BFS and DFS

First, we start by including the necessary libraries. ‘omp.h’ for open multi-processing, ‘stack’ for dfs and ‘queue’ for bfs.

```
#include<iostream>
#include<omp.h>
#include<stack>
#include<queue>
using namespace std;
```

‘struct’ defines a user-defined data type struct that allows you to group related data together.

TreeNode has value(integer) and pointers to its child nodes (i.e left and right) constructor- takes integer value x and initializes it to member variable val , left and right pointers are set null indicating that the node has no children initially, val is an integer value that represents data stored in the node, left is pointer to the left child node an its initialized to null by default, right is pointer to right child node and is initialized to null by default:

```
struct TreeNode{
   int val;
   TreeNode* left;
   TreeNode* right;
   TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};
```

pBFS function:

Implements parallel BFS traversal of a binary tree using a queue.

Takes the root node of the tree as input.

Creates a queue (q) and enqueues the root node.

Executes a loop until the queue is empty.

Within the loop, it determines the size of the queue (qs) and creates a parallel region using #pragma omp parallel for.

Each thread in the parallel region dequeues a node from the queue (q) using a critical section (#pragma omp critical), prints its value, and enqueues its children (if any).

The parallel execution helps process multiple nodes at the same level simultaneously.

```
Void pBFS(TreeNode* root){
   queue<TreeNode*> q;
   q.push(root);
   while(!q.empty()){
      int qs = q.size();
      #pragma omp parallel for
      for(int I = 0; I < qs; i++){
         TreeNode* node;
         #pragma omp critical
         {
            node = q.front();
            cout << node->val << " ";
            q.pop();
            if(node->left) q.push(node->left);
            if(node->right) q.push(node->right);
         }
      }
   }
}
```

`pDFS` function:

Implements parallel DFS traversal of a binary tree using a stack.

Takes the root node of the tree as input.

Creates a stack (`s`) and pushes the root node onto it.

Executes a loop until the stack is empty.

Within the loop, it determines the size of the stack (`ss`) and creates a parallel region using `#pragma omp parallel for`.

Each thread in the parallel region pops a node from the stack (`s`) using a critical section (`#pragma omp critical`), prints its value, and pushes its children (if any).

The parallel execution allows multiple branches of the tree to be explored simultaneously.

```
Void pDFS(TreeNode* root){
   stack<TreeNode*> s;
   s.push(root);
   while(!s.empty()){
      int ss = s.size();
      #pragma omp parallel for
      for(int I = 0; I < ss; i++){
         TreeNode* node;
         #pragma omp critical
         {
            node = s.top();
            cout << node->val << " ";
            s.pop();
            if(node->right) s.push(node->right);
            if(node->left) s.push(node->left);
         }
      }
   }
}
```

'main` function:

- Creates a binary tree with the provided structure.
- Calls the `pBFS` function to perform parallel BFS traversal and prints the result.
- Calls the `pDFS` function to perform parallel DFS traversal and prints the result.

```
Int main(){
   // Construct Tree
```

```
    TreeNode* tree = new TreeNode(1);
    tree->left = new TreeNode(2);
    tree->right = new TreeNode(3);
    tree->left->left = new TreeNode(4);
    tree->left->right = new TreeNode(5);
    tree->right->left = new TreeNode(6);
    tree->right->right = new TreeNode(7);

    /*
    Our Tree Looks like this:
              1
           2      3
       4     5   6    7

    */

    cout << "Parallel BFS: ";
    pBFS(tree);
    cout << "\n";
    cout << "Parallel DFS: ";
    pDFS(tree);
}
```

MIN,MAX, AVG AND SUM REDUCTION

```
#include<iostream>

#include<omp.h>

using namespace std;
```

`minval` function:

- Takes an array (`arr`) and its size (`n`) as input.
- Initializes a variable `minval` with the value of the first element of the array.
- Uses the `#pragma omp parallel for` directive to parallelize the loop.
- The `reduction(min : minval)` clause performs a reduction operation on `minval` across different threads, ensuring that each thread has a local copy of `minval` and updates it with the minimum value it finds.
- Within the loop, each thread compares the current element of the array with its local copy of `minval` and updates it if the current element is smaller.
- After the loop, the function returns the final minimum value.

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

`maxval` function:

- Takes an array (`arr`) and its size (`n`) as input.
- Initializes a variable `maxval` with the value of the first element of the array.
- Uses the `#pragma omp parallel for` directive to parallelize the loop.
- The `reduction(max : maxval)` clause performs a reduction operation on `maxval` across different threads, ensuring that each thread has a local copy of `maxval` and updates it with the maximum value it finds.

- Within the loop, each thread compares the current element of the array with its local copy of `maxval` and updates it if the current element is larger.
- After the loop, the function returns the final maximum value.

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

`sum` function:

- Takes an array (`arr`) and its size (`n`) as input.
- Initializes a variable `sum` with a value of 0.
- Uses the `#pragma omp parallel for` directive to parallelize the loop.
- The `reduction(+ : sum)` clause performs a reduction operation on `sum` across different threads, ensuring that each thread has a local copy of `sum` and updates it by adding the corresponding element of the array.
- Within the loop, each thread adds the current element of the array to its local copy of `sum`.
- After the loop, the function returns the final sum of the array elements.

```
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;
}
```

`average` function:

- Takes an array (`arr`) and its size (`n`) as input.
- Calls the `sum` function to calculate the sum of the array elements.
- Divides the sum by the size of the array to compute the average.
- Returns the average value.

```
Int average(int arr[], int n){
  return (double)sum(arr, n) / n;
}
```

`main` function:

- Initializes variables, including the size of the array (`n`) and an input array (`arr`).
- Calls the `minval`, `maxval`, `sum`, and `average` functions to perform reduction operations on the array.
- Prints the minimum value, maximum value, sum, and average of the array.

```
Int main(){
  int n = 5;
  int arr[] = {1,2,3,4,5};
  cout << “The minimum value is: “ << minval(arr, n) << ‘\n’;
  cout << “The maximum value is: “ << maxval(arr, n) << ‘\n’;
  cout << “The summation is: “ << sum(arr, n) << ‘\n’;
  cout << “The average is: “ << average(arr, n) << ‘\n’;
  return 0;
}
```

BUBBLE SORT

```
#include<iostream>
#include<omp.h>
using namespace std;
```

1. `bubble` function:

   - Implements the sequential version of the Bubble Sort algorithm.
   - Takes an array (`array`) and its size (`n`) as input.
   - Uses nested loops to iterate through the array and compare adjacent elements.
   - If an element is greater than its adjacent element, it swaps them.
   - The outer loop controls the number of passes, and the inner loop compares adjacent elements.
   - After each pass, the largest element "bubbles" up to its correct position at the end of the array.
   - The sorting process continues until the array is sorted in ascending order.

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

2. `pBubble` function:

   - Implements the parallel version of the Bubble Sort algorithm using OpenMP.
   - Takes an array (`array`) and its size (`n`) as input.
   - Divides the sorting process into two phases: sorting odd-indexed numbers and sorting even-indexed numbers.
   - Uses OpenMP directives to parallelize the sorting process.
   - In the first phase, the loop iterates over odd indices (`j += 2`) and compares each element with its previous element (`j-1`).

- If an element is smaller than its previous element, it swaps them.

- After completing the first phase, a barrier (`#pragma omp barrier`) ensures synchronization before proceeding to the second phase.

- In the second phase, the loop iterates over even indices (`j += 2`) and compares each element with its previous element (`j-1`).

- If an element is smaller than its previous element, it swaps them.

- The parallel execution allows multiple comparisons and swaps to occur simultaneously, potentially improving performance.

```
void pBubble(int array[], int n){
  //Sort odd indexed numbers
  for(int i = 0; i < n; ++i){
    #pragma omp for
    for (int j = 1; j < n; j += 2){
    if (array[j] < array[j-1])
    {
      swap(array[j], array[j - 1]);
    }
  }

  // Synchronize
  #pragma omp barrier

  //Sort even indexed numbers
  #pragma omp for
  for (int j = 2; j < n; j += 2){
   if (array[j] < array[j-1])
   {
     swap(array[j], array[j - 1]);
   }
  }
  }
}
```

3. `printArray` function:

   - Takes an array (`arr`) and its size (`n`) as input.

   - Prints the elements of the array separated by spaces.

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

4. `main` function:

   - Initializes variables, including the size of the array (`n`), an input array (`arr`), and a backup array (`brr`).

   - Fills the input array with numbers in descending order from `n` to 1.

   - Measures the execution time of the sequential Bubble Sort algorithm using `omp_get_wtime()` before and after the sorting process.

   - Prints the execution time and the sorted array.

   - Resets the input array to its original descending order.

   - Measures the execution time of the parallel Bubble Sort algorithm using `omp_get_wtime()` before and after the sorting process.

   - Prints the execution time and the sorted array.

```
int main(){
    // Set up variables
    int n = 10;
    int arr[n];
    int brr[n];
    double start_time, end_time;
    // Create an array with numbers starting from n to 1
    for(int i = 0, j = n; i < n; i++, j--) arr[i] = j;

    // Sequential time
    start_time = omp_get_wtime();
    bubble(arr, n);
    end_time = omp_get_wtime();
    cout << "Sequential Bubble Sort took : " << end_time - start_time << " seconds.\n";
    printArray(arr, n);

    // Reset the array
    for(int i = 0, j = n; i < n; i++, j--) arr[i] = j;

    // Parallel time
    start_time = omp_get_wtime();
    pBubble(arr, n);
    end_time = omp_get_wtime();
    cout << "Parallel Bubble Sort took : " << end_time - start_time << " seconds.\n";
    printArray(arr, n);
}
```

MERGE SORT

```
#include <iostream>
#include <omp.h>
using namespace std;
```

merge` function:

   - Takes an array (`arr`), starting index (`low`), middle index (`mid`), and ending index (`high`) as input.

   - Creates two temporary arrays (`left` and `right`) to store the left and right partitions of the array.

   - Copies the elements from the original array to the left and right partitions.

   - Compares the elements from the left and right partitions and places them in the original array in sorted order.

   - Handles the case where any elements are left out in either partition.

```
void merge(int arr[], int low, int mid, int high) {
   // Create arrays of left and right partititons
   int n1 = mid - low + 1;
   int n2 = high - mid;

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

   // Copy all left elements
   for (int i = 0; i < n1; i++) left[i] = arr[low + i];

   // Copy all right elements
   for (int j = 0; j < n2; j++) right[j] = arr[mid + 1 + j];

   // Compare and place elements
   int i = 0, j = 0, k = low;
```

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

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

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

parallelMergeSort` function:

  - Takes an array (`arr`), starting index (`low`), and ending index (`high`) as input.
  - Checks if the array size is greater than 1.

- Uses the `#pragma omp parallel sections` directive to parallelize the two recursive calls for the left and right partitions of the array.

- The `#pragma omp section` directive ensures that each section is executed by a different thread.

- Calls `parallelMergeSort` recursively on the left and right partitions.

- After the recursive calls, merges the sorted left and right partitions using the `merge` function.

```
void parallelMergeSort(int arr[], int low, int high) {
  if (low < high) {
    int mid = (low + high) / 2;

    #pragma omp parallel sections
    {
      #pragma omp section
      {
        parallelMergeSort(arr, low, mid);
      }

      #pragma omp section
      {
        parallelMergeSort(arr, mid + 1, high);
      }
    }
    merge(arr, low, mid, high);
  }
}
```

mergeSort` function:

- Similar to `parallelMergeSort`, but it performs merge sort sequentially without parallelization.

```
void mergeSort(int arr[], int low, int high) {
   if (low < high) {
      int mid = (low + high) / 2;
      mergeSort(arr, low, mid);
      mergeSort(arr, mid + 1, high);
      merge(arr, low, mid, high);
   }
}
```

main` function:

- Initializes variables, including the size of the array (`n`) and an input array (`arr`).
- Measures the time taken by the sequential merge sort algorithm.
- Resets the array.
- Measures the time taken by the parallel merge sort algorithm.
- Prints the time taken by each algorithm.

```
int main() {
   int n = 10;
   int arr[n];
   double start_time, end_time;

   // Create an array with numbers starting from n to 1.
   for(int i = 0, j = n; i < n; i++, j--) arr[i] = j;

   // Measure Sequential Time
   start_time = omp_get_wtime();
   mergeSort(arr, 0, n - 1);
   end_time = omp_get_wtime();
   cout << "Time taken by sequential algorithm: " << end_time - start_time << " seconds\n";
```

```
    // Reset the array
    for(int i = 0, j = n; i < n; i++, j--) arr[i] = j;

    //Measure Parallel time
    start_time = omp_get_wtime();
    parallelMergeSort(arr, 0, n - 1);
    end_time = omp_get_wtime();
    cout << "Time taken by parallel algorithm: " << end_time - start_time << " seconds";

    return 0;
}
```