**LAB : 2**

1. **Importing Necessary Libraries**:

from mpi4py import MPI import numpy as np

* **mpi4py**: This library provides Python bindings for the Message Passing Interface (MPI), allowing Python programs to interact with MPI-based parallel computing.
* **numpy**: This library is used for numerical computing in Python, especially for handling arrays and mathematical operations efficiently.

1. **Initializing MPI Communication**:

comm = MPI.COMM\_WORLD rank = comm.rank

* **MPI.COMM\_WORLD**: This represents the communicator for all processes in the MPI environment.
* **comm.rank**: This retrieves the rank of the current process within the communicator **MPI.COMM\_WORLD**. Each process is assigned a unique rank, starting from 0.

1. **Getting the Total Number of Processors**:

num\_procs = comm.Get\_size()

* **comm.Get\_size()**: This retrieves the total number of processes in the communicator **MPI.COMM\_WORLD**.

1. **Initializing Send Buffer on Rank 0**:

send\_buf = np.array([])

* **send\_buf**: This is an empty numpy array. It will be used to store the array chunks to be sent to different processes.

1. **Setting Up Data on Rank 0**:

if rank == 0: N = 100 # Total number of elements num\_chunks = num\_procs # Number of chunks (equal to number of processors) arr = np.arange(N) # Generate an array of N elements chunk\_size = N // num\_chunks # Calculate the chunk size send\_buf = np.array\_split(arr, num\_chunks) # Split the array into chunks

* On rank 0, the total number of elements **N** is defined, an array **arr** of **N** elements is generated, and it is split into **num\_chunks** chunks. Each chunk will be sent to different processes for computation.

1. **Initializing Receive Buffer**:

recv\_buf = np.empty(1, dtype=int) # Create an empty array to receive data

* **recv\_buf**: This is an empty numpy array. It will be used to receive data from other processes.

1. **Scattering Data**:

send\_buf = comm.scatter(send\_buf, root=0) # Scatter the chunks to all processes

* **comm.scatter()**: This function scatters the chunks of the array **send\_buf** from rank 0 to all processes in **MPI.COMM\_WORLD**. Each process receives one of the chunks.

1. **Calculating Local Sum**:

local\_sum = np.sum(send\_buf) # Calculate the local sum

* **np.sum()**: This calculates the sum of elements in the chunk **send\_buf** received by each process.

1. **Printing Local Sum**:

print("Local sum at rank {0}: {1}".format(rank, local\_sum)) # Print the local sum

* This prints the local sum calculated by each process.

1. **Reducing Local Sums to Obtain Global Sum**:

recv\_buf = comm.reduce(local\_sum, op=MPI.SUM, root=0) # Reduce local sums to obtain global sum

* **comm.reduce()**: This function reduces the local sums (**local\_sum**) from all processes using the specified reduction operation (**MPI.SUM** in this case) and gathers the result in **recv\_buf** at the root process (rank 0).

1. **Printing Global Sum on Rank 0**:

if rank == 0: global\_sum = recv\_buf # Global sum is received by the root process print("Global sum: " + str(global\_sum)) # Print the global sum

* On rank 0, the global sum calculated from the reduction operation is printed.

1. **Initialization**:

if rank == 0:

* + This conditional statement checks if the current process has a rank of 0. In MPI, rank 0 is often designated as the "root" process responsible for coordinating certain tasks.

1. **Defining Parameters**:

N = 100 # Total number of elements num\_chunks = num\_procs # Number of chunks (equal to number of processors)

* + Here, we define the total number of elements **N** in the array and the number of chunks **num\_chunks**. In this example, let's assume we have 100 elements in the array and 7 processors, so **N = 100** and **num\_chunks = 7**.

1. **Generating the Array**:

arr = np.arange(N) # Generate an array of N elements

* + We generate an array **arr** containing **N** elements using **np.arange(N)**. This creates an array with values from 0 to 99.

1. **Calculating Chunk Size**:

chunk\_size = N // num\_chunks # Calculate the chunk size

* + We calculate the size of each chunk by dividing the total number of elements (**N**) by the number of chunks (**num\_chunks**). This ensures that each chunk will have roughly the same number of elements.

1. **Splitting the Array into Chunks**:

send\_buf = np.array\_split(arr, num\_chunks) # Split the array into chunks

* + Finally, we split the array **arr** into **num\_chunks** chunks using **np.array\_split()**. Each chunk will contain approximately **N / num\_chunks** elements. The resulting chunks are stored in the **send\_buf** variable.

**Example**:

* Let's assume we have 100 elements in the array (**N = 100**) and 7 processors (**num\_procs = 7**).
* After splitting the array into chunks, each chunk will contain approximately **100 / 7 ≈ 14** elements.
* The resulting **send\_buf** will be a list containing 7 chunks, each with around 14 elements.

Here's a simplified example of what **send\_buf** might look like:

send\_buf = [array([0, 1, 2, ..., 13]), # Chunk 1 array([14, 15, 16, ..., 27]), # Chunk 2 array([28, 29, 30, ..., 41]), # Chunk 3 array([42, 43, 44, ..., 55]), # Chunk 4 array([56, 57, 58, ..., 69]), # Chunk 5 array([70, 71, 72, ..., 83]), # Chunk 6 array([84, 85, 86, ..., 99])] # Chunk 7

Each chunk is assigned to a different process for parallel computation of the sum of array elements.

In the command **recv\_buf = np.empty(1, dtype=int)**, the number 1 specifies the size of the array to be created.

Here's what each part of the command means:

* **np.empty**: This function from the NumPy library creates a new array with uninitialized (garbage) values.
* **(1, dtype=int)**: This part of the command specifies the shape and data type of the array to be created.
  + **1**: This specifies the size or shape of the array. In this case, it indicates that the array will have a single element.
  + **dtype=int**: This specifies the data type of the elements in the array. Here, **int** indicates that the array will contain integer values.

The comm.scatter() function in MPI distributes chunks of data from the root process (rank 0) to all processes in the MPI communicator (MPI.COMM\_WORLD). Each process receives one of the chunks.

In short, comm.scatter() divides the data among processes, ensuring that each process gets a portion of the data to work on independently. It's a way to distribute work among multiple processes in a parallel computing environment, enabling parallel computation on different parts of the dataset.

1. **print("Local sum at rank {0}: {1}".format(rank, local\_sum))**:
   * This command prints the local sum calculated by each process along with its rank.
   * **{0}** is a placeholder for the rank of the current process, and **{1}** is a placeholder for the local sum calculated by that process.
   * **format(rank, local\_sum)** substitutes the placeholders with the rank and local sum values, respectively, for each process.
2. **recv\_buf = comm.reduce(local\_sum, op=MPI.SUM, root=0)**:
   * This command performs a reduction operation to obtain the global sum.
   * Each process computes its own local sum (**local\_sum**), which represents the sum of elements in the chunk of data it processed independently.
   * The reduction operation (**MPI.SUM**) specifies that the local sums should be summed together to obtain the global sum.
   * **comm.reduce()** gathers the local sums from all processes and applies the reduction operation to obtain the global sum.
   * The result of the reduction operation is stored in **recv\_buf**, which will hold the global sum when the operation is complete.

Sure, let's walk through the working of the provided code with an example:

Suppose we have an array of 100 elements: **[0, 1, 2, ..., 99]**.

1. **Initialization**:
   * We have 7 processors (ranks 0 to 6) executing the code in parallel using MPI.
2. **Array Splitting**:
   * On rank 0, the total number of elements **N** is defined as 100, and the array **arr** containing elements **[0, 1, 2, ..., 99]** is generated.
   * The array **arr** is split into 7 chunks (**num\_chunks = num\_procs = 7**), each containing approximately **100 / 7 ≈ 14** elements.
3. **Data Distribution**:
   * The chunks of the array are scattered to all processes using **comm.scatter()**. Each process receives one of the chunks for computation.
4. **Local Sum Calculation**:
   * Each process calculates the sum of elements in its assigned chunk independently.
   * For example, process with rank 0 computes the sum of elements **[0, 1, 2, ..., 13]**, process with rank 1 computes the sum of elements **[14, 15, 16, ..., 27]**, and so on.
5. **Printing Local Sums**:
   * Each process prints its local sum along with its rank. For example:

bash

Copy code

Local sum at rank 0: <local\_sum\_0> Local sum at rank 1: <local\_sum\_1> ...

1. **Reduction Operation**:
   * After computing the local sums, all processes participate in a reduction operation (**MPI.SUM**) using **comm.reduce()**.
   * The local sums from all processes are gathered and summed together to obtain the global sum of all elements in the array.
2. **Printing Global Sum**:
   * The global sum is obtained and stored in **recv\_buf** on the root process (rank 0).
   * Rank 0 prints the global sum:

bash

Copy code

Global sum: <global\_sum>

**Example**:

* Suppose each chunk of the array is distributed to the processes as follows:
  + Process 0: **[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]**
  + Process 1: **[14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27]**
  + Process 2: **[28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41]**
  + Process 3: **[42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55]**
  + Process 4: **[56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69]**
  + Process 5: **[70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83]**
  + Process 6: **[84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]**
* After the reduction operation, if the local sums are **sum\_0 = 91**, **sum\_1 = 112**, ..., **sum\_6 = 105**, then the global sum would be **global\_sum = sum\_0 + sum\_1 + ... + sum\_6 = 495**.

To modify the code so that the array is provided by the user instead of being generated internally, you can prompt the user to input the array elements. Here's how you can make the necessary changes:

python

Copy code

from mpi4py import MPI import numpy as np comm = MPI.COMM\_WORLD rank = comm.rank # Get the total number of processors num\_procs = comm.Get\_size() if rank == 0: # Prompt the user to input the array N = int(input("Enter the total number of elements in the array: ")) arr = np.empty(N, dtype=int) print("Enter the elements of the array separated by space:") arr[:] = input().split() # Read space-separated input elements and store in the array else: arr = None # Placeholder for non-root processes # Broadcast the array from root process to all other processes arr = comm.bcast(arr, root=0) # Get the total number of elements in the array N = len(arr) # Split the array into chunks num\_chunks = num\_procs # Number of chunks (equal to number of processors) chunk\_size = N // num\_chunks # Calculate the chunk size send\_buf = np.array\_split(arr, num\_chunks) # Split the array into chunks # Initialize an array to store local sums local\_sum = np.empty(1, dtype=int) # Scatter the chunks to all processes send\_buf = comm.scatter(send\_buf, root=0) # Calculate the local sum local\_sum[0] = np.sum(send\_buf) # Print the local sum for each process print("Local sum at rank {0}: {1}".format(rank, local\_sum[0])) # Reduce local sums to obtain global sum recv\_buf = comm.reduce(local\_sum, op=MPI.SUM, root=0) # Print the global sum on root process if rank == 0: global\_sum = recv\_buf[0] # Global sum is received by the root process print("Global sum: " + str(global\_sum)) # Print the global sum

In this modified code:

* The root process (rank 0) prompts the user to input the total number of elements in the array and the array elements themselves.
* The array is broadcasted from the root process to all other processes using **comm.bcast()**.
* All other parts of the code remain the same, including array splitting, local sum calculation, and reduction operation for obtaining the global sum.