Assignment No. 1

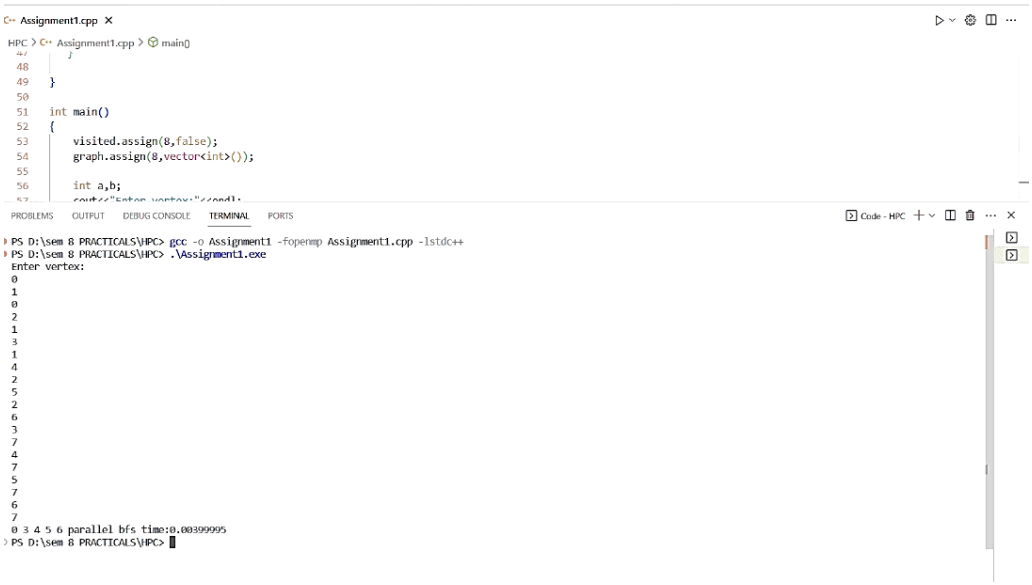
BFS - Code

```
#include <iostream>
#include <vector>
#include <queue>
#include <omp.h>
#include <bits/stdc++.h>
#define pb push_back
using namespace std;
vector<bool> visited;
vector<vector<int>> graph;
void edge(int a,int b)
{
   graph[a].pb(b);
}
void bfs(int start)
{
   queue<int> q;
   q.push(start);
   visited[start] = true;
   #pragma omp parallel
   {
      #pragma omp single
      {
         while (!q.empty())
         {
            int vertex = q.front();
            q.pop();
            cout<<vertex<<" ";
            #pragma omp task firstprivate(vertex)
            {
```

```
                for (auto i=graph[vertex].begin();i!=graph[vertex].end();i++)
                {
                    if (!visited[*i])
                    {
                        q.push(*i);
                        visited[*i] = true;
                    }
                }
            }
        }
    }
}
int main()
{
    visited.assign(8,false);
    graph.assign(8,vector<int>());
    int a,b;
    cout<<"Enter vertex:"<<endl;
    for (int i = 0; i < 10; i++)
    {
        cin >> a >> b;
        edge(a, b);
    }
    double st=omp_get_wtime();
    for (int i = 0; i < 8; i++)
    {
        if (!visited[i])
            bfs(i);
    }
    double et=omp_get_wtime();
```

```
    cout <<"parallel bfs time:"<<et-st <<endl;
    return 0;
}
```

Output



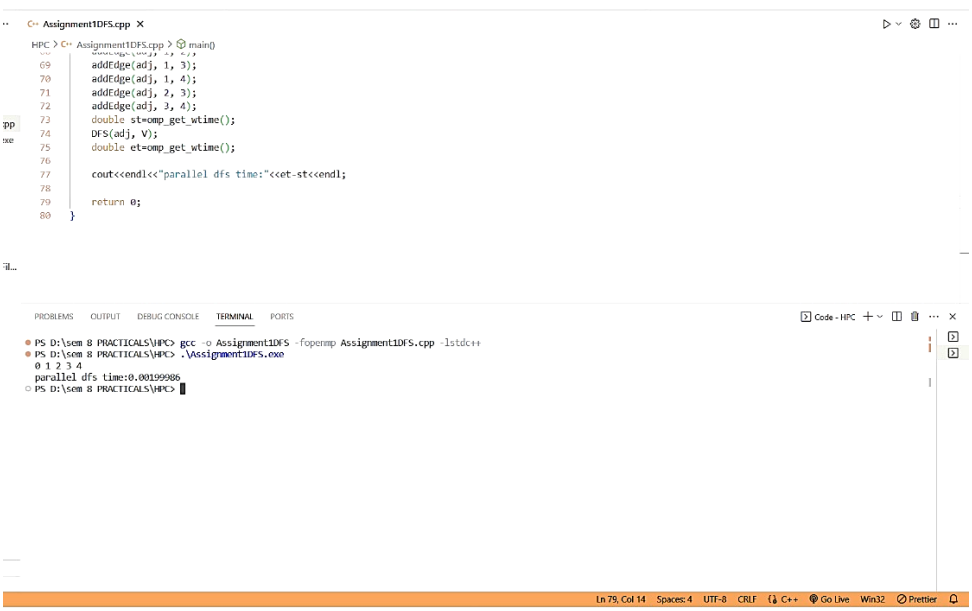
DFS – Code

```
#include <bits/stdc++.h>

#include <stdio.h>

#include <iostream>

#include <omp.h>

using namespace std;

void addEdge(vector<int> adj[], int u, int v)

{

    adj[u].push_back(v);

    adj[v].push_back(u);

}

void DFSUtil(int u, vector<int> adj[], vector<bool> &visited)

{

    visited[u] = true;
```

```
    cout << u << " ";
    #pragma omp parallel
    {
        #pragma omp single
        {
            for (int i = 0; i < adj[u].size(); i++)
                if (visited[adj[u][i]] == false)
                    DFSUtil(adj[u][i], adj, visited);
        }
    }
}

void DFS(vector<int> adj[], int V)
{
    vector<bool> visited(V, false);
    #pragma omp parallel
    {
        #pragma omp single
        {
            for (int u = 0; u < V; u++)
                if (visited[u] == false)
                    DFSUtil(u, adj, visited);
        }
    }
}
```

```
int main()
{
    int V = 5;
    vector<int> adj[V];
    addEdge(adj, 0, 1);
    addEdge(adj, 0, 4);
    addEdge(adj, 1, 2);
    addEdge(adj, 1, 3);
    addEdge(adj, 1, 4);
    addEdge(adj, 2, 3);
    addEdge(adj, 3, 4);
    double st=omp_get_wtime();
    DFS(adj, V);
    double et=omp_get_wtime();
       cout<<endl<<"parallel dfs time:"<<et-st<<endl;
    return 0;
}
```

Output



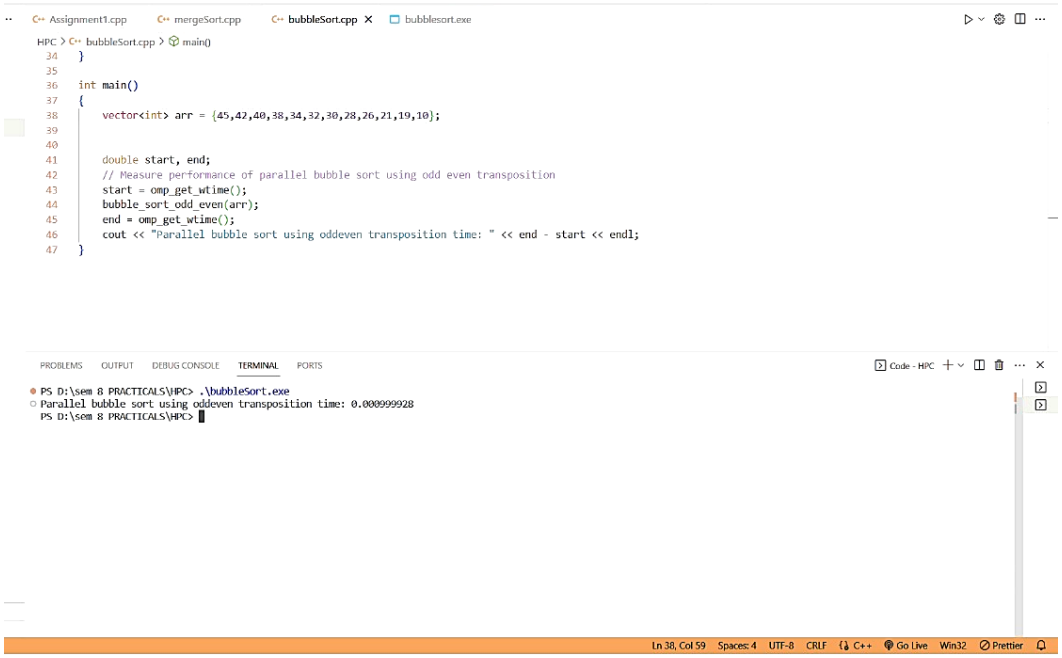
Assignment 2 –

Bubble Sort – Code

```
#include <iostream>
#include <vector>
#include <omp.h>
using namespace std;
void bubble_sort_odd_even(vector<int> &arr)
{
   bool isSorted = false;
   while (!isSorted)
   {
      isSorted = true;
      #pragma omp parallel for
      for (int i = 0; i < arr.size() - 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 < arr.size() - 1; i += 2)
      {
         if (arr[i] > arr[i + 1])
         {
            swap(arr[i], arr[i + 1]);
            isSorted = false;
         }
      }
   }
```

```
}
int main()
{
    vector<int> arr = {5, 2, 9, 1, 7, 6, 8, 3, 4};
    double start, end;
    // Measure performance of parallel bubble sort using odd even transposition
    start = omp_get_wtime();
    bubble_sort_odd_even(arr);
    end = omp_get_wtime();
    cout << "Parallel bubble sort using oddeven transposition time: " << end - start << endl;
}
```

Output –



Merge Sort – Code

```
#include <stdio.h>
#include <omp.h>
#include <iostream>
void merge(int array[],int low,int mid,int high)
{
```

```
int temp[30];
int i,j,k,m;
j=low;
m=mid+1;
for(i=low; j<=mid && m<=high ; i++)
{
  if(array[j]<=array[m])
  {
     temp[i]=array[j];
     j++;
  }
  else
  {
     temp[i]=array[m];
     m++;
  }
}
if(j>mid)
{
  for(k=m; k<=high; k++)
  {
     temp[i]=array[k];
     i++;
  }
```

```
 }
 else
 {
   for(k=j; k<=mid; k++)
   {
     temp[i]=array[k];
     i++;
   }
 }
 for(k=low; k<=high; k++)
   array[k]=temp[k];
}
void mergesort(int array[],int low,int high)
{
 int mid;
 if(low<high)
 {
  mid=(low+high)/2;
 #pragma omp parallel sections num_threads(2)
  {
    #pragma omp section
     {
       mergesort(array,low,mid);
     }
```

```
      #pragma omp section
       {
         mergesort(array,mid+1,high);
       }
    }
   merge(array,low,mid,high);
    }
}
int main()
{
 int array[50];
 int i,size;
 double start,end;
 printf("Enter total no. of elements:\n");
 scanf("%d",&size);
 printf("Enter %d elements:\n",size);
 for(i=0; i<size; i++)
 {
   scanf("%d",&array[i]);
 }
 start=omp_get_wtime();
 printf("\nStart time:%f",start);
 mergesort(array,0,size-1);
 end=omp_get_wtime();
```

```
printf("\nEnd time:%f",end);

printf("\nSorted Elements as follows:\n");

for(i=0; i<size; i++)

  printf("%d ",array[i]);

printf("\n");

printf("\nparallel mergesort time:%f",(end-start));

return 0;

}
```

Output –

Assignment No. 3 –

Code –

```
#include <iostream>
#include <vector>
#include <omp.h>
using namespace std;
void min_reduction(vector<int> &arr)
{    int min_value = arr[0];
   omp_set_num_threads(4);
   #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 = 0;
   #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 = {5, 2, 9, 1, 7, 6, 8, 3, 4};

   double st=omp_get_wtime();

   min_reduction(arr);

   max_reduction(arr);

   sum_reduction(arr);

   average_reduction(arr);

   double et=omp_get_wtime();

   cout<<"Parallel operations time:"<<et-st<<endl;

   return 0;

}
```

Output –

Assignment No. 4 –

Code -

```
import multiprocessing
def process_data(data_chunk):
    """
    Function to process a chunk of data.
    """
    # Your data processing logic goes here
    processed_data = [d * 2 for d in data_chunk]
    return processed_data

if __name__ == '__main__':
    # Create some sample data
    data = list(range(100000))

    # Split the data into chunks
    chunk_size = 1000
    data_chunks = [data[i:i+chunk_size] for i in range(0, len(data), chunk_size)]

    # Create a pool of worker processes
    num_processes = multiprocessing.cpu_count()
    print(num_processes)
    pool = multiprocessing.Pool(num_processes)

    # Process the data in parallel
    processed_chunks = pool.map(process_data, data_chunks)

    # Concatenate the processed chunks into a single list
    processed_data = []
    for chunk in processed_chunks:
```

processed_data.extend(chunk)

print(processed_data[:10])

Output –