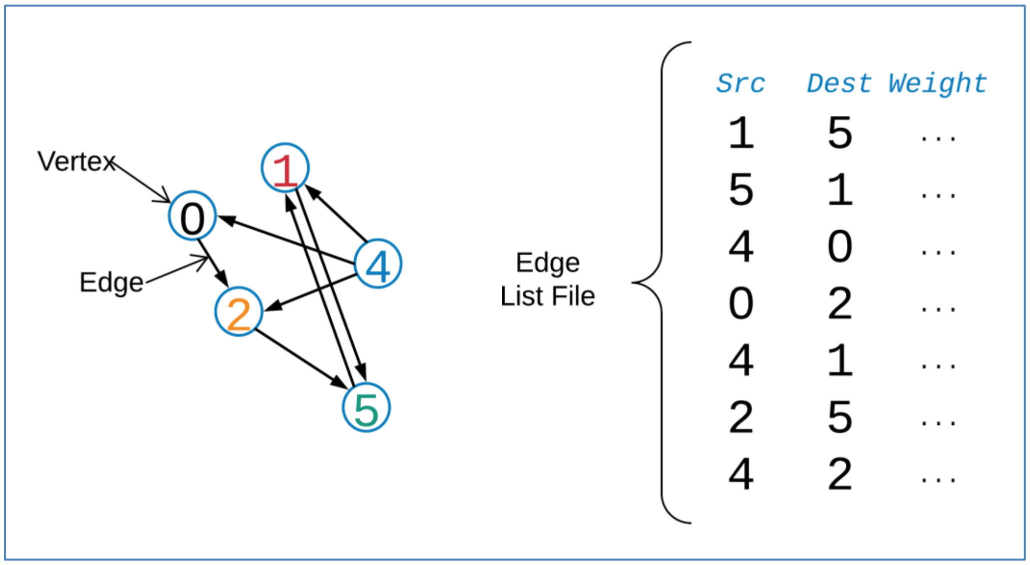
ECE/CSC 506: Architecture of Parallel Computers Program 2

“Stage two, OpenMP”

Parallelizing Breadth First Search (BFS) with OpenMP

Due: Friday 11:55 PM, November 9th, 2018 (Individual Project)



Src	Dest	Weight
1	5	...
5	1	...
4	0	...
0	2	...
4	1	...
2	5	...
4	2	...

Tasks:

In this machine problem you will

- Be provided with two serial implementations of BFS Pull (bottom-up-step)/Push (top-down-step), Queue, Bitmap data structures.
- You will parallelize the BFS algorithms using OpenMP.
- This parallelization will require you to change how you update the Queue and Bitmap.
- You need to solve the race conditions when transitioning from serial to parallel implementation, for BFS.
- Queues need to be multithread friendly.
- Bitmap need to be multithread friendly
- You need to provide atomic implementations for BFS, Queue and Bitmap using
    - OpenMP atomics
    - GCC atomic built-ins
    - OpenMP locks
    - OpenMP critical sections

- Report the difference in performance using these four approaches.
- Mention in you report the difference and elaborate why.
- Explain why you needed to use/not use atomics with both BFS implementations (Push/Pull).
- For extra credit (5 points) look at the assembly code and show what instruction are generated for each approach, this will help you to reason why your performance is same/different/degraded.

Motivation

In the previous machine problem, we showed how to build an optimized memory friendly graph to be used for fast graph processing algorithms. In this machine problem we will be utilizing the adjacency list structure we built for Breadth Search First (BFS). An algorithm for searching or traversing tree or graph data structure, using two approaches. One that uses Push data flow where nodes in the frontier (will be explained later) update neighboring nodes, another is Pull where nodes update themselves and pull information from neighboring nodes in the frontier. Each approach has its own benefits.

Understanding BFS is not only useful for passing this assignment. It is considered a core behavioral pattern for many other graph algorithms and applications. So, optimizing and understanding it might come in handy in the future. Here are some of the places you might find BFS.

- Shortest Path and Minimum Spanning Tree for unweighted graph.
- Peer to Peer Networks.
- Crawlers in Search Engines.
- GPS Navigation systems.
- Social Networking Websites.
- Broadcasting in Network.
- In Garbage Collection.
- Cycle detection in undirected graph.
- Ford–Fulkerson algorithm.
- To test if a graph is Bipartite.
- Path Finding.
- Finding all nodes within one connected component.
- Many algorithms like Prim's Minimum Spanning Tree and Dijkstra's Single Source Shortest Path use structure like Breadth First Search.

Breadth First Search (BFS)

The goal of BFS is to visit each vertex in the Graph, the algorithm starts with a specified root/source vertex and visits all its neighbors (outgoing if directed graph). Then, it visits all the unvisited neighbors of the root's neighbors and continues to process each level of neighbors in one step.

Basic Idea

- We start traversing the nodes of graph in layers (frontiers.)
- Starting at a source node and keep traversing(searching) until we find a target node (search) or explored our entire possible paths (travers).
- The frontier contains nodes that we have seen but have not explored yet.

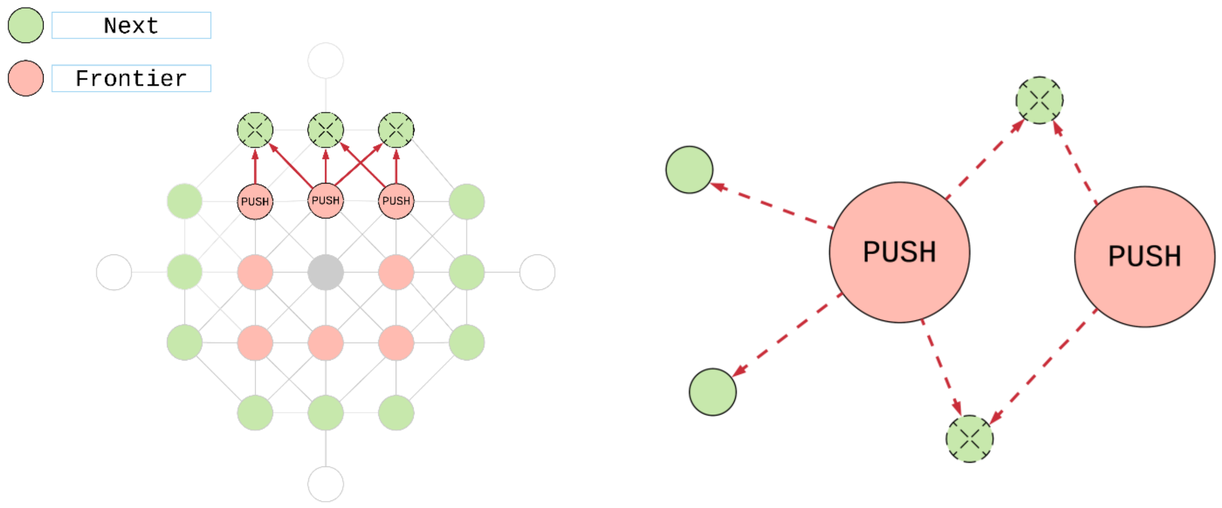
- Each iteration we take a node off the frontier, then add its neighbors to the next frontier(using queues).

Algorithm (Please visit the supplementary slides for detailed explanation)

As mentioned before BFS comes in two flavors Pull (bottom-up-step)/Push (top-down-step), with each implementation has its own tradeoffs.

```
function breadth-first-search(vertices,source)
    frontier←{source}
    next←{}
    parents←[-1,-1,...-1]
    while frontier≠{} do
        top-down-step(vertices,frontier,next,parents)
        frontier←next
        next←{}
    end while
    return tree
```

Push (top-down-step)

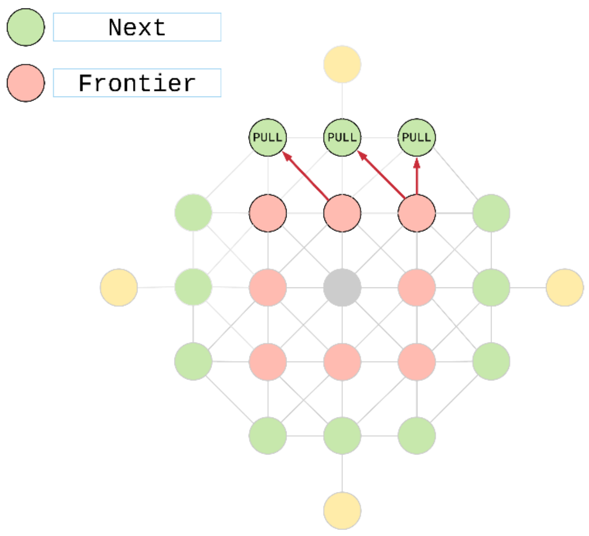


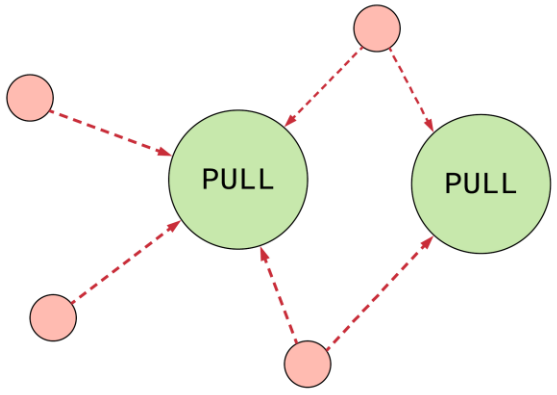
```
function top-down-step(vertices,frontier,next,parents)
   for v ∈ frontier do
     for n ∈ neighbors[v] do
       if parents[n] = -1 then
          parents[n]←v
          next←next⋃{n}
       end if
     end for
   end for
```

For push implementation you will notice that once we visit the neighbors of each node v. v is updating its own out-neighbors (current children) of n, in this case we assign v as a parent of node n also a marker for it to be visited.

The shortcoming of this approach is when the frontier gets too large we end up examining already visited nodes doing unnecessary extra work. also, two or more vertices might end up updating a shared out-neighbor requiring an atomic operation to avoid any race condition adding more overhead.

Pull (bottom-up-step)



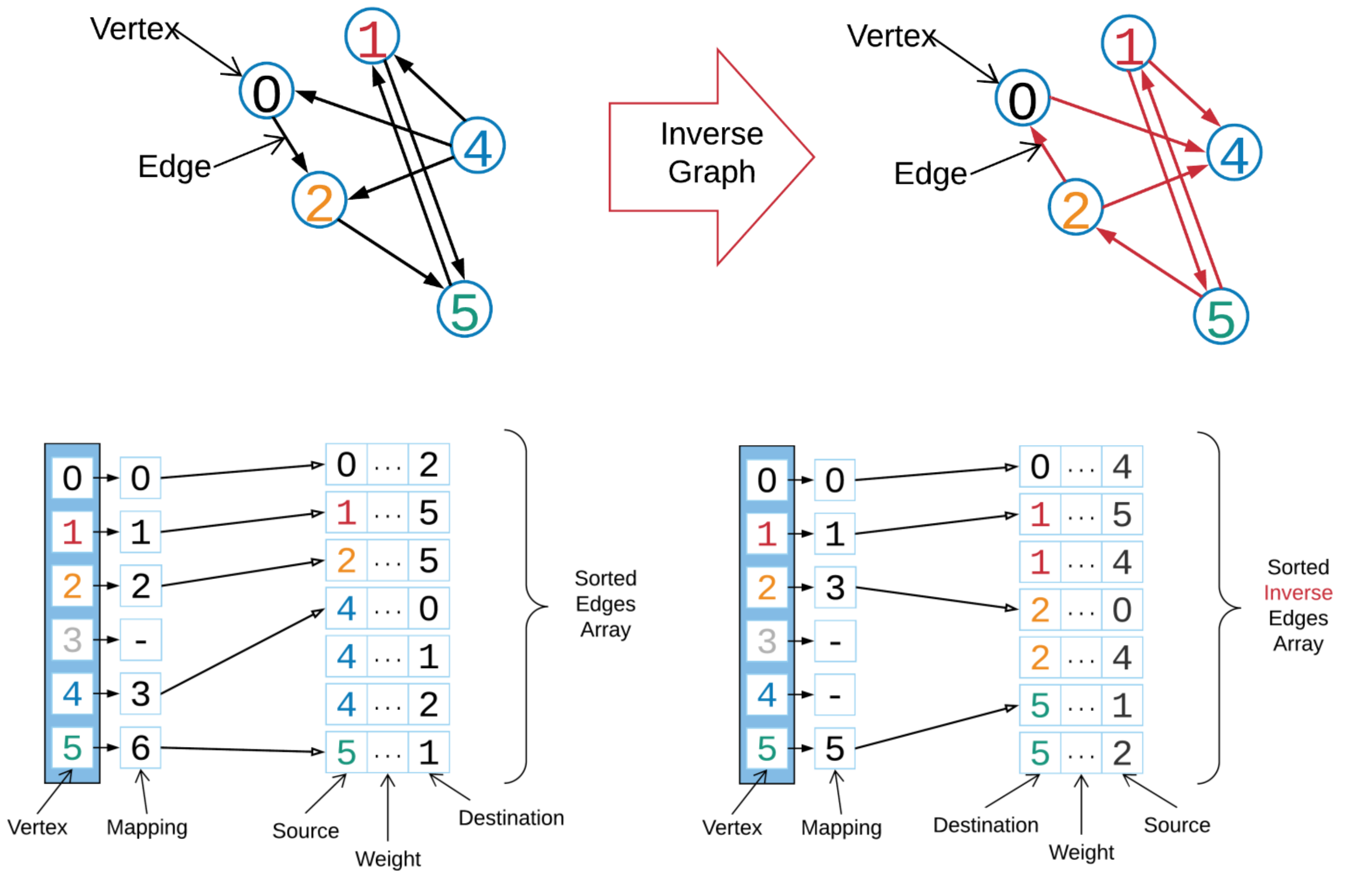


```
function bottom-up-step(vertices,frontier,next,parents)
   for v ∈ vertices do
     if parents[v] = -1 then
       for n ∈ neighbors[v] do
         if n ∈ frontier do
            parents[v]←n
            next←next⋃{v}
            break
         end if
       end for
     end if
   end for
```

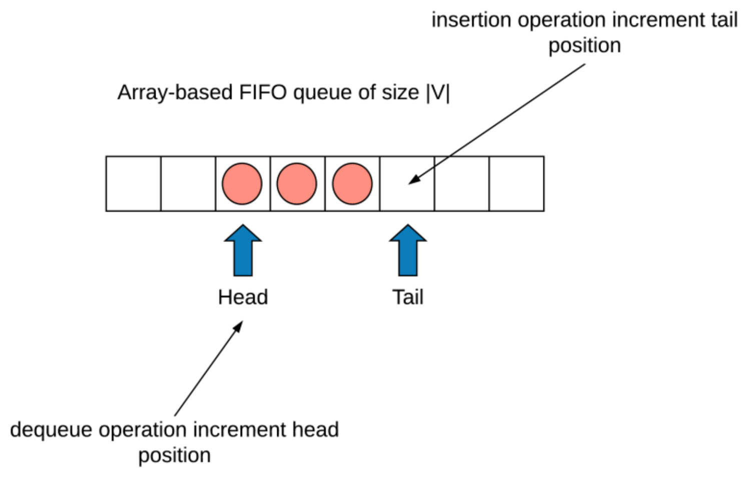
At Pull approach you will notice we have to scan the whole graph for the nodes to detect if it has in-neighbor on the frontier and update its own state as visited. Since each vertex is updating its own state then parallelizing the for loop will not require any atomic operations. Furthermore, to mitigate the next queue bottleneck we chose to use a bitmap where we mark each vertex visited on the bitmap with 1/0.

This operation of bit set/unset will require atomicity since we can only retrieve a whole byte at a time to update one bit, creating conflicts between threads if they share vertices that map at the same byte.

Another drawback of using this approach is the need for double the memory for generating the inverse graph to see the in-neighbors of the nodes/vertices.



Bitmap/Queues explanation (please visit the supplementary slides for further explanation)


insertion operation increment tail position
Array-based FIFO queue of size |V|
Head
Tail
dequeue operation increment head position

In this project we will be using a simple array queue of size graph as worst case scenario. Your task is to make this queue multithreaded aware each time you push vertices to the frontier.

The framework skeleton

- /bin *hold the executable
- /datasets *holds the edgelist benchmarks you need to sort
- /include *hold the *.h files
    - /include/bfs.h *push/pull BFS implementation
    - /include/edgelist.h
    - /include/sort.h
    - /include/timer.h
    - /include/vertex.h
    - /include/queueArray.h // atomic operations added here
    - /include/bitmap.h // atomic operations added here

- /obj *holds the *.o files you make
- /src *holds the main with the source files you need
    - / src /bfs.c *push/pull BFS implementation
    - / src /edgelist.c
    - / src /sort.c
    - / src /timer.c
    - / src /vertex.c
    - /include/queueArray.c // atomic operations added here
    - /include/bitmap.c // atomic operations added here
    - / src /main.c

Grading

20%: Your code compiles successfully

40%: Your output matches exactly for runs on all (provided 5 benchmarks)(points will be equally distributed).

40%: Report. Credit will be given on the statistics shown and discussion presented.

submission Format

In order to grade all submissions promptly, we have to ask you to follow the submission format.

Your final submission should be in a zip file named “unityid1_program2.zip”.

Do not include the parent folder in your zip file “solved by cd into the directory”. Also, no need to include the input and output files. This command should help you generate the zip file:

```
zip -r unityid1_program2.zip ./*
```

Not following the submission format property will result in a maximum of 5 points penalty.

Suggestions

- Read the main and structs carefully, and understand how the program works
- Most of the code given to you is well encapsulated, so you do not have to modify most of the existing functions. You just need to define the incomplete functions Commented
- Make sure there are no memory leaks in your program by de-allocating memory after you are done with it.
- Most of the code given to you is well encapsulated, so you do not have to modify most of the existing functions. You just need to complete the definition of the incomplete functions.

References

- http://lagodiuk.github.io/computer_science/2016/12/19/efficient_adjacency_lists_in_c.html#the-commonly-used-approaches
- http://www.scottbeamer.net/pubs/beamer-thesis.pdf
- https://htor.inf.ethz.ch/publications/img/pushpull.pdf
- https://parlab.eecs.berkeley.edu/sites/all/parlab/files/main.pdf