# Homework 1 Design and Testing

Michael Tang

April 9, 2019

## 1 Serial and Parallel Design

### 1.1 Serial

Phase 1 involves reading and processing the adjacency matrix. An array, *edges*, will be created to represent the edges of the graph. In the process of reading the matrix if an adjacency exists the according entry will be filled in *edges*. I.e., if there are 7 total vertices and row 2 column 3 of the matrix equals 5 (signifying that vertices 2 and 3 have an edge between them of weight 5), then $edges[9] = 5$.
Phase 2 will follow the pseudocode of Floyd-Warshall given by Wikipedia, given as follows:
let $dist := |V| \times |V|$ array of minimum distances initialized to 1000000
for each edge in *edges* $(u, v)$:
  | $dist[u][v] := w(u, v)$ // the weight of the edge $(u, v)$
for each vertex $v$:
  | $dist[v][v] := 0$
for $k$ from 1 to $|V|$:
  | for $i$ from 1 to $|V|$:
  |  | for $j$ from 1 to $|V|$:
  |  |  | if $dist[i][j] > dist[i][k] + dist[k][j]$:
  |  |  |  | $dist[i][j] := dist[i][k] + dist[k][j]$
return *dist*
Phase 3 will output *dist* in the same format as the input adjacency matrix.
Including *edges*, only two data structures are needed. *dist* is represented by a 2-dimensional array, written in C as an array of array pointers.

### 1.2 Parallel

Phases 1 and 3 will remain the same. In Phase 2, work can be divided amongst threads in each iterative block. The first iterative block, adding direct path distances via given edge weights, can have each thread iterate over its own section of *edges* to reduce the work. The second iterative block can do likewise for the set of vertices.
The iteration "for $k$ from 1 to $|V|$" itself iterates over every element of the 2d array *dist*. Here parallelization will shorten the work - the 2d array can be divided equally for each thread to iterate over a non-overlapping section. The pthreads library will be used to implement this.
To prevent race conditions, each element pointed to by the 2d array will be a pthreads mutex variable. A thread that is actively comparing and possibly overwriting an element will have it locked for that entire

duration. This will ensure comparisons of path length aren't interrupted by overwrites, and that competing overwrites aren't made. Note that in the division of labor for the beginning iterations no mutex is needed because no two threads read or write to the same space.
The most significant invariants are:

- that at every iteration of $k$ the shortest paths considering that intermediate step $k$ have been updated for all vertex pairs. Correctness of the k-restricted problem at every step is crucial for the final answer to be correct.
- that the adjacency matrix was read correctly.
- that initial direct distances and self-distances are correctly inputted.

# 2 Correctness Testing

We test the hypothesis that no correct set of inputs will violate the invariants.
For the second invariant, for any input the set of edges created in Phase 1 can be double checked with the adjacency matrix. There are no foreseeable edge cases for this step, but different vertex set sizes are good to make sure indexing is correct.
For the third invariant, the inputted weights can also be easily double checked with the adjacency matrix. Indexing correctness can be similarly tested.
For the serial version, graphs with known minimum paths can be constructed for testing. In the case that the returned minimum paths aren't correct, the test case that returns an error can be further expanded to analyze every step of $k$ for errors. Good cases include graphs with identical weight on every edge, only disconnected edges, fully connected graphs, a one-vertex graph, graphs with no edges, and empty graphs, as well as other cases with a mix of connectivity and amount of vertices.
For the parallel version the worry is that parallelized updates at each $k$ iteration in Phase 2 are incorrect. To this end the same tests as for serial version can be used and the *dist* table can be compared with serial at every step of $k$ for correctness.

# 3 Performance Testing

## 3.1 Parallel Overhead

The hypothesis is that parallel overhead increases cubically with vertex amounts while remaining on a single thread. The overhead would be sourced solely from mutex operations, which would be performed during the path-minimizing iterations over each element. For number of vertices $|V|$ there are $|V|^3$ iterations - therefore, increasing $|V|$ would increase this overhead cubically.
To test this, the provided stopwatch object can be used to measure runtime from the beginning of the path-minimizing iterations ("for $k$ from 1 to $|V|$") to the end of those iterations. That is the sole area where parallel overhead comes from and plotting runtime increases from there would be sufficient and show the trend more clearly.

## 3.2 Parallel Speedup

The hypothesis is that parallel speedup would be time $1/T$ for the entire program where $T$ is the number of threads. This is reasoned as the ideal speedup because it is the division of work for each thread in each

iteration. To test, stopwatch can be used over the entire Phase 2 section. Stopwatch can also be used individually for each iterative block to see if $1/T$ speedup is accurate for each block.