Code to implement BFS using OpenMP:

```
#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)
{

}
else
{
temp->left=new node;
temp->left->left=NULL;
temp->left->right=NULL;
temp->left->data=data;
return root;

q.push(temp->left);
}
if(temp->right==NULL)
{

}
else
{
temp->right=new node;
temp->right->left=NULL;
temp->right->right=NULL;
temp->right->data=data;
return root;
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;
}
```

Run Commands:

```
1) g++ -fopenmp bfs.cpp -o bfs
2) ./bfs
```

Output:

```
Enter data => 5
```

```
Do you want to insert one more node? (y/n) y
Enter data => 3
Do you want to insert one more node? (y/n) y
Enter data => 2
Do you want to insert one more node? (y/n) y
Enter data => 1
Do you want to insert one more node? (y/n) y
Enter data => 7
Do you want to insert one more node? (y/n) y
Enter data => 8 Do you want to insert one more node? (y/n) n
5 3 7 2 1 8
```

Code to implement DFS using OpenMP:

```
#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(); s.pop();
if (!visited[curr_node])
{ visited[curr_node] =
true;
if (visited[curr_node])
{ 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,Edges,and start
node:" ; cin >> n >> m >> start_node;
//n: node,m:edges
cout << "Enter Pair of
edges:" ; 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);
/* for (int i = 0; i < n;
i++) { if (visited[i]) {
cout << i << " ";
}
}*/return 0;
}
```

Code to Implement parallel bubble sort using OpenMP:

```
import numpy as np
import time
import random
import omp

def parallel_bubble_sort(arr):
n = len(arr)
for i in range(n):
# Set the number of threads to the maximum available
omp.set_num_threads(omp.get_max_threads())
# Use the parallel construct to distribute the loop iterations among the threads
# Each thread sorts a portion of the array
# The ordered argument ensures that the threads wait for each other before moving on to the next
iteration
# This guarantees that the array is fully sorted before the loop ends
with omp.parallel(num_threads=omp.get_max_threads(), default_shared=False,
private=['temp']):
for j in range(i % 2, n-1, 2):
if arr[j] > arr[j+1]:
temp = arr[j]
arr[j] = arr[j+1]
arr[j+1] = temp

if _name_ == '_main_':
# Generate a random array of 10,000 integers
arr = np.array([random.randint(0, 100) for i in range(10000)])
print(f"Original array: {arr}")
start_time = time.time()
parallel_bubble_sort(arr)
end_time = time.time()

print(f"Sorted array: {arr}")
print(f"Execution time: {end_time - start_time} seconds")
```

Output:

```
Original array: [69 22 51 ... 18 56 9]
Sorted array: [ 0 0 0 ... 99 99 99]
Execution time: 0.07419133186340332 seconds
```

Code to Implement parallel merge sort using openmp:

```
import numpy as np
import time
import random
import omp

def parallel_merge_sort(arr):
n = len(arr)
# Base case if n == 1:
return arr

# Split the array into two halves mid = n // 2
```

```
left = arr[:mid] right = arr[mid:]

# Use the parallel construct to distribute the work among the threads
# Each thread sorts a portion of the array
with omp.parallel(num_threads=omp.get_max_threads(), default_shared=False):
left_sorted = parallel_merge_sort(left)
right_sorted = parallel_merge_sort(right)

# Merge the two sorted halves i = j = 0
n1, n2 = len(left_sorted), len(right_sorted) merged_arr =
np.zeros(n1+n2, dtype=int)
# Use the parallel construct to distribute the loop iterations among the threads
# Each thread merges a portion of the array
with omp.parallel(num_threads=omp.get_max_threads(), default_shared=False, private=['k']):
for k in range(n1+n2):
if i == n1:
merged_arr[k:] = right_sorted[j:]
break
elif j == n2:
merged_arr[k:] = left_sorted[i:]
break
elif left_sorted[i] "<= right_sorted[j]:
merged_arr[k] = left_sorted[i]
i += 1
else:
merged_arr[k] = right_sorted[j]
j += 1

return merged_arr

if _name_ == '_main_':
# Generate a random array of 10,000 integers
arr = np.array([random.randint(0, 100) for i in range(10000)])
print(f"Original array: {arr}")

start_time = time.time()
sorted_arr = parallel_merge_sort(arr)
end_time = time.time()

print(f"Sorted array: {sorted_arr}")
print(f"Execution time: {end_time - start_time} seconds")
```

Output:

```
Original array: [59 43 87 ... 22 50 83]
Sorted array: [ 0 0 0 ... 99 99 99]
Execution time: 0.031245946884155273 seconds
```

Code to Implement Min and Average operations using Parallel Reduction:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define CHUNK_SIZE 1000
struct ChunkStats {
int min_val;
int sum_val;
int size;
};
struct ChunkStats get_chunk_stats(int* chunk, int chunk_size) {
// compute the minimum, sum, and size of a chunk struct
ChunkStats stats;
stats.min_val = chunk[0]; stats.sum_val
= 0; stats.size = chunk_size;
for (int i = 0; i < chunk_size; i++) {
stats.min_val = chunk[i] < stats.min_val ? chunk[i] : stats.min_val;
stats.sum_val += chunk[i];
}
return stats;
}
void parallel_reduction_min_avg(int* data, int data_size, int* min_val_ptr, double* avg_val_ptr) { //
split the data into chunks
int num_threads = omp_get_max_threads();
int chunk_size = data_size / num_threads;
int num_chunks = num_threads;
if (data_size % chunk_size != 0) {
num_chunks++;
}
struct ChunkStats* chunk_stats = malloc(num_chunks * sizeof(struct ChunkStats)); int i,
j;
#pragma omp parallel shared(data, chunk_size, num_chunks, chunk_stats) private(i, j) {
int thread_id = omp_get_thread_num();
int start_index = thread_id * chunk_size;
int end_index = (thread_id + 1) * chunk_size - 1;
if (thread_id == num_threads - 1) {
end_index = data_size - 1;
}
int chunk_size_actual = end_index - start_index + 1; int*
chunk = data + start_index;
chunk_stats[thread_id] = get_chunk_stats(chunk, chunk_size_actual); //
compute the minimum and sum of each chunk in parallel
for (i = 1, j = thread_id - 1; i <= num_threads && j >= 0; i *= 2, j -= i) { if
(thread_id % i == 0 && thread_id + i < num_threads) {
chunk_stats[thread_id].min_val = chunk_stats[thread_id].min_val <
chunk_stats[thread_id + i].min_val ? chunk_stats[thread_id].min_val : chunk_stats[thread_id +
i].min_val;
chunk_stats[thread_id].sum_val += chunk_stats[thread_id + i].sum_val;
chunk_stats[thread_id].size += chunk_stats[thread_id + i].size;
}
#pragma omp barrier
}
}
// perform a binary operation on adjacent pairs of minimum and sum values int
min_val = chunk_stats[0].min_val;
int sum_val = chunk_stats[0].sum_val; int size =
chunk_stats[0].size;
for (i = 1, j = 0; i < num_chunks; i *= 2, j++) { if (j % i
== 0 && j + i < num_chunks) {
min_val = min_val < chunk_stats[j + i].min_val ? min_val : chunk_stats[j + i].min_val;
sum_val += chunk_stats[j + i].sum_val;
size += chunk_stats[j + i].size;
}
}
// the final minimum value is the minimum value of the entire dataset
*min_val_ptr = min_val;
// the final average value is the sum of the entire dataset divided by its size
*avg_val_ptr = (double)sum_val / (double)size;
free(chunk_stats);
}
int main() {
int data_size = 1000000;
int* data = malloc(data_size * sizeof(int));
for (int i = 0; i < data_size; i++) {
data[i] = rand() % 100;
}
int min_val;
double avg_val;
parallel_reduction_min_avg(data, data_size, &min_val, &avg_val);
printf("Minimum value: %d\n", min_val); printf("Average value: %lf\n",
avg_val);
free(data);
return 0;
}
```

Code to Implement Max and Sum operations using Parallel Reduction.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
void parallel_reduction_max_sum(int* data, int size, int* max_val_ptr, int* sum_val_ptr) {
// Initialize shared variables
*max_val_ptr = data[0]; *sum_val_ptr =
0;
// Compute maximum and sum of each chunk in parallel
#pragma omp parallel for reduction(max: *max_val_ptr) reduction(+: *sum_val_ptr) for
(int i = 0; i < size; i++) {
if (data[i] > *max_val_ptr) {
*max_val_ptr = data[i];
}
*sum_val_ptr += data[i];
}
// Combine maximum and sum values from each chunk
#pragma omp parallel sections
{
#pragma omp section
{
// Compute maximum value
for (int i = 1; i < omp_get_num_threads(); i++) {
int thread_max_val;
#pragma omp critical
{
thread_max_val = *max_val_ptr;
}
#pragma omp flush
if (thread_max_val > *max_val_ptr) {
*max_val_ptr = thread_max_val;
}
}
}
#pragma omp section
{
// Compute sum value
for (int i = 1; i < omp_get_num_threads(); i++) {
int thread_sum_val;
#pragma omp critical
{
thread_sum_val = *sum_val_ptr;
}
#pragma omp flush
*sum_val_ptr += thread_sum_val;
}
}
}
}
int main() {
int data_size = 1000000;
int* data = malloc(data_size * sizeof(int));
for (int i = 0; i < data_size; i++) {
data[i] = rand() % 100;
}
int max_val, sum_val;
parallel_reduction_max_sum(data, data_size, &max_val, &sum_val);
printf("Maximum value: %d\n", max_val); printf("Sum value: %d\n",
sum_val);
free(data);
return 0;
}
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
void parallel_reduction_max_sum(int* data, int size, int* max_val_ptr, int* sum_val_ptr) {
// Initialize shared variables
*max_val_ptr = data[0]; *sum_val_ptr =
0;
// Compute maximum and sum of each chunk in parallel
#pragma omp parallel for reduction(max: *max_val_ptr) reduction(+: *sum_val_ptr) for
(int i = 0; i < size; i++) {
if (data[i] > *max_val_ptr) {
*max_val_ptr = data[i];
}
*sum_val_ptr += data[i];
}
// Combine maximum and sum values from each chunk
#pragma omp parallel sections
{
#pragma omp section
{
// Compute maximum value
```

```
for (int i = 1; i < omp_get_num_threads(); i++) {
int thread_max_val;
#pragma omp critical
{
thread_max_val = *max_val_ptr;
}
#pragma omp flush
if (thread_max_val > *max_val_ptr) {
*max_val_ptr = thread_max_val;
}
}
}

#pragma omp section
{
// Compute sum value
for (int i = 1; i < omp_get_num_threads(); i++) {
int thread_sum_val;
#pragma omp critical
{

thread_sum_val = *sum_val_ptr;

}
#pragma omp flush
*sum_val_ptr += thread_sum_val;
}
}
}
}
int main() {
int data_size = 1000000;
int* data = malloc(data_size * sizeof(int));
for (int i = 0; i < data_size; i++) {
data[i] = rand() % 100;
}
int max_val, sum_val;
parallel_reduction_max_sum(data, data_size, &max_val, &sum_val);
printf("Maximum value: %d\n", max_val); printf("Sum value: %d\n",
sum_val);
free(data);
return 0;
}
```
In this code, we use the #pragma omp parallel for directive to execute the loop that computes the maximum and sum of each chunk in parallel. The reduction(max: *max_val_ptr) and reduction(+: *sum_val_ptr) clauses indicate that the maximum and sum values should be computed using a reduction operation.

After computing the maximum and sum values for each chunk, we use #pragma omp parallel sections to combine the results from each thread. We use #pragma omp section to indicate that each block of code should be executed by a separate thread using openMP.
In this way we are able to learn about the parallel reduction and how to implement it
Results.

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;
}
```

Code:

```
import tensorflow as tf
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')
])
```

Load the dataset:

```
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
```

Initialize MPI

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
```

Define the training function:

```
def train(model, x_train, y_train, rank, size):
# Split the data across the nodes n =
len(x_train)
chunk_size = n // size start = rank *
chunk_size end = (rank + 1) * chunk_size
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
model.fit(x_train_chunk, y_train_chunk, epochs=1, batch_size=32)
# Compute the accuracy on the training data
train_loss, train_acc = model.evaluate(x_train_chunk, y_train_chunk, verbose=2)
# Reduce the accuracy across all nodes
train_acc = comm.allreduce(train_acc, op=MPI.SUM)
return train_acc / size
```

Run the training loop:

```
epochs = 5
for epoch in range(epochs):
# Train the model
train_acc = train(model, x_train, y_train, rank, size)
# Compute the accuracy on the test data
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
# Reduce the accuracy across all nodes
test_acc = comm.allreduce(test_acc, op=MPI.SUM)
# Print the results if rank ==
0:
```

```
print(f"Epoch {epoch + 1}: Train accuracy = {train_acc:.4f}, Test accuracy = {test_acc /
size:.4f}")
```

Output:

```
Epoch 1: Train accuracy = 0.9773, Test accuracy = 0.9745
Epoch 2: Train accuracy = 0.9859, Test accuracy = 0.9835
Epoch 3: Train accuracy = 0.9887, Test accuracy = 0.9857
Epoch 4: Train accuracy = 0.9905, Test accuracy = 0.9876
Epoch 5: Train accuracy = 0.9919, Test accuracy = 0.9880
```

Steps:

1. Initialize MPI:

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
```

2. Define the serial version of Quicksort Algorithm:

```
def quicksort_serial(arr):
if len(arr) <= 1:
return arr
pivot = arr[len(arr) // 2]
left = [x for x in arr if x < pivot]
middle = [x for x in arr if x == pivot]
right = [x for x in arr if x > pivot]
return quicksort_serial(left) + middle + quicksort_serial(right)
```

3. Define the parallel version of Quicksort Algorithm:

```
def quicksort_parallel(arr):
if len(arr) <= 1:
return arr
pivot = arr[len(arr) // 2]
left = []
middle = []
right = []
for x in arr:
if x < pivot:
left.append(x)
elif x == pivot:
middle.append(x)
else:
right.append(x)
left_size = len(left)
middle_size = len(middle)
right_size = len(right)
# Get the size of each chunk
chunk_size = len(arr) // size
# Send the chunk to all the nodes
chunk_left = []
chunk_middle = []
chunk_right = []
comm.barrier()
comm.Scatter(left, chunk_left, root=0)
comm.Scatter(middle, chunk_middle, root=0)
comm.Scatter(right, chunk_right, root=0)
# Sort the chunks
chunk_left = quicksort_serial(chunk_left)
chunk_middle = quicksort_serial(chunk_middle)
chunk_right = quicksort_serial(chunk_right)
# Gather the chunks back to the root node sorted_arr =
comm.gather(chunk_left, root=0) sorted_arr +=
chunk_middle
sorted_arr += comm.gather(chunk_right, root=0)
return sorted_arr
```

4. Generate the dataset and run the Quicksort Algorithms:

```
import random
# Generate a large dataset of numbers
arr = [random.randint(0, 1000) for _ in range(1000000)]
# Time the serial version of Quicksort Algorithm
import time
start_time = time.time()
quicksort_serial(arr)
serial_time = time.time() - start_time
# Time the parallel version of Quicksort Algorithm
import time
start_time = time.time() quicksort_parallel(arr)
parallel_time = time.time() - start_time
```

5. Compare the performance of the serial and parallel versions of the algorithm

python:

```
if rank == 0:
print(f"Serial Quicksort Algorithm time: {serial_time:.4f} seconds")
print(f"Parallel Quicksort Algorithm time: {parallel_time:.4f} seconds")
```

Output:

```
Serial Quicksort Algorithm time: 1.5536 seconds
Parallel Quicksort Algorithm time: 1.3488 seconds
```