SRM Institute of Science and Technology
Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204
Department of Computer Applications
Circular – 2023-24
BCA DS 2nd Sem

Introduction to Computing with Distributed Data Processing (UDS23201J)

Lab Manual

Lab 1: Write a program to implement Remote Method Invocation (RMI)

Title: Remote Method Invocation (RMI) Implementation

Aim: To implement a simple distributed application using Java RMI.

Procedure:

1. Define a remote interface.
2. Implement the remote interface in a server class.
3. Create a server program that instantiates the server object and registers it with the RMI registry.
4. Create a client program that looks up the remote object from the registry and invokes its methods.
5. Compile and run the server.
6. Compile and run the client.

Source Code:

```
// Remote Interface (MyRemote.java)
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface MyRemote extends Remote {
    String sayHello() throws RemoteException;
}

// Server Implementation (MyRemoteImpl.java)
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
    public MyRemoteImpl() throws RemoteException {
        super();
    }

    public String sayHello() throws RemoteException {
        return "Hello from the remote server!";
    }
}

// Server Program (Server.java)
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class Server {
    public static void main(String[] args) {
```

```
        try {
            MyRemoteImpl obj = new MyRemoteImpl();
            // Bind the remote object's stub in the registry.
            Registry registry = LocateRegistry.getRegistry();
            registry.bind("Hello", obj);
            System.out.println("Server ready");
        } catch (Exception e) {
            System.err.println("Server exception: " + e.toString());
            e.printStackTrace();
        }
    }
}

// Client Program (Client.java)
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.RemoteException;
import java.rmi.NotBoundException;

public class Client {
    public static void main(String[] args) {
        try {
            Registry registry = LocateRegistry.getRegistry(null); // null for
localhost
            MyRemote stub = (MyRemote) registry.lookup("Hello");
            String response = stub.sayHello();
            System.out.println("Response from server: " + response);
        } catch (Exception e) {
            System.err.println("Client exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

Input: None (for this simple example).

Expected Output:

Server: "Server ready" (printed on the server console)

Client: "Response from server: Hello from the remote server!" (printed on the client console)

Lab 2: Write a Program to implement Remote Procedure Call (RPC)

Title: Remote Procedure Call (RPC) Implementation

Aim: To implement a basic RPC mechanism.

Procedure:

1. Define an Interface Definition Language (IDL) to specify the remote procedure.
2. Use an RPC generator (like rpcgen on Linux) to create client and server stubs from the IDL.
3. Write the server-side implementation of the remote procedure.
4. Compile the server code.
5. Write the client program that calls the remote procedure through the client stub.
6. Compile the client code.
7. Run the server.
8. Run the client.

Source Code: (Since rpcgen is typically used on Linux, this is a conceptual example. A full, cross-platform example is more complex.)

```
//  my_interface.x (IDL file - Conceptual)
program MY_PROG {
    version MY_VERS {
        int add(int a, int b) = 1;
    } = 1;
}

// my_server.c (Server - Conceptual)
#include "my_interface.h"
#include <stdio.h>

int *
add_1_svc(int *argp, struct svc_req *rqstp)
{
    static int result;
    int a = argp->a; // Accessing structure members (conceptual)
    int b = argp->b;
    result = a + b;
    printf("Server: Received %d and %d, returning %d\n", a, b, result);
    return &result;
}

// my_client.c (Client - Conceptual)
#include "my_interface.h"
#include <stdio.h>

int
main(int argc, char *argv[])
{
    CLIENT *clnt;
    int *result;
    intpair arg; // Assuming intpair is defined in my_interface.h
(conceptual)

    if (argc != 3) {
        fprintf(stderr, "usage: %s hostname a b\n", argv[0]);
        exit(1);
    }
    clnt = clnt_create(argv[1], MY_PROG, MY_VERS, "tcp"); // Simplified
```

```
    if (clnt == NULL) {
        clnt_pcreateerror(argv[1]);
        exit(1);
    }
    arg.a = atoi(argv[2]);
    arg.b = atoi(argv[3]);
    result = add_1(&arg, clnt);
    if (result == NULL) {
        clnt_perror(clnt, "call failed");
        exit(1);
    }
    printf("Result: %d\n", *result);
    clnt_destroy(clnt);
    exit(0);
}
```

Input:

Server: None (waits for client requests)

Client: Two integers as command-line arguments (e.g., client localhost 5 3)

Expected Output:

Server: "Server: Received 5 and 3, returning 8"

Client: "Result: 8"

Lab 3: Case study: PaaS (Facebook, Google App Engine)

Title: Platform as a Service (PaaS) Case Study: Facebook and Google App Engine

Aim: To understand the concepts and features of Platform as a Service (PaaS) by studying Facebook's platform and Google App Engine.

Procedure:

1. Research the architecture and services provided by Facebook as a platform.
2. Investigate the features and capabilities of Google App Engine.
3. Compare and contrast the two platforms in terms of:

    Supported programming languages and frameworks

    Scalability and performance

    Deployment and management

    Pricing models

    Use cases and target audience

4. Document the findings in a report, including diagrams and examples where applicable.

Source Code: This lab is a case study, so there isn't a single source code listing. Instead, the report should reference the official documentation and any relevant code examples from Facebook's and Google App Engine's developer resources. For example, you might include a snippet of a Python app deployed on Google App Engine.

Input: Research and documentation from Facebook and Google App Engine.

Expected Output: A report that includes:

Overview of Facebook's platform and Google App Engine.

Comparison of their features and capabilities.

Analysis of their strengths and weaknesses.

Examples of applications built on each platform.

Conclusion summarizing the key differences and similarities.

Lab 4: Virtualization in Cloud using KVM and VMware

Title: Cloud Virtualization with KVM and VMware

Aim: To explore and compare virtualization technologies, specifically KVM and VMware, in a cloud environment.

Procedure:

1. Research Kernel-based Virtual Machine (KVM) and VMware ESXi.
2. Set up a lab environment (if possible) with either KVM or VMware.
3. Create and manage virtual machines (VMs) using the chosen virtualization platform.
4. Explore features such as:

    VM creation and deletion

    Resource allocation (CPU, memory, storage)

    Networking configuration

    Snapshots and cloning

    Live migration (if applicable)

5. Compare KVM and VMware in terms of:

    Architecture

    Performance

    Scalability

    Cost

    Management tools

    Operating system support

6. Document the findings.

Source Code: This lab involves system configuration and management. Instead of application source code, include the commands or scripts used to create and manage VMs (e.g., virsh commands for KVM, or ESXi CLI commands). Also, include configuration files.

Input: Installation media or software for KVM or VMware, operating system ISO images for the VMs.

Expected Output: A lab report that includes:

Overview of KVM and VMware.

Steps taken to set up the virtualization environment.

Commands/scripts used for VM management.

Comparison of KVM and VMware features.

Analysis of their suitability for different cloud scenarios.

Lab 5: MongoDB Atlas Installation

Title: MongoDB Atlas Installation

Aim: To set up a cloud-based MongoDB database using MongoDB Atlas.

Procedure:

1. Create an account on MongoDB Atlas (cloud.mongodb.com).
2. Create a new cluster in MongoDB Atlas.
3. Configure the cluster settings (region, tier, cluster name).
4. Configure network access (add your IP address to the whitelist).
5. Create a database user with appropriate privileges.
6. Connect to the cluster using the connection string provided by Atlas.
7. Verify the installation by performing a simple database operation.

Source Code: This lab involves cloud service configuration. Include the connection string and any code snippets used to verify the connection (e.g., a Python script using pymongo).

Input: MongoDB Atlas account credentials.

Expected Output:

A successfully created MongoDB Atlas cluster.

A database user with appropriate permissions.

Confirmation of successful connection to the cluster.

Lab 6: MongoDB CRUD operations

Title: MongoDB CRUD Operations

Aim: To perform Create, Read, Update, and Delete (CRUD) operations on a MongoDB database.

Procedure:

1. Connect to a MongoDB database (either a local instance or a cloud-based instance like MongoDB Atlas).
2. Create a new database and collection.
3. Perform the following CRUD operations:

   Create: Insert one or multiple documents into the collection.

   Read: Retrieve documents from the collection using various query criteria.

   Update: Modify existing documents in the collection.

   Delete: Remove documents from the collection.

4. Use a programming language (e.g., Python with pymongo) or the MongoDB shell (mongosh) to perform the operations.

Source Code: Provide code snippets for each CRUD operation. Here's an example using Python:

```
from pymongo import MongoClient

# Connect to MongoDB (replace with your connection string)
client =
MongoClient("mongodb+srv://<username>:<password>@<cluster_url>/test?retryWrit
es=true&w=majority")
db = client.test_database #use a database named "test_database"
collection = db.my_collection  # Create or access a collection

# Create (Insert)
def create_documents():
    document1 = {"name": "Alice", "age": 30, "city": "New York"}
    document2 = {"name": "Bob", "age": 25, "city": "London"}
    collection.insert_many([document1, document2])
    print("Documents inserted.")

# Read (Find)
def read_documents():
    print("All documents:")
    for doc in collection.find():
        print(doc)

    print("\nDocuments with age 25:")
    for doc in collection.find({"age": 25}):
        print(doc)

# Update
def update_documents():
    collection.update_one({"name": "Alice"}, {"$set": {"age": 31}})
    print("Document updated.")
```

```
# Delete
def delete_documents():
    collection.delete_one({"name": "Bob"})
    print("Document deleted.")

# Call the functions
create_documents()
read_documents()
update_documents()
read_documents() # Read again to see the update
delete_documents()
read_documents() # Read again to see the deletion
client.close()
```

Input: Data to be inserted, query criteria for reading, update criteria and values, deletion criteria.

Expected Output:

Confirmation messages for successful operations.

Output of read operations, showing the retrieved documents.

Lab 7: Data modeling in MongoDB

Title: Data Modeling in MongoDB

Aim: To learn how to design schemas and model data in MongoDB, considering relationships and performance.

Procedure:

1. Understand the differences between relational and NoSQL data modeling.
2. Explore MongoDB's document data model.
3. Learn about embedding and referencing documents.
4. Design schemas for different use cases, such as:

    One-to-one relationships

    One-to-many relationships

    Many-to-many relationships

5. Consider factors like data redundancy, query performance, and data consistency.
6. Implement the designed schemas in MongoDB.

Source Code: This lab involves schema design. Provide examples of document structures (JSON-like) that represent the designed schemas. Include code snippets for creating collections and inserting sample data.

Input: Descriptions of data entities and their relationships.

Expected Output:

Documented schema designs for different use cases.

MongoDB collection structures that implement the schemas.

Sample data that conforms to the schemas.

Lab 8: Creation of Queries in MongoDB

Title: Creation of Queries in MongoDB

Aim: To write various types of queries in MongoDB to retrieve data from collections.

Procedure:

1. Learn about MongoDB's query language.
2. Practice using different query operators, including:

    Equality and comparison operators ($eq, $gt, $lt, etc.)

    Logical operators ($and, $or, $not)

    Element operators ($exists, $type)

    Array operators ($in, $all, $size)

    Projection ({field: 1})

    Sorting (sort())

    Limiting and skipping results (limit(), skip())

3. Write queries for different scenarios, such as:

    Retrieving documents based on specific criteria

    Filtering documents based on multiple conditions

    Projecting specific fields

    Sorting and paginating results

    Using aggregate functions

Source Code: Provide MongoDB query examples using the MongoDB shell or a programming language like Python.

```
from pymongo import MongoClient

client =
MongoClient("mongodb+srv://<username>:<password>@<cluster_url>/test?retryWrit
es=true&w=majority")
db = client.test_database
collection = db.my_collection

# Insert some sample data
collection.insert_many([
    {"name": "Alice", "age": 30, "city": "New York", "hobbies": ["reading",
"traveling"]},
    {"name": "Bob", "age": 25, "city": "London", "hobbies": ["sports",
"music"]},
```

```
    {"name": "Charlie", "age": 35, "city": "New York", "hobbies": ["reading",
"cooking"]},
    {"name": "David", "age": 28, "city": "Paris", "hobbies": ["traveling",
"photography"]}
])

# Query 1: Find all documents
print("All documents:")
for doc in collection.find():
    print(doc)

# Query 2: Find documents where age is greater than 28
print("\nDocuments with age > 28:")
for doc in collection.find({"age": {"$gt": 28}}):
    print(doc)

# Query 3: Find documents where city is "New York" and age is less than 35
print("\nDocuments from New York and age < 35:")
for doc in collection.find({"city": "New York", "age": {"$lt": 35}}):
    print(doc)

# Query 4: Project only the name and city fields
print("\nName and city of all documents:")
for doc in collection.find({}, {"_id": 0, "name": 1, "city": 1}):
    print(doc)

# Query 5: Sort documents by age in descending order
print("\nDocuments sorted by age (descending):")
for doc in collection.find().sort("age", -1):
    print(doc)

# Query 6: Find documents where "reading" is in the hobbies array
print("\nDocuments with 'reading' as a hobby:")
for doc in collection.find({"hobbies": "reading"}):
    print(doc)
client.close()
```

Input: Query requirements.

Expected Output: The results of the MongoDB queries.

Lab 9: Write a program to sort a single field in MongoDB

Title: Sorting a Single Field in MongoDB

Aim: To write a program that sorts documents in a MongoDB collection based on a single field.

Procedure:

1. Connect to a MongoDB database.
2. Select a collection.
3. Use the sort() method in MongoDB to sort the documents.
4. Specify the field to sort by and the sort order (ascending or descending).
5. Retrieve and display the sorted documents.

Source Code:

def sort_collection(connection_string, database_name, collection_name, sort_field, sort_order): """ Sorts a MongoDB collection by a single field and prints the sorted documents.

```
  Args:
      connection_string (str): The MongoDB connection string.
      database_name (str): The name of the database.
      collection_name (str): The name of the collection.
      sort_field (str): The field to sort by.
      sort_order (int): 1 for ascending, -1 for descending.
  """
  try:
      # Connect to MongoDB
      client = MongoClient(connection_string)
      db = client[database_name]
      collection = db[collection_name]

      # Sort the collection
      sorted_documents = collection.find().sort(sort_field, sort_order)

      # Print the sorted documents
      print(f"Sorted documents (sorted by {sort_field}, order:
{'ascending' if sort_order == 1 else 'descending'}):")
      for doc in sorted_documents:
          print(doc)

      # Close the connection
      client.close()

  except Exception as e:
      print(f"An error occurred: {e}")
```

if **name** == "**main**": # Replace with your actual connection string, database, and collection connection_string = "mongodb+srv://:@<cluster_url>/test?retryWrites=true&w=majority" database_name = "test_database" collection_name = "my_collection" # Make sure this collection exists and has data

```
  # Example usage: Sort by 'age' in ascending order
  sort_collection(connection_string, database_name, collection_name,
"age", 1)

  # Example usage: Sort by 'name' in descending order
```

```
  sort_collection(connection_string, database_name, collection_name,
"name", -1)

</details>
```

Input:

6. MongoDB connection string.
7. Database name.
8. Collection name.
9. Field to sort by.
10. Sort order (ascending or descending).

Expected Output: The documents in the specified collection, sorted according to the given field and sort order.

Lab 10: Hadoop installation - Setting up a Single Node

Title: Hadoop Single-Node Setup

Aim: To install and configure Hadoop in a pseudo-distributed mode on a single machine.

Procedure:

1. Download a stable version of Hadoop.
2. Install Java Development Kit (JDK) if not already installed.
3. Configure environment variables (JAVA_HOME, HADOOP_HOME, PATH).
4. Configure Hadoop configuration files (core-site.xml, hdfs-site.xml, mapred-site.xml, yarn-site.xml).
5. Format the NameNode.
6. Start the Hadoop services (NameNode, DataNode, ResourceManager, NodeManager).
7. Verify the installation by accessing the Hadoop web interfaces and running a simple MapReduce job.

Source Code: This lab involves system configuration. Include the contents of the modified configuration files (core-site.xml, etc.) and the commands used to start and stop Hadoop services.

Input: Hadoop distribution, JDK.

Expected Output:

A running Hadoop single-node setup.

Verification through web interfaces (NameNode, ResourceManager).

Successful execution of a simple MapReduce job.

Lab 11: Example programs - Hadoop Streaming

Title: Hadoop Streaming Examples

Aim: To write and execute simple MapReduce programs using Hadoop Streaming with languages like Python or Bash.

Procedure:

1. Write a mapper script in Python or Bash that reads input from stdin, processes it, and writes key-value pairs to stdout.
2. Write a reducer script in Python or Bash that reads key-value pairs from stdin, aggregates the values for each key, and writes the output to stdout.
3. Prepare input data and copy it to the Hadoop Distributed File System (HDFS).
4. Run the Hadoop Streaming job, specifying the mapper, reducer, input path, and output path.
5. Retrieve the output from HDFS and examine the results.

Source Code: Provide the mapper and reducer scripts (e.g., Python code).

```
# mapper.py (Python)
#!/usr/bin/env python3
import sys

for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
        print(f"{word}\t1")

# reducer.py (Python)
#!/usr/bin/env python3
import sys

current_word = None
current_count = 0
word = None

for line in sys.stdin:
    line = line.strip()
    word, count = line.split('\t', 1)

    try:
        count = int(count)
    except ValueError:
        continue

    if current_word == word:
        current_count += count
    else:
        if current_word:
            print(f"{current_word}\t{current_count}")
        current_count = count
        current_word = word

if current_word == word:
    print(f"{current_word}\t{current_count}")
```

Input: Input data (e.g., text files) in HDFS.

Expected Output: The output of the MapReduce job, stored in HDFS. For example, for a word count program, the output would be a list of words and their counts.

Lab 12: Writing a Hadoop MapReduce program in Python

Title: Hadoop MapReduce in Python

Aim: To write a complete MapReduce program in Python using the hadoop command.

Procedure:

1. Write a mapper script in Python to process the input data.
2. Write a reducer script in Python to aggregate the intermediate data.
3. Ensure the scripts are executable.
4. Copy the input data to HDFS.
5. Run the MapReduce job using the hadoop command, specifying the mapper, reducer, input path, and output path.6. Retrieve the output from HDFS and verify the results.

Source Code: Provide the Python mapper and reducer scripts. (See example in Lab 11).

Input: Input data in HDFS.

Expected Output: The processed output from the MapReduce job, stored in HDFS.

Lab 13: Create an Application using Apache Spark (e.g., Similarity word count during searching)

Title: Apache Spark Application: Similarity Word Count

Aim: To create a Spark application that calculates a similarity word count (e.g., counting similar words in search queries).

Procedure:

1. Set up a Spark development environment.
2. Write a Spark application (in Python, Scala, or Java) that:

   Reads input data (e.g., search query logs).

   Defines a function to identify similar words (e.g., using a predefined dictionary or a similarity metric).

   Applies the function to the input data using Spark transformations (e.g., map, flatMap).

   Counts the occurrences of similar words using Spark transformations (e.g., reduceByKey).

   Outputs the results.

3. Run the Spark application.
4. Analyze the output.

Source Code: Provide the Spark application code. Here's a Python example:

```
from pyspark import SparkContext, SparkConf

def get_similar_words(word, similarity_dict):
    """
    Returns a list of words similar to the given word, based on a dictionary.
    """
    return similarity_dict.get(word, [word]) # Returns the word itself if no
similar words are found

def calculate_similarity_word_count(input_file, output_path):
    """
    Calculates the similarity word count from an input text file using Spark.
    """
    # Create Spark configuration and context
    conf = SparkConf().setAppName("SimilarityWordCount")
    sc = SparkContext(conf=conf)

    # Define a dictionary of similar words
    similarity_dict = {
        "search": ["find", "lookup", "query"],
        "computer": ["pc", "laptop", "desktop"],
        "help": ["assist", "support", "aid"]
    }

    # Read the input text file
    lines = sc.textFile(input_file)
```

```
    # Split each line into words
    words = lines.flatMap(lambda line: line.split())

    # For each word, get its similar words
    similar_words = words.flatMap(lambda word: get_similar_words(word,
similarity_dict))

    # Count the occurrences of each similar word
    word_counts = similar_words.map(lambda word: (word,
1)).reduceByKey(lambda a, b: a + b)

    # Save the results to a text file
    word_counts.saveAsTextFile(output_path)

    # Print the results
    print("Results:")
    for word, count in word_counts.collect():
        print(f"{word}: {count}")
    sc.stop()

if __name__ == "__main__":
    # Replace with your input file path and output path
    input_file_path = "input.txt"  #  Create a file named input.txt with some
text
    output_path = "output"
    calculate_similarity_word_count(input_file_path, output_path)
```

Input: A text file containing search queries or sentences.

Expected Output: The count of each word and its similar words in the input data.

Lab 14: Writing Spark applications

Title: Writing Spark Applications

Aim: To gain hands-on experience in developing Apache Spark applications for various data processing tasks.

Procedure:

1. Set up a Spark development environment.
2. Choose a data processing task (e.g., log analysis, data cleaning, simple machine learning).
3. Write a Spark application (in Python, Scala, or Java) that:

   Reads input data from a file or other source.

   Processes the data using Spark transformations (e.g., map, filter, reduceByKey, groupByKey).

   Performs the desired data processing task.

   Outputs the results to a file or other destination.

4. Run the Spark application.
5. Analyze the results and evaluate the performance of the application.

Source Code: Provide the complete Spark application code. The specific code will depend on the chosen data processing task.

Input: Input data relevant to the chosen task.

Expected Output: The processed data, as determined by the Spark application.

Lab 15: Write a MPI Program to send data across all processes. Perform a Simple Vector Addition using OpenMP Programming.

Title: MPI Data Transfer and OpenMP Vector Addition

Aim:

To write an MPI program to distribute data among all processes.

To implement vector addition using OpenMP for parallelization.

Procedure:

- **MPI Data Transfer:**

  Initialize the MPI environment.

  Determine the rank of each process and the total number of processes.

  Use MPI functions (e.g., MPI_Bcast, MPI_Scatter, MPI_Gather) to distribute data from one process to all others or to distribute data in chunks from one process to many.

  Print the data received by each process.

  Finalize the MPI environment.

- **OpenMP Vector Addition:**

  Initialize a vector (array) with data.

  Use OpenMP directives (e.g., #pragma omp parallel for) to parallelize the loop that performs the vector addition.

  Print the original vectors and the result vector.

  Compile the code with OpenMP support (e.g., -fopenmp flag in GCC).

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <omp.h>

#define VECTOR_SIZE 10

int main(int argc, char *argv[]) {
    int rank, size, i;
    int send_data;
    int *recv_data;
    int vector_a[VECTOR_SIZE], vector_b[VECTOR_SIZE],
vector_sum[VECTOR_SIZE];

    // Initialize MPI
```

```
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // MPI Data Transfer
    if (rank == 0) {
        send_data = 100;
        printf("Process %d: Sending data %d to all processes\n", rank,
send_data);
    }

    // Use MPI_Bcast to send data from process 0 to all other processes
    MPI_Bcast(&send_data, 1, MPI_INT, 0, MPI_COMM_WORLD);
    printf("Process %d: Received data %d\n", rank, send_data);

   // Allocate memory for receive buffer
    recv_data = (int *)malloc(sizeof(int));
    if (recv_data == NULL) {
        fprintf(stderr, "Error allocating memory on process %d\n", rank);
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    // Example of using MPI_Scatter and MPI_Gather (optional, for more
complex data distribution)
    int send_counts[size];
    int displacements[size];
    int send_buffer[size];

    if (rank == 0)
    {
        for (i = 0; i < size; i++)
        {
            send_counts[i] = 1;
            displacements[i] = i;
            send_buffer[i] = i * 10; // Example data
        }
    }
    MPI_Scatterv(send_buffer, send_counts, displacements, MPI_INT,
                 recv_data, 1, MPI_INT, 0, MPI_COMM_WORLD);
    printf("Process %d received %d\n", rank, *recv_data);

    MPI_Gatherv(recv_data, 1, MPI_INT, send_buffer, send_counts,
displacements, MPI_INT, 0, MPI_COMM_WORLD);
     if (rank == 0) {
         printf("Process 0 gathered: ");
            for (i = 0; i < size; i++) {
                printf("%d ", send_buffer[i]);
            }
            printf("\n");
     }
    free(recv_data);

    // OpenMP Vector Addition
    if (rank == 0) {
        // Initialize vectors
        for (i = 0; i < VECTOR_SIZE; i++) {
            vector_a[i] = i;
            vector_b[i] = VECTOR_SIZE - i;
        }

        printf("Vector A: ");
        for (i = 0; i < VECTOR_SIZE; i++)
            printf("%d ", vector_a[i]);
        printf("\n");

        printf("Vector B: ");
        for (i = 0; i < VECTOR_SIZE; i++)
```

```
            printf("%d ", vector_b[i]);
        printf("\n");

        // Perform vector addition in parallel using OpenMP
        #pragma omp parallel for
        for (i = 0; i < VECTOR_SIZE; i++) {
            vector_sum[i] = vector_a[i] + vector_b[i];
        }

        printf("Vector Sum: ");
        for (i = 0; i < VECTOR_SIZE; i++)
            printf("%d ", vector_sum[i]);
        printf("\n");
    }

    // Finalize MPI
    MPI_Finalize();
    return 0;
}
```

Input: None. The data for vector addition is initialized within the program.

Expected Output:

Each process prints the data it receives from the MPI data transfer.

Process 0 prints the original vectors and their sum after the OpenMP vector addition.

Lab 16: Create a Simple Virtual Machine on Google Compute Engine

Title: Creating a Virtual Machine on Google Compute Engine

Aim: To create and configure a virtual machine (VM) instance on Google Compute Engine (GCE).

Procedure:

1. Create a Google Cloud Platform (GCP) account (if you don't have one).
2. Create a new project in GCP.
3. Enable the Compute Engine API for your project.
4. Go to the Compute Engine section in the Google Cloud Console.
5. Click "Create Instance" to create a new VM instance.
6. Configure the VM instance settings:

    Name

    Zone

    Machine type

    Boot disk (operating system)

    Firewall rules (allow HTTP/HTTPS traffic if needed)

7. Click "Create" to launch the VM instance.
8. Connect to the VM instance using SSH.
9. Verify the VM instance configuration.

Source Code: This lab involves cloud platform configuration. Include the gcloud commands used to create and manage the VM instance (if using the command-line tool). If using the web console, document the steps with screenshots.

Input: Google Cloud Platform account credentials.

Expected Output:

A running VM instance on Google Compute Engine.

Confirmation of successful connection to the VM via SSH.

Verification of the VM's configuration (OS, resources).