1. BFS_DFS

In [ ]:

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

using namespace std;

class Graph {
private:
    int numVertices;
    vector<vector<int>> adj;

public:
    Graph(int vertices): numVertices(vertices), adj(numVertices) {}

    void addEdge(int src, int dest)
    {
        adj[src].push_back(dest);
        adj[dest].push_back(src);
    }

    void viewGraph() {
        cout<<"Graph : ";
        for(int i=0; i<numVertices; i++)
        {
            cout<<"Vertex: "<<i<<"->";
            for(int neighbour: adj[i])
            {
                cout<<neighbour<<" ";
            }
            cout<<endl;
        }
    }

    void bfs(int startVertex)
    {
        vector<bool> visited(numVertices, false);
        queue<int> q;

        visited[startVertex] = true;
        q.push(startVertex);

        while(!q.empty())
        {
            int curr = q.front();
            q.pop();

            cout<<curr;

            #pragma omp parallel for
            for(int neighbour: adj[curr])
            {
                if(!visited[neighbour])
                {
                    visited[neighbour] = true;
                    q.push(neighbour);
                }
            }
        }
    }

    void dfs(int startVertex)
    {
        vector<bool> visited(numVertices, false);
        stack<int> s;

        visited[startVertex] = true;
        s.push(startVertex);

        while(!s.empty())
        {
            int curr = s.top();
            s.pop();

            cout<<curr;

            #pragma omp parallel for
            for(int neighbour: adj[curr])
            {
                if(!visited[neighbour])
                {
                    visited[neighbour] = true;
                    s.push(neighbour);
                }
            }
        }
    }
};

int main()
{
    int numVertices;
    cout<<"VERTICES: ";
    cin>>numVertices;

    int numEdges;
    cout<<"EDGES: ";
    cin>>numEdges;

    Graph graph(numVertices);
    for(int i=0; i<numEdges; i++)
    {
        int src, dest;
        cout<<"EDGE (SRC DEST): ";
        cin>>src>>dest;
        graph.addEdge(src, dest);
    }

    graph.viewGraph();

    int startVertex;
    cout<<"startVertex: ";
    cin>>startVertex;

    cout<<"BFS: ";
    graph.bfs(startVertex);
    cout<<endl;

    cout<<"DFS: ";
    graph.dfs(startVertex);

    return 0;
}
```

2. BUBBLE SORT

In [ ]:

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

using namespace std;

void bubbleSort(int arr[], int n)
{
    for(int i=0; i<n-1; ++i)
    {
        for(int j=0; j<n-i-1; ++j)
        {
            if(arr[j] > arr[j+1])
            {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}

void printArr(int arr[], int n)
{
    for(int i=0; i<n; ++i)
    {
        cout<<arr[i]<<" ";
    }
    cout<<endl;
}

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

    int *arr= new int[n];
    srand(time(0));

    for(int i=0; i<n; ++i)
    {
        arr[i] = rand() % 100;
    }

    // ARRAY BEFORE
    printArr(arr, n);

    // SEQUENTIAL BUBBLE SORT
    clock_t start = clock();
    bubbleSort(arr, n);
    clock_t end = clock();

    double sequential_bubblesort_time= double(end - start) / CLOCKS_PER_SEC;

    // PARALLE BUBBLE SORT
    start = clock();
    #pragma omp parallel
    {
        bubbleSort(arr, n);
    }
    end = clock();

    double parallel_bubblesort_time = double(end - start) / CLOCKS_PER_SEC;

    // ARRAY AFTER
    printArr(arr, n);

    //PRINT
    cout<<"seq bubble sort time: ";
    cout<<sequential_bubblesort_time<<endl;
    cout<<"parallel bubble sort time: ";
    cout<<parallel_bubblesort_time<<endl;

    return 0;
}
```

2. MERGE_SORT

In [ ]:

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

using namespace std;

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

    int left[n1];
    int right[n2];

    for(int i=0; i<n1; i++) left[i] = arr[l + i];
    for(int j=0; j<n2; j++) right[j] = arr[m + 1 + j];

    int i =0, j = 0, k = l;

    while(i < n1 && j < n2)
    {
        if(left[i] <= right[j])
        {
            arr[k] = left[i];
            i++;
        } else
        {
            arr[k] = right[j];
            j++;
        }
        k++;
    }

    //adding any left elements
    while(i < n1)
    {
        arr[k] = left[i];
        i++;
        k++;
    }

    while(j < n2)
    {
        arr[k] = right[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int l, int r)
{
    if(l<r)
    {
        int m = l + (r - l)/2;
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);
        merge(arr, l, m, r);
    }
}

void pmergeSort(int arr[], int l, int r)
{
    if(l<r)
    {
        int m = l + (r - l)/2;
        #pragma omp sections
        {
            #pragma omp section
            {
                pmergeSort(arr, l, m);
            }

            #pragma omp section
            {
                pmergeSort(arr, m+1, r);
            }
        }
        merge(arr, l, m, r);
    }
}

void printArr(int arr[], int n)
{
    for(int i=0; i<n; ++i)
    {
        cout<<arr[i]<<" ";
    }
    cout<<endl;
}

int main() {
    int n;
    cout<"Enter size of array: ";
    cin>>n;

    int *arr = new int[n];
    for(int i=0; i<n; ++i)
    {
        arr[i] = rand() % 100;
    }

    // ARRAY BEFORE
    printArr(arr, n);

    // SEQUENTIAL MERGE SORT
    clock_t start = clock();
    mergeSort(arr, 0, n-1);
    clock_t end = clock();

    double sequential_mergesort_time = double(end - start) / CLOCKS_PER_SEC;

    // PARALLEL MERGE SORT
    start = clock();
    pmergeSort(arr, 0, n-1);
    end = clock();

    double parallel_mergesort_time = double(end - start) / CLOCKS_PER_SEC;

    // ARRAY AFTER
    printArr(arr, n);

    // PRINTING TIME
    cout<"sequential merge sort time: ";
    cout<<sequential_mergesort_time<<endl;
    cout<"parallel merge sort time: ";
    cout<<parallel_mergesort_time<<endl;

    return 0;
}
```

MIN_MAX_SUM_AVG

#pragma omp parallel for reduction(min : min_val)

#pragma omp parallel for reduction(max : max_val)

#pragma omp parallel for reduction(+ : sum)

In [ ]:
```cpp
#include <bits/stdc++.h>
#include <omp.h>

using namespace std;

int min(int arr[], int n)
{
    int min_val = INT_MAX;
    #pragma omp parallel for reduction(min : min_val)
    for(int i=0; i<n; i++)
    {
        if(arr[i] < min_val)
        {
            min_val = arr[i];
        }
    }
    return min_val;
}

int max(int arr[], int n)
{
    int max_val = INT_MIN;
    #pragma omp parallel for reduction(max : max_val)
    for(int i=0; i<n; i++)
    {
        if(arr[i] > max_val)
        {
            max_val = arr[i];
        }
    }
    return max_val;
}

int sum(int arr[], int n)
{
    int sum = 0;
    #pragma omp parallel for reduction(+ : sum)
    for(int i=0; i<n; i++)
    {
        sum += arr[i];
    }
    return sum;
}

double avg(int arr[], int n)
{
    return (double)sum(arr, n)/n;
}

void printArr(int arr[], int n)
{
    for(int i=0; i<n; i++)
    {
        cout<<arr[i]<<" ";
    }
    cout<<endl;
}

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

    int *arr = new int[n];
    for(int i=0; i<n; i++)
    {
        arr[i] = rand() % 100;
    }

    printArr(arr, n);

    cout<<"MIN: "<<min(arr, n)<<endl;
    cout<<"MAX: "<<max(arr, n)<<endl;
    cout<<"SUM: "<<sum(arr, n)<<endl;
    cout<<"AVG: "<<avg(arr, n)<<endl;

    return 0;

}
```

CHRONO CLOCK

In [ ]:
```cpp
#include <chrono>

auto start = chrono::high_resolution_clock::now();
// CALL FUNCTION
auto end = chrono::high_resolution_clock::now();
chrono::duration<double, milli> time = end - start;
```

In [ ]:

In [ ]: