Parallel BFS and DFS

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

using namespace std;

// -------------------- Parallel BFS --------------------
void parallelBFS(const vector<vector<int>> &graph, int start) {
    int n = graph.size();
    vector<bool> visited(n, false);
    queue<int> q;

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

    cout << "Parallel BFS: ";

    while (!q.empty()) {
        int levelSize = q.size();

        #pragma omp parallel for
        for (int i = 0; i < levelSize; ++i) {
            int curr;

            #pragma omp critical
            {
                if (!q.empty()) {
                    curr = q.front();
                    q.pop();
                } else {
                    continue;
                }
            }

            cout << curr << " ";

            for (int neighbor : graph[curr]) {
                if (!visited[neighbor]) {
                    #pragma omp critical
                    {
                        if (!visited[neighbor]) {
                            visited[neighbor] = true;
                            q.push(neighbor);
                        }
                    }
                }
            }
        }
    }
    cout << endl;
}

// -------------------- Parallel DFS --------------------
void parallelDFSUtil(const vector<vector<int>> &graph, int node,
vector<bool> &visited) {
    visited[node] = true;
    cout << node << " ";

    #pragma omp parallel for
    for (int i = 0; i < graph[node].size(); ++i) {
        int neighbor = graph[node][i];
        if (!visited[neighbor]) {
            #pragma omp task
            parallelDFSUtil(graph, neighbor, visited);
        }
    }
}

void parallelDFS(const vector<vector<int>> &graph, int start) {
    int n = graph.size();
    vector<bool> visited(n, false);

    cout << "Parallel DFS: ";

    #pragma omp parallel
    {
        #pragma omp single
        parallelDFSUtil(graph, start, visited);
    }

    cout << endl;
}

// -------------------- Main --------------------
int main() {
    int n, e, start;
    cout << "Enter number of nodes: ";
    cin >> n;

    cout << "Enter number of edges: ";
    cin >> e;

    vector<vector<int>> graph(n);
    cout << "Enter " << e << " undirected edges (u v):" << endl;
    for (int i = 0; i < e; ++i) {
        int u, v;
        cin >> u >> v;
        graph[u].push_back(v);
        graph[v].push_back(u);
    }

    cout << "Enter starting node: ";
    cin >> start;

    parallelBFS(graph, start);
    parallelDFS(graph, start);

    return 0;
}
```

Parallel Bubble sort and Merge Sort

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

// ----------------- Merge Sort ------------------
void merge(int arr[], int low, int mid, int high) {
    int n1 = mid - low + 1, 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)
        arr[k++] = (left[i] <= right[j]) ? left[i++] : 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);
    }
}

// ----------------- Bubble Sort ------------------
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 parallel for
        for (int j = 1; j < n - 1; j += 2)
            if (array[j] > array[j + 1])
                swap(array[j], array[j + 1]);

        #pragma omp parallel for
        for (int j = 0; j < n - 1; j += 2)
            if (array[j] > array[j + 1])
                swap(array[j], array[j + 1]);
    }
}

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

// ----------------- Main ------------------
int main() {
    const int n = 1000;
    int arr1[n], arr2[n], arr3[n], arr4[n];

    for (int i = 0; i < n; i++) {
        int val = rand() % 10000;
        arr1[i] = arr2[i] = arr3[i] = arr4[i] = val;
    }

    // Sequential Bubble Sort
    auto start = high_resolution_clock::now();
    bubble(arr1, n);
    auto stop = high_resolution_clock::now();
    cout << "Sequential Bubble Sort took: "
         << duration_cast<microseconds>(stop - start).count() / 1e6 << " seconds.\n";

    // Parallel Bubble Sort
    start = high_resolution_clock::now();
    pBubble(arr2, n);
    stop = high_resolution_clock::now();
    cout << "Parallel Bubble Sort took: "
         << duration_cast<microseconds>(stop - start).count() / 1e6 << " seconds.\n";

    // Sequential Merge Sort
    start = high_resolution_clock::now();
    mergeSort(arr3, 0, n - 1);
    stop = high_resolution_clock::now();
    cout << "Sequential Merge Sort took: "
         << duration_cast<microseconds>(stop - start).count() / 1e6 << " seconds.\n";

    // Parallel Merge Sort
    start = high_resolution_clock::now();
    parallelMergeSort(arr4, 0, n - 1);
    stop = high_resolution_clock::now();
    cout << "Parallel Merge Sort took: "
         << duration_cast<microseconds>(stop - start).count() / 1e6 << " seconds.\n";

    return 0;
}
```

Min Max Sum Average

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

using namespace std;

int main() {
   // int n;
   // cout << "Enter number of elements: ";
   // cin >> n;
   // vector<int> data(n);
   // cout << "Enter elements:\n";
   // for (int i = 0; i < n; ++i) cin >> data[i];

   const int n = 10000;
   int data[n], brr[n];
   for (int i = 0; i < n; i++){
      data[i] = rand() % 100000;
   }

   int minVal = data[0], maxVal = data[0];
   long long sum = 0;

       #pragma omp parallel for reduction(min:minVal)
reduction(max:maxVal) reduction(+:sum)
   for (int i = 0; i < n; ++i) {
      minVal = min(minVal, data[i]);
      maxVal = max(maxVal, data[i]);
      sum += data[i];
   }

   double avg = static_cast<double>(sum) / n;
   cout << "Min: " << minVal << "\nMax: " << maxVal << "\nSum: " << sum
<< "\nAverage: " << avg << endl;

   return 0;
}
```

CUDA addition

```
%%writefile add.cu

 #include <stdio.h>

__global__ void add(float *A, float *B, float *C, int N)
{

   int i = blockIdx.x * blockDim.x + threadIdx.x;

   if(i <N)
   {
     C[i]=A[i]+B[i];
   }
}
int main()
{
  int N = 4;
   size_t size = N  *sizeof(float);
   float A[] = {1,2,3,4};
   float B[] = {5,6,7,8};
   float C[4];

   float *d_A,*d_B,*d_C;

   cudaMalloc(&d_A,size);
   cudaMalloc(&d_B,size);
   cudaMalloc(&d_C,size);

   cudaMemcpy(d_A,A,size,cudaMemcpyHostToDevice);
   cudaMemcpy(d_B,B,size,cudaMemcpyHostToDevice);

   add<<<1,N>>>(d_A,d_B,d_C,N);
   cudaMemcpy(C,d_C,size,cudaMemcpyDeviceToHost);

   for(int i=0;i< N;i++)
   {
      printf(" %f",C[i]);
      printf("\n");
   }
}

!nvcc -arch=sm_75 add.cu -o add

!./add
```

CUDA Multiplication

```
%%writefile matmul.cu

 #include <stdio.h>

__global__ void matmul(float *A, float *B, float *C, int N)
{

   int col = blockIdx.x * blockDim.x + threadIdx.x;
   int row = blockIdx.y * blockDim.y + threadIdx.y;
   if(row < N && col < N)
   {
      float sum = 0;
      for(int k = 0; k < N;k++)
         sum = sum + A[row * N + k] * B[N * k + col];
      C[row * N + col] = sum;
   }

}
int main()
{
  int N = 2;
   size_t size = N * N *sizeof(float);
   float A[] = {1,2,3,4};
   float B[] = {5,6,7,8};
   float C[4];

   float *d_A,*d_B,*d_C;

   cudaMalloc(&d_A,size);
   cudaMalloc(&d_B,size);
   cudaMalloc(&d_C,size);

   cudaMemcpy(d_A,A,size,cudaMemcpyHostToDevice);
   cudaMemcpy(d_B,B,size,cudaMemcpyHostToDevice);

    dim3 blocks(N,N);
   dim3 threads(1,1);

   matmul<<<blocks,threads>>>(d_A,d_B,d_C,N);
   cudaMemcpy(C,d_C,size,cudaMemcpyDeviceToHost);

   for(int i=0;i< N*N;i++)
   {
      printf(" %f",C[i]);
      printf("\n");
   }

}

!nvcc -arch=sm_75 matmul.cu -o matmul

!./matmul
```

Mini Project

Implement Parallelization of Database Query Optimization

```
import sqlite3
import concurrent.futures
import time

def exe_sql_qry(query):

    try:
        conn=sqlite3.connect('mydb.db')

        cursor=conn.cursor()

        cursor.execute(query)
        if query.strip().lower().startswith('select'):
            result=cursor.fetchall()

        else:
            conn.commit()
            result='Query Executed...'



        conn.close()

        return result

    except Exception as e:
        return f'error occured {str(e)}'

queries=[
     #"CREATE TABLE users(id number primary key,name varchar,mono
number(10));",
    #"INSERT INTO USERS VALUES(01,'SANKET',90000000);",
    #"INSERT INTO USERS VALUES(02,'TANMAY',80000000);",
    #"INSERT INTO USERS VALUES(03,'SANKET',90000000);",
    #"INSERT INTO USERS VALUES(04,'SANKET',90000000);",
    #"INSERT INTO USERS VALUES(05,'SANKET',90000000);",

    #"SELECT * from users;",
    #"SELECT * from users;",
    #"SELECT * from users;",
    #"SELECT * from users;",
    #"SELECT * from users;",

    ]
n=int(input("Enter no of queries: "))
for i in range(n):
    ip=input("Enter the query: ")
    queries.append(ip)

s1time=time.time()
with concurrent.futures.ThreadPoolExecutor() as executor:
    results1=list(executor.map(exe_sql_qry,queries))
e1time=time.time()

for result in results1:

    print(result)

print('total time1',e1time-s1time)
```