UNIVERSIDAD AUTÓNOMA DE CHIHUAHUA

Facultad de Ingeniería

Ingeniería en Ciencias de la Computación

COMPUTO PARALELO Y DISTRIBUÍDO

2.8 Actividad 1: MPI en Python

Trabajo de: ADRIAN (ADORA) GONZÁLEZ DOMÍNGUEZ *[359834]*

Asesora: JOSE SAUL DE LIRA MIRAMONTES

15 de octubre de 2024

1. ¿Qué es MPI4Py y cómo se relaciona con la computación paralela en Python?

MPI4Py es una biblioteca de Python que proporciona enlaces a la Interfaz de Paso de Mensajes (MPI), una API estandarizada y ampliamente utilizada para la computación paralela y distribuida. Permite que los programas en Python se ejecuten en paralelo en múltiples procesadores o computadoras, utilizando técnicas de paso de mensajes para la comunicación entre diferentes procesos. MPI4Py facilita la implementación de algoritmos paralelos y la gestión del intercambio de datos entre procesos, ofreciendo una forma de aprovechar múltiples núcleos y sistemas distribuidos para tareas de computación de alto rendimiento (HPC).

2. ¿Cómo se inicializa y finaliza un entorno MPI utilizando MPI4Py?

Para inicializar y finalizar el entorno MPI en MPI4Py, se utilizan las siguientes funciones:

- MPI.Init(): Esta función inicializa el entorno MPI y debe ser llamada antes de cualquier otra función relacionada con MPI.
- MPI.Finalize(): Esta función finaliza el entorno MPI y debe llamarse después de que todas las tareas MPI hayan sido completadas, señalando el final de las operaciones MPI.

Sin embargo, en MPI4Py, no es estrictamente necesario llamar explícitamente a `MPI.Init()` y `MPI.Finalize()`, ya que la biblioteca las gestiona automáticamente.

3. Explica la diferencia entre la comunicación bloqueante y no bloqueante en MPI4Py.

En MPI4Py, las operaciones de comunicación bloqueante esperan a que la comunicación se complete antes de que el programa continúe ejecutándose, mientras que la comunicación no bloqueante permite que el programa continúe ejecutándose mientras la comunicación ocurre en segundo plano.

Comunicación bloqueante: Funciones como `Send` y `Recv` son ejemplos de comunicación bloqueante, donde un proceso espera hasta que el mensaje sea completamente enviado o recibido.

- Comunicación no bloqueante: Funciones como `Isend` e `Irecv` inician la comunicación pero regresan inmediatamente, permitiendo que el programa realice otras tareas mientras espera que la comunicación se complete. Es necesario usar `Request.Wait()` o `Request.Test()` para asegurar que la operación se complete.

4. ¿Cómo puedes determinar el rango de un proceso y el número total de procesos en un programa MPI utilizando MPI4Py?

- Rango: El rango de un proceso se refiere a su ID único en el comunicador, que a menudo se utiliza para identificar procesos individuales en un programa paralelo.
- Tamaño: El número total de procesos involucrados en la comunicación.

5. ¿Cuáles son las operaciones básicas de comunicación punto a punto en MPI4Py y cómo se usan?

Las operaciones básicas de comunicación punto a punto en MPI4Py incluyen:
- Send: Envía datos de un proceso a otro.
- Recv: Recibe datos de otro proceso.

6. Describe el propósito y uso de las operaciones de comunicación colectiva en MPI4Py.

Las operaciones de comunicación colectiva involucran a todos los procesos dentro de un comunicador, a diferencia de las operaciones punto a punto que solo involucran a dos procesos. Algunas operaciones colectivas comunes incluyen:
- Bcast: Envía datos desde un proceso a todos los demás procesos.
- Scatter: Distribuye partes de un array desde un proceso a todos los procesos.
- Gather: Recoge partes de un array de todos los procesos a un solo proceso.
- Reduce: Combina valores de todos los procesos usando una operación (ej. suma, máximo) y envía el resultado a un proceso.

7. ¿Cómo se crean y usan tipos de datos personalizados en MPI4Py?

Los tipos de datos personalizados permiten la comunicación de objetos complejos de Python, como estructuras, arrays o cualquier otro tipo de datos definido por el usuario.

En MPI4Py, puedes crear tipos de datos personalizados usando `MPI.Datatype.Create_struct()` y funciones relacionadas.

8. Explica el concepto de comunicadores en MPI y cómo se implementan en MPI4Py.

En MPI, un **comunicador** es un grupo de procesos que pueden comunicarse entre sí. El comunicador por defecto, `MPI.COMM_WORLD`, incluye a todos los procesos en un programa MPI. Puedes crear nuevos comunicadores para dividir los procesos en subgrupos, lo que permite un control más detallado sobre la comunicación.

En MPI4Py, los comunicadores se implementan como objetos. Por ejemplo, `MPI.COMM_WORLD` es el comunicador global que incluye a todos los procesos.

9. ¿Cuáles son las ventajas y los desafíos de usar MPI4Py para la computación distribuida en comparación con otras bibliotecas de paralelización en Python?

Ventajas

- **Escalabilidad**: MPI4Py es altamente escalable y puede ejecutarse en muchos nodos en entornos de computación distribuida, lo que lo hace adecuado para tareas de HPC.
- **Flexibilidad**: MPI4Py soporta tanto la comunicación punto a punto como la colectiva, ofreciendo un control preciso sobre el intercambio de datos.
- **Interoperabilidad**: MPI4Py está construido sobre el estándar MPI, lo que permite la integración con otras aplicaciones y bibliotecas basadas en MPI.

Desafíos

- **Complejidad**: MPI4Py es más complejo de usar que otras bibliotecas, especialmente para principiantes.
- **Gestión manual de la comunicación**: El usuario es responsable de gestionar la comunicación entre procesos, lo que puede ser propenso a errores.
- **Sobrecarga**: La sobrecarga de rendimiento del paso de mensajes puede ser significativa para paralelismo de grano fino, especialmente en sistemas pequeños o de memoria compartida.

10. ¿Cómo se puede implementar un algoritmo simple de multiplicación de matrices en paralelo usando MPI4Py?

Una multiplicación de matrices en paralelo se puede implementar distribuyendo las filas de una matriz entre los procesos y realizando la multiplicación local con las columnas de la segunda matriz.

```
import numpy as np
from mpi4py import MPI

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

# Definir el tamaño de la matriz
N = 4

# Inicializar matrices
if rank == 0:
    A = np.random.rand(N, N)
    B = np.random.rand(N, N)
else:
    A = np.zeros((N, N))
    B = np.zeros((N, N))

# Transmitir la matriz B a todos los procesos
comm.Bcast(B, root=0)

# Distribuir las filas de A entre todos los procesos
filas_por_proceso = N // size
A_local = np.zeros((filas_por_proceso, N))
comm.Scatter(A, A_local, root=0)

# Realizar la multiplicación local de matrices
C_local = np.dot(A_local, B)

# Recoger la matriz resultante
C = None
```

```
if rank == 0:
    C = np.zeros((N, N))
comm.Gather(C_local, C, root=0)

if rank == 0:
    print("Matriz resultante C:\n", C)
```