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

Elements of Distributed Data Processing (UDS24201J)

Lab Manual

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

Title: Remote Method Invocation (RMI)

Aim: To implement a simple distributed application using Java RMI, where a client can invoke a method on a remote server object.

Procedure:

1. Define a remote interface that declares the methods to be called remotely.
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 in the registry and invokes its methods.
5. Compile all the Java files.
6. Start the RMI registry.
7. Run the server program.
8. Run the client program.

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 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;

public class Client {
    public static void main(String[] args) {
        try {
            Registry registry = LocateRegistry.getRegistry(null); //null
means 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: (How you run the programs)

Start the RMI registry: rmiregistry in a terminal.

Run the server: java Server

Run the client: java Client

Expected Output:

Server: Prints "Server ready"

Client: Prints "Response from server: Hello from the server!"

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

Title: Remote Procedure Call (RPC)

Aim: To implement a basic RPC mechanism to execute a procedure on a remote machine.

Procedure:

1. Define the interface definition language (IDL) to specify the procedure.
2. Use an RPC generator (like rpcgen on Linux) to create client and server stubs.
3. Implement the server-side procedure.
4. Compile the server code and create the server executable.
5. Compile the client code and create the client executable.
6. Run the server program.
7. Run the client program.

Source Code: (Example using rpcgen on Linux)

```
// IDL File (my_rpc.x)
program MY_RPC_PROG {
    version MY_RPC_VERS {
        int add(int a, int b) = 1;
    } = 1;
}

// Server Implementation (my_rpc_svc.c) - *This is generated by rpcgen,
you'll need to add the function implementation*
#include "my_rpc.h"
#include <stdio.h>

int *
add_1_svc(int *argp, struct svc_req *rqstp)
{
    static int result;
    int a = argp->a; // Access the structure members
    int b = argp->b;
    result = a + b;
    printf("Server received a = %d, b = %d\n", a, b); //Added printf
    return &result;
}

// Client Program (my_rpc_clnt.c) - *This is generated by rpcgen, you might
need to adapt it*
#include <stdio.h>
#include <rpc/client.h>
#include "my_rpc.h"

int main(int argc, char *argv[]) {
    CLIENT *clnt;
    int *result;
    input_data numbers; // Use the struct
    if (argc != 3) {
        fprintf(stderr, "usage: %s hostname a b\n", argv[0]);
        exit(1);
    }
    clnt = clnt_create(argv[1], MY_RPC_PROG, MY_RPC_VERS, "tcp");
    if (clnt == NULL) {
        clnt_pcreateerror(argv[1]);
        exit(1);
    }
```

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

Input:

Compile the IDL: rpcgen my_rpc.x

Compile server: cc my_rpc_svc.c -o server (and link with RPC libraries, may vary by system)

Compile client: cc my_rpc_clnt.c -o client (and link with RPC libraries)

Run server: ./server

Run client: ./client <server_hostname> 5 3 (for example)

Expected Output:

Server: Prints "Server received a = 5, b = 3"

Client: Prints "Result: 8"

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

Title: Platform as a Service (PaaS) Case Study

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

Procedure:

1. Research the architecture and services provided by Facebook's platform for developers.
2. Research the architecture and services provided by Google App Engine.
3. Compare and contrast the features, advantages, and limitations of both platforms.
4. Document your findings, including specific services, scaling capabilities, and development workflows.

Source Code: (This is a case study, so there's no single source code. You'll be documenting your findings.)

Documentation of Facebook Platform features.

Documentation of Google App Engine features.

Comparison table.

Input: Research on Facebook for developers and Google App Engine.

Expected Output: A written report or presentation that includes:

Overview of Facebook's developer platform.

Overview of Google App Engine.

Comparison of features, scaling, and workflows.

Analysis of the benefits and drawbacks of each platform.

Lab 4: Virtualization in Cloud by using KVM and VMware

Title: Cloud Virtualization with KVM and VMware

Aim: To gain practical experience with virtualization technologies by setting up and using Kernel-based Virtual Machine (KVM) and VMware.

Procedure:

1. Install a hypervisor (KVM or VMware) on a suitable machine.
2. Create virtual machines (VMs) using the installed hypervisor.
3. Configure the VMs with appropriate operating systems, memory, and storage.
4. Explore different virtualization features, such as snapshots, cloning, and networking.
5. Compare the performance and features of KVM and VMware.

Source Code: (This involves configuration, not direct coding, but you'll document the steps)

Instructions for installing KVM (or VMware).

Instructions for creating and configuring VMs.

Configuration files (e.g., virsh commands for KVM, VM settings in VMware).

Input:

Download and install KVM or VMware.

Download ISO images for guest operating systems (e.g., Linux distributions, Windows).

Expected Output:

A running virtual machine.

Documentation of the steps taken.

Comparison of KVM and VMware (features, performance).

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.
2. Create a new cluster in MongoDB Atlas.
3. Configure the cluster settings, including cloud provider, region, and cluster tier.
4. Configure network access to allow connections to the cluster.
5. Create a database user with appropriate permissions.
6. Connect to the MongoDB Atlas cluster using the provided connection string.

Source Code: (This is a cloud setup, so the "source code" is the configuration)

Screenshots of the MongoDB Atlas setup process.

Connection string obtained from MongoDB Atlas.

Input:

Access to a web browser and internet connection.

MongoDB Atlas account credentials.

Expected Output:

A running MongoDB cluster in MongoDB Atlas.

A connection string to access the cluster.

Verification that you can connect to the database.

Lab 6: MongoDB CRUD operations

Title: MongoDB CRUD Operations

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

Procedure:

1. Connect to a MongoDB database (either local or cloud-based).
2. Create a new database and collection.
3. Insert documents into the collection.
4. Read documents from the collection using various query methods.
5. Update existing documents in the collection.
6. Delete documents from the collection.

Source Code:

```
# Python example using pymongo
from pymongo import MongoClient

# Establish a connection
client = MongoClient("mongodb+srv://<your_connection_string>") # Replace with
your connection string
db = client.test_database #use your database name
collection = db.my_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 document in collection.find():
        print(document)

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

# 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.")

#main method
if __name__ == '__main__':
    create_documents()
    read_documents()
    update_documents()
    delete_documents()
```

```
read_documents() # Read again to show changes
client.close()
```

Input:

A running MongoDB instance.

Python and the pymongo library (if using the example code).

Replace <your_connection_string> with your actual MongoDB connection string.

Expected Output:

Output showing the inserted documents.

Output showing the results of read queries.

Output confirming the update operation.

Output confirming the delete operation.

Final output showing the documents after the update and delete.

Lab 7: Data modeling in MongoDB

Title: Data Modeling in MongoDB

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

Procedure:

1. Understand the differences between relational and NoSQL data modeling.
2. Explore MongoDB's document-oriented approach.
3. Implement data models using embedded documents and linked documents.
4. Consider factors like data redundancy, query patterns, and data consistency when designing schemas.
5. Model different types of relationships (one-to-one, one-to-many, many-to-many) in MongoDB.

Source Code: (This is more about schema design than code, but you'll create example documents)

```
// Example of embedded documents (one-to-many relationship)
{
  "product_name": "Laptop",
  "price": 1200,
  "reviews": [
    {
      "user": "Alice",
      "rating": 5,
      "comment": "Excellent product"
    },
    {
      "user": "Bob",
      "rating": 4,
      "comment": "Good value"
    }
  ]
}

// Example of linked documents (one-to-many relationship)
// Products collection
{
  "_id": ObjectId("product1"),
  "name": "Laptop",
  "price": 1200
}
// Orders collection
{
  "_id": ObjectId("order1"),
  "product_id": ObjectId("product1"),  // Reference to the product
  "quantity": 1,
  "customer": "Alice"
}
```

Input:

A MongoDB instance.

A clear understanding of the data to be modeled (e.g., products, customers, orders).

Expected Output:

Example schemas (JSON documents) for different data models.

Explanation of the chosen modeling approach.

Discussion of the trade-offs between embedding and linking.

Diagrams (if applicable) to illustrate the data models.

Lab 8: Creation of Queries in MongoDB

Title: MongoDB Querying

Aim: To write various MongoDB queries to retrieve data from a database, including filtering, sorting, and projecting.

Procedure:

1. Connect to a MongoDB database.
2. Use the find() method to retrieve documents.
3. Apply query operators (e.g., $gt, $lt, $eq, $in) to filter documents.
4. Use projection to select specific fields to return.
5. Use the sort() method to order the results.
6. Use aggregation pipeline stages ($match, $group, $sort, etc.) for complex queries.

Source Code:

```
# Python example using pymongo
from pymongo import MongoClient

client = MongoClient("mongodb+srv://<your_connection_string>")  # Replace
db = client.test_database
collection = db.users

# Insert some data for querying
collection.insert_many([
    {"name": "Alice", "age": 30, "city": "New York", "role": "admin"},
    {"name": "Bob", "age": 25, "city": "London", "role": "user"},
    {"name": "Charlie", "age": 35, "city": "New York", "role": "user"},
    {"name": "David", "age": 28, "city": "London", "role": "editor"},
    {"name": "Eve", "age": 32, "city": "Paris", "role": "user"}
])

# 1. Find all users
def find_all_users():
    print("All users:")
    for user in collection.find():
        print(user)

# 2. Find users in New York
def find_users_in_new_york():
    print("\nUsers in New York:")
    for user in collection.find({"city": "New York"}):
        print(user)

# 3. Find users older than 30
def find_users_older_than_30():
    print("\nUsers older than 30:")
    for user in collection.find({"age": {"$gt": 30}}):
        print(user)

# 4. Find names and ages of users in London
def find_names_and_ages_in_london():
    print("\nNames and ages of users in London:")
    for user in collection.find({"city": "London"}, {"_id": 0, "name": 1,
"age": 1}):
        print(user)

# 5. Find users sorted by age
```

```
def find_users_sorted_by_age():
    print("\nUsers sorted by age:")
    for user in collection.find().sort("age"):
        print(user)

# 6. Using Aggregation: Count users by city
def count_users_by_city():
    print("\nUser count by city:")
    pipeline = [
        {"$group": {"_id": "$city", "count": {"$sum": 1}}}
    ]
    results = collection.aggregate(pipeline)
    for result in results:
        print(result)

if __name__ == '__main__':
    find_all_users()
    find_users_in_new_york()
    find_users_older_than_30()
    find_names_and_ages_in_london()
    find_users_sorted_by_age()
    count_users_by_city()
    client.close()
```

Input:

A MongoDB instance with data.

Python and the pymongo library (if using the example). Replace the connection string.

Expected Output:

Output showing the results of each query.

The output should match the filtering, projection, sorting, and aggregation criteria specified in the 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, in ascending or descending order.

Procedure:

1. Connect to a MongoDB database.
2. Use the find() method with the sort() method to sort documents.
3. Specify the field to sort by and the sort order (1 for ascending, -1 for descending).

Source Code:

```
# Python example using pymongo
from pymongo import MongoClient

client = MongoClient("mongodb+srv://<your_connection_string>") # Replace
db = client.test_database
collection = db.products  # Example collection

# Insert sample data
collection.insert_many([
    {"name": "Laptop", "price": 1200},
    {"name": "Mouse", "price": 25},
    {"name": "Keyboard", "price": 75},
    {"name": "Monitor", "price": 300},
    {"name": "Headphones", "price": 100}
])

# Sort by price in ascending order
def sort_by_price_ascending():
    print("Products sorted by price (ascending):")
    for product in collection.find().sort("price", 1):
        print(product)

# Sort by price in descending order
def sort_by_price_descending():
    print("\nProducts sorted by price (descending):")
    for product in collection.find().sort("price", -1):
        print(product)

if __name__ == '__main__':
    sort_by_price_ascending()
    sort_by_price_descending()
    client.close()
```

Input:

A MongoDB instance with data.

Python and the pymongo library (if using the example). Replace connection string.

Expected Output:

Output showing the documents sorted by the specified field in ascending order.

Lab 10: Hadoop installation – Setting up a Single Node

Title: Single-Node Hadoop Installation

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

Procedure:

1. Download a stable version of Apache Hadoop.
2. Install Java Development Kit (JDK) if it is 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 Hadoop NameNode.
6. Start the Hadoop services (NameNode, DataNode, ResourceManager, NodeManager).
7. Verify the installation by accessing the Hadoop web interfaces.

Source Code: (This is mostly configuration, but you'll document the steps and configuration files)

Instructions for downloading and extracting Hadoop.

Example contents of the configuration files.

Commands used to start and stop Hadoop services.

Input:

A Linux machine (recommended for Hadoop).

JDK installed.

Hadoop distribution.

Expected Output:

A running single-node Hadoop installation.

Verification that you can access the Hadoop web interfaces (e.g., NameNode UI, ResourceManager UI).

Confirmation that you can run basic Hadoop commands.

Lab 11: Example programs-Hadoop Streaming Practice

Title: Hadoop Streaming Practice

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

Procedure:

1. Write a mapper script to process input data.
2. Write a reducer script to aggregate the processed data.
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, and output paths.
5. Retrieve the output from HDFS and analyze the results.

Source Code:

```
# mapper.py (Python)
#!/usr/bin/env python
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 python
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:

A running Hadoop installation.

Input data (e.g., a text file).

Mapper and reducer scripts (e.g., mapper.py, reducer.py).

Expected Output:

Output in HDFS containing the results of the MapReduce job (e.g., word counts).

Verification that the mapper and reducer scripts processed the data correctly.

Lab 12: Writing a Hadoop MapReduce program in Python

Title: Hadoop MapReduce in Python

Aim: To implement a MapReduce program in Python to process data on a Hadoop cluster.

Procedure:

1. Write a mapper script in Python to process the input data and emit key-value pairs.

2. Write a reducer script in Python to aggregate the values for each key and produce the final output. 3. Transfer the input data to the Hadoop Distributed File System (HDFS). 4. Execute the MapReduce job using Hadoop Streaming, specifying the mapper and reducer scripts, input and output paths. 5. Retrieve the output data from HDFS and verify the results.

Source Code: (See example in Lab 11. The structure is the same for basic MapReduce in Python using Hadoop Streaming)

Input:

A running Hadoop cluster.

Input data (e.g., text files).

Python mapper and reducer scripts.

Expected Output:

Output data in HDFS, representing the processed results of the MapReduce job.

Verification that the Python scripts correctly implemented the MapReduce logic.

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

Title: Apache Spark Application: Similarity Word Count

Aim: To develop an Apache Spark application that performs a similarity word count, which could be used in a search context.

Procedure:

1. Set up an Apache Spark environment.
2. Load the input data (e.g., a set of documents or search queries) into an RDD or DataFrame.
3. Implement a function to calculate word counts for each document.
4. Implement a function to calculate the similarity between documents based on their word counts (e.g., using cosine similarity).
5. Apply the functions to the data using Spark transformations and actions.
6. Output the similarity scores between documents.

Source Code:

```
# Python example using PySpark
from pyspark import SparkContext, SparkConf
from pyspark.sql import SparkSession
import math

# Function to calculate word counts
def get_word_counts(text):
    words = text.lower().split()
    word_counts = {}
    for word in words:
        word_counts[word] = word_counts.get(word, 0) + 1
    return word_counts

# Function to calculate cosine similarity
def cosine_similarity(vec1, vec2):
    dot_product = 0
    magnitude1 = 0
    magnitude2 = 0
    for key in set(vec1.keys()) | set(vec2.keys()):
        val1 = vec1.get(key, 0)
        val2 = vec2.get(key, 0)
        dot_product += val1 * val2
        magnitude1 += val1 ** 2
        magnitude2 += val2 ** 2
    if magnitude1 == 0 or magnitude2 == 0:
        return 0
    return dot_product / (math.sqrt(magnitude1) * math.sqrt(magnitude2))

if __name__ == "__main__":
    conf = SparkConf().setAppName("SimilarityWordCount")
    sc = SparkContext(conf=conf)
    spark = SparkSession(sc)

    # Sample data (list of documents)
    data = [
        "This is a sample document",
        "This is another document",
        "A sample document is this",
        "Another sample document"
```

```
    ]

    # Create an RDD
    documents = sc.parallelize(data)

    # Calculate word counts for each document
    word_counts = documents.map(get_word_counts)

    # Collect the word counts
    word_count_list = word_counts.collect()

    # Calculate and print similarity between each pair of documents
    for i in range(len(word_count_list)):
        for j in range(i + 1, len(word_count_list)):
            similarity = cosine_similarity(word_count_list[i],
word_count_list[j])
            print(f"Similarity between document {i+1} and {j+1}:
{similarity:.4f}")

    sc.stop()
```

Input:

An Apache Spark cluster or a local Spark setup.

Input data consisting of text documents or search queries.

Python and PySpark.

Expected Output:

Output showing the similarity scores (e.g., cosine similarity) between pairs of documents.

The output demonstrates how similar the documents are based on their word content.

Lab 14: Writing Spark applications

Title: Writing Spark Applications

Aim: To develop complete Apache Spark applications for various data processing tasks.

Procedure:

1. Define the problem to be solved using Spark (e.g., data cleaning, transformation, analysis).
2. Write Spark code using RDDs, DataFrames, or Datasets, depending on the requirements.
3. Utilize Spark transformations (e.g., map, filter, reduceByKey) and actions (e.g., collect, saveAsTextFile).
4. Optimize the Spark application for performance (e.g., by using appropriate partitioning, caching, and data serialization).
5. Test the Spark application with sample data and analyze the results.

Source Code: (This will vary greatly depending on the specific application. Provide a general structure and an example.)

```
# Example:  Spark Application - Log Analysis
from pyspark import SparkContext, SparkConf
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

if __name__ == "__main__":
    conf = SparkConf().setAppName("LogAnalysis")
    sc = SparkContext(conf=conf)
    spark = SparkSession(sc)

    # 1. Load the log data into a DataFrame
    log_file = "path/to/your/logfile.txt"  # Replace with your log file path
    df = spark.read.text(log_file)

    # 2. Extract relevant information using regular expressions
    df = df.select(
        regexp_extract(col("value"), r"(\d{4}-\d{2}-\d{2}
\d{2}:\d{2}:\d{2})", 1).alias("timestamp"),
        regexp_extract(col("value"), r"\[(\w+)\]", 1).alias("level"),
        regexp_extract(col("value"), r"\[.*?\] (.*)", 1).alias("message")
    )

    # 3. Filter log messages by level
    errors_df = df.filter(col("level") == "ERROR")

    # 4. Count the number of errors
    error_count = errors_df.count()
    print(f"Total number of errors: {error_count}")

    # 5. Show the first 10 error messages
    errors_df.show(10, truncate=False)

    # 6. Group by timestamp and count occurrences
    error_counts_by_time =
errors_df.groupBy("timestamp").count().orderBy("timestamp")
    error_counts_by_time.show()

    # Stop Spark Session
    spark.stop()
```

```
sc.stop()
```

Input:

An Apache Spark cluster.

Input data relevant to the application (e.g., log files, text data, CSV files).

Python and PySpark.

Expected Output:

The output will depend on the specific application. For the log analysis example:

Total number of errors.

A sample of error messages.

Error counts grouped by timestamp.

The output demonstrates that the Spark application has processed the data according to the defined logic.

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 multiple processes.

To write an OpenMP program to perform parallel vector addition.

Procedure:

MPI (Data Transfer):

1. Initialize the MPI environment.
2. Determine the rank (ID) of each process and the total number of processes.
3. Use MPI functions (e.g., MPI_Send, MPI_Recv, MPI_Bcast, or MPI_Allgather) to distribute the data from one process to all others.
4. Print the data received by each process.
5. Finalize the MPI environment.

OpenMP (Vector Addition):

6. Initialize the vectors to be added.
7. Use OpenMP directives (e.g., #pragma omp parallel for) to parallelize the loop that performs the vector addition.
8. Ensure that the loop iterations are distributed among the available threads.
9. Print the resulting sum vector.

Source Code:

```
// MPI (Data Transfer)
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    MPI_Init(NULL, NULL);
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    int data;
    if (world_rank == 0) {
        data = 100; // Root process has the data
        printf("Process 0 sending data %d to all processes\n", data);
        for (int i = 1; i < world_size; i++) {
            MPI_Send(&data, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
        }
    } else {
        MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process %d received data %d from process 0\n", world_rank,
data);
    }
```

```
    MPI_Finalize();
    return 0;
}

// OpenMP (Vector Addition)
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define VECTOR_SIZE 1000

int main() {
    int a[VECTOR_SIZE], b[VECTOR_SIZE], sum[VECTOR_SIZE];
    int i;

    // Initialize vectors a and b
    for (i = 0; i < VECTOR_SIZE; i++) {
        a[i] = i;
        b[i] = VECTOR_SIZE - i;
    }

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

    // Print the result (optional, print a few elements)
    printf("Vector Addition Result:\n");
    for (i = 0; i < 10; i++) { // Print first 10 elements
        printf("sum[%d] = %d\n", i, sum[i]);
    }

    return 0;
}
```

Input:

MPI: A working MPI installation (e.g., OpenMPI, MPICH).

OpenMP: A C/C++ compiler with OpenMP support (e.g., GCC with -fopenmp flag).

Expected Output:

MPI: Each process prints the data it received. Process 0 prints the data it sent.

OpenMP: The program prints the sum of the two vectors.

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 instance on Google Compute Engine (GCE).

Procedure:

1. Create a Google Cloud Platform (GCP) account.
2. Create a new project in GCP.
3. Enable the Compute Engine API for the project.
4. Navigate to the Compute Engine section in the GCP console.
5. Create a new VM instance, specifying the machine type, operating system, and other settings.
6. Configure firewall rules to allow access to the VM instance (e.g., for SSH).
7. Connect to the VM instance using SSH.
8. Verify that the VM instance is running and accessible.

Source Code: (This is a cloud platform setup, so the "source code" is the configuration)

Instructions for creating a GCP project and enabling the Compute Engine API.

Screenshots of the VM instance creation process in the GCP console.

Commands used to connect to the VM instance via SSH.

Input:

A Google Cloud Platform account.

Internet access.

SSH client.

Expected Output:

A running virtual machine instance on Google Compute Engine.

Confirmation that you can connect to the VM instance via SSH.

Verification of the VM instance's configuration (e.g., operating system, resources).