Programación Paralela: Práctica 6

6 – 10 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

Las primitivas de `mpi4py` tienen una versión con la primera letra en mayúsculas que está optimizada para datos de tipo *buffer* de Python. Un ejemplo típico de ese tipo de datos son los vectores y matrices de `numpy`.

Estas rutinas son mucho más eficientes por que transmiten los datos sin convertir ni serializar como ocurre con los tipos generales de Python. El inconveniente es que la variable que recibe el mensaje debe estar inicializada conforme al mensaje, con el mismo tipo y número de elementos. Esta es la forma de trabajar de otros lenguajes utilizados para cálculo como C++ y Fortran.

En el caso de la comunicación punto a punto, la sintaxis de las versiones en mayúsculas difiere un poco de las versiones en minúscula. A continuación, tienes las dos versiones para poder compararlas.

Objetos Python

```
comm.send(obj, dest=0, tag=0)
obj = comm.recv(src=0, tag=0)
```

Datos tipo buffer (por ejemplo, vectores numpy)

```
comm.Send(sendarray, dest=0, tag=0)
comm.Recv(recvarray, src=0, tag=0)
```

Teniendo en cuenta esta información vamos a volver a implementar el ejercicio 2 de la práctica pasada, pero en este caso la lista que enviaremos será un vector de numpy. El programa tendrá que hacer lo siguiente.

1. Inicializa MPI y obtén el rango del proceso actual y el número total de procesos.
2. Si el rango del proceso actual es 0, debe generar un vector de numpy de números enteros aleatorios entre 1 y 100 de longitud n y enviarla al proceso con rango 1.
3. El proceso con rango 1 debe recibir el vector de números del proceso 0 y calcular la suma de todos los números.
4. El proceso con rango 1 debe enviar esta suma de vuelta al proceso 0.
5. El proceso 0 debe recibir la suma de vuelta y mostrarla por pantalla.
6. Los procesos con rango diferente de 0 y 1 deben esperar sin hacer nada.

- Realiza dos versiones, en la primera modificamos el envío y recepción de la lista de datos para que sea un vector de numpy, pero no modificamos el envió de la suma calculada. En la segunda versión modificaremos también el envío de la suma calculada (en este caso tendremos que enviarla como un vector de 1 solo elemento)

Ejercicio 2

De la misma forma que sucede con las operaciones punto a punto (send y recv), la operación colectiva de distribución "broadcast" también presenta 2 variantes, mayúscula y minúscula, en función del tipo de datos a transferir.

Al igual que en el caso anterior la función en mayúscula está optimizada para datos tipo buffer (como son los vectores de Numpy).

Objetos Python

```
datarecv = comm.bcast(datasend, root=0)
```

Datos tipo buffer (por ejemplo, arrays numpy)

```
comm.Bcast(array, root=0)
```

Para comprobar esta mejora vamos a trabajar con un pequeño programa con las siguientes características:

1. Inicializa MPI y obtén el rango del proceso actual y el número total de procesos.
2. Si el rango del proceso actual es 0, el proceso debe generar una lista de números enteros aleatorios entre 1 y 100 de longitud n.
3. El proceso 0 debe transmitir esta lista a todos los otros procesos utilizando la función de broadcast de MPI.
4. Se debe medir el tiempo de ejecución que tarda en realizar la operación de difusión MPI
5. El proceso 0 mostrará por pantalla el tiempo consumido en la operación.

Recuerda que para medir el tiempo debes utilizar la función MPI.Wtime

- Realiza una primera versión del ejercicio utilizando la función bcast (es decir haciendo difusión de una lista de Python)
- Realiza una segunda versión del ejercicio utilizando la función Bcast (es decir haciendo difusión de un vector de Numpy)
- Ejecuta las 2 versiones cambiando el valor de n (100,1000,10000,100000,1000000,10000000…) y compara cómo evolucionan los tiempos. ¿qué ves? ¿cómo lo explicas? Recuerda que al trabajar con tiempos hay que realizar varias mediciones.

Ejercicio 3

Una primitiva muy útil cuando trabajamos con vectores en MPI es scatter donde los elementos de un vector se reparten desde un proceso a todos (incluyendo el origen).

En el siguiente ejemplo el proceso 0 genera una lista de 5 elementos y la reparte entre los procesos.

```
from mpi4py import MPI
import random

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

if rank == 0:
    # Datos a distribuir
    n=5
    data = [random.randint(1, 100) for _ in
range(n)]
    print(f"Proceso {rank} reparte: {data}")
else:
    data = None

# Distribuir los datos
data=comm.scatter(data, root=0)

# Cada proceso imprime su parte de los datos
print(f"Proceso {rank} recibe: {data}")
```

Si ejecutas este código podrás comprobar que solo funciona correctamente si lo lanzamos con 5 procesos. Esto se debe a que al utilizar la función scatter debe haber correspondencia entre el número de datos a distribuir y el número de procesos.

Al igual que con el resto de primitivas que hemos visto hasta hora, scatter también tiene una variante en mayúsculas que optimizada para datos tipo buffer. La sintaxis de esta variante es:

comm.Scatter(data, recbuff, root=0)

- Modifica el ejemplo anterior para que la lista a repartir entre los procesos sea un vector de Numpy y utiliza la función Scatter.
- Modifica el tamaño de n (10 en lugar de 5) y comprueba que pasa al ejecutar el ejemplo y tu versión del ejercicio. Prueba diferentes valores de n (tamaño de la lista) y diferente número de procesos e indica en qué casos funciona cada opción (scatter y Scatter)

Ejercicio 4

Para evitar los problemas de la función Scatter, MPI proporciona la función Scatterv que, en su forma más sencilla, recibe un vector con el número de elementos en cada proceso:

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

El vector tamanyos debe estar definido en todos los procesos e indica el número de elementos del vector que recibe cada proceso.

Mediante Scatterv se puede repartir un número diferente de elementos a cada proceso, lo que soluciona la limitación del ejercicio anterior. El enfoque trivial, repartir los elementos entre p − 1 procesos y asignar el resto al proceso p, no siempre funciona bien. Por ejemplo, si tenemos 101 elementos y los repartimos entre 11 nos quedarán 10 elementos en todos los procesos menos en el último que tendrá uno solo. Este proceso acabará muy pronto y no participará casi en la ejecución del programa, efecto que se conoce como carga no balanceada.

La mejor opción es hacer la división entera entre el número de elementos n y de procesos p, y luego repartir el resto de la división entre los procesos. La siguiente expresión obtiene precisamente esto:

```
tamanyo_local = (n + rank) // p
```

- Modifica el ejercicio anterior para que utilice la función Scatterv y pueda funcionar con distintos valores de n y p. El procesador 0 tendrá que calcular cuantos elementos se van a repartir a cada procesador y enviarlo al resto de procesadores.

Ejercicio 5

La inversa a scatter es la operación gather que combina un vector (o un elemento) de cada proceso en un único vector en un proceso específico.

La función gather (igual que el resto de funciones de mpi) presenta dos variantes: una en mayúsculas y otro en minúsculas. La sintaxis es la siguiente:

Objetos Python

```
datarecv = comm.gather(datasend, root=0)
```

Datos tipo buffer (por ejemplo, arrays numpy)

```
comm.Gather(datasend, datareceived, root=0)
```

- Para comprobar el funcionamiento de la función gather vamos a trabajar con un pequeño programa con las siguientes características:

1. Cada proceso MPI genera un número entero aleatorio (entre 1 y 100).

2. Cada proceso MPI calcula el cuadrado del número aleatorio generado.

3. Todos los procesos MPI envían los cuadrados de los números generados al proceso con rango 0.

4. El proceso con rango 0 suma todos los cuadrados recibidos para obtener la suma total de los cuadrados de los números generados por todos los procesos.

5. El proceso con rango 0 imprime la suma total de los cuadrados.

Ejercicio 6

El objetivo del ejercicio anterior es reducir los datos que se han generado en los diferentes procesos a un solo valor que estará en el proceso 0.

- Optimiza el código del ejercicio anterior teniendo en cuenta el objetivo final del ejercicio.

Ejercicio 7

Como hemos comentado anteriormente con gather cada proceso puede enviar un único dato al proceso destino o más de un dato.

En el siguiente código cada proceso envía una lista con 5 elementos.

```
from mpi4py import MPI
import random
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()


# Cantidad de elementos que cada proceso generará
local_size = 5


# Generar datos aleatorios en cada proceso
local_data  =[random.randint(1,  100)  for  _  in
range(local_size)]
print(f"Proceso {rank} envia: {local_data}")


# Reunir los datos de todos los procesos en el proceso
raíz
global_data=comm.gather(local_data, root=0)


# Imprimir los resultados en el proceso raíz
```

```
if rank == 0:
    print("Global    data    gathered    from    all
processes:")
    print(global_data)
```

- Modifica el ejercicio anterior para que cada proceso envíe un array de numpy y por tanto se tenga que utilizar la función Gather (con mayúsculas). Recuerda que en estas versiones hay que inicializarla variable que recibe el mensaje (en este caso global_data)

- Compara los resultados obtenidos con ambas versiones ¿Qué diferencias observas?