////////////////parallel dfs and bfs

```
#include <bits/stdc++.h>
# include <omp.h>
using namespace std;

// Function to perform Parallel BFS
void parallelBFS(vector<vector<int>>& adj_list, int source, vector<bool>& visited, vector<int>&
bfs_order) {
    queue<int> q;
    q.push(source);

    // Parallel loop over the queue
    #pragma omp parallel
    {
        while (!q.empty()) {
            // Get the next vertex from the queue
            #pragma omp for
            for (int i = 0; i < q.size(); i++) {
                int curr = q.front();
                q.pop();

                // If the current vertex has not been visited, mark it as visited
                // and explore all its neighbors
                if (!visited[curr]) {
                    #pragma omp critical
                    {
                        visited[curr] = true;
                        bfs_order.push_back(curr); // add the visited node to the bfs_order vector
                    }
                    for (int j = 0; j < adj_list[curr].size(); j++) {
                        int neighbor = adj_list[curr][j];

                        // Add the neighbor to the queue if it has not been visited
                        if (!visited[neighbor]) {
                            q.push(neighbor);
                        }
                    }
                }
            }
        }
    }
}
```

```
// Function to perform Parallel DFS
void parallelDFS(vector<vector<int>>& adj_list, int source, vector<bool>& visited, vector<int>&
dfs_order) {
    stack<int> s;
    s.push(source);

    // Parallel loop over the stack
    #pragma omp parallel
    {
        while (!s.empty()) {
            // Get the next vertex from the stack
            #pragma omp for
            for (int i = 0; i < s.size(); i++) {
                int curr = s.top();
                s.pop();

                // If the current vertex has not been visited, mark it as visited
                // and explore all its neighbors
                if (!visited[curr]) {
                    #pragma omp critical
                    {
                        visited[curr] = true;
                        dfs_order.push_back(curr); // add the visited node to the dfs_order vector
                    }
                    for (int j = 0; j < adj_list[curr].size(); j++) {
                        int neighbor = adj_list[curr][j];

                        // Add the neighbor to the stack if it has not been visited
                        if (!visited[neighbor]) {
                            s.push(neighbor);
                        }
                    }
                }
            }
        }
    }
}

int main() {
    // Construct the adjacency list
    vector<vector<int>> adj_list = {
        {1, 2},
```

```
        {0, 3, 4},
        {0, 5, 6},
        {1},
        {1},
        {2},
        {2}
    };

    // Perform Parallel BFS from node 0
    int source = 0;
    int n = adj_list.size();
    vector<bool> visited(n, false);
    vector<int> bfs_order;
    parallelBFS(adj_list, source, visited, bfs_order);

    // Print the visited nodes and the BFS order
    cout << "BFS order: ";
    for (int i = 0; i < bfs_order.size(); i++) {
        cout << bfs_order[i] << " ";
    }
    cout << endl;

    for (int i = 0; i < n; i++) {
        if (visited[i]) {
            cout << "Node " << i << " has been visited" << endl;
        }
    }

    // Perform Parallel DFS from
// reset the visited vector
fill(visited.begin(), visited.end(), false);
vector<int> dfs_order;
parallelDFS(adj_list, source, visited, dfs_order);

// Print the visited nodes and the DFS order
cout << "DFS order: ";
for (int i = 0; i < dfs_order.size(); i++) {
    cout << dfs_order[i] << " ";
}
cout << endl;

for (int i = 0; i < n; i++) {
    if (visited[i]) {
```

```
        cout << "Node " << i << " has been visited" << endl;
    }
}

return 0;
}
```

//

Parallel mergersort

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

using namespace std;

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

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++) {
        L[i] = arr[l + i];
    }
    for (int j = 0; j < n2; j++) {
        R[j] = arr[m + 1 + j];
    }

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
```

```
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

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

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

int main() {
    srand(time(nullptr));
    const int size = 10000;
    int arr[size];

    for (int i = 0; i < size; i++) {
        arr[i] = rand() % 10000;
    }

    double start = omp_get_wtime();
    mergeSort(arr, 0, size - 1);
    double end = omp_get_wtime();

    cout << "Sorted array: " << endl;
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
```

```
    }
    cout << endl;
    cout << "Time taken: " << end - start << " seconds" << endl;

    return 0;
}
```

//

Parallel bubble sort

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

void bubble_sort(int arr[], int n) {
    int i, j;
    for (i = 0; i < n - 1; i++) {
        if (i % 2 == 0) {
            #pragma omp parallel for shared(arr)
            for (j = 0; j < n - 1; j += 2) {
                if (arr[j] > arr[j+1]) {
                    int temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
            }
        }
        else {
            #pragma omp parallel for shared(arr)
            for (j = 1; j < n - 1; j += 2) {
                if (arr[j] > arr[j+1]) {
                    int temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
            }
        }
    }
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

    bubble_sort(arr, n);
```

```
    printf("Sorted array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    return 0;
}
```

//min, max , average and sum in parallel reduction

```
#include <iostream>

#include <vector>

#include <omp.h>

using namespace std;


void min_reduction(vector<int>& arr) {

        int min_value = INT_MAX;

        #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 = INT_MIN;
    #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};

  min_reduction(arr);

  max_reduction(arr);

  sum_reduction(arr);

  average_reduction(arr);

}
```

/////////////////// cuda addition of two large vectors

```
#include <iostream>
#include <cuda_runtime.h>
#include <bits/stdc++.h>
// Kernel function for vector addition
__global__ void vectorAdd(const float* a, const float* b, float* c, int size)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < size)
        c[idx] = a[idx] + b[idx];
}

int main()
{
    int size = 1000000;  // Size of the vectors
    size_t bytes = size * sizeof(float);

    // Allocate memory on the host (CPU)
    float* h_a = new float[size];
    float* h_b = new float[size];
    float* h_c = new float[size];
```

```
    // Initialize input vectors
    for (int i = 0; i < size; ++i) {
        h_a[i] = i;
        h_b[i] = i;
    }

    // Allocate memory on the device (GPU)
    float* d_a, * d_b, * d_c;
    cudaMalloc((void**)&d_a, bytes);
    cudaMalloc((void**)&d_b, bytes);
    cudaMalloc((void**)&d_c, bytes);

    // Copy input data from host to device
    cudaMemcpy(d_a, h_a, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, bytes, cudaMemcpyHostToDevice);

    // Define block and grid sizes
    int threadsPerBlock = 256;
    int blocksPerGrid = (size + threadsPerBlock - 1) / threadsPerBlock;

    // Launch kernel on the GPU
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, d_c, size);

    // Copy result from device to host
    cudaMemcpy(h_c, d_c, bytes, cudaMemcpyDeviceToHost);

    // Print the first 10 elements of the result
    for (int i = 0; i < 10; ++i) {
        std::cout << h_c[i] << " ";
    }
    std::cout << std::endl;

    // Free memory
    delete[] h_a;
    delete[] h_b;
    delete[] h_c;
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
}
```

//////////////// cuda matrix multiplication

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

// CUDA kernel for matrix multiplication
__global__ void matrixMultiply(int *a, int *b, int *c, int N)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < N && col < N) {
        int sum = 0;
        for (int k = 0; k < N; ++k) {
            sum += a[row * N + k] * b[k * N + col];
        }
        c[row * N + col] = sum;
    }
}

int main()
{
    int N = 4; // Matrix size

    int *a, *b, *c; // Host matrices
    int *d_a, *d_b, *d_c; // Device matrices

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

    // Allocate host memory
    a = (int*)malloc(matrixSize);
    b = (int*)malloc(matrixSize);
    c = (int*)malloc(matrixSize);

    // Initialize host matrices
    for (int i = 0; i < N * N; ++i) {
        a[i] = i + 1;
        b[i] = i + 1;
    }

    // Allocate device memory
```

```
    cudaMalloc((void**)&d_a, matrixSize);
    cudaMalloc((void**)&d_b, matrixSize);
    cudaMalloc((void**)&d_c, matrixSize);

    // Transfer data from host to device
    cudaMemcpy(d_a, a, matrixSize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, matrixSize, cudaMemcpyHostToDevice);

    // Define block and grid dimensions
    dim3 threadsPerBlock(2, 2);
    dim3 blocksPerGrid((N + threadsPerBlock.x - 1) / threadsPerBlock.x,
                       (N + threadsPerBlock.y - 1) / threadsPerBlock.y);

    // Launch kernel
    matrixMultiply<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, d_c, N);

    // Transfer results from device to host
    cudaMemcpy(c, d_c, matrixSize, cudaMemcpyDeviceToHost);

    // Print result
    for (int i = 0; i < N * N; ++i) {
        std::cout << c[i] << " ";
        if ((i + 1) % N == 0)
            std::cout << std::endl;
    }

    // Free memory
    free(a);
    free(b);
    free(c);
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
}
```

Or

```
#define N 16
 #include <bits/stdc++.h>
#include <cuda_runtime.h>
```

```
Using namespace std;
__global__ void matrixMult (int *a, int *b, int *c, int width);
int main() { int a[N][N], b[N][N], c[N][N];
 int *dev_a, *dev_b, *dev_c;
// initialize matrices a and b with appropriate values
 int size = N * N * sizeof(int);
  cudaMalloc((void **) &dev_a, size);
 cudaMalloc((void **) &dev_b, size);
cudaMalloc((void **) &dev_c, size);
cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);
dim3 dimGrid(1, 1);
dim3 dimBlock(N, N);
 matrixMult<<dimGrid, dimBlock>>(dev_a, dev_b, dev_c, N);
cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);
 for (int i = 0; i < N * N; ++i) {
        std::cout << c[i] << " ";
        if ((i + 1) % N == 0)
            std::cout << std::endl;
    }
cudaFree(dev_a);
cudaFree(dev_b);
cudaFree(dev_c);
__global__ void matrixMult (int *a, int *b, int *c, int width)
 { int k, sum = 0;
int col = threadIdx.x + blockDim.x * blockIdx.x;
int row = threadIdx.y + blockDim.y * blockIdx.y;
 if(col < width && row < width)
 { for (k = 0; k < width; k++)
   sum += a[row * width + k] * b[k * width + col];
   c[row * width + col] = sum;
}
 }
```

http://users.wfu.edu/choss/CUDA/docs/Lecture%205.pdf

nvcc program.cu -o program

Huffman

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

__global__ void buildHuffmanTree(int* frequencies, int* tree, int n) {
```

```
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < n) {
        // Find the two lowest frequency nodes
        int min1 = INT_MAX, min2 = INT_MAX;
        int minIndex1, minIndex2;
        for (int j = 0; j < n; j++) {
            if (frequencies[j] != 0 && frequencies[j] < min1) {
                min2 = min1;
                minIndex2 = minIndex1;
                min1 = frequencies[j];
                minIndex1 = j;
            } else if (frequencies[j] != 0 && frequencies[j] < min2) {
                min2 = frequencies[j];
                minIndex2 = j;
            }
        }
        // Combine the two lowest frequency nodes into a new node
        int newNodeIndex = n + i;
        frequencies[newNodeIndex] = min1 + min2;
        tree[newNodeIndex] = 0;
        tree[newNodeIndex + n] = 0;
        if (minIndex1 < minIndex2) {
            tree[newNodeIndex] = minIndex1;
            tree[newNodeIndex + n] = minIndex2;
        } else {
            tree[newNodeIndex] = minIndex2;
            tree[newNodeIndex + n] = minIndex1;
        }
    }
}

int main() {
    int n = 256;
    int* frequencies;
    int* tree;
    cudaMalloc(&frequencies, n * sizeof(int));
    cudaMalloc(&tree, 2 * n * sizeof(int));

    // Initialize frequencies
    for (int i = 0; i < n; i++) {
        frequencies[i] = i + 1;
    }
```

```
    int numBlocks = (n + 255) / 256;
    buildHuffmanTree<<<numBlocks, 256>>>(frequencies, tree, n);

    // Encode the data using the Huffman tree
    // ...

    cudaFree(frequencies);
    cudaFree(tree);
    return 0;
}
```

Huffman encoding;

```
#include <iostream>
#include <queue>
#include <vector>

// Node structure for the Huffman tree
struct Node {
    char data;
    unsigned frequency;
    Node* left;
    Node* right;

    Node(char data, unsigned frequency)
        : data(data), frequency(frequency), left(nullptr), right(nullptr) {}

    ~Node() {
        delete left;
        delete right;
    }
};

// Comparison function for priority queue
struct Compare {
    bool operator()(Node* left, Node* right) {
        return left->frequency > right->frequency;
    }
};

// Kernel function for generating Huffman codes on the GPU
__global__ void generateCodesKernel(Node* root, char* codes, int* codeLengths, int
```

```
codesSize) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    if (tid < codesSize) {
        Node* node = root;
        int codeIndex = tid * codesSize;
        int codeLength = 0;

        while (node) {
            if (node->left && tid < node->left->frequency) {
                node = node->left;
                codes[codeIndex + codeLength] = '0';
            } else if (node->right) {
                node = node->right;
                codes[codeIndex + codeLength] = '1';
            } else {
                break;
            }
            codeLength++;
        }

        codeLengths[tid] = codeLength;
    }
}

// Huffman encoding function
void huffmanEncodeGPU(const char* input, char* output, int size, const char* codes,
const int* codeLengths, int codesSize) {
    char* d_input;
    char* d_output;
    char* d_codes;
    int* d_codeLengths;

    // Allocate device memory
    cudaMalloc((void**)&d_input, size * sizeof(char));
    cudaMalloc((void**)&d_output, size * codesSize * sizeof(char));
    cudaMalloc((void**)&d_codes, codesSize * codesSize * sizeof(char));
    cudaMalloc((void**)&d_codeLengths, codesSize * sizeof(int));

    // Copy input data to device memory
    cudaMemcpy(d_input, input, size * sizeof(char), cudaMemcpyHostToDevice);

    // Copy Huffman codes to device memory
```

```
    cudaMemcpy(d_codes, codes, codesSize * codesSize * sizeof(char),
cudaMemcpyHostToDevice);

    // Copy code lengths to device memory
    cudaMemcpy(d_codeLengths, codeLengths, codesSize * sizeof(int),
cudaMemcpyHostToDevice);

    // Configure kernel execution parameters
    int blockSize = 256;
    int gridSize = (codesSize + blockSize - 1) / blockSize;

    // Launch the kernel to generate codes on the GPU
    generateCodesKernel<<<gridSize, blockSize>>>(root, d_codes, d_codeLengths,
codesSize);

    // Copy the encoded data from device to host memory
    cudaMemcpy(output, d_output, size * codesSize * sizeof(char),
cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_input);
    cudaFree(d_output);
    cudaFree(d_codes);
    cudaFree(d_codeLengths);
}

int main() {
    std::string text = "Hello, world!";
    int size = text.size();

    // Count frequencies of characters
    std::vector<unsigned> frequencies(256, 0);
    for (char c : text) {
        frequencies[c]++;
    }

    // Create a priority queue to store nodes
    std::priority_queue<Node*, std::vector<Node*>, Compare> pq
```