COMP 429/529 Parallel Programming: Project 3

Due: midnight on Friday, Jan 7, 2022

Notes: You may discuss the problems with your peers but the submitted work must be your own work. Late assignment will be accepted with a penalty of -20 points per day. Please submit your source code through blackboard. This assignment is worth 20% of your total grade. In this assignment, you can work with a partner or do it alone.

Description

In this assignment you will implement an MPI parallel version of Breadth First Search (BFS) graph traversal algorithm.

BFS

Breadth First Search is an algorithm for traversing a graph data structure, in which nodes are discovered level by level. A **graph** is a data structure, consisting of a set of **vertices** (nodes) V and a set of connections E between them named **edges**. Traversal is the process of discovering all nodes through the edges between them, starting with a **source** node. Unlike a more specific type of a graph named *tree*, BFS on a general graph needs to take into account possible edges to already visited nodes, as cycles are allowed. Additionally, in this assignment you will need to provide the depth (level) at which each of the nodes has been discovered, recorded in the *result* array. For example, for the source vertex it would be 0, for its directly adjacent vertices (neighbors) it would be 1, and so on.

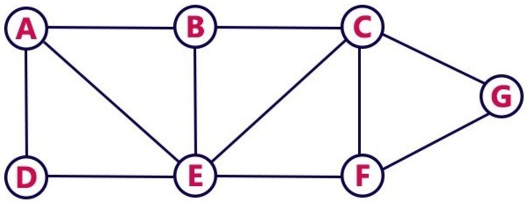
The research around this algorithm has given us many possible ways of implementing this algorithm, as well as ways to parallelize it. In this assignment we will try 2 of them.

- Distributed approach: simply splitting the graph.
- Work-efficient approach: splitting the workload at every iteration.

Sequential BFS

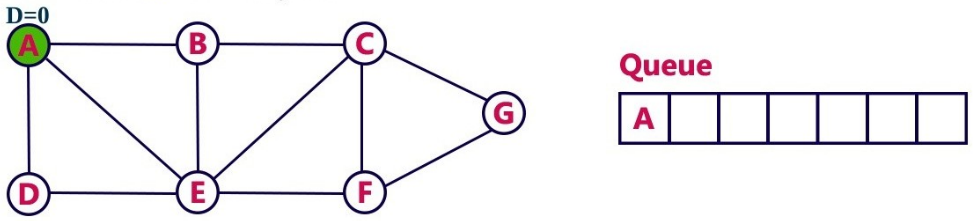
```
BFS (G, s) //Where G is the graph and s is the source node
      let Q be queue.
      Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked.
      mark s as visited. set result[s] to 0
      while ( Q is not empty)
           //Removing that vertex from queue, whose neighbour will be visited now
           v  =  Q.dequeue( )
          for all neighbours w of v in Graph G //processing all the neighbours of v
               if w is not visited
                         Q.enqueue( w )  //Stores w in Q to further visit its neighbour
                         mark w as visited
                         result[w] = result[v] + 1  //Saving the depth value
```

Consider the following example graph to perform BFS traversal



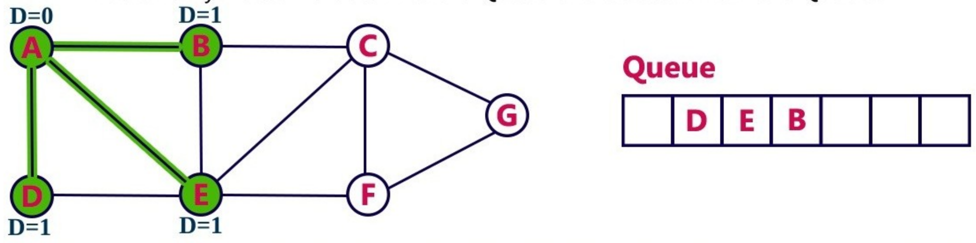
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.



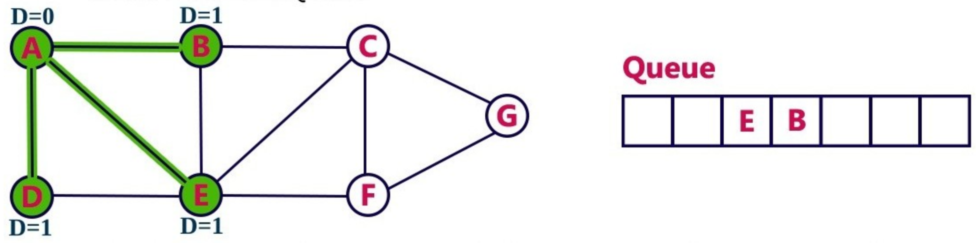
Step 2:

D++ - Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..



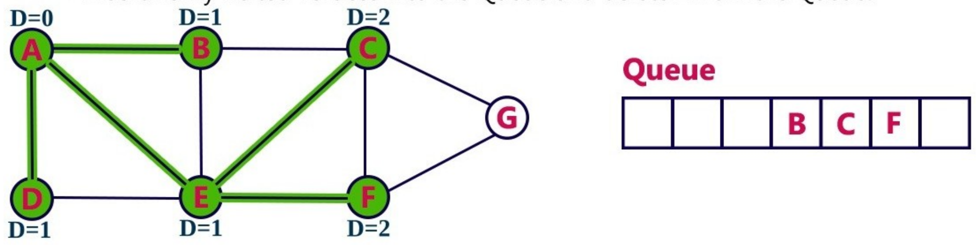
Step 3:

D++ - Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.



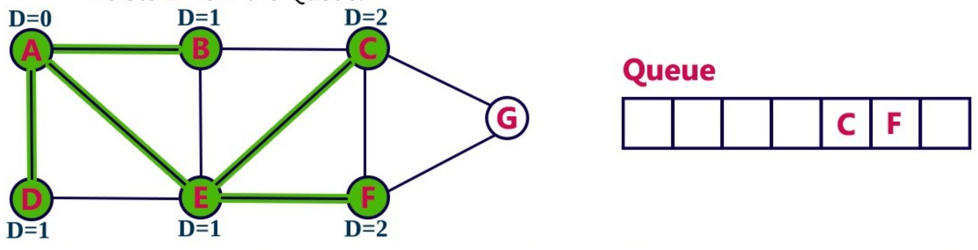
Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.



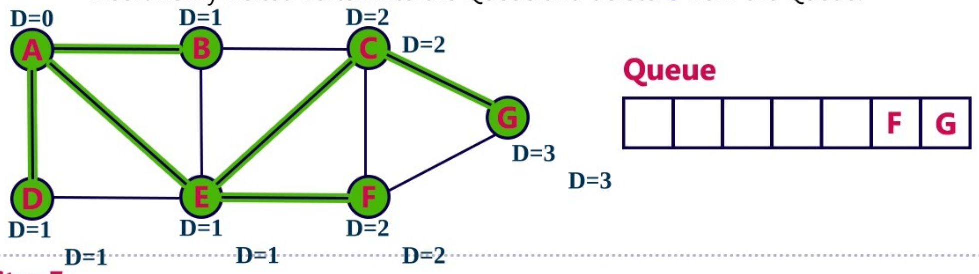
Step 5:

D++ - Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.



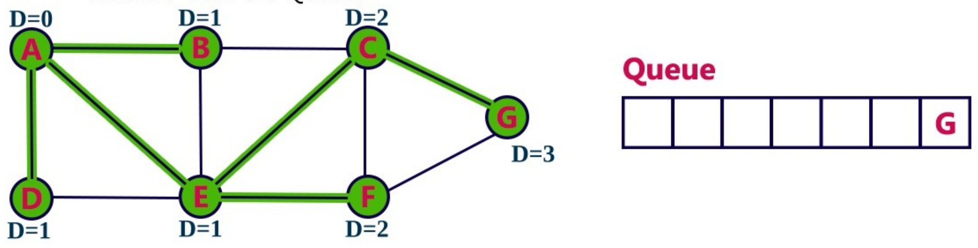
Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.



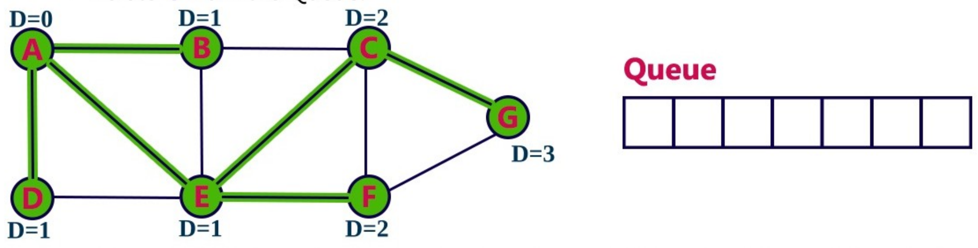
Step 7:

D++ - Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

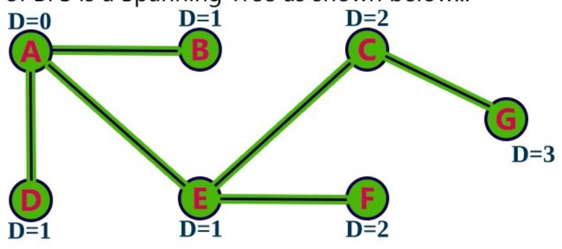


Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.

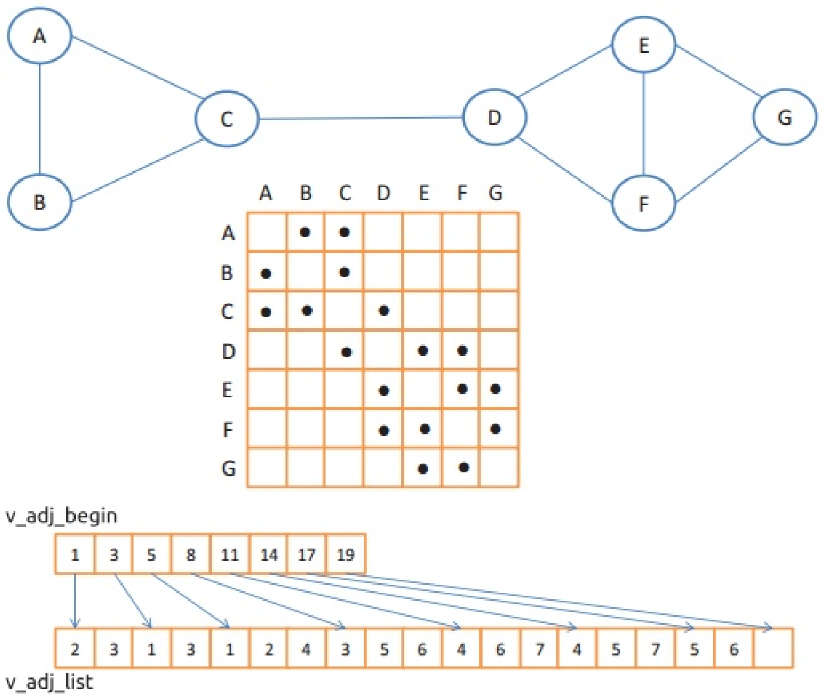


- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



CSR graph

Every graph can be represented in a form of a sparse binary matrix, where each *row* index represents a source vertex and each bit in every *column* signifies whether there is an edge going from vertex *row* to vertex *column*. So, a row is an adjacency list of a vertex associated with the row index. In this assignment, you will be using the graph in Compressed Sparse Row (CSR) format. As it is shown on Figure 1, it allows to reduce the matrix to an array of edge destination vertices `v_adj_list`, and an array of offsets to `v_adj_list` named `v_adj_begin`. Every item in `v_adj_begin` is an index of `v_adj_list`, which marks the start of the new row in the graph matrix. In such a way, in CSR the neighbors of vertex `v` are `v_adj_list[v_adj_begin[v]:v_adj_begin[v+1]`. The code provided to you also gives an array of node degrees (degree is a number of outgoing edges of a vertex) `v_adj_length`, in which `v_adj_length[v]=v_adj_begin[v+1]-v_adj_begin[v]`.



	A	B	C	D	E	F	G
A		●	●				
B	●		●				
C	●	●		●			
D			●		●	●	
E				●		●	●
F				●	●		●
G					●	●	

Figure 1: A representation of a graph as a sparse matrix and a CSR vector.

Assignment

Part I: Distributed approach

In this part, you will have to split the graph vertices. This might seem trivial at first, but since every vertices' neighbor might be assigned to a different process, communication is required. To implement it, you have to split the data structures that represent the graph evenly across the processes. However, you would still need a set of variables or a data structure that will be shared across the processes to make sure each of them has relevant and updated information about the already visited nodes and their corresponding depths.

Part II: Work efficient approach

You might have noticed that the previous approach relies on randomness for workload balancing. In this part, let's assume we don't need memory distribution and we can store a whole copy of a graph on each processor. Similar to the serial queue-based code, you will have an array of vertices that have been discovered in the previous level, called **frontier**, which will be updated with a new set of vertices as search depth is increased. However, unlike in the queue implementation, here you will iterate over all the vertices in the **frontier**, so each iteration of an algorithm will correspond to traversing a level. For balanced parallelization, instead of splitting the vertices of the graph, at each iteration you will be splitting the **frontier**. This way, at every iteration each processor will traverse the neighbors of an equal number of nodes.

The serial pseudocode for frontier-based BFS is given below.

```
BFS_frontier (G, s) //Where G is the graph and s is the source node
      let frontier_in and frontier_out be array.
      frontier_in[0] = s //Inserting s to the frontier
      mark s as visited, set result[s] to 0
      while ( frontier_in is not empty)
           out_size = 0
           for all vertices v in frontier
                for all neighbours w of v in Graph G
                    if w is not visited
                        frontier_out[out_size++] = w  //Inserting w to the frontier
                        mark w as visited
                        result[w] = result[v] + 1  //Saving the depth value
            swap frontier_in and frontier_out
```

Serial Code

We are providing you with a working serial program that has the 3 different BFS implementations. All of them can be found in `bfs_sequential.cpp` file. `bfs_main.cpp` handles launching, parameter parsing and graph reading. Graph reading code will provide you with information about the graph, which you can use in order to see in what scenario a specific method is beneficial and provide answers to the questions being asked.

- **Queue**: the one described in the first pseudo-code (also in the figure) that uses a queue
- **Naive**: a less efficient version, which goes through all the vertices at every iteration. Use this implementation as a basis for Part I. The function `mpi_vertex_dist.cpp` for Part I already contains the same code, you will have to add the MPI functionality and (if needed) additional data structures.
- **Frontier**: an approach explained in the pseudocode from Part II. Use this implementation as a basis for Part II. The function `mpi_frontier.cpp` for Part II already

contains the same code, you will have to add the MPI functionality and (if needed) additional data structures.

```
The program may be run via the command line using mpirun (openmpi) or mpiexec (mpich):
mpirun [-np <int>] ./bfs <str>filename <int>source <int>num_runs <int>method
mpiexec [-np <int>] ./bfs <str>filename <int>source <int>num_runs <int>method

With the arguments interpreted as follows:
<str> filename -> Path to the .mtx graph file
<int> source -> The source vertex.
<int> num_runs -> Number of runs of each method, better to just use 1
<int> method -> 0 = Queue, 1 = Naive, 2 = Frontier, 3 = Part I, 4 = Part II.

NOTE:
if method = -1 -> runs all the methods
if method = -1 and source = -1 -> runs all the methods, consecutively using
every vertex of the graph as a starting vertex (limited by 1024 vertices).
```

Requirements

To run the program, you will only need a C++11 compiler and an MPI library installed. On KUACC machines, C++ compilers (intel and gcc) and MPI are available as loadable modules. There are two MPI options: **mpich/3.2.1** and **openmpi/3.0.0** (note that openmpi/4.0.1 is also available, but you may experience problems with it).

Important! When mpich/3.2.1 is loaded, gcc/7.3.0 is loaded together with it. When openmpi/3.0.0 is being used, you need to specifically load gcc (versions 7.3.0 and 9.3.0 were tested).

```
To compile type
      mpic++ -O3 bfs_main.cpp -o bfs
Example run with mpich: (runs with all (up to 1024) vertices and methods)
      mpiexec -np 4 ./bfs mygraph.mtx -1 1
Example run with openmpi: (runs Part II with source vertex 0)
      mpirun -np 2 ./bfs mygraph.mtx 0 1 4
```

Experiments

You are going to conduct an experimental performance study on KUACC. The aim of this experimental study is to get performance data across different numbers of processors for a given graph. Make sure nothing other than the graph information and timing is being printed out.

- The only experiment you need to do is a strong scaling study such that you will run your parallelized code multiple times under different processor number. You will have to run this study for every graph provided to you and **report the results in a form of figures**. The command that you will run is the following.

```
using mpich: bash$ mpiexec -np <num_of_processors> ./bfs <graph_file.mtx> -1 1
using openmpi: bash$ mpirun -np <num_of_processors> ./bfs <graph_file.mtx> -1 1
```

`<num_of_processors>` will be 1, 2, 4, 8 and 16. Plot the speedup and execution time figures as a function of processor count and include them in your report, separately for each graph. You can include all 5 methods on each graph.

You will notice a difference in your algorithms' efficiency on different graphs. In your report, you need to explain why the parallelization methods worked better for some graphs and failed for others. Compare both of your algorithm's performance to its serial version and report the scaling. Also, compare them to each other. The graph reading code can provide you with more information about the graphs if needed (like the degree histogram). The set of graphs is provided to you can be found in **this google drive folder**. No need to report the performance for the `test.mtx` graph. It is provided only for testing

Important Remarks

In this assignment, you will only have to work on two files: `mpi_vertex_dist.cpp` and `mpi_frontier.cpp`. You don't need to do any changes to other files, although you are not restricted. Additionally, you can do any changes to data structures and their types if you like.
Important! To ensure that all of your performance data is taken from the same machine on the cluster, all performance data for experiments can be submitted as a single job. In addition, all the performance data should be collected on a single node with multiple cores (processors), no need to use multiple nodes.

More details on how to run your code in KUACC cluster can be found in Section Environment below.

Submission

- Document your work in a well-written report which discusses your findings.
- Your report should present a clear evaluation of the design of your code, including bottlenecks of the implementation, and describe any special coding or tuned parameters.
- We have provided a timer to allow you to measure the running time of your code.
- Submit both the report and source code electronically through blackboard.

- Please create a parent folder named after your username(s). Your parent parent folder should include a report in pdf and a subdirectory for source code. Include all the necessary files to compile your code. Be sure to delete all object and executable files before creating a zip file.
- GOOD LUCK.

Environment

Even if you develop and test your implementations on your local machine or on the computer labs on campus, you must collect performance data on the KUACC cluster.

- Accessing KUACC outside of campus requires VPN. You can install VPN through this link: https://my.ku.edu.tr/faydali-linkler/
- A detailed explanation is provided in http://login.kuacc.ku.edu.tr/ to run programs in the KUACC cluster. In this document, we briefly explain it for the Unix-based systems. For other platforms you can refer to the above link.
- In order to log in to the KUACC cluster, you can use ssh (Secure Shell) in a command line as follows: The user name and passwords are the same as your email account.

```
     bash$ ssh $<$username$>$@login.kuacc.ku.edu.tr
     bash$ ssh dunat@login.kuacc.ku.edu.tr //example
```

- The machine you logged into is called login node or front-end node. **You are not supposed to run jobs in the login node** but only compile them at the login node. The jobs run on the compute nodes by submitting job scripts.
- To run jobs in the cluster, you have to change your directory to your scratch folder and work from there. The path to your scratch folder is

```
bash$ cd  /scratch/users/username/
```

- To submit a job to the cluster, you can create and run a shell script with the following command:

```
bash$ sbatch <scriptname>.sh
```

- To check the status of your currently running job, you can run the following command:

```
bash$ squeue -u <your-user-name>
```

  A sample of the shell script is provided in Blackboard along with the assignment folder. In the website of the KUACC cluster, a lot more details are provided.
- To copy any file from your local machine to the cluster, you can use the scp (secure copy) command on your local machine as follows:

```
scp -r <filename> <username>@login.kuacc.ku.edu.tr:/kuacc/users/<username>/
scp -r src_folder dunat@login.kuacc.ku.edu.tr:/kuacc/users/dunat/  //example
```

-r stands for recursive, so it will copy the src_folder with its subfolders.

- Likewise, in order to copy files from the cluster into the current directory in your local machine, you can use the following command on your local machine:

```
scp -r <username>@login.kuacc.ku.edu.tr:/kuacc/users/<username>/fileToBeCopied  ./
scp -r dunat@login.kuacc.ku.edu.tr:/kuacc/users/dunat/src_code ./  //example
```

- To compile the assignment on the cluster, you can use the mpich/3.2.1 or openmpi/3.0.0+gcc/7.3.0

```
bash$ module avail //shows all available modules in KUACC
bash$ module list   //list currently loaded modules.
bash$ module load mpich/3.2.1
bash$ module load openmpi/3.0.0
bash$ module load gcc/7.3.0
```

- If you have problems compiling or running jobs on KUACC, first check the website provided by the KU IT. If you cannot find the solution there, you can always post a question on Blackboard.
- Don't leave the experiments on KUACC to the last minutes of the deadline as the machine gets busy time to time. Note that there are many other people on campus using the cluster.

Grading

- 40% Part I: implementation, efficiency, scalability, experiments
- 40% Part II: implementation, efficiency, scalability, experiments
- 20% Report: speedup and scaling figures, answering questions, explaining your findings, reporting performance, etc.
- You may be asked to perform a demo if needed.