CSE 6010
Assignment 5
Bellman-Ford with OpenMP

Initial Submission Due Date: 11:59pm on Thursday, November 16
Final Submission Due Date: 11:59pm on Thursday, November 30
Submit codes as described herein to Gradescope through Canvas
48-hour grace period applies to all deadlines

In this assignment, you will implement the Bellman-Ford algorithm for single-source shortest paths in graphs that we discussed in class. When we introduced single-source shortest-path algorithms previously, we mentioned that one advantage of the Bellman-Ford algorithm over Dijkstra's algorithm was that it could be parallelized. Here, you will do exactly that!

Your task is to write a program in C, including OpenMP directives, to calculate shortest paths in a directed graph using a parallel version of the Bellman-Ford algorithm. In this algorithm, the shortest-path estimates are updated each iteration by considering whether each edge can improve the estimates, with different threads handling different edges. Note that the number of iterations required may vary across different runs because of the asynchronous nature of the threads.

You may test your code against the **autograder** for the final submission; you may submit as many times as you like. Please note that you still must submit your initial submission on time.

Specifications:

- Input and command-line arguments: Your program should take 2 required command-line arguments and 1 optional command-line argument, in the following order.
    - ***First command-line argument (required):*** Filename that specifies the graph information. The first line of the file will include two integers, the number of vertices (numvertices) in the graph followed by the number of edges. The remaining lines correspond to the edges, with each line including three values associated with an edge: the "from" vertex (specified by an integer from 0 to numvertices-1), the "to" vertex (similarly specified), and a (positive) double value for the weight of the edge. The total number of lines in the file will be one more than the number of edges. You may assume that a valid filename is specified and are not required to perform any validation. Several sample input files corresponding to graphs of various sizes will be provided.
    - ***Second command-line argument (required):*** The number of threads to use in the parallel section(s) of the code. You may assume this value is a positive integer that does not require validation.
    - ***Third command-line argument (optional):*** destination vertex used for abbreviated output. Specifically, your code should provide one of two kinds of output, depending on whether this command-line argument has been provided. More details on output specifications are provided later.
        - If provided: this command-line argument represents the index of a "destination" vertex for a shortest path beginning at vertex 0 for which output should be provided. The output will include the length of the shortest path from vertex 0 to this vertex, as well as the vertices of the path in the proper order (originating at the source vertex 0 and finishing at the

specified destination vertex). You may assume this argument, if provided, is an integer from 0 to numvertices-1.

        - If not provided: the same information should be provided, but for the shortest paths to all vertices (a total of numvertices, *including the source vertex*) instead of just to a single destination vertex.

- Graph and storage setup: The file given as a command-line argument will include the graph information, as discussed above. You should define an edge struct so that you can dynamically allocate an array of edges to loop over as part of the Bellman-Ford algorithm, along with arrays to track distance estimates and predecessors (for tracing the shortest paths). If you save the edge weights in your edge struct, you will not need to save the graph in any other form; this information will be enough for the algorithm.
- Algorithm: Use the Bellman-Ford algorithm to find shortest paths. Note that in the parallel version, edges may not be evaluated in the order in which they are listed because synchrony is not enforced across the threads, which is acceptable; you should not add any barriers to force a certain order. For this assignment, the graph is ***directed***, so be sure the relaxation step is properly specified. Because OpenMP can be added with a few carefully placed and planned directives, you may wish to write a working code without OpenMP first.
- Parallelization:
    - Set the number of threads in the code to the value provided on the command line. You may do this with the function `omp_set_num_threads()` or by including the `num_threads()` clause in the `#pragma omp parallel` directive.
    - You should parallelize across the edges: that is, distribute over the threads the work of considering whether a shorter distance estimate from the source vertex (0) to the "to" vertex of an edge can be found by going to the "from" vertex of that edge and then including that edge in the path.
    - Note that the "flag" in the algorithm that is set to indicate whether anything changed, and thus whether further iterations are needed, is a prime candidate for a reduction!
    - Use `omp_get_wtime()` to get the start time before the parallel section and the end time after exiting the parallel section, and use the difference to represent the time spent in the parallel section. If you spawn threads multiple times, you should accumulate all the time spent in the parallel sections.
    - Note that for a variety of reasons, you may not always see the speedup you may expect when running in parallel. (It may depend on the specifics of your implementation, your running environment, details of the graph, etc.) We will not test for any particular scaling, but ***we will test that your code runs in less time with multiple threads, as well as testing that it runs correctly.***
- Output and cleanup:
    - Your code should output one or more lines that look like the following:
      ```
      4: 3.21033; 0 3 5 4
      ```
      The output represents the index of the vertex that is the destination of the shortest path from vertex 0, shortest-path length from 0 (always the source vertex!) to the destination vertex, and then the actual path. Here, the output represents information about the shortest path from 0 to 4; the shortest path has a distance of 3.21038, and the shortest path follows edges going from vertex 0 to 3, then from vertex 3 to 5, and finally from vertex 5 to 4. There is one space after the colon, after

the semicolon, and between the vertex listing for the shortest path, followed by a new line immediately after the last vertex in the shortest path. The path length should include 5 values beyond the decimal (format code %.5f).

  - If there is no path from the source vertex 0 to a destination, only output the node index and the colon: e.g., something like:
    `8: INFTY; 8`
    where `INFTY` can be any high value you have set during initialization and for path just the node index.

- If a destination vertex was specified as a command-line argument, only output this information for the specified destination vertex. We will test your code on different graphs, including very large graphs with thousands of vertices and millions of edges (not only to demonstrate that your code works in such cases but to ensure there will be a reliable and meaningful decrease in runtime when using multiple threads), and in such cases we do not want to output shortest-path information for all vertices!
- If no destination vertex was specified as a command-line argument, output this information for all vertices from 0 to (numvertices-1), including to the source vertex (distance will be 0). The information for each destination vertex should be listed on a separate line, and they should proceed in order beginning with 0. (Do not parallelize the output!)
- As a final output, on a separate line, write the elapsed time in the parallel section(s), including 5 values beyond the decimal (format code %.5f); e.g.:
  `0.00292`
- All dynamically allocated memory should be freed. ***Note that valgrind may show a memory leak ("still reachable" leak) with OpenMP even when none exists. We will be testing for leaks where memory is "definitely lost" or "indirectly lost", which will happen when you don't free the memory which was allocated using malloc.***

To receive full credit, your code must be well structured and documented so that it is easy to understand. Be sure to include comments that explain your code statements and structure.

A graph test file is provided for you; we will check results for other graphs.

Final submission: You should submit to Gradescope through Canvas the following files:

(1) your code (all .c and .h files); no specific filenames are required for your code but ***note the specification for your executable file below***.

(2) the **Makefile** you use to compile your program such that your executable file that will run your program is named **'bellmanford'**. You may use the Makefile provided for previous assignments as a starting place. **Don't forget to include any needed compiler flags!**

(3) a series of 2-3 slides **saved as a PDF** and named **slides.pdf**, structured as follows:

- Slide 1: your name and a brief explanation of how you developed/structured your program. This should not be a recitation of material included in this assignment document but should focus on the main structural and functional elements of your program (e.g., **where you added parallelization and why,** the purpose of any loops you used, the purpose of any if statements you used to change the flow of the program, the purpose of any functions you

created, etc.). The goal is to help us understand how you developed your code and your intentions. You are limited to one slide.
- Slides 2-3: a discussion of why you believe your program functions properly.

Initial submission: Your initial submission does not need to include the slides or Makefile. It will not be graded for correctness or even tested but rather will be graded based on the appearance of a good-faith effort to complete the majority of the assignment.

Some hints:

- Start early!
- Identify a logical sequence for implementing the desired functionality. Then implement one piece at a time and verify each piece works properly before proceeding to implement the next.
- **We highly recommend implementing the algorithm serially before adding OpenMP directives; parallel code is more complex to debug.**