HPC program

Assignment no - 1

```
#include <iostream>
#include <queue>
#include <vector>
#include <omp.h>
using namespace std;

int main() {
    // Input: Number of vertices, edges, and source node
    int num_vertices, num_edges, source;
    cout << "Enter number of vertices, edges, and source node: ";
    cin >> num_vertices >> num_edges >> source;

    // Initialize adjacency list (undirected graph)
    vector<vector<int>> adj_list(num_vertices + 1);
    cout << "Enter edges (u v):" << endl;
    for (int i = 0; i < num_edges; i++) {
        int u, v;
        cin >> u >> v;
        adj_list[u].push_back(v);
        adj_list[v].push_back(u);
    }

    // BFS setup
    queue<int> q;
    vector<bool> visited(num_vertices + 1, false);

    // Start BFS from the source node
    q.push(source);
    visited[source] = true;

    cout << "BFS Traversal: ";
    while (!q.empty()) {
```

```
        int curr_vertex = q.front();
        q.pop();
        cout << curr_vertex << " ";

        // Parallel processing of neighbours
        #pragma omp parallel for shared(adj_list, visited, q) schedule(dynamic)
        for (int i = 0; i < adj_list[curr_vertex].size(); i++) {
            int neighbour = adj_list[curr_vertex][i];

            // Critical section to avoid race conditions
            #pragma omp critical
            {
                if (!visited[neighbour]) {
                    visited[neighbour] = true;
                    q.push(neighbour);
                }
            }
        }
    }

    cout << endl;
    return 0;
}

#include <iostream>
#include <vector>
#include <omp.h>
using namespace std;

const int MAXN = 1e5; // Maximum number of nodes
vector<int> adj[MAXN + 5]; // Adjacency list
bool visited[MAXN + 5]; // Visited status for nodes

/**
 * Performs parallel DFS traversal starting from a given node.
 * @param node The starting node for DFS.
 */
void dfs(int node) {
    visited[node] = true;
```

```
    #pragma omp parallel for
    for (int i = 0; i < adj[node].size(); i++) {
        int next_node = adj[node][i];
        if (!visited[next_node]) {
            dfs(next_node);
        }
    }
}

int main() {
    cout << "=== Parallel DFS Traversal ===\n";

    // Input: Number of nodes and edges
    int n, m;
    cout << "Enter number of nodes and edges: ";
    cin >> n >> m;

    // Input: Edges of the graph
    cout << "Enter " << m << " edges (u v):\n";
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u); // Undirected graph
    }

    // Input: Starting node for DFS
    int start_node;
    cout << "Enter the starting node for DFS: ";
    cin >> start_node;

    // Reset visited array (for safety)
    fill(visited, visited + MAXN + 5, false);

    // Perform DFS
    dfs(start_node);

    // Output: Visited nodes
```

```
    cout << "\nVisited nodes: ";
    for (int i = 1; i <= n; i++) {
        if (visited[i]) {
            cout << i << " ";
        }
    }
    cout << "\n";

    return 0;
}
```

Assignment no - 2

```
#include <iostream>
#include <omp.h>
using namespace std;

void bubble_sort_odd_even(int arr[], int n) {
    bool isSorted = false;
    while (!isSorted) {
        isSorted = true;
        #pragma omp parallel for
        for (int i = 0; i < n - 1; i += 2) {
            if (arr[i] > arr[i + 1]) {
                swap(arr[i], arr[i + 1]);
                isSorted = false;
            }
        }
        #pragma omp parallel for
        for (int i = 1; i < n - 1; i += 2) {
            if (arr[i] > arr[i + 1]) {
                swap(arr[i], arr[i + 1]);
                isSorted = false;
            }
        }
    }
}

int main() {
```

```
    int n;
    cout << "Enter the number of elements: ";
    cin >> n;

    int* arr = new int[n];  // Dynamic array allocation

    cout << "Enter " << n << " elements:\n";
    for (int i = 0; i < n; ++i) {
        cin >> arr[i];
    }

    double start, end;

    // Measure performance of parallel bubble sort
    start = omp_get_wtime();
    bubble_sort_odd_even(arr, n);
    end = omp_get_wtime();

    cout << "\nParallel bubble sort (odd-even) time: " << end - start << " seconds\n";

    cout << "Sorted array: ";
    for (int i = 0; i < n; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;

    delete[] arr;  // Free allocated memory
    return 0;
}

#include <iostream>
#include <omp.h>
using namespace std;

void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;
```

```
    // Create temporary arrays
    int* L = new int[n1];
    int* R = new int[n2];

    // Copy data to temp arrays
    for (i = 0; i < n1; i++) {
        L[i] = arr[l + i];
    }
    for (j = 0; j < n2; j++) {
        R[j] = arr[m + 1 + j];
    }

    // Merge the temp arrays back into arr[l..r]
    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k++] = L[i++];
        } else {
            arr[k++] = R[j++];
        }
    }

    // Copy remaining elements of L[] (if any)
    while (i < n1) {
        arr[k++] = L[i++];
    }

    // Copy remaining elements of R[] (if any)
    while (j < n2) {
        arr[k++] = R[j++];
    }

    // Free temporary memory
    delete[] L;
    delete[] R;
}
```

```
void merge_sort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        #pragma omp task
        merge_sort(arr, l, m);
        #pragma omp task
        merge_sort(arr, m + 1, r);
        #pragma omp taskwait
        merge(arr, l, m, r);
    }
}

void parallel_merge_sort(int arr[], int n) {
    #pragma omp parallel
    {
        #pragma omp single
        merge_sort(arr, 0, n - 1);
    }
}

int main() {
    int n;
    cout << "Enter the number of elements: ";
    cin >> n;

    int* arr = new int[n];  // Dynamic array allocation

    cout << "Enter " << n << " elements:\n";
    for (int i = 0; i < n; ++i) {
        cin >> arr[i];
    }

    double start, end;

    // Measure performance of sequential merge sort
    start = omp_get_wtime();
    merge_sort(arr, 0, n - 1);
    end = omp_get_wtime();
    cout << "\nSequential merge sort time: " << end - start << " seconds\n";
```

```
    // Reset array for parallel sort
    cout << "Re-enter " << n << " elements for parallel sort:\n";
    for (int i = 0; i < n; ++i) {
        cin >> arr[i];
    }

    // Measure performance of parallel merge sort
    start = omp_get_wtime();
    parallel_merge_sort(arr, n);
    end = omp_get_wtime();
    cout << "Parallel merge sort time: " << end - start << " seconds\n";

    cout << "Sorted array: ";
    for (int i = 0; i < n; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;

    delete[] arr;  // Free allocated memory
    return 0;
}
```

Assignment no - 3

```
#include <iostream>
#include <omp.h>
#include <climits>

using namespace std;

// Function to find the minimum value in an array using parallel reduction
void min_reduction(int arr[], int n) {
  int min_value = INT_MAX;  // Initialize min_value to positive infinity

  // Use OpenMP parallel for loop with reduction clause (min)
  #pragma omp parallel for reduction(min: min_value)
  for (int i = 0; i < n; i++) {
    if (arr[i] < min_value) {
```

```
      min_value = arr[i];  // Update min_value if a smaller element is found
    }
  }

  cout << "Minimum value: " << min_value << endl;
}

// Function to find the maximum value in an array using parallel reduction
void max_reduction(int arr[], int n) {
  int max_value = INT_MIN;  // Initialize max_value to negative infinity

  // Use OpenMP parallel for loop with reduction clause (max)
  #pragma omp parallel for reduction(max: max_value)
  for (int i = 0; i < n; i++) {
    if (arr[i] > max_value) {
      max_value = arr[i];  // Update max_value if a larger element is found
    }
  }

  cout << "Maximum value: " << max_value << endl;
}

// Function to calculate the sum of elements in an array using parallel reduction
void sum_reduction(int arr[], int n) {
  int sum = 0;

  // Use OpenMP parallel for loop with reduction clause (+)
  #pragma omp parallel for reduction(+: sum)
  for (int i = 0; i < n; i++) {
    sum += arr[i];  // Add each element to the sum
  }

  cout << "Sum: " << sum << endl;
}

// Function to calculate the average of elements in an array using parallel reduction
void average_reduction(int arr[], int n) {
  int sum = 0;
```

```
  // Use OpenMP parallel for loop with reduction clause (+)
  #pragma omp parallel for reduction(+: sum)
  for (int i = 0; i < n; i++) {
    sum += arr[i];  // Add each element to the sum
  }

  // Calculate average using the reduced sum (note: consider division by n-1 for unbiased
average)
  double average = (double)sum / (n - 1);
  cout << "Average: " << average << endl;
}

int main() {
  int n;
  cout << "\nEnter the total number of elements: ";
  cin >> n;

  int *arr = new int[n]; // Allocate memory for the array

  for (int i = 0; i < n; i++) {
    cout << "Enter element : ";
    cin >> arr[i];
  }

  min_reduction(arr, n);
  max_reduction(arr, n);
  sum_reduction(arr, n);
  average_reduction(arr, n);

  delete[] arr;  // Deallocate memory after use
  return 0;
}
```

Assignment no - 4

```
!ls /usr/local

!which nvcc
```

```
!nvidia-smi

%%writefile vector_add.cu

#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>

#define N 1000000

// CUDA Kernel to perform vector addition
__global__ void vectorAdd(int* A, int* B, int* C, int n) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n) {
    C[i] = A[i] + B[i];
  }
}

// Fill array with random integers
void fillArray(int *arr, int n){
  for (int i = 0; i < n; i++) {
    arr[i] = rand() % 100;
  }
}

int main() {
  int size = N * sizeof(int);

  // Allocate memory on host
  int *h_A = (int*)malloc(size);
  int *h_B = (int*)malloc(size);
  int *h_C = (int*)malloc(size);

  // Initialize arrays on host
  fillArray(h_A, N);
  fillArray(h_B, N);

  // Allocate memory on device
  int *d_A, *d_B, *d_C;
```

```
  cudaMalloc((void**)&d_A, size);
  cudaMalloc((void**)&d_B, size);
  cudaMalloc((void**)&d_C, size);

  // Copy data from host to device
  cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

  // Launch kernel on GPU
  int threadsPerBlock = 256;
  int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
  vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

  // Copy result back to host
  cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

  // Print the first 10 elements of the result
  printf("Vector Addition Result (first 10 element):\n");
  for (int i = 0; i < 10; i++) {
    printf("%d + %d = %d\n", h_A[i], h_B[i], h_C[i]);
  }

  // Free memory
  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);
  free(h_A);
  free(h_B);
  free(h_C);

  return 0;
}
```

```
!nvcc -arch=sm_75 vector_add.cu -o vector_add
```

```
!./vector_add
```

```
%%writefile matrix_mul.cu
```

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>

#define N 16

// CUDA Kernel to perform matrix multiplication
__global__ void matrixMul(int *A, int *B, int *C, int width) {
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;

  // Check for valid matrix indices within bounds
  if (row < width && col < width) {
    int sum = 0;
    for (int k = 0; k < width; ++k) {
      sum += A[row * width + k] * B[k * width + col];
    }
    C[row * width + col] = sum;
  }
}

void fillMatrix(int *matrix, int width) {
  for (int i = 0; i < width * width; i++) {
    matrix[i] = rand() % 10;
  }
}

void printMatrix(int *matrix, int width) {
  for (int i = 0; i < width; i++) {
    for (int j = 0; j < width; j++) {
      printf("%4d ", matrix[i * width + j]);
    }
    printf("\n");
  }
}

int main() {
  int size = N * N * sizeof(int);
```

```
// Allocate memory on host
int *h_A = (int*)malloc(size);
int *h_B = (int*)malloc(size);
int *h_C = (int*)malloc(size);

// Initialize matrices on host
fillMatrix(h_A, N);
fillMatrix(h_B, N);

// Allocate memory on device
int *d_A, *d_B, *d_C;
cudaMalloc((void**)&d_A, size);
cudaMalloc((void**)&d_B, size);
cudaMalloc((void**)&d_C, size);

// Copy data from host to device
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

// Define grid and block dimensions
dim3 dimBlock(16, 16);
dim3 dimGrid((N + dimBlock.x - 1) / dimBlock.x, (N + dimBlock.x - 1) / dimBlock.x);

//Launch kernel on GPU
matrixMul<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, N);

// Copy result back to host
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

//Print results
printf("Matrix A:\n");
printMatrix(h_A, N);
printf("\nMatrix B:\n");
printMatrix(h_B, N);
printf("\nMatrix C (A x B):\n");
printMatrix(h_C, N);

//Free memory
cudaFree(d_A);
```

```
  cudaFree(d_B);
  cudaFree(d_C);
  free(h_A);
  free(h_B);
  free(h_C);

  return 0;
}

!nvcc -arch=sm_75 matrix_mul.cu -o matrix_mul

!./matrix_mul
```

Assignment no - 5

```
# Import required libraries
import tensorflow as tf
from mpi4py import MPI

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

# Define the neural network architecture
def create_model():
    model = tf.keras.models.Sequential([
        tf.keras.layers.Conv2D(32, (3, 3),
        activation='relu',
        input_shape=(28, 28, 1)),
        tf.keras.layers.MaxPooling2D((2, 2)),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(10, activation='softmax')
    ])
    return model

# Load and preprocess the MNIST dataset
def load_data():
    mnist = tf.keras.datasets.mnist
```

```
    (x_train, y_train), (x_test, y_test) = mnist.load_data()
    # Normalize pixel values to [0, 1]
    x_train, x_test = x_train / 255.0, x_test / 255.0
    return (x_train, y_train), (x_test, y_test)

# Training function with distributed data processing
def train_model(model, x_train, y_train, rank, size):
    # Split data across nodes
    n = len(x_train)
    chunk_size = n // size
    start = rank * chunk_size
    end = (rank + 1) * chunk_size

    # Handle remainder for last node
    if rank == size - 1:
        end = n

    x_train_chunk = x_train[start:end]
    y_train_chunk = y_train[start:end]

    # Compile the model
    model.compile(
        optimizer='adam',
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )

    # Train the model on local data
    model.fit(x_train_chunk, y_train_chunk, epochs=1, batch_size=32)

    # Evaluate local accuracy
    _, train_acc = model.evaluate(x_train_chunk, y_train_chunk, verbose=2)

    # Aggregate accuracy across all nodes
    global_train_acc = comm.allreduce(train_acc, op=MPI.SUM)
    return global_train_acc / size

# Main execution block
if __name__ == "__main__":
```

```
# Create model and load data
model = create_model()
(x_train, y_train), (x_test, y_test) = load_data()

# Training parameters
epochs = 5

# Training loop
for epoch in range(epochs):
    # Distributed training
    train_acc = train_model(model, x_train, y_train, rank, size)

    # Evaluate on test data
    _, test_acc = model.evaluate(x_test, y_test, verbose=2)
    global_test_acc = comm.allreduce(test_acc, op=MPI.SUM)

    # Print results from rank 0 only
    if rank == 0:
        print(f"\nEpoch {epoch + 1}/{epochs}")
        print(f"Train Accuracy: {train_acc:.4f}")
        print(f"Test Accuracy:  {global_test_acc/size:.4f}")
        print("-" * 40)
```

DL program

Assignment no - 1

```
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score
```

```
import matplotlib.pyplot as plt
import seaborn as sns

# Load the dataset
data = pd.read_csv('/content/sample_data/Boston.csv')

# Assume the target column is named 'MEDV'
X = data.drop("MEDV", axis=1)
Y = data["MEDV"]

data.head()

data.shape

data.describe()

# Preprocess the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split the data
X_train, X_test, Y_train, Y_test = train_test_split(X_scaled, Y, test_size=0.2,
random_state=42)

# Build the model
model = Sequential()
model.add(Dense(128, activation='relu', input_shape=(X_train.shape[1],)))
model.add(Dense(64, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(1))  # Output layer for regression

# Compile the model
model.compile(loss='mse', optimizer='adam', metrics=['mae'])

# Train the model
history = model.fit(X_train, Y_train, epochs=100, batch_size=1, verbose=1, validation_data=
(X_test, Y_test))

# Evalute the model
```

```
mse = model.evaluate(X_test, Y_test)
print("Mean Squared Error:", mse)

# Predictions
y_pred = model.predict(X_test)
print(y_pred[:5])
# Visualizing Predicted vs Actual Pricesplt.figure(figsize=(10, 6))
plt.scatter(Y_test, y_pred, alpha=0.6)
plt.plot([Y_test.min(), Y_test.max()], [Y_test.min(), Y_test.max()], 'r--', lw=2)
plt.title('Predicted vs Actual Prices')
plt.xlabel('Actual Prices')
plt.ylabel('Predicted Prices')
plt.xlim([0, 60])
plt.ylim([0, 60])
plt.grid()
plt.show()
```

Assignment no - 2

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing.sequence import pad_sequences
import matplotlib.pyplot as plt

# Load IMDB dataset
vocab_size = 10000
maxlen = 200
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=vocab_size)

# Pad sequences to ensure uniform input length
x_train = pad_sequences(x_train, maxlen=maxlen)
x_test = pad_sequences(x_test, maxlen=maxlen)

# Build the model
model = keras.Sequential([
    layers.Embedding(input_dim=vocab_size, output_dim=32, input_length=maxlen),
    layers.GlobalAveragePooling1D(),
```

```
    layers.Dense(64, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Train the model
history = model.fit(x_train, y_train, epochs=10, batch_size=512, validation_split=0.2,
verbose=1)

# Evaluate the model
loss, accuracy = model.evaluate(x_test, y_test, verbose=1)
print(f"Test Accuracy: {accuracy:.4f}")

# Plot accuracy and loss
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

# Show sample predictions
y_pred_probs = model.predict(x_test[:10])
y_pred_classes = (y_pred_probs > 0.5).astype("int32")

for i in range(10):
```

```
    print(f"Review {i+1} - Predicted: {'Positive' if y_pred_classes[i][0] == 1 else 'Negative'}, Actual: {'Positive' if y_test[i] == 1 else 'Negative'}")
```

Assignment no - 3

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt
import numpy as np

# Load Fashion MNIST dataset
fashion_mnist = keras.datasets.fashion_mnist
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

# Normalize the data
x_train = x_train / 255.0
x_test = x_test / 255.0

# Class names
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

# Build the model
model = keras.Sequential([
    layers.Reshape((28, 28, 1), input_shape=(28, 28)),
    layers.Conv2D(32, (3, 3), activation='relu'),
    layers.MaxPooling2D(2, 2),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D(2, 2),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

```
# Train the model
history = model.fit(x_train, y_train, epochs=10, validation_split=0.2)

# Evaluate the model
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test accuracy: {test_acc:.4f}")

# Plot training history
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

# Predict on test images
predictions = model.predict(x_test[:5])

# Show predictions for first 5 test images
for i in range(5):
    plt.imshow(x_test[i], cmap='gray')
    plt.title(f"Predicted: {class_names[np.argmax(predictions[i])]} | Actual:
{class_names[y_test[i]]}")
    plt.axis('off')
    plt.show()
```

Assignment no - 4

```
import tensorflow as tf
```

```
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from sklearn.preprocessing import MinMaxScaler
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Load Google stock price dataset
data = pd.read_csv('/content/sample_data/Google_Stock_Price_Train.csv')

# Assume 'Close' is the target column for prediction
data = data[["Close"]]
data.dropna(inplace=True)

# Normalize the data
scaler = MinMaxScaler(feature_range=(0, 1))
data_scaled = scaler.fit_transform(data)

# Create sequences for time series prediction
def create_sequences(data, sequence_length):
    X, Y = [], []
    for i in range(sequence_length, len(data)):
        X.append(data[i-sequence_length:i, 0])
        Y.append(data[i, 0])
    return np.array(X), np.array(Y)

sequence_length = 60
X, Y = create_sequences(data_scaled, sequence_length)

# Reshape input to be 3D for RNN [samples, time steps, features]
X = np.reshape(X, (X.shape[0], X.shape[1], 1))

# Split into training and test sets
split = int(0.8 * len(X))
X_train, X_test = X[:split], X[split:]
Y_train, Y_test = Y[:split], Y[split:]
```

```
# Build RNN model
model = keras.Sequential([
    layers.SimpleRNN(50, return_sequences=True, input_shape=(X_train.shape[1], 1)),
    layers.SimpleRNN(50),
    layers.Dense(1)
])

# Compile model
model.compile(optimizer='adam', loss='mean_squared_error')

# Train model
history = model.fit(X_train, Y_train, epochs=50, batch_size=32, validation_split=0.1,
verbose=1)

# Evaluate model
loss = model.evaluate(X_test, Y_test)
print(f"Test Loss: {loss:.4f}")

# Predict
predicted_stock_price = model.predict(X_test)

# Inverse transform predictions
predicted_stock_price = scaler.inverse_transform(predicted_stock_price.reshape(-1, 1))
y_test_scaled = scaler.inverse_transform(Y_test.reshape(-1, 1))

# Plot results
plt.figure(figsize=(10, 6))
plt.plot(y_test_scaled, color='blue', label='Actual Google Stock Price')
plt.plot(predicted_stock_price, color='red', label='Predicted Google Stock Price')
plt.title('Google Stock Price Prediction')
plt.xlabel('Time')
plt.ylabel('Stock Price')
plt.legend()
plt.show()
```