MSBD5009/DSAA5015 Parallel Programming (Spring 2023)
Final Examination
18:30-21:30 on May 20, 2023 (Saturday)

Name	MSBD/DSAA	Student ID	Email

Instructions

This is a **closed-book** exam. It consists of **four** problems. The first **three** problems require you to fill in missing code in designated boxes. The last problem contains **five** short answer questions. **The programming APIs and serial version reference code are listed in the appendices. You can tear off the appendices for your easy reference.**

Problem 1-3 Description

In this semester's assignments, you worked on finding the (L, R)-dense-subgraph in a bipartite graph. In Problems 1-3 of this exam, we consider the k-core decomposition problem. The idea of k-core is similar to (L, R)-dense-subgraph, except that the degree threshold is k for a general graph instead of (L, R) for a bipartite graph. The formal definition of k-core is as follows.

Definition 1. k-core. Given a graph G and a non-negative integer k, a subgraph H is a **k-core** of G, if (1) each vertex $v \in H$ has at least k neighbors in H, i.e., $|N(v, H)| \geq k$; and (2) H is maximal, i.e., any graph of which H is a subgraph, except H itself, has at least one vertex who has fewer than k neighbors.

k-core can be computed through a simple algorithm:

Algorithm 1: Computing k-core

```
(1) Let $j = 0$;
(2) Repeatedly remove all vertices whose degrees are $j$ or less, until no such vertices remain in the graph. These removed vertices are given a coreness of $j$;
(3) If the graph has remaining vertices, increment $j$ by 1 and go back to Step (2).
```

Definition 2. k-core decomposition. k-core decomposition of graph G is to compute the coreness for each vertex $v \in G$.

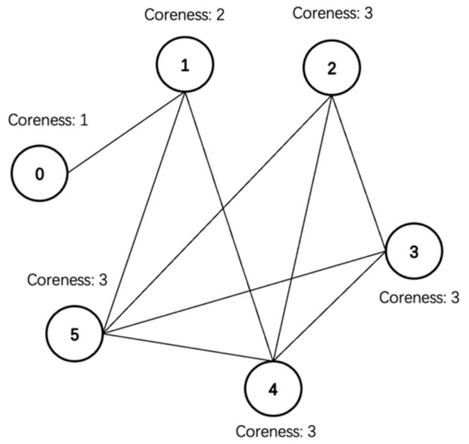


Figure 1. k-core decomposition of an example graph (vertex ID labeled in vertex)

As shown in Figure 1, after k-core decomposition, every vertex is assigned a coreness value. In Problem 1-3, we assume that **the vertex IDs in the input graph are consecutive non-negative integers starting from 0**, i.e., an n-vertex graph has vertex IDs of 0, 1, 2, …, n-1.

Problem 1. MPI Implementation (13 points)

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <queue>
#include <mpi.h>

using namespace std;

int main(int argc, char** argv) {

    // MPI initialization:
                                                                Blank 1 (0.5 pt)
    MPI_Init(&argc, &argv);


    MPI_Comm comm;
    int rank, size;

    // set rank and size:
                                                                  Blank 2 (1 pt)
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);



    int final_numVertices = 6;  // Number of vertices in the graph

    // Create an adjacency list to represent the graph
    vector<vector<int>> graph(final_numVertices);

    // Add edges to the graph
    graph[0] = {1};
    graph[1] = {0, 2, 3};
    graph[2] = {1, 3, 4, 5};
    graph[3] = {1, 2, 4, 5};
    graph[4] = {2, 3, 5};
    graph[5] = {2, 3, 4};

    int final_mindeg = 0;
    int final_maxdeg = 0;
    int *final_degree=nullptr;
    int *local_coreness=nullptr;
    int *final_coreness=nullptr;
    bool *final_removed=nullptr;

    // allocate memory
    final_degree=(int *)calloc(final_numVertices, sizeof(int));
    local_coreness=(int *)calloc(final_numVertices, sizeof(int));
    final_coreness=(int *)calloc(final_numVertices, sizeof(int));
    final_removed=(bool *)calloc(final_numVertices, sizeof(bool));

    for (int i = 0; i < final_numVertices; ++i){
            final_coreness[i] = 0;
    }
    if(rank == 0){
        // Initialize the degree of each vertex
        vector<int> degree(final_numVertices);
        for (int i = 0; i < final_numVertices; ++i){
            degree[i] = graph[i].size();
            final_degree[i] = degree[i];
            local_coreness[i] = degree[i];
            final_removed[i] = false;
        }
        // Find the minimum and maximum degrees in the graph
        final_mindeg = *min_element(degree.begin(), degree.end());
        final_maxdeg = *max_element(degree.begin(), degree.end());
    }

    // send the minimum and maximum degrees of the graph to all processes:
                                                                Blank 3 (1 pt)
    MPI_Bcast(&final_mindeg, 1, MPI_INT, 0 , MPI_COMM_WORLD);
    MPI_Bcast(&final_maxdeg, 1, MPI_INT, 0 , MPI_COMM_WORLD);


   //send final_degree, local_coreness, final_removed arrays to all processes:
                                                              Blank 4 (1.5 pt)
    MPI_Bcast(final_degree, final_numVertices, MPI_INT, 0 , MPI_COMM_WORLD);
    MPI_Bcast(local_coreness, final_numVertices, MPI_INT, 0 , MPI_COMM_WORLD);
    MPI_Bcast(final_removed, final_numVertices, MPI_C_BOOL, 0 , MPI_COMM_WORLD);


    int local_n = 0;
    int my_first_i=0;
    int my_last_i=0;

    // Set local_n, my_first_i and my_last_i for the current process
    if (rank < size-1) {
                                                                Blank 5 (3 pt)
        local_n = final_numVertices / size;
        my_first_i = local_n * rank;
        my_last_i = my_first_i + local_n;

    } else {
        local_n = final_numVertices - (final_numVertices / size) * (size-1);
        my_first_i = (final_numVertices / size)*rank;
        my_last_i = my_first_i + local_n;
    }

    // Create a queue to store the vertices with degree less than k
    queue<int> q;

    // Perform k-core decomposition for each k value
    for (int k = final_mindeg+1; k <= final_maxdeg; ++k) {

        // Add all the vertices with degree less than k to the queue
        bool finish_flag = true;
        for (int i = my_first_i; i < my_last_i; ++i) {
            if (!final_removed[i]) finish_flag = false;
        }
        if(finish_flag) break;
        // Add all the vertices with degree less than k to the queue
        for (int i = 0; i < final_numVertices; ++i) {
            if (final_degree[i] < k) { q.push(i); }
        }

        while (!q.empty()) {
            int vertex = q.front();
            q.pop();

            // Skip isolated vertices
            if (final_degree[vertex] == 0 && final_removed[vertex])
                continue;

            // Reduce the degree of the neighboring vertices and update their coreness
            for (int neighbor : graph[vertex]) {
                if (final_degree[neighbor] > 0) {
                    final_degree[neighbor]--;
                    if (final_degree[neighbor] < k) {
                        q.push(neighbor);
                        local_coreness[neighbor] = max(local_coreness[neighbor], k-1);
                    }
                }
            }

            // Set the degree of the current vertex to 0
            final_removed[vertex] = true;
            final_degree[vertex] = 0;
            local_coreness[vertex] = k - 1;

       } // end while

    } // end for

    //set local_coreness of vertices that are not assigned to this process to 0.
                                                                   Blank 6 (2 pts)
    for(int i = 0; i<final_numVertices; i++){
        if(i >= my_last_i || i < my_first_i) local_coreness[i] = 0;
    }


    // Merge local_coreness to final_coreness
                                                                   Blank 7 (2 pts)
    MPI_Allreduce(local_coreness, final_coreness, final_numVertices, MPI_INT, MPI_MAX,
MPI_COMM_WORLD);


    // Wait for all processes to arrive here
                                                                    Blank 8 (1 pt)
    MPI_Barrier(MPI_COMM_WORLD);

    // Output the coreness of each vertex for all k values
    if(rank ==0){
        for (int i = 0; i < final_numVertices; ++i)
            cout << "Vertex ID: " << i << ", Coreness: " << final_coreness[i] << endl;
    }

    // Terminate MPI execution environment
                                                                    Blank 9 (1 pt)
    MPI_Finalize();
    return 0;
}
```

Problem 2. Pthread Implementation (14 points)

```
// Pthread Version
// Compile: g++ -std=c++11 -pthread kcore_Park.cpp -o kcore_pthread
// Run: ./kcore_pthread

#include <ctime>
#include <cmath>
#include <iostream>
#include <chrono>
#include <pthread.h>
#include "string.h"
#include <cstring>
#include <vector>
#include <algorithm>

using namespace std;

vector<int> curr;
int curr_size;
vector<int> next_vec;
int next_size;
vector<int> degrees;

// Mutex to protect access to shared variables
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex3 = PTHREAD_MUTEX_INITIALIZER;


struct AllThings
{
    int num_threads;
    int my_rank;
    vector<int> offsets;
    vector<int> neighbors;
    int numVertices;
    int numEdges;
    int k;

    AllThings(int inum_threads, int imy_rank, vector<int> ioffsets,
        vector<int> ineighbors, int inumVertices,
        int inumEdges, int ik)
    {
        num_threads = inum_threads;
        my_rank = imy_rank;
        offsets = ioffsets;
        neighbors = ineighbors;
        numVertices = inumVertices;
        numEdges = inumEdges;
        k = ik;
    };
};
```

```
void *scanKernel(void *allthings)
{
    // Use curr_size to store the number of k-core
    // Use curr to store the the k-core information. curr[i] stores the index of the vertex in
vector ‘degrees’ that has degree k.
    AllThings *all = (AllThings *)allthings;

    // set num_queries_local to number of vertices divided by number of threads
    int num_queries_local =
                                                                              Blank 10 (1 pt)
        all->numVertices / all->num_threads;
    int beg_local = 0;
    // get the begin index for each thread
    if (all->numVertices % all->num_threads > all->my_rank)
    {
        num_queries_local++;
        beg_local = num_queries_local * all->my_rank;
    }
    else
    {   // set beg_local to be the starting index of vertices for the thread.
                                                                              Blank 11 (1 pt)
    beg_local = num_queries_local * all->my_rank + all->numVertices % all->num_threads;


    }
    int numVertices = all->numVertices;
    for(int i=0; i<num_queries_local; i++){
        int vid = beg_local+i;
        if (vid < numVertices && degrees[vid] == all->k)
        {
            //increment curr_size by 1 and store vid into curr.
       //You may find vector::void push_back (value_type&& val) useful.
                                                                              Blank 12 (2 pts)
            pthread_mutex_lock(&mutex);
            curr_size += 1;
            curr.push_back(vid);
            pthread_mutex_unlock(&mutex);






        }
    }
    return 0;
}


void *peelKernel(void *allthings)
{
    // Use curr_size to store the number of k-core
    // Use curr to store the the k-core information. curr[i] stores the index of the vertex in
vector ‘degrees’ that has degree k.
    AllThings *all = (AllThings *)allthings;
    // set num_queries_local to curr_size divided by number of threads
    int num_queries_local =
                                                                              Blank 13 (1 pt)
        curr_size / all->num_threads;

    int beg_local = 0;
    if (curr_size % all->num_threads > all->my_rank)
    {
        num_queries_local++;
        beg_local = num_queries_local * all->my_rank;
    }
    else
    {
        // update beg_local
```
Blank 14 (1 pt)
```
        beg_local = num_queries_local*all->my_rank + curr_size % all->num_threads;
```
```
    }
    vector<int> offsets = all->offsets;
    vector<int> neighbors = all->neighbors;
    for (int i=0; i<num_queries_local; i++){
        int vid = i+beg_local;
        if (vid < curr_size)
        {
            const int v = curr[vid];
            for (int offset = offsets[v]; offset < offsets[v + 1]; offset ++)
            {
                const int neighbor_of_v = neighbors[offset];
                int curr_degree = degrees[neighbor_of_v];
                if (curr_degree > all->k)
                {
// check if degrees[neighbor_of_v] is greater than k;
// if so, decrement it by 1 and set curr_degree to the new degrees[neighbor_of_v] value.
```
Blank 15 (2 pts)
```
                    pthread_mutex_lock(&mutex2);
                    if(degrees[neighbor_of_v] > all->k) {
                        degrees[neighbor_of_v] -= 1;
                        curr_degree = degrees[neighbor_of_v];
                    }
                    pthread_mutex_unlock(&mutex2);
```
```
                    if (curr_degree == all->k){

                        // increment next_size by 1 and store neighbor_of_v into next_vec.
                        // You may find vector::void push_back (value_type&& val) useful.
```
Blank 16 (2 pts)
```
                        pthread_mutex_lock(&mutex3);
                        next_size += 1;
                        next_vec.push_back(neighbor_of_v);
                        pthread_mutex_unlock(&mutex3);
```
```
                    } //end if (curr_degree == all->k)
                } // end if (curr_degree > all->k)
            } //end for loop on offsets
        } // end if (vid < curr_size)
    } // end for loop on num_queries_local
} //end peelKernel
```

```
// Function to perform k-core decomposition
void kCoreDecomposition(const vector<int>& offsets, const vector<int>& neighbors, int
numVertices, int numEdges){
    int num_of_threads = 3;
    // Find the minimum and maximum degrees in the graph
    int minDegree = *min_element(degrees.begin(), degrees.end());
    int maxDegree = *max_element(degrees.begin(), degrees.end());

    //for (int k = minDegree; k <= maxDegree; ++k) {
    for (int k = minDegree; k <= maxDegree; ++k) {
        // Create threads to perform k-core decomposition
        // thread_handles initialization
        pthread_t *thread_handles = (pthread_t *)malloc(num_of_threads * sizeof(pthread_t));
        for (int thread = 0; thread < num_of_threads; thread++){
            AllThings *all_t = new AllThings(num_of_threads, thread, offsets, neighbors,
numVertices, numEdges, k);

            // Perform pthread_create on scanKernel
                                                                    Blank 17 (2 pts)
            pthread_create(&thread_handles[thread], NULL, scanKernel, (void *)all_t);


        }
        // Wait for threads to complete
        for (int thread = 0; thread < num_of_threads; thread++){
            pthread_join(thread_handles[thread], NULL);
        }
        while(curr_size > 0){
            next_size = 0;
            next_vec.clear();
            // thread_handles initialization
            thread_handles = (pthread_t *)malloc(num_of_threads * sizeof(pthread_t));
            for (int thread = 0; thread < num_of_threads; thread++){
                AllThings *all_t = new AllThings(num_of_threads, thread, offsets, neighbors,
numVertices, numEdges, k);
                //perform pthread_create on peelKernel
                                                                    Blank 18 (2 pts)
                pthread_create(&thread_handles[thread], NULL, peelKernel, (void *)all_t);


            }
            // Wait for threads to complete
            for (int thread = 0; thread < num_of_threads; thread++){
                pthread_join(thread_handles[thread], NULL);
            }
            curr = next_vec;
            curr_size = next_size;
        }
        free(thread_handles);
    }
}
```

```
int main() {

    int numVertices = 6;  // Number of vertices in the graph
    int numEdges = 18;    // Number of edges in the graph

     // use the CSR data structure to store the graph
     // neighbor of vertex 0: 1
     // neighbor of vertex 1: 0, 2, 3;
     // neighbor of vertex 2: 1, 3, 4, 5;
     // neighbor of vertex 3: 1, 2, 4, 5;
     // neighbor of vertex 4: 2, 3, 5;
     // neighbor of vertex 5: 2, 3, 4;
    vector<int> offsets {0, 1, 4, 8, 12, 15, 18};
     vector<int> neighbors {1, 0, 2, 3, 1, 3, 4, 5, 1, 2, 4, 5, 2, 3, 5, 2, 3, 4};

    // Initialize the degree of each vertex
    degrees.resize(numVertices);
    for (int i = 0; i < numVertices; ++i)
        degrees[i] = offsets[i + 1] - offsets[i];

    // Perform k-core decomposition
     kCoreDecomposition(offsets, neighbors, numVertices, numEdges);

    // Output the coreness of each vertex for all k values
    for (int i = 0; i < numVertices; ++i)
        cout << "Vertex ID: " << i << ", Coreness: " << degrees[i] << endl;
    return 0;
}
```

Problem 3. CUDA Implementation (13 points)

```
#include <stdio.h>
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
#define cudaErrorCheck(ans) { gpuAssert((ans), __FILE__, __LINE__); }

inline void gpuAssert(cudaError_t code, const char *file, int line, bool abort=true)
{
   if (code != cudaSuccess)
   {
      fprintf(stderr,"GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
      if (abort) exit(code);
   }
}

using namespace std;

__global__ void scanKernel(
    const int numVertices,
    const int *degree,
    const int k,
    int *d_curr,
    int *d_curr_size
    // Use d_curr_size to store the number of vertices in the current k-core
    // Use d_curr to store the the k-core information. d_curr[i] stores the index of the
vertex in vector ‘degrees’ that has degree k.

) {
    //calculate the global tid
    int tid =
                                                                    Blank 19 (0.5 pt)
    blockIdx.x * blockDim.x + threadIdx.x;


    // if tid is a valid vertex ID and degree[tid] is k
     if (tid < numVertices && degree[tid] == k)
    {
        // increment d_curr_size atomically. write tid to the end of the d_curr array
        int write_offset_thread = atomicAdd(d_curr_size, 1);
        d_curr[write_offset_thread] = tid;
    }
}
__global__ void peelKernel(
    const int *offsets,
    const int *neighbors,
    int *degree,
    const int k,
    const int *d_curr,
    const int h_curr_size,
    int *d_next,
    int *d_next_size
) {
    //calculate the global tid
    int tid =
                                                                    Blank 20 (0.5 pt)
    blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < h_curr_size)
    {
        // pick the corresponding vertex from d_curr
        const int v = d_curr[tid];
        // iterate over each neighbor of v
        for (int offset = offsets[v]; offset < offsets[v + 1]; offset ++)
        {
            const int neighbor_of_v = neighbors[offset];
            // check whether the degree of the neighbor is greater than k
            if (degree[neighbor_of_v]  > k)
            {
                // atomically decrement the degree of the neighbor by 1
                // hint: please refer to the scanKernel function for the usage of
                // atomical operations in cuda. Same for the following two blanks.
                int previous_degree =
                                                                        Blank 21 (1 pt)
                    atomicSub(&degree[neighbor_of_v], 1);

                // if the degree is k + 1 before the subtraction, update d_next
                // if the degree is less than or equal to k before the subtraction, update
               degree[neighbor_of_v]

                if (previous_degree == k + 1)
                {
    // increment d_next_size atomically. write neighbor_of_v to the end of the d_next array
                                                                       Blank 22 (2 pts)
                    int write_offset_thread = atomicAdd(d_next_size, 1);
                    d_next[write_offset_thread] = neighbor_of_v;



                }
                else if (previous_degree <= k)
                {
                    //atomically increment degree[neighbor_of_v] by 1
                                                                        Blank 23 (1 pt)
                    atomicAdd(&degree[neighbor_of_v], 1);

                }
            }
        }
    }
}
// Function to perform k-core decomposition
void kCoreDecomposition(const vector<int>& offsets, const vector<int>& neighbors, vector<int>&
degrees, const int numVertices, const int numEdges) {
    // Find the minimum and maximum degrees in the graph
    int minDegree = *min_element(degrees.begin(), degrees.end());
    int maxDegree = *max_element(degrees.begin(), degrees.end());
    int *d_offsets, *d_neighbors, *d_degrees;
    //cudaMalloc and cudaMemcpy for d_offsets, d_neighbors, d_degrees
    cudaErrorCheck(cudaMalloc(&d_offsets, sizeof(int) * (numVertices + 1)));
    cudaErrorCheck(cudaMemcpy(d_offsets, offsets.data(), sizeof(int) * (numVertices + 1),
cudaMemcpyHostToDevice));
    cudaErrorCheck(cudaMalloc(&d_neighbors, sizeof(int) * numEdges));
    cudaErrorCheck(cudaMemcpy(d_neighbors, neighbors.data(), sizeof(int) * numEdges,
cudaMemcpyHostToDevice));
    cudaErrorCheck(cudaMalloc(&d_degrees, sizeof(int) * numVertices));
    cudaErrorCheck(cudaMemcpy(d_degrees, degrees.data(), sizeof(int) * numVertices,
cudaMemcpyHostToDevice));
```

```
    int *d_curr, *d_curr_size, h_curr_size;
    cudaErrorCheck(cudaMalloc(&d_curr, sizeof(int) * numVertices));
    cudaErrorCheck(cudaMalloc(&d_curr_size, sizeof(int)));
    cudaErrorCheck(cudaMemset(d_curr_size, 0, sizeof(int)));
    int *d_next, *d_next_size, h_next_size;
    cudaErrorCheck(cudaMalloc(&d_next, sizeof(int) * numVertices));
    cudaErrorCheck(cudaMalloc(&d_next_size, sizeof(int)));
    cudaErrorCheck(cudaMemset(d_next_size, 0, sizeof(int)));
    cudaErrorCheck(cudaDeviceSynchronize());

    // Perform k-core decomposition for each k value
    for (int k = minDegree; k <= maxDegree; ++k) {
        // in the scanKernel, each thread processes a vertex in the graph.
        const int scan_block_dim = 1024;
        // set scan_grid_dim to be the number of thread blocks for the scanKernel
        const int scan_grid_dim =
                                                                       Blank 24 (1 pt)
  numVertices /
  scan_block_dim + ((numVertices % scan_block_dim) != 0);

        // call scanKernel with scan_grid_dim and scan_block_dim
                                                                       Blank 25 (2 pt)
        scanKernel<<<scan_grid_dim, scan_block_dim>>>(numVertices, d_degrees, k, d_curr,
d_curr_size);

        // copy the value stored in d_curr_size to h_curr_size
                                                                       Blank 26 (1 pt)
        cudaMemcpy(&h_curr_size, d_curr_size, sizeof(int), cudaMemcpyDeviceToHost);

       while (h_curr_size > 0)
        {
           cudaErrorCheck(cudaMemset(d_next_size, 0, sizeof(int)));
            cudaErrorCheck(cudaDeviceSynchronize());
            const int peel_block_dim = 1024;

            //set peel_grid_dim to be the number of thread blocks for the peelKernel
            const int peel_grid_dim =
                                                                       Blank 27 (1 pt)
h_curr_size / peel_block_dim + ((h_curr_size % peel_block_dim) != 0);


            //call the kernel function peelKernel with peel_grid_dim and peel_block_dim
                                                                      Blank 28 (2 pts)
            peelKernel<<<peel_grid_dim, peel_block_dim>>>(d_offsets, d_neighbors, d_degrees, k,
d_curr, h_curr_size, d_next, d_next_size);


            // copy the value stored in d_next_size to h_next_size
                                                                       Blank 29 (1 pt)
            cudaMemcpy(&h_next_size, d_next_size, sizeof(int), cudaMemcpyDeviceToHost);

            // Update d_cur, d_next, h_curr_size, d_next_size
            swap(d_curr, d_next);
            swap(h_next_size, h_curr_size);
            swap(d_next_size, d_curr_size);
        }
    }
    cudaErrorCheck(cudaMemcpy(degrees.data(), d_degrees, sizeof(int) * numVertices,
cudaMemcpyDeviceToHost));
}
```

```
int main() {
    int numVertices = 6;  // Number of vertices in the graph
    int numEdges = 18;    // Number of edges in the graph

    // use the CSR data structure to store the graph
    // neighbor of vertex 0: 1
    // neighbor of vertex 1: 0, 2, 3;
    // neighbor of vertex 2: 1, 3, 4, 5;
    // neighbor of vertex 3: 1, 2, 4, 5;
    // neighbor of vertex 4: 2, 3, 5;
    // neighbor of vertex 5: 2, 3, 4;
    std::vector<int> offsets {0, 1, 4, 8, 12, 15, 18};
    std::vector<int> neighbors {1, 0, 2, 3, 1, 3, 4, 5, 1, 2, 4, 5, 2, 3, 5, 2, 3, 4};

    // Initialize the degree of each vertex
    vector<int> degrees(numVertices);
    for (int i = 0; i < numVertices; ++i)
        degrees[i] = offsets[i + 1] - offsets[i];

    // Perform k-core decomposition
    kCoreDecomposition(offsets, neighbors, degrees, numVertices, numEdges);

    // Output the coreness of each vertex for all k values
    for (int i = 0; i < numVertices; ++i)
        cout << "Vertex ID: " << i << ", Coreness: " << degrees[i] << endl;
    return 0;
}
```

Problem 4. Short Answer Questions (20 points)

4. (a). (4 points) Read the following MPI code snippet and specify if there is any problem. If so, briefly describe the problem(s) and how to solve them.

```
int main(int argc, char *argv[]) {
    int size, rank;
    const int root = 0;

    int datasize = atoi(argv[1]);
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int nodeDest = (rank + 1) % size;
    int nodeFrom = (rank - 1 + size) % size;

    MPI_Status status;
    int *data = new int[datasize];
    for (int i = 0; i < datasize; i++) { data[i] = rank; }

    MPI_Send(data, datasize, MPI_INT, nodeDest, 0, MPI_COMM_WORLD);
    MPI_Recv(data, datasize, MPI_INT, nodeFrom, 0, MPI_COMM_WORLD, &status);
    delete[] data;

    MPI_Finalize();
    return 0;
}
```

Your answer to question 4. (a):

```
The main problem is that all processes first send and then receive. Depending on MPI
implementations, it is possible that all processes are stuck in the sending and no one can
receive. One simple solution is to use MPI_Sendrecv so that the MPI system will schedule send
and receive; other solutions include a ring communication pattern for processes to send to
their following neighbors and receive from their preceding neighbors.
```

4. (b). (4 points) Which of the following two CUDA functions will run faster? Explain your answer.

```
// *** Function 1 ***
__global__ void kernel1(int* d_data, const int numElement) {
        const int tid = blockDim.x*blockIdx.x + threadIdx.x;
        const int nthread = blockDim.x*gridDim.x;

        for(int i = tid; i < numElement; i += nthread) {
                d_data[i] += 1;
        }
}

// *** Function 2 ***
__global__ void kernel2(int* d_data, const int numElement) {
        const int tid = blockDim.x*blockIdx.x + threadIdx.x;
        const int nthread = blockDim.x*gridDim.x;
        const int numElementPerThread = numElement/nthread;
        const int start = tid*numElementPerThread;
        int end = start + numElementPerThread;

        for(int i = start; i < end; i++) {
                d_data[i] += 1;
        }
}
```

Your answer to question 4. (b):

Kernel 1 will run faster because threads can perform coalesced memory access.

4. (c). (4 points) Is the following OpenMP code snippet correct? If so, explain why there is no scope specified for the variable "sum"; otherwise, explain what problem(s) the code has.

```
# pragma omp parallel for num_threads(thread_count) \
      reduction(+: sum) default(none) private(factor,i) shared(n)
for (i = 0; i < n; i++) {
    factor = (i % 2 == 0) ? 1.0 : -1.0;
    sum += factor/(2*i+1);
}
```

Your answer to question 4. (c):

The code is correct. The variable “sum” is a reduction variable so is protected for shared access. No need to specify its scope.

4. (d). (4 points) Are there any problems with the OpenMP parallelization in the function? If so, briefly describe the problem(s).

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

// Function to partition the array and return the pivot index
int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    #pragma omp parallel for
    for (int j = low; j <= high - 1; ++j) {
        if (arr[j] <= pivot) {
            i++;
            #pragma omp critical
            {
                swap(arr[i], arr[j]); //swap the two elements
            }
        }
    }

    swap(arr[i + 1], arr[high]); //swap the two elements

    return (i + 1);
}
```

Your answer to question 4. (d):

Yes there are problems with this OpenMP parallelization. First, the iterations of the for loop depend on each other's swap result, because some iteration's arr[j] may be other iteration's arr[i]. Second, i is a shared variable defined before the parallel for, but its increment (i++;) is not protected so there will be race condition.

4. (e). (4 points) The following Pthreads code snippet compares each pair of elements at the same index of arrays A and B of the same length, and adds the value A[i] to a global variable "sum" if A[i]>=B[i] and adds value B[i] to "sum" if A[i]<B[i], where i=0,1, ..., array_length-1. Check if there are any problems in the thread function thread_func. If so, briefly describe the problems and how to fix them.

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

int sum = 0;
int len = 1000;
vector<int> A(len);
vector<int> B(len);
int num_threads = 5;
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;

void* thread_func (void*) {
    for (int i = 0; i < len; i++) {
        if(A[i] >= B[i]){
            pthread_mutex_lock(&mutex1);
            sum += A[i];
            pthread_mutex_unlock(&mutex1);
        }
        else if(A[i] < B[i]){
            pthread_mutex_lock(&mutex2);
            sum += B[i];
            pthread_mutex_unlock(&mutex2);
        }
    }
    pthread_exit(NULL);
}

int main() {
    for (int i = 0; i < len; i++) {
        A[i] = rand() % len;
        B[i] = rand() % len;
    }

    pthread_t workers[num_threads];
    for (long i = 0; i<num_threads; i++)
        pthread_create(&workers[i], NULL, thread_func, NULL);
    for (long i = 0; i<num_threads; i++)
        pthread_join(workers[i], NULL);

    std::cout << "Final value of sum: " << sum << std::endl;

    pthread_mutex_destroy(&mutex1);
    pthread_mutex_destroy(&mutex2);

    return 0;
}
```

Your answer to question 4. (e):

One problem with this program is all threads added A or B elements to the sum. So how many threads we have, how many times A or B elements will be added to the sum. A solution to this problem is to let each thread work on one partition of the array instead of the entire array.

Another problem is the addition of A elements and that of B elements to the sum are protected by different mutexes. Depending on the progress of individual threads, there may be concurrent additions of A and B elements to the sum, which is a race condition. A simple solution is to use the same mutex for both addition statements.

[Appendices]

Appendix A. MPI APIs

1. int MPI_Init(int *argc, char ***argv)

2. int MPI_Barrier(MPI_Comm comm)

3. int MPI_Comm_size(MPI_Comm comm, int *size)

4. int MPI_Comm_rank(MPI_Comm comm, int *rank)

5. int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

6. int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

7. int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

8. MPI_Send(void* data, int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm communicator)

9. MPI_Recv(void* data, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm communicator, MPI_Status* status)

10. int MPI_Finalize()

11. Predefined reduce operations:

MPI_MAX : maximum	MPI_MIN : minimum
MPI_SUM : sum	MPI_PROD : product
MPI_LAND : logical and	MPI_BAND : bit-wise and
MPI_LOR : logical or	MPI_BOR : bit-wise or
MPI_LXOR : logical xor	MPI_BXOR : bit-wise xor
MPI_MAXLOC : max value and location	MPI_MINLOC : min value and location

12. Predefined MPI Datatypes:

MPI_CHAR	MPI_DOUBLE
MPI_FLOAT	MPI_INT
MPI_LONG	MPI_LONG_DOUBLE
MPI_LONG_LONG	MPI_LONG_LONG_INT
MPI_SHORT	MPI_C_BOOL

Appendix B. Pthreads APIs

1. int pthread_mutex_lock(pthread_mutex_t *mutex);

2. int pthread_mutex_unlock(pthread_mutex_t *mutex);

3. int pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr, void *(*start_routine)(void*), void *restrict arg);

4. int pthread_join(pthread_t *thread*, void ***value_ptr*);

Appendix C. CUDA APIs

1. cudaError_t cudaMalloc (void **devPtr, size_t size)

2. cudaError_t cudaFree (void *devPtr)

3. cudaError_t cudaMemcpy (void *dst, const void *src, size_t count, cudaMemcpyKind kind), kind: cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost

4. cudaError_t cudaDeviceSynchronize (void)

5. int atomicAdd(int *address, int val);

6. int atomicSub(int *address, int val);

Appendix D. OpenMP APIs

1. #pragma omp parallel [clause[[,]clause] ...]

2. #pragma omp for [clause[[,] clause] ...]

3. #pragma omp critical [(name)]

4. #pragma omp barrier

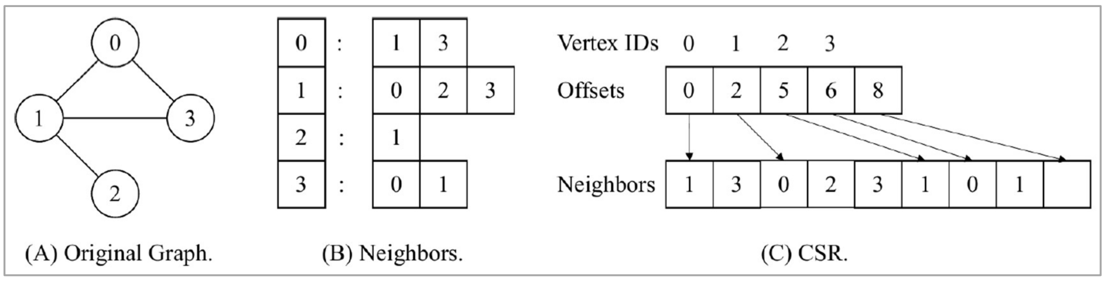
5. int omp_get_num_threads(void)

Appendix E. Sequential *k*-core Decomposition

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

using namespace std;

// Function to perform k-core decomposition
vector<int> kCoreDecomposition(vector<vector<int>>& graph) {
    int numVertices = graph.size();

    // Initialize the degree of each vertex
    vector<int> degree(numVertices);
    for (int i = 0; i < numVertices; ++i)
        degree[i] = graph[i].size();

    // Find the minimum and maximum degrees in the graph
    int minDegree = *min_element(degree.begin(), degree.end());
    int maxDegree = *max_element(degree.begin(), degree.end());

    // Create a copy of the degree vector for coreness calculation
    vector<int> coreness(degree.begin(), degree.end());
    vector<bool> removed(numVertices, false);

    // Create a queue to store the vertices with degree less than k
    queue<int> q;

    // Perform k-core decomposition for each k value
    for (int k = minDegree+1; k <= maxDegree; ++k) {
        // Add all the vertices with degree less than k to the queue
        for (int i = 0; i < numVertices; ++i) {
            if (degree[i] < k)
                q.push(i);
        }

        while (!q.empty()) {
            int vertex = q.front();
            q.pop();

            // Skip isolated vertices
            if (degree[vertex] == 0 && removed[vertex])
                continue;

            // Reduce the degree of the neighboring vertices and update their coreness
            for (int neighbor : graph[vertex]) {
                if (degree[neighbor] > 0) {
                    degree[neighbor]--;
                    if (degree[neighbor] < k) {
                        q.push(neighbor);
                        coreness[neighbor] = max(coreness[neighbor], k - 1);
                    }
                }
            }

            // Set the degree of the current vertex to 0
            removed[vertex] = true;
            degree[vertex] = 0;
            coreness[vertex] = k - 1;
        }
    }
    return coreness;
}
```

```
int main() {
    int numVertices = 6;  // Number of vertices in the graph

    // Create an adjacency list to represent the graph
    vector<vector<int>> graph(numVertices);

    // Add edges to the graph
    graph[0] = {1};
    graph[1] = {0, 2, 3};
    graph[2] = {1, 3, 4, 5};
    graph[3] = {1, 2, 4, 5};
    graph[4] = {2, 3, 5};
    graph[5] = {2, 3, 4};

    // Perform k-core decomposition
    vector<int> coreness = kCoreDecomposition(graph);

    // Output the coreness of each vertex for all k values
    for (int i = 0; i < numVertices; ++i)
        cout << "Vertex ID: " << i << ", Coreness: " << coreness[i] << endl;

    return 0;
}
```

Appendix F. CSR Data Structure (Used in Pthread and CUDA implementations)



A graph can be stored in a CSR structure, where one array is used to store the neighborhood information in sequential order, and the other array stores the offsets for corresponding vertices.

Note: The length of the array for storing offsets is the number of vertices plus one, as the first element of offsets is zero and points to the first element of the array of neighbors. Accordingly, the length of the array for storing neighbors is the number of edges plus one.