Programación Paralela: Práctica 7

13 – 17 mayo 2024

Las respuestas deben entregarse en un único documento (formato pdf). Este documento debe incluir el nombre completo de todos los integrantes del grupo. Si el ejercicio es de implementación, el código fuente se puede entregar en ficheros .py independientes dentro de un archivo comprimido.

Ejercicio 1

La función Allgather es una operación colectiva que recoge datos de todos los procesos en una comunicación y los distribuye a todos los procesos, incluido el proceso que proporcionó los datos originales. En otras palabras, Allgather reúne los datos de todos los procesos y los distribuye de vuelta a todos los procesos.

En términos de sintaxis y parámetros, la función Allgather es similar a Gather, pero con la diferencia clave de que los datos se distribuyen a todos los procesos, no solo al proceso raíz.

Objetos Python

```
datarecv = comm.allgather(datasend)
```

Datos tipo buffer (por ejemplo, arrays numpy)

```
comm.AllGather(datasend, datareceived)
```

La operación allgather podría verse como una operación gather seguida de un broadcast.

En el ejercicio 4, que utiliza la función Scatterv, para saber cuántos datos se reparten a cada procesador, el proceso 0 ha calculado el vector tamanyos y ha enviado esa información al resto de procesadores.

- Modifica el ejercicio 4 para que cada proceso calcule el número de elementos que debe recibir y todos los procesadores tengan esa información.

Ejercicio 2

Hasta el momento, hemos visto la función Scatterv en su versión más sencilla

```
comm.Scatterv([array_global, tamanyos], array_local, root = 0)
```

Sin embargo, puede darse el caso de que no queramos repartir todos los elementos del vector o no queramos que el reparto comience en el elemento 0. Para estos casos se debe utilizar la sintaxis:

```
comm.Scatterv( [array_global, tamanyos,  desplazamiento,
MPI.DOUBLE],array_local, root = 0)
```

En este caso, además de indicar el array a repartir (array_global) y el número de elementos que recibirá cada procesador (tamanyos) hay que especificar otro array (desplazamiento) con la posición inicial de los elementos dentro del vector original y el tipo de los elementos a enviar (MPI.INT o MPI.DOUBLE)

Del mismo modo que gather es la inversa de scatter, MPI nos ofrece Gatherv como inversa de Scatterv. Utilizaremos Gatherv cuando el número de elementos a recopilar de cada proceso sea distinto (es decir no todos los procesos envían el mismo número de datos). Los parámetros para la primitiva Gatherv son similares a los del Scatterv, intercambiando origen y destino:

```
comm.Gatherv(array_local, [array_global, tamanyos], root = 0)
```

```
comm.Gatherv(array_local, [array_global, tamanyos,
desplazamiento, MPI.DOUBLE], root = 0)
```

Para practicar estas funciones vamos a suponer una ejecución con cuatro procesos, P2 reparte datos del vector B (de 16 enteros) de la siguiente manera: a P0: B[3], B[4], B[5]; a P1: B[7], B[8]; a P2: B[10]; y a P3: B[12], B[13], B[14], B[15]. Tras ello, cada proceso suma 100 a los elementos recibidos, y, finalmente, se recopilan los datos finales en P2, en las mismas posiciones iniciales del vector B.

- Completa el programa que aparece a continuación para que realice las funcionalidades descritas: al principio, P2 debe inicializar el vector a B[i] = i, y, al final, imprimir el nuevo vector B.

  El programa debe imprimir:

  B in rank=2 after the calculation

  0 1 2 103 104 105 6 107 108 9 110 11 112 113 114 115

```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
```

```
rank = comm.Get_rank()
size = comm.Get_size()


if rank == 2:
    B=numpy.empty(16, dtype='i')
    for i in range(16):
        B[i] = i
else:
    B=None

# Scattering of B from P2


# Local calculation


# Gathering of B in P2


# Print results
if rank == 2:
    print("\n B in rank=2 after the calculation \n")
    for i in range(16):
        print("%4d" % B[i], end="")
    print("\n\n")
```

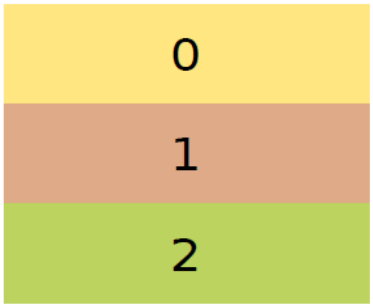
Ejercicio 3

Para practicar el uso de mpi vamos a elegir un problema con alto coste computacional como es el caso del cálculo del número PI, en el que se realizan gran cantidad de cálculos repetitivos. A continuación, tienes un código secuencial para calcular una aproximación del número PI.

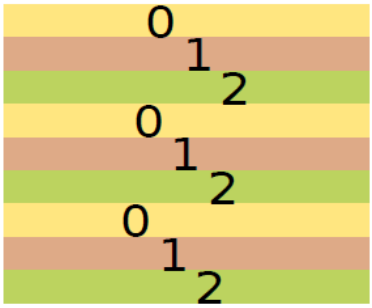
```
import math

def compute_pi(n):
    h = 1.0 / n
    s = 0.0

    for i in range(n):
        x = h * (i + 0.5)
        s += 4.0 / (1.0 + x**2)
    return s * h

n = 100
pi = compute_pi(n)
error = abs(pi - math.pi)

print ("pi is approximately %.16f, error is %.16f" %
(pi, error))
```

- Modifica el ejercicio para que el cálculo se realice en paralelo con p

procesadores. Deberás dividir el número de repeticiones entre los procesadores involucrados. Utiliza una distribución cíclica para conseguir mejores resultados. Finalmente tendrás que combinar los resultados parciales obtenidos por cada procesador para obtener el resultado final de PI.



Block distribution



Cyclic distribution

Ejercicio 4

Modifica el ejercicio anterior para que utilice Numpy. Recuerda que puedes convertir un int o float de Python en un valor escalar de numpy con la siguiente expresión x = numpy.array(x).

Ejercicio 5

Otro ejemplo de algoritmo que se puede paralelizar con MPI es el algoritmo que genera y visualiza el conjunto de Julia (también puede hacerse con el conjunto de Mandelbrot). El conjunto de Julia es un conjunto de puntos en el plano complejo que exhiben ciertas propiedades matemáticas. Estos puntos se calculan iterativamente utilizando una función específica y se asigna un color a cada punto según ciertos criterios. La forma y el patrón del conjunto de Julia dependen de la función utilizada y de los valores iniciales (semillas) que se les da a los cálculos. El conjunto de Julia se visualiza comúnmente como un conjunto de puntos en el plano complejo, donde cada punto se colorea según el número de iteraciones necesarias para que una secuencia de cálculos específica diverja hacia el infinito o se mantenga acotada. Los puntos dentro del conjunto de Julia se suelen colorear de negro o con un color específico, mientras que los puntos fuera del conjunto se colorean de manera diferente según la rapidez con la que divergen.

Aquí hay un ejemplo básico de cómo generar y visualizar el conjunto de Julia en Python utilizando la biblioteca matplotlib. Este código generará y visualizará el conjunto de Julia con una configuración específica de parámetros. Puedes experimentar con diferentes valores de c, así como ajustar otros parámetros como xmin, xmax, ymin, ymax, width, height, y max_iter para

obtener diferentes visualizaciones del conjunto de Julia.

```
import numpy as np
import matplotlib.pyplot as plt

def julia_set(z, c, max_iter):
    for i in range(max_iter):
        if abs(z) > 2:
            return i
        z = z**2 + c
    return max_iter

def  generate_julia_set(xmin,  xmax,  ymin,  ymax,
width, height, c, max_iter):
    x_vals = np.linspace(xmin, xmax, width)
    y_vals = np.linspace(ymin, ymax, height)
    julia = np.zeros((height, width))

    for i in range(width):
        for j in range(height):
            x = x_vals[i]
            y = y_vals[j]
            z = complex(x, y)
            julia[j, i] = julia_set(z, c, max_iter)

    return julia

def  plot_julia_set(julia_set,xmin,  xmax,  ymin,
ymax):
    plt.imshow(julia_set, cmap='hot', extent=(xmin,
xmax, ymin, ymax))
    plt.colorbar()
    plt.title("Conjunto de Julia")
    plt.xlabel("Parte real")
    plt.ylabel("Parte imaginaria")
    plt.show()

if __name__ == "__main__":
    xmin, xmax, ymin, ymax = -2, 2, -2, 2
    width, height = 500, 500
    c = complex(-0.7, 0.27015)
    max_iter = 300

    julia  =  generate_julia_set(xmin,  xmax,  ymin,
ymax, width, height, c, max_iter)
    plot_julia_set(julia, xmin, xmax, ymin, ymax)
```

- Paraleliza el cálculo del conjunto de Julia utilizando MPI, para ello dividiremos el plano complejo en regiones y asignaremos cada región a un proceso. Cada proceso será responsable de calcular una parte del conjunto de Julia y luego los resultados se combinarán para formar la imagen completa.

Nota: Debes hacer modificaciones en las siguientes funciones:

- generate_julia_set para que cada proceso calcule el conjunto de Julia sobre las filas que le correspondan. En este caso utiliza una distribución

por bloques. Cada proceso calculará su fila de inicio y fila de fin y realizará el bucle for sobre esas filas.

- plot_julia_set para recopilar los conjuntos de julia parciales calculados por cada proceso. Además, una vez reunidos todos los conjuntos parciales en el conjunto final, sólo el procesador 0 deberá hacer la visualización de la imagen.