CS 314 Principles of Programming Languages
Project 3: Efficient Parallel Graph Matching

THIS IS NOT A GROUP PROJECT! You may talk about the project and possible solutions in general terms, but must not share code. In this project, you will be asked to implement two parallel graph matching algorithms. Your program should take a legal matrix-market matrix file as input and output the matching results.

This document is not a complete specification of the project. You will encounter important design and implementation issues that need to be addressed in your project solution. Identifying these issues is part of the project. As a result, you need to start early, allowing time for possible revisions of your solution.

1 Background

1.1 The Graph Matching Problem

Given a graph $G = (V, E)$, while V is the set of vertices (also called nodes) and $E \subset |V|^2$. A **matching** M in G is a set of pairwise non-adjacent edges which means that no two edges share a common vertex. A vertex is **matched** if it is an endpoint of one of the edges in the matching. Otherwise the vertex is **unmatched** [3]. In Figure 1, we show three different matchings for the same graph.

A **maximum matching** can be defined as a matching where the total weight of the edges in the matching is maximized. In Figure 1, (c) is a **maximum matching** because the total weight of the edges in the matching is 7, and there could be no other matching that has total weight greater than 7 for this graph.

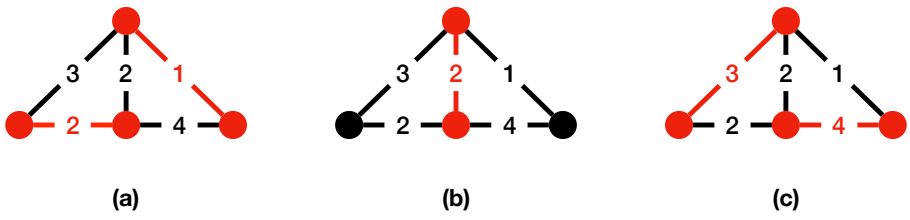

Figure 1: Graph Matching Examples

1.2 Parallel Graph Matching Algorithms

The graph matching problem is not easy to parallelize. Most existing matching algorithms such as **blossom algorithm** are embarrassingly sequential. Here we describe two *handshaking*-based algorithms [2] that are amenable to parallelization.

1.2.1 One-Way Handshaking Matching

Given a graph G, two vertices *shake hands* only when there is an edge between these two and they are the strongest neighbor of each other. We will define "strongest neighbor". The edge that connects the two handshaking vertices is added to the matching M. We show an example of **one-way handshaking** matching in Figure 2.

To identify handshaking vertices, each vertex v in G extends a hand to its strongest neighbor. Here the strongest neighbor of a vertex v is the neighbor vertex that sits on the **maximum-weight** edge incident to v. This is a greedy algorithm that aims to maximize the total weight in the matching. If there are multiple incident edges of v that have maximum weight, the neighbor vertex that has the **smallest vertex index** will be chosen as the strongest neighbor. For example, in Figure 2(b), the strongest neighbor of vertex E is vertex B because it has an edge weight of 4 and a smaller index (alphabetical order) than vertex F.

For each vertex, we check if its strongest neighbor extends a hand back. If so, these two vertices are matched. Then the corresponding edge will be added to the matching. For example, in Figure 2(c), vertex B extends a hand to vertex E and vertex E also extends a hand to vertex B, so edge BE will be added to the matching.

At every pass of the process, we check all the vertices that are not matched from previous passes (each vertex is checked once in each pass), and identify if there is any new edge that can be added to the matching. We repeat this until no more edges can be added to the matching. We show the passes in Figure 2(b), (c), (d),(e) and (f). The handshaking method is highly data parallel, since each vertex can be processed independently and there is no data race.

1.2.2 N-Way Handshaking Matching

In one-way handshaking matching, it is possible that one vertex will have extended hands from multiple neighbors, but at most one of these neighbors can be matched. This may affect the matching efficiency. For example, in Figure 2 (b), both vertex D and vertex H extend hands to vertex E. However, since vertex E's strongest neighbor is vertex B, so neither vertex D nor H will be matched at this pass of handshaking in Figure 2 (b).

Instead of extending one hand, **N-way handshaking matching** allows each vertex to extend N hands ($N > 1$). We show an example of 2-way handshaking matching in Figure 3. In Figure 3(b), each vertex extends (up to) 2 hands at once, which extends to its strongest and second strongest neighbors. For example, vertex H extends one hand to vertex E which is its strongest neighbor and another hand to vertex I which is the second strongest neighbor.

In the next step, we take the edges whose two end points do not shake hands out of consideration (as if we "discard" edges). In Figure 3(b), there is no handshaking between

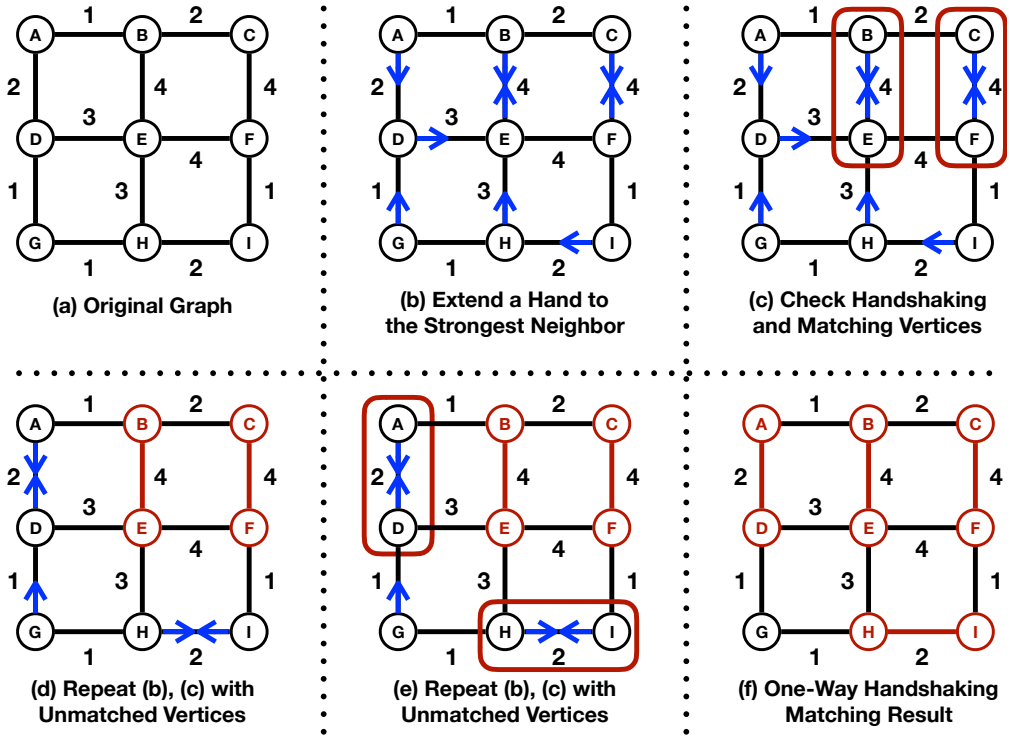

Figure 2: One-Way Handshaking Matching Example

vertex H and vertex E, so the edge HE is "as if" discarded (before next step in the same pass). After this, we will obtain an updated graph with a max degree N (N=2 in Figure 3(c)), we will refer to this graph as *N-way graph* in the remaining of the project description.

We now do **one-way handshaking matching** on the updated *N-way graph*. The matching on the *N-way graph* may yield more matched vertices. For example, in Figure 3(e), both vertex H and vertex D can be matched in the first pass, while compared with *one-way matching* in Figure 2, both vertex H and vertex D have to be matched in the second pass.

1.3 POSIX threads - pthreads

A thread is defined as an *independent* stream of instructions that can be scheduled to run in its own context. Multiple threads run in multiple active contexts.

Historically, hardware vendors have implemented their own proprietary versions of threads. For UNIX systems, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard. Implementations that adhere to this standard are referred to as POSIX threads, or **pthreads**[1].

For more details on how to use pthreads API to write a parallel program and how to compile a pthreads program, please refer to the *pthread tutorial* [1] and the recitations.

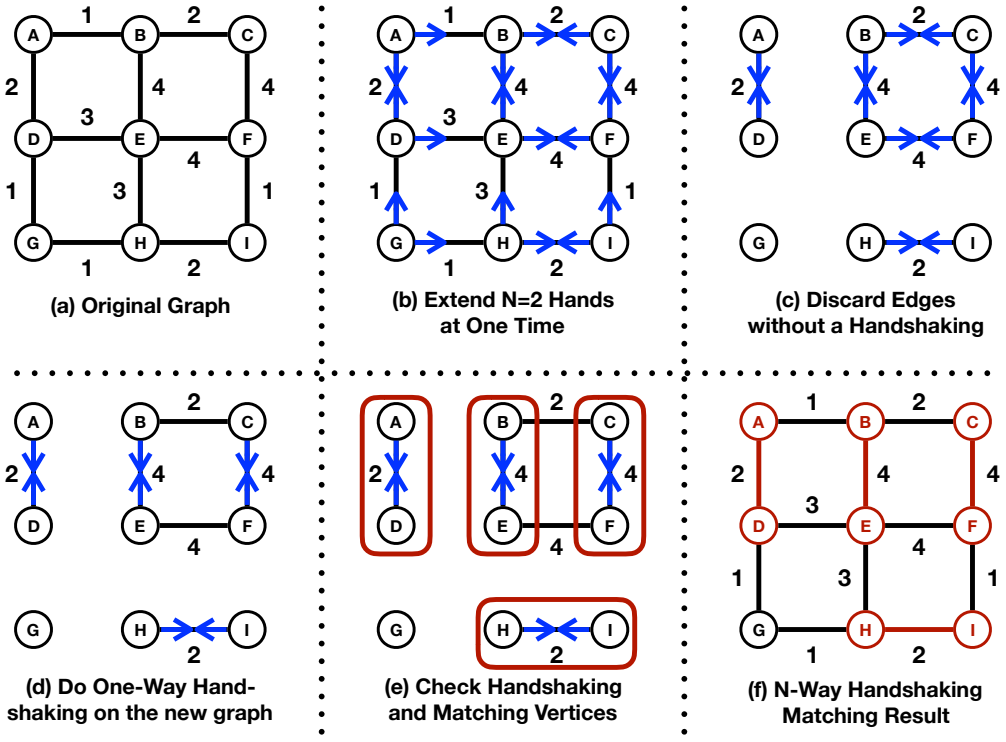

Figure 3: N-Way Handshaking Matching Example (N=2)

2 Implementation

In this project, you will be asked to do the following:

1) implement the parallel **one-way handshaking matching** algorithm
2) **Extra Credit:** implement the parallel **N-way handshaking matching** algorithm

2.1 Data Structure

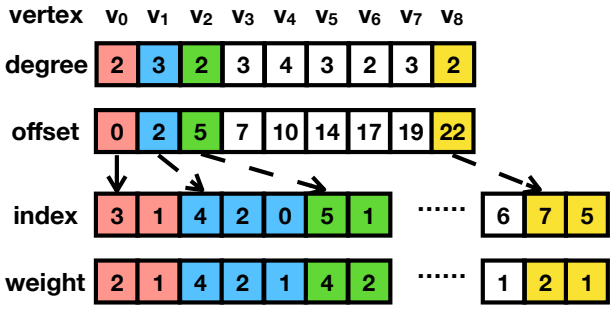

Figure 4: Adjacency Array Data Structure

Un-directed weighted graph In this project, we use adjacency lists to represent an un-directed weighted graph. There are four arrays: index[], offset[], degree[], and weight[]. The array index[] keeps the neighbor lists of all vertices, for instance, node v_0's neighbor list is followed by neighbor v_1's neighbor list, and so on. The array offset[] stores where a node's neighbor list starts in the index[] array. The corresponding weight[] array stores the weight of each edge. The array degree[] stores the degree of each vertex, which is the number of neighbors for each vertex.

An example of adjacency array representation is shown in Figure 4. For vertex v_2, its degree is 2 (degree[2] = 2). The offset[2] value is 5 which means its neighbor list starts at index[5], thus index[5] = 5 (corresponding to vertex v_5), index[6] = 1 (corresponding to vertex v_1), and vertex v_5 and vertex v_1 are two neighbors of vertex v_2.

Pleas be aware that within a neighbor list (of a specific vertex), the neighbors are sorted in descending order such that the strongest neighbor is always placed as the first item and the second strongest neighbor is placed as the second item and so on. We already sorted the neighbors of each vertex for you. You do not have to sort them to find the strongest neighbor, however, you do need to filter out the nodes that are already matched from previous passes.

We have provided graph I/O functions and code for reading/parsing the graphs, the pointers to the four arrays: index[], offset[], weight[], and degree[] are stored in *GraphData* struct in the provided code package. Here is what *GraphData* looks like:

```
struct GraphData {
    int nNodes;
    int nEdges;
    int *offset;
    int *degree;
    int *index;
    double *weight;
}
```

In the main function of the provided code package, it calls readmm(*inputFile*, &*graph*) to read and parse an input graph file into the data object *graph*. Please check **DataStructure.h** and **utils.c** for more details.

***N-way* graph** In **N-way handshaking matching** algorithm, you need to generate the *N-way graph*. We use two arrays to represent the *N-way graph*: nWayGraphDegree[] and nWayGraph[]. nWayGraphDegree[] is an array that stores the degree of each vertex in the *N-way graph*. nWayGraph[] represents the adjacency list of the nodes in the *N-way graph*. Since every node in the *N-way graph* has at most N neighbors, we allocate N*$node\_number$ elements for the nWayGraph[] array, such that the neighbors of the vertex v_i are stored starting from $nWayGraph[i * N]$ to at most $nWayGraph[i * N + nWayGraphDegree[i]\text{-}1]$ in descending order such that the strongest neighbor are placed first. In Figure 5, we show an example of the *2-way graph*.

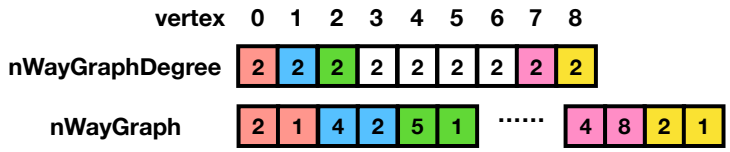

Figure 5: 2-Way Graph Data Structure

Matching results The matching algorithm runs for as many passes as it needs, until no edge can be added into the matching. It is possible that some vertices cannot be matched to any neighbor.

In this project, we use the array res[] to store the matching results. We provide several defined constants to represent the status of a vertex in **match.h**. The array res[] is initialized to **UNMATCHED** (-1) for each vertex. When the graph matching program terminates, for vertex i, res[i] is either its matched vertex index or **NO_MATCHED_NODES** (-2) which represents vertex i is not matched. In general:

$$res[i] = \begin{cases} j, & \text{if vertex i and vertex j are matched} \\ -1, & \text{initial value} \\ -2, & \text{vertex i is not matched} \end{cases}$$

In the main function, it calls write_match_result($outputFile$, res, $nNodes$) to write the matching results stored in res into output file $outputFile$. $nNodes$ represents the total number of vertices.

2.2 Work Balancing by Vertices

```
i = current thread index
nodeToProcess = the list of nodes to process
workchunk = (numNodes + threadNum - 1) / threadNum
beg = i * workchunk
end = min(beg + workchunk, numNodes)
for each v in { nodeToProcess[beg], nodeToProcess[beg+1], ..., nodeToProcess[end-1] }
do
    ... your code for matching
end for
```

Figure 6: Balancing Based on Vertex Number

To implement parallel handshaking-based matching algorithms, we need to map the tasks into different threads in a load balanced way. In this project, each thread will be in charge of a subset of vertices. A global barrier synchronization is performed after each pass of matching (the barrier code is already provided for you). The algorithm distributes the vertices evenly to the co-running threads. Assuming there are $nNodes$ vertices to be processed, the total number of threads is $threadNum$, each thread will process around $nNodes/threadNum$

nodes. Since $nNodes$ is not necessarily a multiply of $threadNum$, the last thread might be assigned $\leq nNodes/threadNum$ vertices. The code snippet in Figure 6 shows how each thread should find the set of the vertices it is in charge of. Please use this vertex balancing method for all your parallel function implementation.

2.3 One-Way Handshaking

We have described the **one-way handshaking matching** algorithm in Section 1.2.1. In **one-way handshaking**, each vertex extends a hand to its strongest neighbor. Next it checks if there is a handshaking between itself and its strongest neighbor.

You only need to complete the following functions in *oneway.c*:

1. **extend_one_hand(int threadId, int threadNum, GraphData graph, int nodeNum, int *nodeToProcess, int *res, int *strongNeighbor)**

   Function Description:
   Each thread needs to be assigned a subset of vertices from nodeToProcess[] array. See Section 2.2 for load balancing method. For each vertex with ID v in this subset, find its strongest neighbor and store the vertex index in the array $strongNeighbor[v]$. The pseudo code is shown below.

```
Vi = the set of vertices assigned to thread i
N(k) = the set of (sorted) neighbors for vertex k
for each v in Vi
    for each u in N(v)  do
        if (res[u] == UNMATCHED) then
            strongNeighbor[v] = u
            break out loop u
        end if
    end for
end for
```

   Input Parameters:
   threadId: The thread index
   threadNum: Total thread number
   graph: The graph object
   nodeNum: The number of vertices
   nodeToProcess Each element is a vertex ID, and this array is usually used to pass the unmatched vertices to the processing function. The size of the array is *nodeNum*.
   res: The array that stores the matching status for each vertex. The size of the array is the total number of vertices in the graph.

   Output Parameters:
   strongNeighbor: The array that stores the index of the strongest neighbor for each node. The size of the array is the total number of vertices in the graph.
   e.g. $strongNeighbor[] = \{3,4,5,4,1,2,3,4,7\}$ for the example in Figure 2

2. **check_handshaking(int threadId, int threadNum, int nodeNum, int *nodeToProcess, int *strongNeighbor, int *res)**

   Function Description:
   Each thread needs to be assigned a subset of vertices from nodeToProcess[] array. See Section 2.2 for load balancing method. For each vertex v in this subset, given its strongest neighbor $strongNeighbor$[v], update res[v] correspondingly. The pseudo code is shown below.

```
Vi = the set of vertices assigned to thread i
for each v in Vi do
    s = strongNeighbor[v]
    if (v == strongNeighbor[s]) then
        res[v] = s
    end if
end for
```

   Input Parameters:
   threadId: Thread index
   threadNum: Total thread number
   nodeNum: The number of vertices
   nodeToProcess Each element is a vertex ID, and this array is usually used to pass the unmatched vertices to the processing function. The size of the array is *nodeNum*.
   strongNeighbor: The array that stores the index of the strongest neighbor for each vertex. The size of the array is the total number of vertices in the graph.

   Output Parameters:
   res The array that stores the matching status for each vertex. The size of the array is the total number of vertices in the graph.
   E.g. $res[] = \{-1,4,5,-1,1,2,-1,-1,-1\}$ for the example in Figure 2

2.4 Extra Credit (20%): N-Way Handshaking Matching

For **N-way handshaking matching**, each vertex extends N hands, then the algorithm neglects (as if "discards") those edges whose two end nodes have no handshaking, resulting in the *N-way graph*. Upon the *N-way graph*, *one-way* matching is performed.

You only need to implement the following functions in *nways.c*:

1. **generate_n_way_graph(int threadId, int threadNum, GraphData graph, int nWays, int nodeNum, int *nodeToProcess, int *res, int *nWayGraphDegree, int *nWayGraph)**

Function Description:
Each thread needs to be assigned a subset of vertices from nodeToProcess[] array. See Section 2.2 for load balancing method. For each vertex v, find its N strongest neighbors. Then the algorithm obtains the *N-way graph* information which is represented by $nWayGraphDegree[]$ and $nWayGraph[]$. The pseudo code is shown below. Note that the neighbors of each vertex in the *N-way graph* also need to be **sorted** such that the strongest neighbor is placed first.

```
Vi = the set of vertices assigned to thread i
N(k) = the set of (sorted) neighbors for vertex k
for each v in Vi
    degree = 0
    for each u in N(v) in sorted order do
        if (res[u] == UNMATCHED) then
            nWayGraph[N * v + degree] = u
            degree++
            if (degree == nWays) then
                break for loop u
            end if
        end if
    end for
end for
```

Input:
threadId: Thread index
threadNum: Total thread number
graph: Graph object
nWays: the number of hands extended
nodeNum: The number of vertices
nodeToProcess: Each element is a vertex ID, and this array is usually used to pass the unmatched vertices to the processing function. The size of the array is *nodeNum*.
res: The array that stores the matching status for each vertex. The size of the array is the total number of vertices in the graph.

Output:
nWayGraphDegree: The array that stores the vertex degree of the *N-way graph*. The size of the array is the total number of vertices in the graph,
nWayGraph: The array that stores the N strongest neighbors of each vertex. The size of the array is the total number of vertices multiplied by N.
e.g. $nWayGraphDegree[] = \{2,2,2,2,2,2,2,2,2\}$
$nWayGrap[] = \{3,1,4,2,5,1,4,0,1,5,2,4,3,7,4,8,7,5\}$ for the example in Figure 3

2. **prune_n_way_graph(int threadId, int threadNum, int nWays, int nodeNum, int *nodeToProcess, int *nWayGraphDegree, int *nWayGraph, int**

***strongNeighbor)**

Function Description:
Each thread needs to be assigned a subset of vertices from nodeToProcess[] array. See Section 2.2 for load balancing method. For each vertex, it finds the strongest neighbor in the N-way graph. The pseudo code is shown below. Note that the neighbors of each vertex are **sorted** such that the strongest neighbor is placed first.

```
Vi = the set of vertices assigned to thread i
S(k) = the set of N strongest neighbors for
vertex k
for each v in Vi
    for each u in S(v) in sorted order do
        for each x in S(u) in sorted order do
            If (x == v) then
                strongNeighbor[x] = u
                break for loop u
            end if
        end for
    end for
end for
```

Input Parameters:
threadId: Thread index
threadNum: Total thread number
nWays: The number of hands extended
nodeNum: The number of vertices
nodeToProcess: Each element is a vertex ID, and this array is usually used to pass the unmatched vertices to the processing function. The size of the array is *nodeNum*.
nWayGraphDegree: The array that stores the vertex degree of the *N-way graph*. The size of the array is the total number of vertices in the graph,
nWayGraph: The array that stores the N strongest neighbors of each vertex. The size of the array is the total number of vertices multiplied by N.

Output Parameters:
strongNeighbor: The array that stores the the strongest neighbor for each vertex. The size of the array is the total number of vertices in the graph.
e.g. $strongNeighbor[] = \{3,4,5,0,1,2,-2,8,7\}$ for the example in Figure 3

2.5 Filtering Matched Vertices

In handshaking algorithms, each pass except the last one will produce a set of matched vertices. The vertices that are matched from previous passes need not to be considered in

the later passes. You will need to filter out the nodes that are already matched before starting a new pass. Then you can distribute the unmatched vertices evenly to different threads again.

At the first pass, all the vertices need to be processed and they are evenly distributed to co-running threads. Starting from the second pass, you will need to count the number of unmatched nodes. An example is shown in Figure 7. Each thread counts the number of unmatched vertices for the set of vertices it is in charge of, and stores the counts in the array *nodeCount*[]. In Figure 7, for instance, *nodeCount*[0] = 1 since only vertex 0 is not matched among the set of vertices that are processed by thread 0.

Next, an exclusive prefix sum is performed on *nodeCount* array. An exclusive prefix sum of an array $x[]$ is another array $y[]$ that:

$$y[i] = \begin{cases} 0, & \text{if i = 0} \\ \sum_{j=0}^{i-1} x[j], & \text{otherwise} \end{cases}$$

The purpose of using exclusive prefix sum is to find out where to store the unmatched vertices in the array *newNodeToProcess*[]. For example, in Figure 7, *nodeCount* = {1, 1, 3}, after exclusive prefix sum, it is y ={0, 1, 2} and threads i know where to place the unmatched vertices it is in charge of: starting from newNodeToProcess[y[i]]. Every thread then stores the unmatched vertices it finds to the *newNodeToProcess* array that will be used for next pass.

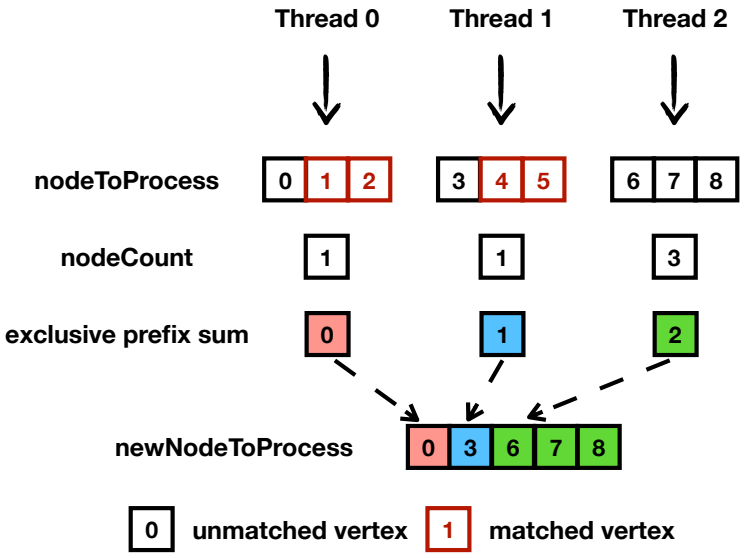

Figure 7: Filter Matched Nodes from Vertex Array

You only need to complete the following functions in *filter.c*:

1. **count_unmatched_vertices(int threadId, int threadNum, int nodeNum, int *nodeToProcess, int *res, int *nodeCount)**

Function Description:
Each thread counts the unmatched vertices in the set of vertices it is in charge of. The count by thread i is stored in $nodeCount[i]$. The pseudo code is shown below.

```
Vi = the set of vertices assigned to thread i
for each v in Vi do
    if (res[v] == UNMATCHED) then
        nodeCount[i]++
    end if
end for
```

Input:
threadId: Thread index
threadNum: Total thread number
nodeNum: The number of vertices
nodeToProcess: Each element is a vertex ID, and this array is usually used to pass the unmatched vertices to the processing function. The size of the array is *nodeNum*.
res: The array that stores the matching status. The size of the array is the number of vertices.

Output:
nodeCount: Each element of the array stores the number of unmatched vertices collected by each thread. The size of the array is the total number of threads.
e.g. $nodeCount[] = \{1,1,3\}$ for the example in Figure 2.

2. **update_remain_nodes_index(int threadId, int threadNum, int *nodeToProcess, int *startLocations, int *res, int nodeNum, int newNodeToProcess)**

Function Description:
Each thread stores the unmatched vertices it finds into the $newNodeToProcess$ array starting at the location provided in $startLocations$. The pseudo code is shown below.

```
Vi = the set of vertices assigned to thread i
for each v in Vi do
    if (res[v] == UNMATCHED) then
        offset = startLocations[i]++
        newNodeToProcess[offset] = v
    end if
end for
```

Input:
threadId: Thread index

threadNum: Total thread number
nodeToProcess: Each element is a vertex ID, and this array is usually used to pass the unmatched vertices to the processing function. The size of the array is *nodeNum*.
startLocations: The array that stores the start location of each thread in $newNodeToProcess$. The size of the array is the number of threads plus one.
res: The array that stores the matching status. The size of the array is the number of vertices.
nodeNum: The number of vertices to process

Output:
newNodeToProcess: Each element is a vertex ID. The size of the array is the number of vertices.
$newNodeToProcess[] = \{0,3,6,7,8\}$ for the example in Figure 3

.

3 Grading

We provide six real world matrices for you to test your code. In the code package, we have provided the function for transforming the input matrix to *GraphData graph* in main.c. Six matrices and four hidden matrices will be used to grade your program. Your programs will be graded based on functionality. You will receive 0 credit if we cannot compile/run your code on *ilab* machines. All grading will be done automatically on *ilab* machines.

4 How to Get Started

The code package for you to start with is provided on **Sakai**. Create your own directory on the ilab cluster, and copy the entire provided project folder to your home directory or any other one of your directories. Make sure that the read, write, and execute permissions for groups and others are disabled (`chmod go-rwx <directory_name>`). Please read the README file before you start.

4.1 Compilation and Execution

Compilation: We have provided a **Makefile** in the sample code package.
make match should compile everything into a single executable called **match**.
make clean should remove all generated executable and intermediate files.

Execution: To see the usage of the program, please compile the program and simply run **./match** and the usage information will be shown. Please **DO NOT** change the usage of the program.

Program Output: The program will output the matching results to the file you specify in the arguments and print the time and the number of iterations to get the maximal matching on the screen. We have handled this part for you. Please **DO NOT** change any other file that is not **oneway.c**, **nways.c**, and **filter.c**.

4.2 Input Matrix List

The following six matrix files are provided for you to test your graph matching implementations. The matrix will be transformed to a weighted un-directed graph automatically by the program. A micro test matrix "test.mtx" is provided in the code package. The test matrix is the example we used in the project description.

af_shell10 https://sparse.tamu.edu/MM/Schenk_AFE/af_shell10.tar.gz
cant https://sparse.tamu.edu/MM/Williams/cant.tar.gz
consph https://sparse.tamu.edu/MM/Williams/consph.tar.gz
dawson5 https://sparse.tamu.edu/MM/GHS_indef/dawson5.tar.gz
hood https://sparse.tamu.edu/MM/GHS_psdef/hood.tar.gz
ldoor https://sparse.tamu.edu/MM/GHS_psdef/ldoor.tar.gz

If the disk space on ilab machine is not enough, please download one matrix at one time or use /freespace to store the matrices.

5 What to Submit

You need to submit two files: **oneway.c** and **filter.c**. If you finish the extra credit part for **N-way handshaking**, please also submit **nways.c**. Please **DO NOT** include any output matching results file and input graph/matrix files in your submission.

6 Questions

All questions regarding this project should be posted on Sakai forum. Enjoy the project!

References

[1] Blaise Barney. Posix threads programming. `https://computing.llnl.gov/tutorials/pthreads/`, 2017.

[2] Jonathan Cohen and Patrice Castonguay. Efficient graph matching and coloring on the gpu.

[3] Wikipedia contributors. Matching (graph theory) — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Matching_(graph_theory)&oldid=869039028`, 2018. [Online; accessed 20-November-2018].