

# BFS

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

class node
{
    public:
        node *left, *right;
        int data;
};

class Breadthfs
{
    public:
        node *insert(node *, int);
        void bfs(node *);
};

node *insert(node *root, int data)
// inserts a node in tree
{
    if(!root)
    {
        root=new node;
        root->left=NULL;
        root->right=NULL;
        root->data=data;
        return root;
    }
    queue<node *> q;
    q.push(root);
    while(!q.empty())
    {
        node *temp=q.front();
        q.pop();
        if(temp->left==NULL)
        {
```

```

        temp->left=new node;
        temp->left->left=NULL;
        temp->left->right=NULL;
        temp->left->data=data;
        return root;
    }
else
{
    q.push(temp->left);
}
if(temp->right==NULL)
{
    temp->right=new node;
    temp->right->left=NULL;
    temp->right->right=NULL;
    temp->right->data=data;
    return root;
}
else
{
    q.push(temp->right);
}
}
}

void bfs(node *head)
{
    queue<node*> q;
    q.push(head);
    int qSize;
    while (!q.empty())
    {
        qSize = q.size();
        #pragma omp parallel for
            //creates parallel threads
        for (int i = 0; i < qSize; i++)
        {
            node* currNode;
            #pragma omp critical

```

```

        {
            currNode = q.front();
            q.pop();
            cout<<"\t"<<currNode->data;

            }// prints parent node
#pragma omp critical
        {
            if(currNode->left)// push parent's left node in queue
                q.push(currNode->left);
            if(currNode->right)
                q.push(currNode->right);
            }// push parent's right node in queue
        }
    }

}

int main(){
    node *root=NULL;
    int data;
    char ans;
    do
    {
        cout<<"\n enter data=";
        cin>>data;
        root=insert(root,data);
        cout<<"do you want insert one more node?";
        cin>>ans;

    }while(ans=='y' || ans=='Y');
    bfs(root);
    return 0;
}

```

# DFS

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

using namespace std;

const int MAX = 100000;
vector<int> graph[MAX];
bool visited[MAX];

void dfs(int node) {
    stack<int> s;
    s.push(node);

    while (!s.empty()) {
        int curr_node = s.top();
        if (!visited[curr_node]) {
            visited[curr_node] = true;
            s.pop();
            cout<<curr_node<<" ";

            #pragma omp parallel for
            for (int i = 0; i < graph[curr_node].size(); i++) {
                int adj_node = graph[curr_node][i];
                if (!visited[adj_node]) {
                    s.push(adj_node);
                }
            }
        }
    }
}

int main() {
    int n, m, start_node;
    cout<<"Enter no. of Node,no. of Edges and Starting Node of graph:\n";
    cin >> n >> m >> start_node;

    //n: node,m:edges
```

```

        cout<<"Enter pair of node and edges:\n";
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;

//u and v: Pair of edges
        graph[u].push_back(v);
        graph[v].push_back(u);
    }
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        visited[i] = false;
    }
    dfs(start_node);
    return 0;
}

/*output
Enter no. of Node,no. of Edges and Starting Node of graph:
4 3 0
Enter pair of node and edges:
0 1
0 2
2 4
0 2 4 1
*/

```

# Bubble Sort

```
#include <iostream>
#include <omp.h>
using namespace std;
void sequentialBubbleSort(int *, int);
void parallelBubbleSort(int *, int);
void swap(int &, int &);
void sequentialBubbleSort(int *a, int n)
{
    int swapped;
    for (int i = 0; i < n; i++)
    {
        swapped = 0;
        for (int j = 0; j < n - 1; j++)
        {
            if (a[j] > a[j + 1])
            {
                swap(a[j], a[j + 1]);
                swapped = 1;
            }
        }
        if (!swapped)
            break;
    }
}

void parallelBubbleSort(int *a, int n)
{
    int swapped;
    for (int i = 0; i < n; i++)
    {
        swapped = 0;
        int first=i%2;
#pragma omp parallel for shared(a,first)
        for (int j = first; j < n - 1; j++)
```

```

        {
            if (a[j] > a[j + 1])
            {
                swap(a[j], a[j + 1]);
                swapped = 1;
            }
        }

        if (!swapped)
            break;
    }
}

```

```

}

```

```

void swap(int &a, int &b)

```

```

{

```

```

    int test;
    test = a;
    a = b;
    b = test;

```

```

}

```

```

int main()

```

```

{

```

```

    int *a, n;
    cout << "\n enter total no of elements=>";
    cin >> n;
    a = new int[n];
    cout << "\n enter elements=>";
    for (int i = 0; i < n; i++)
    {
        cin >> a[i];
    }

```

```

    double start_time = omp_get_wtime(); // start timer

```

```

    for sequential algorithm sequentialBubbleSort(a, n);

```

```

        double end_time = omp_get_wtime(); // end timer for sequential
algorithm
        cout << "\n sorted array is=>";
        for (int i = 0; i < n; i++)
        {
            cout << a[i] << endl;
        }
        cout << "Time taken by sequential algorithm: " << end_time -
start_time << " seconds" << endl;

        start_time = omp_get_wtime(); // start timer
for parallel algorithm parallelBubbleSort(a, n);

        end_time = omp_get_wtime(); // end timer for parallel algorithm

        cout << "\n sorted array is=>";
        for (int i = 0; i < n; i++)
        {
            cout << a[i] << endl;
        }

        cout << "Time taken by parallel algorithm: " << end_time -
start_time << " seconds" << endl;

        delete[] a; // Don't forget to free the allocated memory

        return 0;
}

```

## Output :

Input array : 90, 64, 22, 12, 25, 11, 34

Output array : [11, 12, 22, 25, 34, 64, 90]



## Implement min, max, sum and average operations using parallel reduction

```
#include <iostream>
```

```
#include <vector>
```

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

```
#include <climits>
```

```
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;  
    arr.push_back(5);  
    arr.push_back(2);  
    arr.push_back(9);  
    arr.push_back(1);  
    arr.push_back(7);  
    arr.push_back(6);  
    arr.push_back(8);  
    arr.push_back(3);  
    arr.push_back(4);  
  
    min_reduction(arr);  
    max_reduction(arr);  
    sum_reduction(arr);  
    average_reduction(arr);  
}
```

## Output :

Enter array elements [5, 2, 9, 1, 7, 6, 8, 3, 4]

Minimum value : 1

Maximum value : 9

Sum : 45

Average : 5



# CUDA kernel to add two vectors

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

```
// CUDA kernel to add two vectors
```

```
__global__ void vectorAdd(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];  
    }  
}
```

```
int main() {
```

```
    int size = 1000000; // Size of the vectors  
    int *a, *b, *c; // Host vectors  
    int *d_a, *d_b, *d_c; // Device vectors  
    int blockSize = 256; // Threads per block  
    int numBlocks = (size + blockSize - 1) / blockSize; // Number of blocks
```

```
    / Allocate memory for host vectors a =  
(int*)malloc(size * sizeof(int)); b =  
(int*)malloc(size * sizeof(int)); c =  
(int*)malloc(size * sizeof(int));
```

```
    / Initialize host vectors  
    for (int i = 0; i < size; i++) {  
        a[i] = i;  
        b[i] = i * 2;  
    }
```

```
    / Allocate memory for device vectors  
    cudaMalloc(&d_a, size * sizeof(int));  
    cudaMalloc(&d_b, size * sizeof(int));  
    cudaMalloc(&d_c, size * sizeof(int));
```

```
    / Copy host vectors to device  
    cudaMemcpy(d_a, a, size * sizeof(int), cudaMemcpyHostToDevice);  
    cudaMemcpy(d_b, b, size * sizeof(int), cudaMemcpyHostToDevice);
```

```

// Launch kernel
vectorAdd<<<numBlocks, blockSize>>>(d_a, d_b, d_c, size);

// Copy result back to host
cudaMemcpy(c, d_c, size * sizeof(int), cudaMemcpyDeviceToHost);

// Verify the result
for (int i = 0; i < size; i++) {
    if (c[i] != a[i] + b[i]) {
        printf("Error: Element %d did not match!\n", i); break;
    }
}

/ Free device memory
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);

/ Free host memory
free(a); free(b); free(c);

return 0;
}

```

## Output :

```
$ module load cuda
```

```
$ make
```

```
$ sub submit.lsf
```

```
$ cat add_vec.JOBID
```

```
-----
__SUCCESS__
```

```
-----
N           = 104857
Threads Per Block = 256
Blocks In Grid   = 4096
-----
```

# Matrix multiplication

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

```
#include <iostream>
```

```
__global__ void matmul(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 Pvalue = 0;
        for (int k = 0; k < N; k++) {
            Pvalue += A[Row*N+k] * B[k*N+Col];
        }
        C[Row*N+Col] = Pvalue;
    }
}
```

```
int main() {
    int N = 512;
    int size = N * N * sizeof(int);
    int* A, * B, * C;
    int* dev_A, * dev_B, * dev_C;
    cudaMallocHost(&A, size);
    cudaMallocHost(&B, size);
    cudaMallocHost(&C, size);
    cudaMalloc(&dev_A, size);
    cudaMalloc(&dev_B, size);
    cudaMalloc(&dev_C, size);

    / Initialize matrices A and B for
    (int i = 0; i < N; i++) {
        for (int j = 0; j <
N; j++) { A[i*N+j] = i*N+j;
B[i*N+j] = j*N+i;

        }
    }
```

```

    }

    cudaMemcpy(dev_A, A, size,
cudaMemcpyHostToDevice); cudaMemcpy(dev_B, B,
size, cudaMemcpyHostToDevice);

    dim3 dimBlock(16, 16);
    dim3 dimGrid(N/dimBlock.x, N/dimBlock.y);

    matmul<<<dimGrid, dimBlock>>>(dev_A, dev_B, dev_C, N);

    cudaMemcpy(C, dev_C, size, cudaMemcpyDeviceToHost);

    // Print the result
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 10; j++) {
            std::cout << C[i*N+j] << " ";
        }
        std::cout << std::endl;
    }

    / Free memory
    cudaFree(dev_A);
    cudaFree(dev_B);
    cudaFree(dev_C);
    cudaFreeHost(A);
    cudaFreeHost(B);
    cudaFreeHost(C);

    return 0;
}

```

## Output :

```

please type in m n and k
1024 1024 1024
Time elapsed on matrix multiplication of
1024x1024.

```

1024x1024 on GPU : 13.604608 ms.

Time elapsed on matrix multiplication of  
1024x1024.

On CPU: 9925.121094 ms.

All results are correct !!!, speedup = 729.541138