Assignment No : 01

```
#include<iostream>
#include<vector>
#include<queue>
#include<omp.h>
using namespace std;
void parallelBFS(vector<vector<int>> &graph, int startIndex){
   int numVertices = graph.size();
   vector<bool> visited(numVertices, false);
   queue<int> bfsQueue;
   visited[startIndex] = true;
   bfsQueue.push(startIndex);
   while(!bfsQueue.empty()){
      int currVertex = bfsQueue.front();
      bfsQueue.pop();
      #pragma omp parallel for
      for(int i=0; i<graph[currVertex].size(); i++){
         int neighbour = graph[currVertex][i];
         while(!visited[neighbour]){
            visited[neighbour] = true;
            bfsQueue.push(neighbour);
         }
      }
   }
   cout<<"Parallel BFS : "<<endl;
   for(int i=0; i<numVertices; i++){
      if(visited[i]){
         cout<<i<<" ";
      }
   }
   cout<<endl;
}
int main() {
   vector<vector<int>> graph = {
      {1, 2},
      {0, 3, 4},
      {0, 5},
      {1},
      {1},
      {2}
   };
   int startVertex = 0;
   parallelBFS(graph, startVertex);
}
```

Assignment No: 02

```
#include <iostream>
#include <vector>
#include <omp.h>
using namespace std;
void parallelBubbleSort(vector<int>& arr) {
   int n = arr.size();
   bool swapped;
   do {
      swapped = false;
      #pragma omp parallel for shared(arr, swapped)
      for (int i = 1; i < n; i++) {
         if (arr[i - 1] > arr[i]) {
            swap(arr[i - 1], arr[i]);
            swapped = true;
         }
      }
      #pragma omp barrier
   } while (swapped);
}
void sequentialBubbleSort(vector<int>& arr) {
   int n = arr.size();
   bool swapped;
   do {
      swapped = false;
      for (int i = 1; i < n; i++) {
         if (arr[i - 1] > arr[i]) {
            swap(arr[i - 1], arr[i]);
            swapped = true;
         }
      }
   } while (swapped);
}
int main() {
   vector<int> arr = {64, 34, 25, 12, 22, 11, 90};
   double startTimeSeq = omp_get_wtime();
   sequentialBubbleSort(arr);
   double endTimeSeq = omp_get_wtime();
   double startTimeParallel = omp_get_wtime();
   parallelBubbleSort(arr);
   double endTimeParallel = omp_get_wtime();
   double elapsedTimeSeq = endTimeSeq - startTimeSeq;
   double elapsedTimeParallel = endTimeParallel - startTimeParallel;
   cout << "Sequential Bubble Sort completed in " << elapsedTimeSeq << " seconds." <<endl;
   cout << "Parallel Bubble Sort completed in " << elapsedTimeParallel << " seconds." <<endl;
   return 0;
}
```

Assignment No:03

```
#include<iostream>
#include<vector>
#include<omp.h>
#include<climits>
using namespace std;
void min_reduction(vector<int> &arr){
   int min_val = INT_MAX;
   #pragma omp parallel for reduction(min : min_val)
   for(int i=0; i<arr.size(); i++){
      if(arr[i] < min_val){
         min_val = arr[i];
      }
   }
   cout<<"Minimum Value is : "<<min_val<<endl;
}
void max_reduction(vector<int> &arr){
   int max_val = INT_MIN;
   #pragma omp parllel for reduction(max : max_val)
   for(int i=0; i<arr.size(); i++){
      if(arr[i] > max_val){
         max_val = arr[i];
      }
   }
   cout<<"Maximum Value is : "<<max_val<<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 is : "<<sum<<endl;
}
void avg_reduction(vector<int> &arr){
   int sum = 0;
   #pragma omp parallel for reduction(+ : sum)
   for(int i=0; i<arr.size(); i++){
      sum += arr.size();
   }
   cout<<"Average is : "<<(double)sum/arr.size()<<endl;
}
int main(){
   vector<int> arr = {5, 2, 9, 6, 3, 7, 4, 8};
   min_reduction(arr);
   max_reduction(arr);
   sum_reduction(arr);
   avg_reduction(arr);
   return 0;
}
```

Assignment No:04

```
#include <iostream>
#include <vector>
#include <cuda_runtime.h>
using namespace std;
__global__ void vectorAddition(const int* a, const int* b, int* c, int size) {
   int tid = blockIdx.x * blockDim.x + threadIdx.x;
   if (tid < size) {
      c[tid] = a[tid] + b[tid];
   }
}
int main() {
   int size = 1000000; // Size of the vectors
   int numBytes = size * sizeof(int);
   vector<int> h_a(size);
   vector<int> h_b(size);
   vector<int> h_c(size);
   for (int i = 0; i < size; i++) {
      h_a[i] = i;
      h_b[i] = i;
   }
   int* d_a;
   int* d_b;
   int* d_c;
   cudaMalloc((void**)&d_a, numBytes);
   cudaMalloc((void**)&d_b, numBytes);
   cudaMalloc((void**)&d_c, numBytes);
   cudaMemcpy(d_a, h_a.data(), numBytes, cudaMemcpyHostToDevice);
   cudaMemcpy(d_b, h_b.data(), numBytes, cudaMemcpyHostToDevice);
   int blockSize = 256;
   int gridSize = (size + blockSize - 1) / blockSize;
   vectorAddition<<<gridSize, blockSize>>>(d_a, d_b, d_c, size);
   cudaMemcpy(h_c.data(), d_c, numBytes, cudaMemcpyDeviceToHost);
   for (int i = 0; i < size; i++) {
      if (h_c[i] != h_a[i] + h_b[i]) {
         std::cout << "Error at index " << i << std::endl;
         break;
      }
   }
   cudaFree(d_a);
   cudaFree(d_b);
   cudaFree(d_c);
   return 0;
}
```