} al proceso {nextProc}', flush=True)


if __name__ == "__main__":
        if len(sys.argv) > 1:
            n = int(sys.argv[1])
            main(n)
        else:
            print("Debe ingresar como parametro la cantidad de vueltas del Anillo")
```

```
lean@lean:~/MEGA/MEGAsync/Lean/UADER/4° Lic. en Sistemas de Informacion/Computacion Avanzada/CA [tps]/CA [tp2]/ej3$ mpiexec -n 4 python3 ej3_mpi_anillo.py 2

Vuelta al anillo N°: 0

El proceso 0 recibio el mensaje A del proceso 3
El proceso 0 envia el mensaje A al proceso 1

Vuelta al anillo N°: 0

El proceso 1 recibio el mensaje A del proceso 0
El proceso 1 envia el mensaje A al proceso 2

Vuelta al anillo N°: 0

El proceso 2 recibio el mensaje A del proceso 1
El proceso 2 envia el mensaje A al proceso 3

Vuelta al anillo N°: 0

El proceso 3 recibio el mensaje A del proceso 2
El proceso 3 envia el mensaje A al proceso 0

Vuelta al anillo N°: 1

El proceso 0 recibio el mensaje A del proceso 3
El proceso 0 envia el mensaje A al proceso 1

Vuelta al anillo N°: 1

El proceso 1 recibio el mensaje A del proceso 0
El proceso 1 envia el mensaje A al proceso 2

Vuelta al anillo N°: 1

El proceso 2 recibio el mensaje A del proceso 1
El proceso 2 envia el mensaje A al proceso 3

Vuelta al anillo N°: 1

El proceso 3 recibio el mensaje A del proceso 2
El proceso 3 envia el mensaje A al proceso 0
```

Ejercicio 4: Multiplicación de matriz por un vector.
Analizar diferentes soluciones paralelas para resolver la multiplicación de una matriz por un vector. Analice soluciones paralelas considerando los distintos tipos de particionado de datos. Ejemplo:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 & 17 & 18 \\ 19 & 20 & 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 & 29 & 30 \\ 31 & 32 & 33 & 34 & 35 & 36 \end{pmatrix} \times \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{pmatrix} = \begin{pmatrix} 91 \\ 217 \\ 343 \\ 469 \\ 595 \\ 721 \end{pmatrix}$$

a) Resuelva el problema planteado considerando que cada proceso realiza la multiplicación de una fila de la matriz por el vector completo. Considera en la solución el siguiente esquema de código:

1. El proceso 0 generará la matriz y el vector.
2. Las filas de la matriz se repartirán entre los procesos participantes.
3. El vector se replicará en los procesos.
4. Cada proceso realizará las operaciones que le corresponden.
5. El vector resultado es reunido por el proceso 0.
6. El proceso 0 mostrará el resultado.

Observación: El programa debe ser parametrizable, es decir que el mismo debe solicitar el tamaño de la matriz antes de generarla. Asuma el caso más simple donde cuenta con tantos procesos como filas tenga la matriz.

ej4a_mpi_producto_matricial.py

```
#!/usr/bin/env python3

import numpy as np
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
root = 0

matrix = None # matriz n*n vacía
bufferReceptor = np.zeros((1, size), dtype=np.int) # vector columna vacío
vectorResultOfProduct = np.zeros((size, 1), dtype=np.int) # vector resultante vacío
vector = None # vector columna vacío


if rank == 0:
    matrix = np.random.randint(0, 10, size=(size, size), dtype=np.int) # matriz n*n
    vector = np.random.randint(0, 10, size=(size, 1), dtype=np.int) # vector columna

# repartimos las filas de la matriz entre los procesos participantes
comm.Scatter([matrix, MPI.INT], [bufferReceptor, MPI.INT])
```

```
# repartimos el vector a multiplicar entre los procesos participantes
vectorToMultiply = comm.bcast(vector, root)

# realizamos los productos correspondientes
vectorResult = np.matmul(bufferReceptor, vectorToMultiply)

# enviamos el vector resultante de cada producto realizado por los demas procesos, al proceso 0
productResult = comm.gather(vectorResult[0], root)

if rank == 0:
    for i in range(len(productResult)):
        vectorResultOfProduct.put(i, productResult[i].item())

    print(f'Hola soy el proceso {rank}, el producto {matrix} · {vector} = {vectorResultOfProduct}')
```

```
lean@lean:~/MEGA/MEGAsync/Lean/UADER/4° Lic. en Sistemas de Informacion/Computacion Avanzada/CA [tps]/CA [tp2]/ej4$ mpiexec -n 4 python3 ej4a_mpi_producto_matricial.py
Hola soy el proceso 0, el producto [[8 3 4 6]
 [0 2 9 5]
 [2 2 0 6]
 [7 7 9 9]] · [[6]
 [5]
 [6]
 [3]] = [[105]
 [ 79]
 [ 40]
 [158]]
```

b) Modifique el código anterior pensando que existen más filas que procesos para tratarlas. Considere dos casos:

• El número de filas de la matriz es múltiplo de la cantidad de procesos.

ej4b1_mpi_producto_matricial.py

```
#!/usr/bin/env python3

import sys
import numpy as np
from mpi4py import MPI

def main(matrixSize, procSize):

    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    root = 0

    matrix = None # matriz n*n vacía
    bufferReceptor = np.zeros((matrixSize // procSize, matrixSize), dtype=np.int) # vector columna vacío
    vectorResultOfProduct = np.zeros((matrixSize, 1), dtype=np.int) # vector resultante vacío
    vector = None # vector columna vacío


    if rank == 0:
        matrix = np.random.randint(0, 10, size=(matrixSize, matrixSize), dtype=np.int) # matriz n*n
        vector = np.random.randint(0, 10, size=(matrixSize, 1), dtype=np.int) # vector columna

    # repartimos las filas de la matriz entre los procesos participantes
    comm.Scatter([matrix, MPI.INT], [bufferReceptor, MPI.INT])

    # repartimos el vector a multiplicar entre los procesos participantes
    vectorToMultiply = comm.bcast(vector, root)

    # realizamos los productos correspondientes
    vectorResult = np.matmul(bufferReceptor, vectorToMultiply)

    # enviamos el vector resultante de cada producto realizado por los demas procesos, al proceso 0
    productResult = comm.gather(vectorResult, root)

    if rank == 0:
        aux = 0
        for list1 in productResult:
            for list2 in list1:
                vectorResultOfProduct.put(aux, list2)
                aux += 1

        print(f'Hola soy el proceso {rank}, el producto {matrix} · {vector} = {vectorResultOfProduct}')


if __name__ == "__main__":
        haveArguments = len(sys.argv) > 1
        if not haveArguments:
            print("Debe ingresar como parametro la cantidad el tamaño de la matriz cuadrada")

        else:
            comm = MPI.COMM_WORLD
            matrizSize = int(sys.argv[1])
            procSize = comm.Get_size()
            if matrizSize % procSize == 0:
                main(matrizSize, procSize)
            else:
```

```
            print("La cantidad de filas de la matriz debe ser multiplo de la cantidad de procesos")
```

```
lean@lean:~/MEGA/MEGAsync/Lean/UADER/4° Lic. en Sistemas de Informacion/Computacion Avanzada/CA [tps]/CA [tp2]/ej4$ mpiexec -n 2 python3 ej4b1_mpi_producto_matricial.py 4
Hola soy el proceso 0, el producto [[4 3 4 2]
 [0 9 5 6]
 [4 0 2 9]
 [0 7 1 3]] · [[9]
 [7]
 [8]
 [6]] = [[101]
 [139]
 [106]
 [ 75]]
```

• El número de filas de la matriz no es múltiplo de la cantidad de procesos.

ej4b2_mpi_producto_matricial.py

```
#!/usr/bin/env python3

import sys
import numpy as np
from mpi4py import MPI
from array import array

def main(matrixSize):
    """
    El inconveniente que se produce en este ejercicio es que al realizar la reparticion de las filas de la matriz en el Scatter,
    se produce un error ya que la cantidad de elementos a enviar es menor que la cantidad de elemtos a recibir, con lo cual la
    parcial encontrada fue agregar filas de 0 a la matriz para igualar la cantidad de elementos a enviar y recibir. Por lo tanto
     la reparticion de datos se realiza correctamente y el Scatter no produce fallos. Al realizar el producto matricial, estas
filas
    de ceros añadidas a la matriz no afectan el mismo.
    """

    comm = MPI.COMM_WORLD
    procSize = comm.Get_size()
    rank = comm.Get_rank()
    root = 0

    # Si la cantidad de filas es multiplo de la cantidad de procesos devuelve el resto, sino devuelve el resto mas uno
    bufferReceptorSize = (matrixSize//procSize) if matrixSize%procSize == 0 else round(matrixSize//procSize)+1

    # La cantidad de filas de 0 que debe añadirse a la matriz
    numberOfRowsToAdd = (procSize * bufferReceptorSize) - matrixSize

    matrix = None # matriz n*n vacía
    bufferReceptor = np.zeros((bufferReceptorSize, matrixSize), dtype=np.int) # vector columna vacío
    vectorResultOfProduct = np.zeros((matrixSize + numberOfRowsToAdd, 1), dtype=np.int) # vector resultante vacío
    vector = None # vector columna vacío


    if rank == 0:
        # Si la cantidad de filas de 0 a añadir es multiplo de la cantidad de procesos,
        # la diferencia sera 0, caso contrario distinto de 0
        if numberOfRowsToAdd != 0:
            matrix = np.random.randint(0, 10, size=(matrixSize, matrixSize), dtype=np.int) # matriz random de n*n
            vector = np.random.randint(0, 10, size=(matrixSize, 1), dtype=np.int) # vector columna random

            vectorOfZeros = np.zeros((numberOfRowsToAdd, matrixSize), dtype=np.int) # vector de ceros a añadir
              matrix = np.append(matrix, np.zeros((numberOfRowsToAdd, matrixSize), dtype=np.int), 0) # añadimos el vector a la
matriz
        else:
            matrix = np.random.randint(0, 10, size=(matrixSize, matrixSize), dtype=np.int) # matriz n*n
            vector = np.random.randint(0, 10, size=(matrixSize, 1), dtype=np.int) # vector columna

    # repartimos las filas de la matriz entre los procesos participantes
    comm.Scatter([matrix, MPI.INT], [bufferReceptor, MPI.INT])

    # repartimos el vector a multiplicar entre los procesos participantes
    vectorToMultiply = comm.bcast(vector, root)

    # realizamos los productos correspondientes
    vectorResult = np.matmul(bufferReceptor, vectorToMultiply)

    # enviamos el vector resultante de cada producto realizado por los demas procesos, al proceso 0
    productResult = comm.gather(vectorResult, root)

    # Refactorizamos la matriz resultante para mostrarla de forma adecuada
    if rank == 0:
        aux = 0
        for list1 in productResult:
            for list2 in list1:
                vectorResultOfProduct.put(aux, list2)
                aux += 1

        print(f'Hola soy el proceso {rank}, el producto {matrix} · {vector} = {vectorResultOfProduct[:-numberOfRowsToAdd]}')
```

```
if __name__ == "__main__":
        haveArguments = len(sys.argv) > 1
        if not haveArguments:
            print("Debe ingresar como parametro la cantidad el tamaño de la matriz cuadrada")

        else:
            matrizSize = int(sys.argv[1])
            main(matrizSize)
```

```
lean@lean:~/MEGA/MEGAsync/Lean/UADER/4° Lic. en Sistemas de Informacion/Computacion Avanzada/CA [tps]/CA [tp2]/ej4$ mpiexec -n 5 python3 ej4b2_mpi_producto_matricial.py 8
Hola soy el proceso 0, el producto [[9 9 3 6 0 9 2 4]
 [8 5 6 6 3 4 5 8]
 [1 8 5 3 4 0 8 7]
 [7 1 4 9 4 6 1 3]
 [3 4 0 5 5 4 8 2]
 [3 7 5 9 5 8 7 6]
 [5 4 3 8 7 1 0 0]
 [4 9 7 6 1 9 9 5]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]] · [[8]
 [0]
 [1]
 [6]
 [6]
 [6]
 [3]
 [1]] = [[175]
 [171]
 [ 86]
 [180]
 [134]
 [188]
 [139]
 [167]]
```

Ejercicio 5: Juego de la Vida.

El juego de la vida es un autómata celular diseñado por el matemático británico John Horton Conway en 1970. Es un juego de cero jugadores, lo que quiere decir que su evolución está determinada por el estado inicial y no necesita ninguna entrada de datos posterior.
El "tablero de juego" es una matriz de N x N formada por "células". Cada célula tiene 8 células vecinas, que son las que están próximas a ella, incluidas las diagonales.
Las células tienen dos estados: están "vivas" o "muertas" (1 representa una casilla con vida y 0 sin vida).
El estado de la matriz evoluciona a lo largo de unidades de tiempo discretas (se podría decir que por turnos). El estado de todas las células se tiene en cuenta para calcular el estado de las mismas al turno siguiente. Todas las células se actualizan simultáneamente.

Las transiciones dependen del número de células vecinas vivas según las siguientes reglas:

• Una célula muerta con exactamente 3 células vecinas vivas "nace" (al turno siguiente estará viva).
• Una célula viva con 2 ó 3 células vecinas vivas sigue viva, en otro caso muere o permanece muerta (por "soledad" o "superpoblación").

a) Analice el problema planteado. ¿Considera que es posible encontrar una solución paralela para el mismo? Justifique su respuesta.

Es posible encontrar una solución paralela para el problema planteado, por ejemplo se podria dividír el cálculo de la cuadrícula entre procesadores como se muestra en la ***Figura 1***. Cada el proceso obtiene la fila de la matriz / el número de filas del procesador y luego realiza el cálculo de la misma. Esta operación es proporcionada por la función scatter(). Los resultados son recopilados por la función gather().
Para que el resultado del cálculo sea válido, los procesadores deben intercambiar información en tiempo real de las celdas en los bordes adyacentes. Para hacer esto, se podría utilizar un campo adicional que representa el subtotal de los bordes de la matriz. Este campo es intercambiado por procesos, utilizando las operaciones no bloqueantes Isend() e Irecv().
El uso de comunicación no bloqueante permite anular el cálculo de la matriz interna con el intercambio de campos. Después de calcular la matriz interna, se completa el cálculo de los bordes, mientras que con la función Wait() se está esperando finalización de la comunicación y se finaliza un turno.

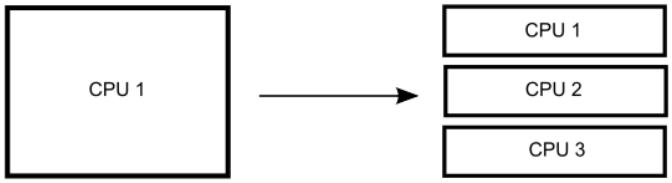


Figura 1: Paralelización del cálculo de la cuadrícula

b) Realice una implementación del juego de a vida de manera secuencial y otra en forma paralela utilizando MPI. Ambas implementaciones serán en pseudocódigo. Considere que el estado inicial es generado aleatoriamente por el programa. Se debe parametrizar el tamaño de la matriz y el número de ciclos que ejecutará el juego.

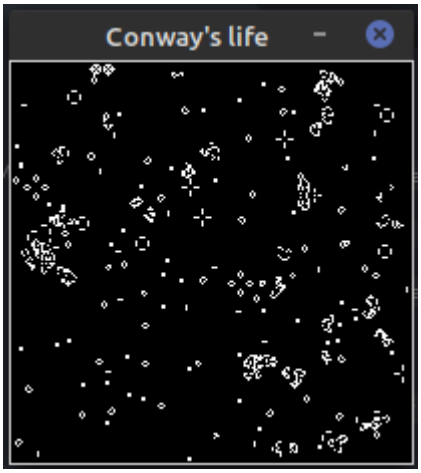
sequential_game_of_life.py

```
#! /usr/bin/env python3

# ==============================================================================
# Name: seq_life.py
# Description: Sequence implementation of Conways Life.
# Author: Milan Munzar (xmunza00@stud.fit.vutbr.cz)

import sys
import numpy
import timeit
```

```
import queue as Queue
import threading
import argparse

from gui_mainWindow import MainWindow

parser = argparse.ArgumentParser(prog='Game of Life Secuencial', description='Computacion Avanzada')
parser.add_argument('-m', help='Tamaño de la matriz', type=int, required=True)
parser.add_argument('-c', help='Numero de ciclos', type=int, required=True)
args = parser.parse_args()

MATRIX_SIZE = args.m
NO_EPOCH = args.c

# ================================================================================
# sequential life
def sequentialLife(m_act, queue):

    ms = [m_act, numpy.zeros((MATRIX_SIZE, MATRIX_SIZE))]
    m_activ_w = numpy.ones((MATRIX_SIZE, MATRIX_SIZE), dtype = bool)
    epoch = 0

    while epoch < NO_EPOCH:

        #print epoch

        # living cells
        index = epoch % 2
        m_r = ms[index]
        m_w = ms[not(index)]
        # active cells
        m_activ_r = m_activ_w
        m_activ_w = numpy.zeros((MATRIX_SIZE, MATRIX_SIZE), dtype = bool)

        # current epoch
        for ro in range(1, MATRIX_SIZE - 1):
            for co in range(1, MATRIX_SIZE - 1):

                if m_activ_r[ro][co] == False:
                    continue

                neighborhood = m_r[ro - 1][co - 1] + m_r[ro - 1][co] + m_r[ro - 1][co + 1] + \
                    m_r[ro][co - 1] + m_r[ro][co + 1] + \
                    m_r[ro + 1][co - 1] + m_r[ro + 1][co] + m_r[ro + 1][co + 1]

                if m_r[ro][co] == 255:
                    if neighborhood == 510 or neighborhood == 765:
                        m_w[ro][co] = 255
                    else:
                        m_w[ro][co] = 0
                        for i in range(-1, 2):
                            for j in range(-1, 2):
                                m_activ_w[ro + i][co + j] = True
                else:
                    if neighborhood == 765:
                        m_w[ro][co] = 255
                        for i in range(-1, 2):
                            for j in range(-1, 2):
                                m_activ_w[ro + i][co + j] = True
                    else:
                        m_w[ro][co] = 0

        epoch = epoch + 1
        queue.put(m_w)
        #! current epoch
    #! life

    return

# ================================================================================
# Main
def main():

    # create initial matrix
    mt_initial = numpy.zeros((MATRIX_SIZE, MATRIX_SIZE))
    for ro in range(1, MATRIX_SIZE - 1):
        for co in range(1, MATRIX_SIZE - 1):
            rand = numpy.random.random_integers(1, 10)
            if rand != 10:
                mt_initial[ro][co] = 0
            else:
                mt_initial[ro][co] = 255

    # gui
    queue = Queue.Queue()
    queue.put(mt_initial)
    thread = threading.Thread(target = MainWindow, args = (queue,))
    thread.setDaemon(True)
    thread.start()
```

```
    sequentialLife(mt_initial, queue)

    sys.exit(0)

if __name__ == "__main__":
    main()

# EOF
# ============================================================================
```

```
lean@lean:~/MEGA/MEGAsync/Lean/UADER/4° Lic. en Sistemas de Informacion/Computacion Avanzada/CA [tps]/CA [tp2]/ej5$ python3 sequential_game_of_life.py -m 200 -c 100
```



paralel_game_of_life.py

```
#! /usr/bin/env python3

# ============================================================================
# Name: prl_life.py
# Description: Sequence implementation of Conways Life.
# Author: Milan Munzar (xmunza00@stud.fit.vutbr.cz)

import sys
import numpy
import queue as Queue
import threading
import argparse

from gui_mainWindow import MainWindow
from mpi4py import MPI


parser = argparse.ArgumentParser(prog='Game of Life Paralelo', description='Computacion Avanzada')
parser.add_argument('-m', help='Tamaño de la matriz', type=int, required=True)
parser.add_argument('-c', help='Numero de ciclos', type=int, required=True)
args = parser.parse_args()

MATRIX_SIZE = args.m
NO_EPOCH = args.c

# ============================================================================
# parallel life
def parallelLife(m_act, queue):


    no_rows = len(m_act)
    ms = [m_act, numpy.zeros((no_rows, MATRIX_SIZE))]
    m_activ_w = numpy.ones((no_rows, MATRIX_SIZE), dtype = bool)

    epoch = 0
    while epoch < NO_EPOCH:

        # receives msgs
        halo_up = numpy.zeros(MATRIX_SIZE, dtype = int)
        halo_down = numpy.zeros(MATRIX_SIZE, dtype = int)

        if rank != 0:
            req2 = comm.Irecv(halo_up, source = rank - 1, tag = 2)
        if rank != size - 1:
            req1 = comm.Irecv(halo_down, source = rank + 1, tag = 1)

        # sends halo region
        index = epoch % 2
        m_r = ms[index]
        m_w = ms[not(index)]
        m_activ_r = m_activ_w
        m_activ_w = numpy.zeros((no_rows, MATRIX_SIZE), dtype = bool)
```

```
        if rank != 0:
            for i in range(1, MATRIX_SIZE - 1):
                halo_up[i] = m_r[0][i - 1] +  m_r[0][i] +  m_r[0][i + 1]

            comm.Isend(halo_up, dest = rank - 1, tag = 1)

        if rank != size - 1:
            for i in range(1, MATRIX_SIZE - 1):
                halo_down[i] = m_r[no_rows - 1][i - 1] +  m_r[no_rows - 1][i] + \
                   m_r[no_rows - 1][i + 1]

            comm.Isend(halo_down, dest = rank + 1, tag = 2)

        # computes inner matrix
        for ro in range(1 , no_rows - 1):
            for co in range(1, MATRIX_SIZE - 1):

                if m_activ_r[ro][co] == False:
                    continue

                m_w[ro][co] = m_r[ro - 1][co - 1] + m_r[ro - 1][co] + m_r[ro - 1][co + 1] + \
                    m_r[ro][co - 1] + m_r[ro][co + 1] + \
                    m_r[ro + 1][co - 1] + m_r[ro + 1][co] + m_r[ro + 1][co + 1]

        # update borders
        if rank != 0:
            req2.Wait()
            for co in range(1, MATRIX_SIZE - 1):

                if m_activ_r[0][co] == False:
                    continue

                m_w[0][co] = halo_up[co] + \
                    m_r[0][co - 1]  + m_r[0][co + 1] + \
                    m_r[1][co - 1] + m_r[1][co] + m_r[1][co + 1]

        if rank != size - 1:
            req1.Wait()
            for co in range(1, MATRIX_SIZE - 1):

                if m_activ_r[no_rows - 1][co] == False:
                    continue

                m_w[no_rows - 1][co] = halo_down[co] + \
                    m_r[no_rows - 1][co - 1]  + m_r[no_rows - 1][co + 1] + \
                    m_r[no_rows - 2][co - 1] + m_r[no_rows - 2][co] + m_r[no_rows - 2][co + 1]

        # compute cells
        for ro in range(0, no_rows):
            for co in range(1, MATRIX_SIZE - 1):

                if m_r[ro][co] == 255:
                    if m_w[ro][co] == 510 or m_w[ro][co] == 765:
                        m_w[ro][co] = 255
                    else:
                        m_w[ro][co] = 0
                        for i in range(0 if ro == 0 else - 1, 1 if ro == no_rows - 1 else 2):
                            for j in range(-1, 2):
                                m_activ_w[ro + i][co + j] = True
                else:
                    if m_w[ro][co] == 765:
                        m_w[ro][co] = 255
                        for i in range(0 if ro == 0 else - 1, 1 if ro == no_rows - 1 else 2):
                            for j in range(-1, 2):
                                m_activ_w[ro + i][co + j] = True
                    else:
                        m_w[ro][co] = 0

        # update GUI
        mt_res = comm.gather(m_w, root = 0)
        if rank == 0:
            mt_res = numpy.vstack(mt_res)
            queue.put(mt_res)

        epoch = epoch + 1
    #! parallel life

    return m_w



# ========================================================================================
# Main
if __name__ == "__main__":

    comm = MPI.COMM_WORLD
    size = comm.Get_size()
```

```
    rank = comm.Get_rank()
    q = Queue.Queue()

    # master
    if rank == 0:

        # create initial matrix
        mt_initial = numpy.zeros((MATRIX_SIZE, MATRIX_SIZE))
        for ro in range(1, MATRIX_SIZE - 1):
            for co in range(1, MATRIX_SIZE - 1):
                rand = numpy.random.random_integers(1, 10)
                if rand != 10:
                    mt_initial[ro][co] = 0
                else:
                    mt_initial[ro][co] = 255

        # gui
        q.put(mt_initial)
        thread = threading.Thread(target = MainWindow, args = (q,))
        thread.setDaemon(True)
        thread.start()

        # create subarrays
        mt_div = []
        for i in range(0, len(mt_initial), int(MATRIX_SIZE/size)):
            mt_div.append(mt_initial[i:i + int(MATRIX_SIZE/size)])

    # slaves
    else:
        mt_div = None

    mt_div = comm.scatter(mt_div, root = 0)
    m_w = parallelLife(mt_div, q)

    sys.exit(0)

# EOF
# ================================================================================
```

```
lean@lean:~/MEGA/MEGAsync/Lean/UADER/4° Lic. en Sistemas de Informacion/Computacion Avanzada/CA [tps]/CA [tp2]/ej5$ mpiexec -n 4 python3 paralel_game_of_life.py -m 200 -c 100
```