DFS-BFS

```
#include<iostream>
#include<omp.h>
#include<stack>
#include<queue>

using namespace std;

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

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);
            }
        }
    }
}

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);
            }
        }
    }
}


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);
}
```

1. The `TreeNode` struct represents a node in the binary tree. It has a value (`val`) and pointers to its left and right child nodes (`left` and `right`).
2. The `pBFS` function performs parallel breadth-first search. It starts by creating a queue (`q`) and pushing the root node into it. The algorithm then enters a loop that continues until the queue becomes empty.
3. Inside the loop, the algorithm retrieves the current size of the queue (`qs`). The `#pragma omp parallel for` directive parallelizes the subsequent loop, dividing the iterations among the available threads.
4. Within the parallel loop, each thread retrieves a node from the front of the queue (`node = q.front()`). This operation is protected by a critical section (`#pragma omp critical`) to ensure that only one thread accesses the queue at a time.

5. The value of the node is printed (`cout << node->val << " ";`), and the node is removed from the queue (`q.pop()`). If the node has left and right children, they are added to the queue (`q.push(node->left)` and `q.push(node->right)`).
6. The `pDFS` function performs parallel depth-first search. It follows a similar structure to `pBFS`, but uses a stack (`s`) instead of a queue. The algorithm continues until the stack becomes empty.
7. Inside the loop, the current size of the stack (`ss`) is retrieved, and the subsequent loop is parallelized using `#pragma omp parallel for`.
8. Each thread retrieves a node from the top of the stack (`node = s.top()`), prints its value, and removes it from the stack (`s.pop()`). If the node has right and left children, they are added to the stack (`s.push(node->right)` and `s.push(node->left)`).
9. In the `main` function, a binary tree is constructed with some example nodes.
10. The `pBFS` function is called to perform parallel breadth-first search, and the result is printed.
11. The `pDFS` function is called to perform parallel depth-first search, and the result is printed.

Note that OpenMP is used to parallelize the loop iterations in both functions. However, the critical sections (`#pragma omp critical`) ensure that only one thread accesses the shared data structures (queue or stack) at a time, preventing data races.

Overall, the code aims to parallelize the traversal of a binary tree using BFS and DFS algorithms, allowing for potential performance improvements when executed on a parallel system with multiple threads.

BUBBLE-MERGE SORT

```
#include<iostream>
#include<omp.h>

using namespace std;

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]);
        }
    }
}

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]);
      }
    }
  }
}

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

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);
}
```

The provided code demonstrates the implementation of both sequential and parallel versions of the Bubble Sort algorithm using OpenMP.

Let's go through the code and understand its functionality:

- The code begins with the necessary header files, including `<iostream>` and `<omp.h>`.
- The `bubble` function implements the sequential version of the Bubble Sort algorithm. It takes an array (`array`) and its size (`n`) as input. The function iterates through the array and compares adjacent elements, swapping them if they are in the wrong order.
- The `pBubble` function implements the parallel version of the Bubble Sort algorithm using OpenMP. It performs an optimized version of Bubble Sort by sorting the odd-indexed numbers in one iteration and the even-indexed numbers in another iteration. The sorting of odd-indexed and even-indexed elements is done in parallel using the `#pragma omp for` directive.
- The `printArray` function prints the elements of an array.
- In the `main` function, an array `arr` of size `n` is initialized with numbers from `n` to 1.
- The sequential Bubble Sort algorithm is then executed on `arr`. The start and end times are recorded using `omp_get_wtime()` to measure the execution time.
- The array `arr` is reset to its original unsorted state.
- Next, the parallel Bubble Sort algorithm (`pBubble`) is executed on `arr`. Again, the start and end times are measured.
- Finally, the execution times and sorted arrays are printed.

Note that Bubble Sort is not the most efficient sorting algorithm, and its parallel implementation may not always provide significant performance improvements. Other sorting algorithms like Quick Sort or Merge Sort generally offer better performance. Bubble Sort is used here for demonstration purposes.

Also, be cautious when parallelizing sorting algorithms with OpenMP since certain optimizations, such as loop dependencies and potential race conditions, need to be taken into account to ensure correctness and efficiency.

MERGE

```cpp
#include <iostream>
#include <omp.h>

using namespace std;

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++;
    }
}
```

```
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);
    }
}

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);
    }
}

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;
}
```

The provided code demonstrates the implementation of the Merge Sort algorithm using both sequential and parallel versions using OpenMP.

Here's how the code works:

- The code begins with the necessary header files, including `<iostream>` and `<omp.h>`.
- The `merge` function takes an array (`arr`), the indices of the low, mid, and high points, and performs the merge operation of Merge Sort. It creates two temporary arrays (`left` and `right`) to store the left and right partitions of the array. The elements are then compared and merged back into the original array.
- The `parallelMergeSort` function is the parallel implementation of Merge Sort using OpenMP. It uses the `#pragma omp parallel sections` directive to divide the sorting tasks into two sections, each responsible for sorting a subarray. This allows for parallel execution of the sorting tasks. After the sections are completed, the `merge` function is called to merge the two sorted subarrays.
- The `mergeSort` function is the sequential implementation of Merge Sort. It recursively divides the array into subarrays until the base case is reached (when `low` is less than `high`). It then performs the merge operation to combine the sorted subarrays.
- In the `main` function, an array `arr` of size `n` is initialized with numbers from `n` to 1.
- The sequential Merge Sort algorithm is executed on `arr`. The start and end times are recorded using `omp_get_wtime()` to measure the execution time.
- The array `arr` is reset to its original unsorted state.
- Next, the parallel Merge Sort algorithm (`parallelMergeSort`) is executed on `arr`. Again, the start and end times are measured.
- Finally, the execution times are printed.

It's important to note that Merge Sort is a divide-and-conquer sorting algorithm known for its stability and efficiency. Parallelizing Merge Sort using OpenMP can lead to improved performance on multi-core processors. However, keep in mind that the

performance gain from parallelization may vary depending on the size of the input array, the number of available cores, and the efficiency of the parallelization.

PARALLEL REDUCTION

```
#include<iostream>
#include<omp.h>

using namespace std;
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;
}

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;
}

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;
}

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

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;
}
```

The provided code demonstrates the use of OpenMP directives for parallel reduction operations such as finding the minimum value, maximum value, sum, and average of an array.

Here's how the code works:

- The code begins with the necessary header files, including `<iostream>` and `<omp.h>`.
- The `minval` function takes an array (`arr`) and its size (`n`) as input. It initializes a variable `minval` with the first element of the array. The OpenMP directive `#pragma omp parallel for reduction(min : minval)` is used to parallelize the loop that iterates through the array and updates the `minval` variable if a smaller value is found. The reduction clause `reduction(min : minval)` ensures that each thread has a private copy of `minval`, and the final reduction operation is performed to obtain the minimum value across all threads. The function returns the minimum value.
- The `maxval` function is similar to `minval`, but it finds the maximum value in the array using the `max` reduction operation.
- The `sum` function calculates the sum of the elements in the array. It initializes a variable `sum` with 0 and uses the reduction operation `reduction(+ : sum)` to parallelize the loop and update the `sum` variable. Each thread maintains a private copy of `sum`, and the final reduction operation adds the partial sums from all threads to obtain the total sum.
- The `average` function calculates the average of the elements in the array. It uses the `sum` function to obtain the sum of the array elements and then divides it by the size of the array.
- In the `main` function, an array `arr` of size `n` is initialized with values {1, 2, 3, 4, 5}.
- The minimum value, maximum value, sum, and average of the array are computed using the respective functions, and the results are printed.
- The program concludes by returning 0.

By using OpenMP reduction clauses (`min`, `max`, and `+`), the provided code allows multiple threads to concurrently perform the reduction operations. This can result in improved performance by leveraging the computational power of multi-core processors.

CUDA-MATRIX MULTIPLICATION

The provided code demonstrates matrix multiplication using CUDA. Here's a breakdown of how the code works:

- The `multiply` kernel function is defined with the `__global__` qualifier, indicating that it will be executed on the GPU. The function takes three integer pointers (`A`, `B`, and `C`) representing matrices, and an additional integer (`size`) indicating the size of the matrices. Each thread is responsible for computing a single element of the resulting matrix `C`. The function uses thread indices and block indices to calculate the corresponding row and column indices. It then performs the matrix multiplication operation by iterating over the corresponding row in matrix `A` and column in matrix `B`, accumulating the product in the `sum` variable. Finally, the result is stored in the corresponding position in matrix `C`.
- The `initialize` function is used to initialize a matrix with random values. It takes a pointer to the matrix (`matrix`) and the size of the matrix (`size`) as input. It iterates over each element of the matrix and assigns a random value using `rand() % 10`.
- The `print` function is used to print a matrix. It takes a pointer to the matrix (`matrix`) and the size of the matrix (`size`) as input. It iterates over each row and column of the matrix and prints the corresponding value.
- In the `main` function, three integer pointers (`A`, `B`, and `C`) are declared to store the matrices.
- The size of the matrices (`N`) and the block size (`blockSize`) are initialized. In this example, `N` is set to 2, and `blockSize` is set to 16.
- The total number of elements in the matrices (`matrixSize`) and the memory size required to store them (`matrixBytes`) are calculated.
- Memory is allocated on the host (CPU) using `new` for matrices `A`, `B`, and `C`. The `initialize` function is called to fill matrices `A` and `B` with random values. The matrices are then printed.
- Memory is allocated on the device (GPU) using `cudaMalloc` for matrices `X`, `Y`, and `Z`.
- The values of matrix `A` are copied from the host to the device using `cudaMemcpy`. Matrix `B` is also copied from the host to the device.
- The number of threads per block (`THREADS`) and the number of blocks per grid (`BLOCKS`) are calculated based on the matrix size and the desired block size.
- The `dim3` structs `threads` and `blocks` are used to define the dimensions of the thread blocks and grid.
- The kernel function `multiply` is launched using the `<<< >>>` syntax, specifying the number of blocks and threads.
- The result matrix `C` is copied from the device to the host using `cudaMemcpy`.
- The resulting matrix `C` is printed.
- Memory is freed on the host using `delete[]`, and memory is freed on the device using `cudaFree`.

- The program concludes by returning 0.

CUDA- VECTOR ADDITION

The provided code demonstrates vector addition using CUDA. Here's a breakdown of how the code works:

- The `add` kernel function is defined with the `__global__` qualifier, indicating that it will be executed on the GPU. The function takes three integer pointers (`A`, `B`, and `C`) representing vectors, and an additional integer (`size`) indicating the size of the vectors. Each thread is responsible for adding a single element of the vectors. The thread index `tid` is calculated based on the block index and thread index within the block. If the thread index is within the valid range (`tid < size`), the corresponding elements from vectors `A` and `B` are added, and the result is stored in vector `C`.
- The `initialize` function is used to initialize a vector with random values. It takes a pointer to the vector (`vector`) and the size of the vector (`size`) as input. It iterates over each element of the vector and assigns a random value using `rand() % 10`.
- The `print` function is used to print a vector. It takes a pointer to the vector (`vector`) and the size of the vector (`size`) as input. It iterates over each element of the vector and prints the corresponding value.
- In the `main` function, three integer pointers (`A`, `B`, and `C`) are declared to store the vectors.
- The size of the vectors (`N`) is initialized. In this example, `N` is set to 4.
- The total number of elements in the vectors (`vectorSize`) and the memory size required to store them (`vectorBytes`) are calculated.
- Memory is allocated on the host (CPU) using `new` for vectors `A`, `B`, and `C`. The `initialize` function is called to fill vectors `A` and `B` with random values. The vectors are then printed.
- Memory is allocated on the device (GPU) using `cudaMalloc` for vectors `X`, `Y`, and `Z`.
- The values of vector `A` are copied from the host to the device using `cudaMemcpy`. Vector `B` is also copied from the host to the device.
- The number of threads per block (`threadsPerBlock`) and the number of blocks per grid (`blocksPerGrid`) are calculated based on the vector size.
- The kernel function `add` is launched using the `<<< >>>` syntax, specifying the number of blocks and threads.
- The result vector `C` is copied from the device to the host using `cudaMemcpy`.
- The resulting vector `C` is printed.

- Memory is freed on the host using `delete[]`, and memory is freed on the device using `cudaFree`.
- The program concludes by returning 0.

CUDA INFO

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA. It allows developers to utilize the computational power of NVIDIA GPUs (Graphics Processing Units) for general-purpose computing tasks, beyond just graphics processing.

CUDA provides a programming model and a set of APIs that enable developers to write parallel programs that can be executed on NVIDIA GPUs. It allows developers to offload computationally intensive tasks from the CPU to the GPU, taking advantage of the parallel processing capabilities of GPUs to accelerate computations.

Key features and concepts of CUDA include:

1. Parallel Execution Model: CUDA enables parallel execution by dividing tasks into parallel threads that can be executed concurrently on the GPU. These threads are organized into blocks, and blocks are organized into a grid.
2. GPU Architecture: CUDA leverages the architecture of NVIDIA GPUs, which consist of hundreds or thousands of cores capable of executing many threads in parallel.
3. Kernel Functions: CUDA programs include kernel functions, which are special functions that are executed on the GPU. Each kernel function is executed by multiple threads in parallel.
4. Memory Hierarchy: CUDA provides a hierarchy of memory types, including global memory, shared memory, and local memory, which can be used to manage data movement between the CPU and GPU, as well as within the GPU itself.
5. CUDA Libraries: NVIDIA provides a set of libraries that are optimized for GPU computing, such as cuBLAS for linear algebra operations, cuFFT for Fast Fourier Transforms, and cuDNN for deep neural networks.
6. CUDA C/C++: CUDA programs can be written using the CUDA C/C++ programming language extensions, which allow developers to write GPU-accelerated code using familiar C/C++ syntax.

CUDA has gained popularity in various fields, including scientific computing, data analytics, machine learning, computer vision, and more, due to its ability to

significantly accelerate computations and solve complex problems by harnessing the power of GPUs.