DR. GABRIEL LEON PAREDES, PhD

gleon@ups.edu.ec

www.linkedin.com/in/gabrielleonp

Cloud Computing, Smart Cities & High-Performance Computing

Cuenca, Ecuador

COMPUTO PARALELO

Message Passing

Intercambio de Mensajes

- El objetivo principal del MPI es establecer un estándar eficiente, flexible y portátil para la comunicación de intercambio de mensajes.
- Principalmente, mostraremos las funciones de la biblioteca que incluyen primitivas de comunicación síncronas y asíncronas, como (enviar / recibir) y (difusión / todos a todos), las operaciones de combinar los resultados parciales del cálculo (reunir / reducir), y finalmente, las primitivas de sincronización entre procesos (barreras).

Requerimientos técnicos

- Necesitará las bibliotecas *mpich* y mpi4py.
- La biblioteca *mpich* es una implementación portátil de MPI. Es un software gratuito y está disponible para varias versiones de Unix (incluidos Linux y macOS)
  - http://www.mpich.org/static/downloads/1.4.1p1/
- Microsoft Windows
  - https://www.microsoft.com/en-us/download/details.aspx?id=100593
  - Además, asegúrese de elegir entre las versiones de 32 bits o 64 bits para obtener la correcta para su máquina.
- El módulo mpi4py Python proporciona enlaces de Python para el estándar MPI (https://www.mpi-forum.org). Se implementa por encima de la especificación MPI-1/2/3 y expone una API que se basa en MPI-2 C++.
  - El procedimiento de instalación de mpi4py es:
    - pip install mpi4py
  - Los usuarios de Anaconda deben escribir lo siguiente:
    - conda install mpi4py

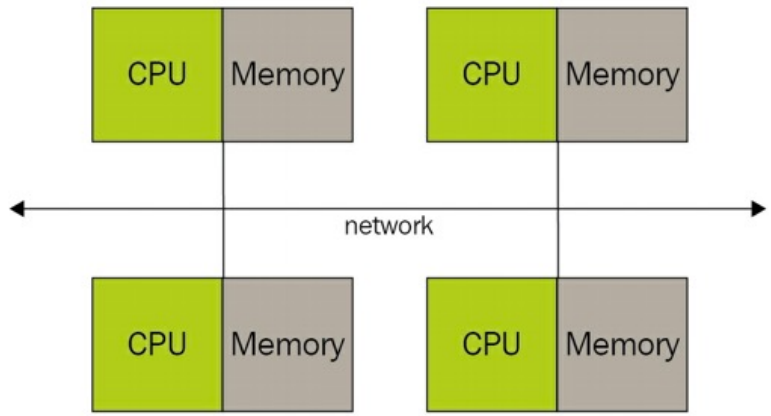
Requerimientos técnicos

- Los ejemplos en este capítulo, usan mpi4py instalado usando pip y son ejecutados:
  - mpiexec –n x python mpi4y_script_name.py
- El comando *mpiexec* es la forma típica de iniciar trabajos paralelos, en donde, **x** es el número total de procesos a utilizar, mientras que **mpi4py_script_name.py** es el nombre del script que se ejecutará.

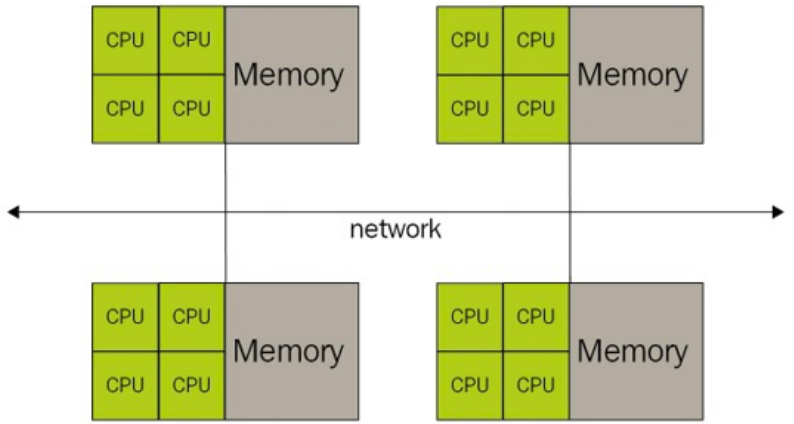
http://manpages.ubuntu.com/manpages/xenial/man1/mpiexec.mpich.1.html

Entendiendo la estructura MPI

- Originalmente, MPI fue diseñado para arquitecturas de memoria distribuida, que comenzó a crecer en popularidad hace 20 años.
- Con el tiempo, los sistemas de memoria distribuida comenzaron a combinarse entre sí, creando sistemas híbridos con memoria distribuida / compartida:
- Hoy, MPI se ejecuta en memoria distribuida, memoria compartida y sistemas híbridos. Sin embargo, el modelo de programación sigue siendo el de la memoria distribuida, aunque la verdadera arquitectura en la que se realiza el cálculo puede ser diferente



The distributed memory architecture schema



The hybrid system architecture schema

Ventajas

- Estandarización: es compatible con todas las plataformas de computación de alto rendimiento (HPC).
- Portabilidad: los cambios aplicados al código fuente son mínimos, lo cual es útil si decide utilizar la aplicación en una plataforma diferente que también sea compatible con el mismo estándar.
- Rendimiento: los fabricantes pueden crear implementaciones optimizadas para un tipo específico de hardware y obtener un mejor rendimiento.
- Funcionalidad: se definen más de 440 rutinas en MPI-3, pero muchos programas paralelos se pueden escribir utilizando menos de 10 rutinas.

Usando el módulo mpi4py

- Según el modelo de ejecución de MPI, nuestra aplicación consta de N (5 en este ejemplo) procesos autónomos, cada uno con su propia memoria local capaz de comunicar datos a través del intercambio de mensajes.
- El comunicador define un grupo de procesos que pueden comunicarse entre sí. El trabajo MPI_COMM_WORLD utilizado aquí es el comunicador predeterminado e incluye todos los procesos.
- La identificación de un proceso se basa en rangos. A cada proceso se le asigna un rango para cada comunicador al que pertenece. El rango es un entero que se asigna, que comienza desde cero e identifica cada proceso individual en el contexto de un comunicador específico.
- La práctica común es definir el proceso con un rango global de 0 como el proceso maestro. A través del rango, el desarrollador puede especificar cuál es el proceso de envío y cuáles son los procesos del destinatario.
- Cabe señalar que, solo con fines ilustrativos, la salida estándar no siempre se ordenará

Para ejecutar el código, escriba la siguiente línea de comando:

mpiexec –n 5 python helloworld_MPI.py

```
#hello.py
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
print ("hello world from process ", rank)
```

```
hello world from process  1
hello world from process  0
hello world from process  2
hello world from process  3
hello world from process  4
```

Usando el módulo mpi4py

- MPI pertenece a la técnica de programación Single Program Multiple Data (SPMD).
- SPMD es una técnica de programación en la cual un solo programa es ejecutado por varios procesos al mismo tiempo, pero cada proceso puede operar con datos diferentes. Al mismo tiempo, los procesos pueden ejecutar la misma instrucción y diferentes instrucciones.
- El programa contendrá instrucciones apropiadas que permiten la ejecución de solo partes del código y / o operar en un subconjunto de datos. Esto se puede implementar usando diferentes modelos de programación, y todos los ejecutables comienzan al mismo tiempo.
- La referencia completa a la librería mpi4py puede ser encontrada en https://mpi4py.readthedocs.io/en/stable/

Comunicación punto a punto

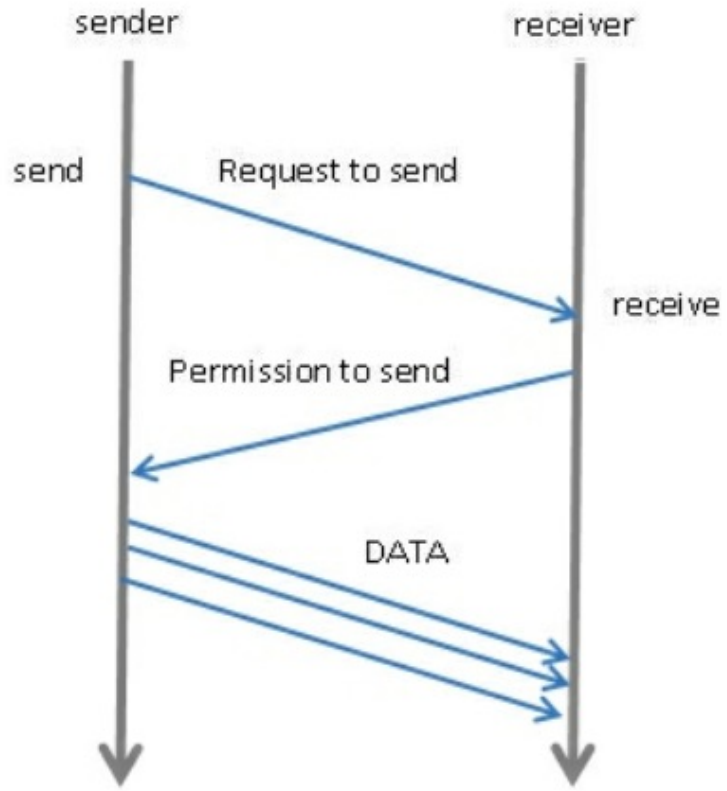
- Las operaciones punto a punto consisten en el intercambio de mensajes entre dos procesos. En un mundo perfecto, cada operación de envío estaría perfectamente sincronizada con la operación de recepción respectiva.
- Obviamente, este no es el caso, y la implementación de MPI debe ser capaz de preservar los datos enviados cuando los procesos del remitente y el destinatario no están sincronizados.
- Por lo general, esto ocurre usando un búfer, que es transparente para el desarrollador y está completamente administrado por la biblioteca mpi4py.
- El módulo mpi4py permite la comunicación punto a punto a través de dos funciones:
    - comm.Send (data, process_destination): esta función envía datos al proceso de destino identificado por su rango en el grupo comunicador.
    - comm.Recv (process_source): esta función recibe datos del proceso de abastecimiento, que también se identifica por su rango en el grupo comunicador.
    - El parámetro Comm, que es la abreviatura de comunicador, define el grupo de procesos que pueden comunicarse mediante el paso de mensajes usando comm = MPI.COMM_WORLD.

```python
from mpi4py import MPI

comm=MPI.COMM_WORLD
rank = comm.rank
print("my rank is : " , rank)

if rank==0:
    data= 10000000
    destination_process = 4
    comm.send(data,dest=destination_process)
    print ("sending data %s " %data +\
           "to process %d" %destination_process)

if rank==1:
    destination_process = 8
    data= "hello"
    comm.send(data,dest=destination_process)
    print ("sending data %s :" %data + \
           "to process %d" %destination_process)


if rank==4:
    data=comm.recv(source=0)
    print ("data received is = %s" %data)


if rank==8:
    data1=comm.recv(source=1)
    print ("data1 received is = %s" %data1)
```

Comunicación punto a punto

- Ejecutamos el ejemplo con un número total de procesos igual a 9. Entonces, en el grupo comunicador de comunicaciones, tenemos nueve tareas que pueden comunicarse entre sí.
- Además, para identificar una tarea o procesos dentro del grupo, utilizamos su valor de rango.
- Tenemos dos procesos emisores y dos procesos receptores.
  - El proceso de rango igual a 0 envía datos numéricos al proceso receptor de rango igual a 4. También notamos que la declaración *comm.recv* debe contener, como argumento, el rango del proceso del remitente.
  - Para los otros procesos de emisor y receptor (el proceso de rango igual a 1 y el proceso de rango igual a 8, respectivamente), la situación es la misma, la única diferencia es el tipo de datos.

Comunicación punto a punto



The send/receive transmission protocol

- El diagrama resume el protocolo de comunicación punto a punto en mpi4py.
- Se describe un proceso de dos pasos, que consiste en enviar algunos DATOS desde una tarea (remitente), y otra tarea (receptor) que recibe estos datos.
- La tarea de envío debe especificar los datos que se enviarán y su destino (el proceso del receptor), mientras que la tarea de recepción debe especificar la fuente del mensaje que se recibirá.
- Para ejecutar el código, escriba la siguiente línea de comando:
  - mpiexec –n 9 python pointToPointCommunication.py

```python
from mpi4py import MPI

comm=MPI.COMM_WORLD
rank = comm.rank
print("my rank is %i" % (rank))

if rank==1:
    data_send= "a"
    destination_process = 5
    source_process = 5

    data_received=comm.recv(source=source_process)
    comm.send(data_send,dest=destination_process)

    print ("sending data %s " %data_send + \
           "to process %d" %destination_process)
    print ("data received is = %s" %data_received)



if rank==5:
    data_send= "b"
    destination_process = 1
    source_process = 1

    comm.send(data_send,dest=destination_process)
    data_received=comm.recv(source=source_process)


    print ("sending data %s :" %data_send + \
           "to process %d" %destination_process)
    print ("data received is = %s" %data_received)
```

Evitando problemas de bloqueo

- Un problema común que enfrentamos es un punto muerto. Esta es una situación en la que dos (o más) procesos se bloquean entre sí y están en estado de espera hasta que el otro realiza una determinada acción que permita continuar al otro proceso y viceversa.
- El módulo mpi4py no proporciona ninguna funcionalidad específica para resolver el problema del punto muerto, pero hay algunas medidas que el desarrollador debe seguir para evitar el problema del punto muerto.
- Ambos procesos se preparan para recibir un mensaje del otro y quedan estancados allí. Esto sucede debido a que la función comm.recv() MPI y el comm.send() MPI los bloquean.
- Esto significa que el proceso de llamada espera su finalización.
    - En cuanto al comm.send() MPI, la finalización se produce cuando se han enviado los datos y puede sobrescribirse sin modificar el mensaje.
    - La finalización del comm.recv () MPI ocurre cuando los datos han sido recibidos y pueden usarse.

```
if rank==1:
    data_send= "a"
    destination_process = 5
    source_process = 5
    comm.send(data_send,dest=destination_process)
    data_received=comm.recv(source=source_process)

   print ("sending data %s " %data_send + \
           "to process %d" %destination_process)
    print ("data received is = %s" %data_received)

if rank==5:
    data_send= "b"
    destination_process = 1
    source_process = 1
    data_received=comm.recv(source=source_process)
    comm.send(data_send,dest=destination_process)

   print ("sending data %s :" %data_send + \
           "to process %d" %destination_process)
    print ("data received is = %s" %data_received)
```

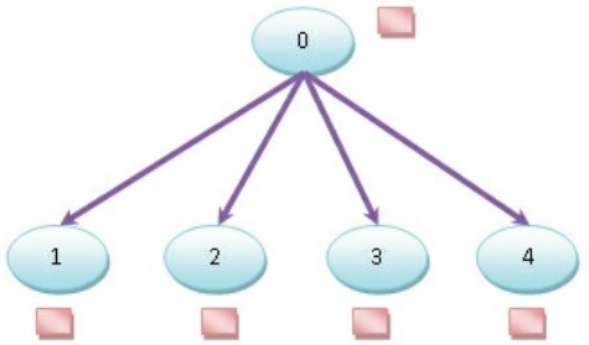
```
if rank==1:
    data_send= "a"
    destination_process = 5
    source_process = 5
    comm.send(data_send,dest=destination_process)
    data_received=comm.recv(source=source_process)

if rank==5:
    data_send= "b"
    destination_process = 1
    source_process = 1
    comm.send(data_send,dest=destination_process)
    data_received=comm.recv(source=source_process)
```

Evitando problemas de bloqueo

- La primera idea es invertir el comm.recv() con el comm.send(),
- La solución que nos permite evitar puntos muertos es utilizar las funciones de envío y recepción de forma asimétricas

Comunicación colectiva usando un broadcast

- Durante el desarrollo del código paralelo, a menudo nos encontramos en una situación en la que debemos compartir, entre múltiples procesos, el valor de una determinada variable en tiempo de ejecución.
- Para resolver este tipo de situaciones, se utilizan árboles de comunicación (por ejemplo, el proceso 0 envía datos a los procesos 1 y 2, que, respectivamente, se encargarán de enviarlos a los procesos 3, 4, 5, 6, etc.).
- En cambio, las bibliotecas MPI proporcionan funciones que son ideales para el intercambio de información o el uso de múltiples procesos que están claramente optimizados para la máquina en la que se realizan.
- Un método de comunicación que involucra todos los procesos que pertenecen a un comunicador se llama comunicación colectiva.
- La comunicación colectiva generalmente involucra más de dos procesos. Sin embargo, en lugar de esto, llamaremos la transmisión de comunicación colectiva, en la que un solo proceso envía los mismos datos a cualquier otro proceso.



Broadcasting data from process 0 to processes 1, 2, 3, and 4

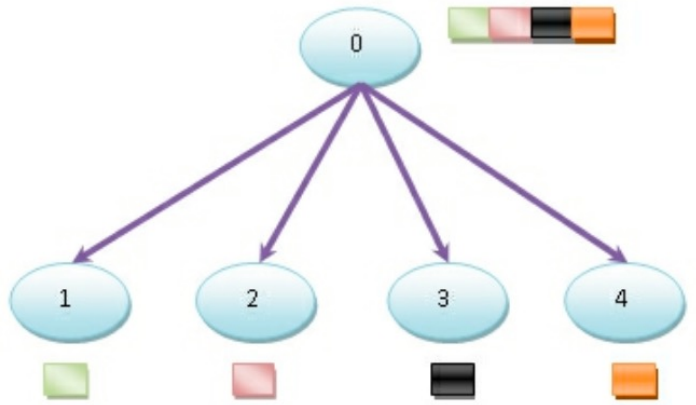
Comunicación colectiva usando un broadcast

- Las funcionalidades de transmisión mpi4py se ofrecen mediante el siguiente método:
  - buf = comm.bcast (data_to_share, rank_of_root_process)
- Esta función envía la información contenida en la raíz del proceso del mensaje a cualquier otro proceso que pertenezca al comunicador de comunicaciones.
- mpiexec –n python 10 broadcast.py

```
1  from mpi4py import MPI
2
3  comm = MPI.COMM_WORLD
4  rank = comm.Get_rank()
5
6  if rank == 0:
7     variable_to_share = 100
8
9  else:
10    variable_to_share = None
11
12 variable_to_share = comm.bcast(variable_to_share, root=0)
13 print("process = %d" %rank + " variable shared  = %d " %variable_to_share)
```

Comunicación colectiva usando scatter

- La funcionalidad de dispersión es muy similar a una difusión de dispersión, pero con una diferencia importante: mientras que comm.bcast envía los mismos datos a todos los procesos de escucha, comm.scatter puede enviar fragmentos de datos en una matriz a diferentes procesos.
- La función comm.scatter toma los elementos de la matriz y los distribuye a los procesos de acuerdo con su rango, para lo cual el primer elemento se enviará al proceso 0, el segundo elemento al proceso 1, y así sucesivamente.
- Las funcionalidades de scatter en mpi4py se ofrecen mediante el siguiente método:
    - ×□□W !"#$%&'()*+,-./0123456789:;□ÿ□□<=□>□□?@ABCDEFGHI



Scattering data from process 0 to processes 1, 2, 3, and 4

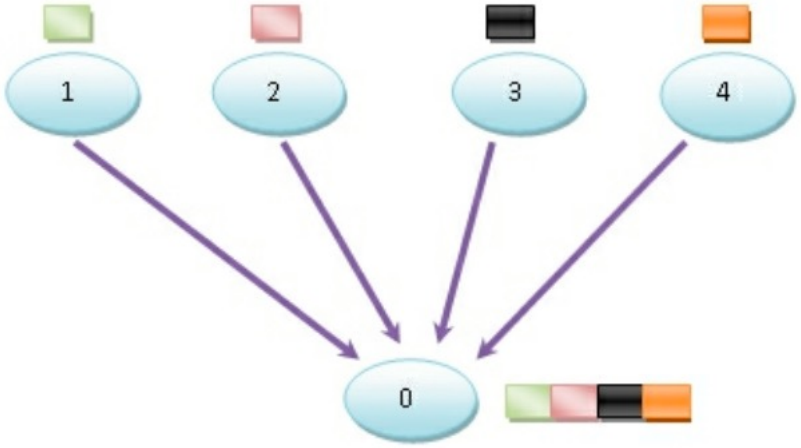
Comunicación colectiva usando scatter

- Una de las restricciones de la funcionalidad *comm.scatter* es que puede dispersar tantos elementos como los procesadores que especifique en la instrucción de ejecución.
- De hecho, si intenta dispersar más elementos que los procesadores especificados (tres, en este ejemplo), obtendrá un error.

```
1  from mpi4py import MPI
2
3  comm = MPI.COMM_WORLD
4  rank = comm.Get_rank()
5
6  if rank == 0:
7     array_to_share = [1, 2, 3, 4 ,5 ,6 ,7, 8 ,9 ,10]
8
9  else:
10    array_to_share = None
11
12 recvbuf = comm.scatter(array_to_share, root=0)
13 print("process = %d" % rank + " variable shared  = %d " % recvbuf )
```

Comunicación colectiva usando gather

- La función gather realiza el inverso de la función scatter. En este caso, todos los procesos envían datos a un proceso raíz que recopila los datos recibidos.
- La función de gather, que se implementa en mpi4py, es la siguiente:
  - recvbuf = comm.gather (sendbuf, rank_of_root_process)
- Aquí, sendbuf son los datos que se envían, y rank_of_root_process representa el procesamiento del receptor de todos los datos.



Gathering data from processes 1, 2, 3, and 4

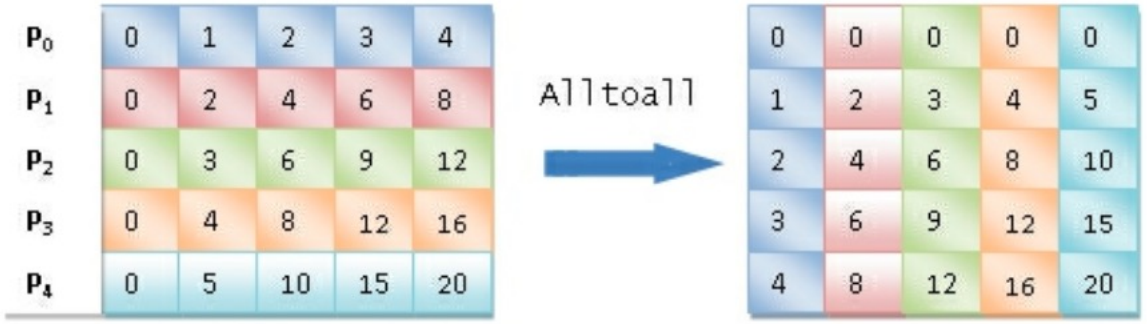
Comunicación colectiva usando gather

- El proceso raíz de 0 recibe datos de los otros cuatro procesos, como se representa en el diagrama anterior.
- Establecemos n = 5 procesos que envían sus datos:
- Si el rango del proceso es 0, entonces los datos se recopilan en un arreglo.
- La recopilación de datos viene dada, en cambio, por la siguiente función:
  - data = comm.gather (data, root = 0)
- Ejecutamos el código configurando el grupo de procesos igual a 5:
  - mpiexec -n 5 python collect.py

```
1  from mpi4py import MPI
2
3  comm = MPI.COMM_WORLD
4  size = comm.Get_size()
5  rank = comm.Get_rank()
6  data = (rank+1)**2
7
8  data = comm.gather(data, root=0)
9  if rank == 0:
10    print ("rank = %s " %rank +\
11           "...receiving data to other process")
12    for i in range(1,size):
13
14       value = data[i]
15       print(" process %s receiving %s from process %s"\
16             %(rank , value , i))
```

Comunicación colectiva usando AllToall

- La comunicación colectiva Alltoall combina las funcionalidades de scatter y gather
- Consideraremos un grupo de procesos, donde cada proceso envía y recibe una matriz de datos numéricos de los otros procesos definidos en el grupo:
- El método comm.alltoall toma el iésimo objeto del argumento sendbuf de la tarea j y lo copia en el objeto jth del argumento recvbuf de la tarea i..



The Alltoall collective communication

Comunicación colectiva usando AllToall

```
C:\>mpiexec -n 5 python alltoall.py
process 0 sending [0 1 2 3 4] receiving [0 0 0 0 0]
process 1 sending [0 2 4 6 8] receiving [1 2 3 4 5]
process 2 sending [ 0 3 6 9 12] receiving [ 2 4 6 8 10]
process 3 sending [ 0 4 8 12 16] receiving [ 3 6 9 12 15]
process 4 sending [ 0 5 10 15 20] receiving [ 4 8 12 16 20]
```

```python
from mpi4py import MPI
import numpy

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


senddata = (rank+1)*numpy.arange(size,dtype=int)
recvdata = numpy.empty(size,dtype=int)
comm.Alltoall(senddata,recvdata)


print(" process %s sending %s receiving %s"\
      %(rank , senddata , recvdata))
```

Operación reduction

- Similar a *comm.gather*, *comm.reduce* toma un arreglo de elementos de entrada en cada proceso y devuelve un arreglo de elementos de salida al proceso raíz. Los elementos de salida contienen el resultado reducido.
- En mpi4py, definimos la operación de reducción a través de la siguiente declaración:
  - comm.Reduce (sendbuf, recvbuf, rank_of_root_process, op = type_of_reduction_operation)
- Debemos tener en cuenta que la diferencia con la declaración *comm.gather* reside en el parámetro *op*, que es la operación que desea aplicar a sus datos, y el módulo mpi4py contiene un conjunto de operaciones de reducción que se pueden utilizar.

Operación reduction

- Ahora, veremos cómo implementar la suma de una matriz de elementos con la operación de reducción MPI.SUM utilizando la funcionalidad de reducción. Cada proceso manipulará una matriz de tamaño 10.
- Para realizar la suma, usamos la declaración comm.Reduce.
- Además, identificamos con el rango cero, que es el proceso raíz que contendrá recvbuf, que representa el resultado final del cálculo:
  - comm.Reduce (senddata, recvdata, root = 0, op = MPI.SUM)
- Tiene sentido ejecutar el código con un grupo comunicador de 10 procesos, ya que este es el tamaño de la matriz manipulada.
  - mpiexec -n 10 python reduce.py

```
1  import numpy
2  from mpi4py import MPI
3  comm = MPI.COMM_WORLD
4  size = comm.size
5  rank = comm.rank
6
7
8  array_size = 10
9  recvdata = numpy.zeros(array_size,dtype=numpy.int)
10 senddata = (rank+1)*numpy.arange(array_size,dtype=numpy.int)
11
12 print(" process %s sending %s " %(rank , senddata))
13
14
15 comm.Reduce(senddata,recvdata,root=0,op=MPI.SUM)
16 print ('on task',rank,'after Reduce:    data = ',recvdata)
```

Operación reduction

- Tenga en cuenta que con la opción op = MPI.SUM, aplica la operación de suma a todos los elementos del arreglo de columnas.
- Las operaciones de envío son las siguientes:
  - El proceso P0 envía la matriz de datos [0 1 2].
  - El proceso P1 envía la matriz de datos [0 2 4].
  - El proceso P2 envía la matriz de datos [0 3 6].
- La operación de reducción suma los i-ésimos elementos de cada tarea y luego coloca el resultado en el i-ésimo elemento de la matriz en el proceso raíz P0.

Reduction in collective communication

Operación reduction

- Algunas de las operaciones de reducción definidas por MPI son las siguientes:
  - MPI.MAX: Esto devuelve el elemento máximo.
  - MPI.MIN: Esto devuelve el elemento mínimo.
  - MPI.SUM: Esto resume los elementos.
  - MPI.PROD: esto multiplica todos los elementos.
  - MPI.LAND: realiza la operación lógica AND a través de los elementos.
  - MPI.MAXLOC: devuelve el valor máximo y el rango del proceso que lo posee.
  - MPI.MINLOC: devuelve el valor mínimo y el rango del proceso que lo posee.

Referencias

- Zaccone, Giancarlo. Python Parallel Programming Cookbook: Over 70 recipes to solve challenges in multithreading and distributed system with Python 3, 2nd Edition . Packt Publishing.