BFS_DFS

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

using namespace std;

// Graph class representing the adjacency list
class Graph {
    int V;  // Number of vertices
    vector<vector<int> > adj;  // Adjacency list

public:
    Graph(int V) : V(V), adj(V) {}

    // Add an edge to the graph
    void addEdge(int v, int w) {
        adj[v].push_back(w);
    }

    // Parallel Depth-First Search
    void parallelDFS(int startVertex) {
        vector<bool> visited(V, false);
        parallelDFSUtil(startVertex, visited);
    }

    // Parallel DFS utility function
    void parallelDFSUtil(int v, vector<bool>& visited) {
        visited[v] = true;
        cout << v << " ";

        #pragma omp parallel for
        for (int i = 0; i < adj[v].size(); ++i) {
            int n = adj[v][i];
            if (!visited[n])
                parallelDFSUtil(n, visited);
        }
    }

    // Parallel Breadth-First Search
    void parallelBFS(int startVertex) {
        vector<bool> visited(V, false);
        queue<int> q;

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

        while (!q.empty()) {
            int v = q.front();
            q.pop();
            cout << v << " ";

            #pragma omp parallel for
            for (int i = 0; i < adj[v].size(); ++i) {
                int n = adj[v][i];
                if (!visited[n]) {
                    visited[n] = true;
                    q.push(n);
                }
            }
        }
    }
};

int main() {
    // Create a graph
    Graph g(7);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(1, 4);
    g.addEdge(2, 5);
    g.addEdge(2, 6);

    cout << "Depth-First Search (DFS): ";
    g.parallelDFS(0);
    cout << endl;

    cout << "Breadth-First Search (BFS): ";
    g.parallelBFS(0);
    cout << endl;

    return 0;
}

```

2. MERGE_BUBBLE

```cpp
#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){
    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]);
            }
        }

        #pragma omp barrier

        #pragma omp for
        for (int j = 2; j < n; j += 2){
            if (array[j] < array[j-1]){
                swap(array[j], array[j - 1]);
            }
        }
    }
}
void merge(int arr[], int low, int mid, int high) {
    int n1 = mid - low + 1;
    int n2 = high - mid;

    int left[n1], right[n2];

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

    int i = 0, j = 0, k = low;
    while (i < n1 && j < n2) {
        if (left[i] <= right[j]){
            arr[k++] = left[i++];
        } else{
            arr[k++] = right[j++];
        }
    }

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

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);
    }
}

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 printArray(int arr[], int n){
    for(int i = 0; i < n; i++) cout << arr[i] << " ";
    cout << "\n";
}
int main() {
    int n = 10;
    int arr[10] = {10, 56, 38, 77, 61, 95, 40, 13, 29, 17};
    double start_time, end_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);

    start_time = omp_get_wtime();
    #pragma omp parallel
    pBubble(arr, n);
    end_time = omp_get_wtime();
    cout << "Parallel Bubble Sort took: " << end_time - start_time << " 
seconds.\n";
    printArray(arr, n);

    start_time = omp_get_wtime();
    mergeSort(arr, 0, n - 1);
    end_time = omp_get_wtime();
    cout << "Sequential Merge Sort took: " << end_time - start_time << " 
seconds.\n";
    printArray(arr, n);

    start_time = omp_get_wtime();
    parallelMergeSort(arr, 0, n - 1);
    end_time = omp_get_wtime();
    cout << "Parallel Merge Sort took: " << end_time - start_time << " 
seconds.\n";
    printArray(arr, n);
    return 0;
}
// Compile with: g++ -fopenmp bubble_merge.cpp -o bubble_merge
// Run with: ./bubble_merge
```

3. PARALLEL REDUCTION

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

int parallelMin(vector<int> vec) {
    int min_val = vec[0];
    #pragma omp parallel for
    for (int i = 1; i < vec.size(); i++) {
        if (vec[i] < min_val) {
            min_val = vec[i];
        }
    }
```

```cpp
    return min_val;
}

int parallelMax(vector<int> vec) {
    int max_val = vec[0];
    #pragma omp parallel for
    for (int i = 1; i < vec.size(); i++) {
        if (vec[i] > max_val) {
            max_val = vec[i];
        }
    }
    return max_val;
}

int parallelSum(vector<int> vec) {
    int sum = 0;
    #pragma omp parallel for
    for (int i = 0; i < vec.size(); i++) {
        sum += vec[i];
    }
    return sum;
}

float parallelAverage(vector<int> vec) {
    int sum = parallelSum(vec);
    float avg = float(sum) / vec.size();
    return avg;
}

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

    vector<int> vec(n);
    cout << "Enter the elements: ";
    for (int i = 0; i < n; ++i) {
        cin >> vec[i];
    }

    int min_val = parallelMin(vec);
    cout << "Minimum value: " << min_val << endl;

    int max_val = parallelMax(vec);
    cout << "Maximum value: " << max_val << endl;
    int sum = parallelSum(vec);
    cout << "Sum of values: " << sum << endl;

    float avg = parallelAverage(vec);
    cout << "Average of values: " << avg << endl;

    return 0;
}
```

4. CUDA MATRIX MULTIPLICATION

```
%%cu
#include <iostream>
using namespace std;


// CUDA code to multiply matrices
__global__ void multiply(int* A, int* B, int* C, int size) {
    // Uses thread indices and block indices to compute each element
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < size && col < size) {
        int sum = 0;
        for (int i = 0; i < size; i++) {
            sum += A[row * size + i] * B[i * size + col];
        }
        C[row * size + col] = sum;
    }
}


void initialize(int* matrix, int size) {
    for (int i = 0; i < size * size; i++) {
        matrix[i] = rand() % 10;
    }
}


void print(int* matrix, int size) {
    for (int row = 0; row < size; row++) {
        for (int col = 0; col < size; col++) {
            cout << matrix[row * size + col] << " ";
        }
        cout << '\n';
    }
    cout << '\n';
}


int main() {
    int* A, * B, * C;

    int N = 2;
    int blockSize =  16;

    int matrixSize = N * N;
    size_t matrixBytes = matrixSize * sizeof(int);

    A = new int[matrixSize];
    B = new int[matrixSize];
    C = new int[matrixSize];

    initialize(A, N);
    initialize(B, N);
    cout << "Matrix A: \n";
    print(A, N);

    cout << "Matrix B: \n";
    print(B, N);


    int* X, * Y, * Z;
    // Allocate space
    cudaMalloc(&X, matrixBytes);
    cudaMalloc(&Y, matrixBytes);
    cudaMalloc(&Z, matrixBytes);

    // Copy values from A to X
    cudaMemcpy(X, A, matrixBytes, cudaMemcpyHostToDevice);

    // Copy values from A to X and B to Y
    cudaMemcpy(Y, B, matrixBytes, cudaMemcpyHostToDevice);

    // Threads per CTA dimension
    int THREADS = 2;

    // Blocks per grid dimension (assumes THREADS divides N evenly)
    int BLOCKS = N / THREADS;

    // Use dim3 structs for block  and grid dimensions
    dim3 threads(THREADS, THREADS);
    dim3 blocks(BLOCKS, BLOCKS);

    // Launch kernel
    multiply<<<blocks, threads>>>(X, Y, Z, N);

    cudaMemcpy(C, Z, matrixBytes, cudaMemcpyDeviceToHost);
    cout << "Multiplication of matrix A and B: \n";
    print(C, N);

    delete[] A;
    delete[] B;
    delete[] C;

    cudaFree(X);
    cudaFree(Y);
    cudaFree(Z);

    return 0;
}
```

CUDA VECTOR ADDITION

```
%%cu
#include <iostream>
using namespace std;

__global__ void add(int* A, int* B, int* C, int size) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid < size) {
        C[tid] = A[tid] + B[tid];
    }
}


void initialize(int* vector, int size) {
    for (int i = 0; i < size; i++) {
        vector[i] = rand() % 10;
    }
}

void print(int* vector, int size) {
    for (int i = 0; i < size; i++) {
        cout << vector[i] << " ";
    }
    cout << endl;
}

int main() {
    int N = 4;
    int* A, * B, * C;

    int vectorSize = N;
    size_t vectorBytes = vectorSize * sizeof(int);

    A = new int[vectorSize];
    B = new int[vectorSize];
    C = new int[vectorSize];

    initialize(A, vectorSize);
    initialize(B, vectorSize);

    cout << "Vector A: ";
    print(A, N);
    cout << "Vector B: ";
    print(B, N);

    int* X, * Y, * Z;
    cudaMalloc(&X, vectorBytes);
    cudaMalloc(&Y, vectorBytes);
    cudaMalloc(&Z, vectorBytes);

    cudaMemcpy(X, A, vectorBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(Y, B, vectorBytes, cudaMemcpyHostToDevice);

    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

    add<<<blocksPerGrid, threadsPerBlock>>>(X, Y, Z, N);

    cudaMemcpy(C, Z, vectorBytes, cudaMemcpyDeviceToHost);
    cout << "Addition: ";
    print(C, N);

    delete[] A;
    delete[] B;
    delete[] C;

    cudaFree(X);
    cudaFree(Y);
    cudaFree(Z);

    return 0;
}
```