DFS:=

```
#include <iostream>
#include <vector>
#include <stack>
#include <omp.h>
using namespace std;
const int MAX = 100000;
vector<int> graph[MAX];
bool visited[MAX];
void dfs(int node) {
stack<int> s;
s.push(node);
while (!s.empty()) {
int curr_node = s.top();
if (!visited[curr_node]) {
visited[curr_node] = true;
s.pop();
cout<<curr_node<<" ";
#pragma omp parallel for
for (int i = 0; i < graph[curr_node].size(); i++) {
int adj_node = graph[curr_node][i];
if (!visited[adj_node]) {
s.push(adj_node);
}
}
}
}
}
int main() {
int n, m, start_node;
cout<<"Enter no. of Node,no. of Edges and Starting Node of graph:\n";
cin >> n >> m >> start_node;
//n: node,m:edges
cout<<"Enter pair of node and edges:\n";
for (int i = 0; i < m; i++) {
```

```
int u, v;
cin >> u >> v;
//u and v: Pair of edges
graph[u].push_back(v);
graph[v].push_back(u);
}
#pragma omp parallel for
for (int i = 0; i < n; i++) {
visited[i] = false;
}
dfs(start_node);
return 0;
}
```

BFS :=

```
#include<iostream>

#include<stdlib.h>

#include<queue>

using namespace std;

class node

{

public:

node *left, *right;

int data;

};

class Breadthfs

{

public:

node *insert(node *, int);

void bfs(node *);

};

node *insert(node *root, int data)

// inserts a node in tree

{

if(!root)

{

root=new node;

root->left=NULL;

root->right=NULL;

root->data=data;

return root;

}

queue<node *> q;

q.push(root);

while(!q.empty())

{

node *temp=q.front();

q.pop();

if(temp->left==NULL)
```

```
{
temp->left=new node;
temp->left->left=NULL;
temp->left->right=NULL;
temp->left->data=data;
return root;
}
else
{
q.push(temp->left);
}
if(temp->right==NULL)
{
temp->right=new node;
temp->right->left=NULL;
temp->right->right=NULL;
temp->right->data=data;
return root;
}
else
{
q.push(temp->right);
}
}
}
void bfs(node *head)
{
queue<node*> q;
q.push(head);
int qSize;
while (!q.empty())
{
qSize = q.size();
#pragma omp parallel for
//creates parallel threads
```

```
for (int i = 0; i < qSize; i++)
{
node* currNode;
#pragma omp critical
{
currNode = q.front();
q.pop();
cout<<"\t"<<currNode->data;
}// prints parent node
#pragma omp critical
{
if(currNode->left)// push parent's left node in queue
q.push(currNode->left);
if(currNode->right)
q.push(currNode->right);
}// push parent's right node in queue
}
}
}
int main(){
node *root=NULL;
int data;
char ans;
do
{
cout<<"\n enter data=>";
cin>>data;
root=insert(root,data);
cout<<"do you want insert one more node?";
cin>>ans;
}while(ans=='y'||ans=='Y');
bfs(root);
return 0;
}
```

Paralle Bubble Sort :=

```
import time

import random


start = time.perf_counter()


def Parallel_Bubble_sort(lst):

 Sorted = 0

 n = len(lst)

 while Sorted == 0:

  Sorted = 1

  for i in range(0, n-1, 2):

   if lst[i] > lst[i+1]:

    lst[i], lst[i+1] = lst[i+1], lst[i]

    Sorted = 0

   for i in range(1, n-1, 2):

    if lst[i] > lst[i+1]:

     lst[i], lst[i+1] = lst[i+1], lst[i]

     Sorted = 0

 print(lst)


lst = [(random.randint(0,100)) for i in range(100)]

Parallel_Bubble_sort(lst)

finish = time.perf_counter()

print(f'Finished in {round(finish-start,2)} second(s)')
```

MERGE SORT :=

```
def merge(arr, l, m, r):
 n1 = m - l + 1
 n2 = r - m
# create temp arrays
 L = [0] * (n1)
 R = [0] * (n2)
 for i in range(0, n1):
  L[i] = arr[l + i]
  for j in range(0, n2):
   R[j] = arr[m + 1 + j]
 i = 0 # Initial index of first subarray
 j = 0 # Initial index of second subarray
 k = l # Initial index of merged subarray
 while i < n1 and j < n2:
  if L[i] <= R[j]:
   arr[k] = L[i]
   i += 1
  else:
   arr[k] = R[j]
   j += 1
   k += 1

 while i < n1:
  arr[k] = L[i]
  i += 1
  k += 1

 while j < n2:
  arr[k] = R[j]
  j += 1
  k += 1

def mergeSort(arr, l, r):
 if l < r:
```

```
    m = l+(r-l)//2

    mergeSort(arr, l, m)

    mergeSort(arr, m+1, r)

    merge(arr, l, m, r)


# Driver code to test above

arr = [12, 11, 13, 5, 6, 7]

n = len(arr)

print("Given array is")

for i in range(n):

  print("%d" % arr[i],end=" ")


mergeSort(arr, 0, n-1)

print("\n\nSorted array is")

for i in range(n):

  print("%d" % arr[i],end=" ")
```

Implement Min, Max, Sum and Average operations using Parallel Reduction :=

```
#include <iostream>

#include <vector>

#include <omp.h>

#include <climits>

using namespace std;

void min_reduction(vector<int>& arr) {

int min_value = INT_MAX;

#pragma omp parallel for reduction(min: min_value)

for (int i = 0; i < arr.size(); i++) {

if (arr[i] < min_value) {

min_value = arr[i];

}

}

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

}

void max_reduction(vector<int>& arr) {

int max_value = INT_MIN;

#pragma omp parallel for reduction(max: max_value)

for (int i = 0; i < arr.size(); i++) {

if (arr[i] > max_value) {

max_value = arr[i];

}

}

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

}

void sum_reduction(vector<int>& arr) {

int sum = 0;

#pragma omp parallel for reduction(+: sum)

for (int i = 0; i < arr.size(); i++) {

sum += arr[i];

}

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

}
```

```
void average_reduction(vector<int>& arr) {

int sum = 0;

#pragma omp parallel for reduction(+: sum)

for (int i = 0; i < arr.size(); i++) {

sum += arr[i];

}

cout << "Average: " << (double)sum / arr.size() << endl;

}

int main() {

vector<int> arr;

arr.push_back(5);

arr.push_back(2);

arr.push_back(9);

arr.push_back(1);

arr.push_back(7);

arr.push_back(6);

arr.push_back(8);

arr.push_back(3);

arr.push_back(4);

min_reduction(arr);

max_reduction(arr);

sum_reduction(arr);

average_reduction(arr);

}
```

Linear Regression :=

```
import pandas as pd
# Load the dataset from a CSV file
df = pd.read_csv('/content/HousingData.csv')
# Display the first few rows of the dataset
print(df.head())
#Step 2: Preprocess the data
from sklearn.preprocessing import StandardScaler # Split the data into input and output variables
X = df.drop('MEDV', axis=1)
y = df['MEDV']
# Scale the input features
scaler = StandardScaler()
X = scaler.fit_transform(X)
# Display the first few rows of the scaled input features print(X[:5])
#Step 3: Split the dataset
from sklearn.model_selection import train_test_split # Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
# Print the shapes of the training and testing sets
print('Training set shape:', X_train.shape, y_train.shape)
print('Testing set shape:', X_test.shape, y_test.shape)
#Step 4: Define the model architecture
from keras.models import Sequential
from keras.layers import Dense, Dropout
# Define the model architecture
model = Sequential()
model.add(Dense(64, input_dim=13, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(32, activation='relu'))
model.add(Dense(1))
# Display the model summary
print(model.summary())
#Step 5: Compile the model
# Compile the model
model.compile(loss='mean_squared_error', optimizer='adam', metrics=['mean_absolute_error'])
```

```
#Step 6: Train the model

from keras.callbacks import EarlyStopping # Train the model

early_stopping = EarlyStopping(monitor='val_loss', patience=5)

history = model.fit(X_train, y_train, validation_split=0.2, epochs=100, batch_size=32, callbacks=[early_stopping])

# Plot the training and validation loss over epochs

import matplotlib.pyplot as plt

plt.plot(history.history['loss'])

plt.plot(history.history['val_loss'])

plt.title('Model Loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend(['Training', 'Validation'])

plt.show()

#Step 7: Evaluate the model

# Evaluate the model on the testing set

loss, mae = model.evaluate(X_test, y_test)

# Print the mean absolute error

print('Mean Absolute Error:', mae)
```

Movie Review Classification using IMDB Dataset :=

```
import numpy as np

from keras.datasets import imdb

from keras.preprocessing.sequence import pad_sequences

from keras.models import Sequential

from keras.layers import Embedding, Bidirectional, LSTM, Dense # Load the IMDB dataset

(x_train, y_train), (x_test, y_test) = imdb.load_data()

# Pad or truncate the sequences to a fixed length of 250 words

max_len = 250

x_train = pad_sequences(x_train, maxlen=max_len)

x_test = pad_sequences(x_test, maxlen=max_len)

# Define the deep neural network architecture

model = Sequential()

model.add(Embedding(input_dim=10000, output_dim=128, input_length=max_len))

model.add(Bidirectional(LSTM(64, return_sequences=True)))

model.add(Bidirectional(LSTM(32)))

model.add(Dense(1, activation='sigmoid')) # Compile the model

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

history = model.fit(x_train, y_train, epochs=10, batch_size=128, validation_split=0.2) # Evaluate the model on the test set

loss, acc = model.evaluate(x_test, y_test, batch_size=128)

print(f'Test accuracy: {acc:.4f}, Test loss: {loss:.4f}')
```

CNN MNIST Fashhion Dataset :=

```
import tensorflow as tf

from tensorflow import keras

import numpy as np

import matplotlib.pyplot as plt # Load the dataset

fashion_mnist = keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data() # Normalize the images

train_images = train_images / 255.0

test_images = test_images / 255.0

# Define the model

model = keras.Sequential([ keras.layers.Flatten(input_shape=(28, 28)), keras.layers.Dense(128, activation='relu'),
keras.layers.Dense(10, activation='softmax')])

# Compile the model

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

# Train the model

model.fit(train_images, train_labels, epochs=10) # Evaluate the model

test_loss, test_acc = model.evaluate(test_images, test_labels)

print('Test accuracy:', test_acc)

# Make predictions

predictions = model.predict(test_images)

predicted_labels = np.argmax(predictions, axis=1)

# Show some example images and their predicted labels

num_rows = 5

num_cols = 5

num_images = num_rows * num_cols

plt.figure(figsize=(2 * 2 * num_cols, 2 * num_rows))

for i in range(num_images):

  plt.subplot(num_rows, 2 * num_cols, 2 * i + 1)

  plt.imshow(test_images[i], cmap='gray')

  plt.axis('off')

  plt.subplot(num_rows, 2 * num_cols, 2 * i + 2)

  plt.bar(range(10), predictions[i])

  plt.xticks(range(10))

  plt.ylim([0, 1])
```

```
plt.tight_layout()

plt.title(f"Predicted label: {predicted_labels[i]}")

plt.show()
```

CUDA Program for Addition of Two Large Vectors:

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

// CUDA kernel for 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];

}

}

int main() {

int n = 1000000; // Vector size int *a, *b, *c; // Host vectors

int *d_a, *d_b, *d_c; // Device vectors int size = n * sizeof(int); // Size in bytes

// Allocate memory for host vectors a = (int*) malloc(size);

b = (int*) malloc(size); c = (int*) malloc(size);

// Initialize host vectors for (int i = 0; i < n; i++) {

a[i] = i;

b[i] = i;}

// Allocate memory for device vectors cudaMalloc((void**) &d_a, size);

cudaMalloc((void**) &d_b, size); cudaMalloc((void**) &d_c, size);

// Copy host vectors to device vectors

cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice); cudaMemcpy(d_b, b, size,

cudaMemcpyHostToDevice);

// Define block size and grid size int blockSize = 256;

int gridSize = (n + blockSize - 1) / blockSize;

// Launch kernel

vectorAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);

// Copy device result vector to host result vector cudaMemcpy(c, d_c, size,

cudaMemcpyDeviceToHost);

// Verify the result

for (int i = 0; i < n; i++) { if (c[i] != 2*i) {

printf("Error: c[%d] = %d\n", i, c[i]); break;

}

}

// Free device memory cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);

// Free host memory free(a); free(b);

free(c);
```

```
return 0;

}
```

Cuda Program for matrix multiplication :=

```
#include <stdio.h> #define BLOCK_SIZE 16

 global void matrix_multiply(float *a, float *b, float *c, int n)

{

int row = blockIdx.y * blockDim.y + threadIdx.y; int col = blockIdx.x * blockDim.x + threadIdx.x; float

sum = 0;

if (row < n && col < n) { for (int i = 0; i < n; ++i) {

sum += a[row * n + i] * b[i * n + col];

}

c[row * n + col] = sum;}

}

int main(){

int n = 1024;

size_t size = n * n * sizeof(float); float *a, *b, *c;

float *d_a, *d_b, *d_c; cudaEvent_t start, stop; float elapsed_time;

// Allocate host memory

a = (float*)malloc(size);

b = (float*)malloc(size);

c = (float*)malloc(size);

// Initialize matrices

for (int i = 0; i < n * n; ++i) { a[i] = i % n;

b[i] = i % n;

}

// Allocate device memory

cudaMalloc(&d_a, size);

cudaMalloc(&d_b, size);

cudaMalloc(&d_c, size);

// Copy input data to device

cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);

cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Set kernel launch configuration

dim3 threads(BLOCK_SIZE, BLOCK_SIZE);

dim3 blocks((n + threads.x - 1) / threads.x, (n + threads.y - 1) / threads.y);

// Launch kernel cudaEventCreate(&start); cudaEventCreate(&stop); cudaEventRecord(start);

matrix_multiply<<<blocks, threads>>>(d_a, d_b, d_c, n); cudaEventRecord(stop);
```

```
cudaEventSynchronize(stop); cudaEventElapsedTime(&elapsed_time, start, stop);

// Copy output data to host

cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Print elapsed time

printf("Elapsed time: %f ms\n", elapsed_time);

// Free device memory cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);

// Free host memory free(a); free(b);

free(c);

return 0;

}
```