# Design Document

Alex Miller

April 13, 2021

## Modules, data structures, interfaces, invariants:

The program will be broken up into two modules, main.c and floyd.c. main.c will describe the following functionality:

- Take as input:
  - The number of threads to use in the parallelized section of the code; call this value $p$
  - A text file representing an adjacency matrix; let it be called $foo.txt$. Integers greater than 1000 in the adjacency matrix represent the lack of an edge between vertices
- Generate a $graph\_t$ instance that describes $foo.txt$ with the following attributes:
  - The number of vertices in the adjacency matrix
  - A 2D array that describes the adjacency matrix
- Call this instance $foo$; it is generated with a function $construct\_graph$ that takes $foo.txt$'s FILE pointer as input
- Compute the shortest paths between vertices in $foo$ using $fw\_serial$; the result of this operation is a 2D array that represents shortest path distances between vertices in $foo$. Call this result $res\_serial$. Note the time it takes to generate this result as $serial\_time$. We will discuss this function later. For the time being understand it as our serial implementation of Floyd-Warshall.
- Compute the shortest paths between vertices in $foo$ using $fw\_parallel$ and $p$ threads; the result of this operation is a 2D array that represents shortest path distances between vertices in $foo$. Call this result $res\_parallel$. Note the time it takes to generate this result as $parallel\_time$. We will discuss this function later. For the time being understand it as our parallelized implementation of Floyd-Warshall.
- If $res\_serial$ and $res\_parallel$ are identical, then we store $res\_parallel$ as a text file named $foo\_res\_ < parallel\_time : serial\_time >$
- Otherwise, we raise an error

The purpose of *main.c* is to run convert a text-based adjacency matrix into a form that we can operate with, namely a 2D array. We use this 2D array firstly to compute a shortest path result using a serialized version of Floyd-Warshall, $fw\_serial$. Noting the time and result for testing purposes, we then run $fw\_parallel$. We can assume that $fw\_serial$ gives a the correct shortest path result, since it is sufficiently verifiable. We can therefore use it's output as a justification for accepting the output of $fw\_parallel$.
The other module, $floyd.c$ describes the datatypes and functions referenced in *main.c*:

- $graph\_t$
  - This data type describes a graph in the form of an adjacency matrix
  - It contains the following attributes:
    - $V$: the number of vertices in the graph
    - $M$: the adjacency matrix

- graph_t $construct\_graph(FILE)$
  - $FILE$ describes a file containing an adjacency matrix
  - This function outputs the appropriate representation in a $graph\_t$ instance
  - This function should also check whether or not it is taking in a trivial example (a matrix with one element, for example) and raise an error accordingly. Additionally, it should also check for negative cycles in resultant graphs, as these aren't necessarily tolerated under Floyd-Warshall.
- int[][] $fw\_serial(g)$
  - $g$ describes a $graph\_t$ instance
  - Name a 2D array $res$, which will eventually contain our shortest path table.
  - Initially $res \leftarrow g.V$
  - $\forall k, \forall i, \forall j \leq g.V$, compute the following:
    - If $res[i][j] \leftarrow MIN(res[i][j], res[i][k] + res[k][j])$
  - This is sufficient for computing shortest paths within an adjacencey matrix (Source).
  - Return $res$
- int[][] $fw\_parallel(g, p)$
  - $g$ describes a $graph\_t$ instance
  - $p$ is an integer specifying the number of threads to use in computing a result
  - Name a 2D array $res$, which will eventually contain our shortest path table. All threads will have access to this array.
  - Initially $res \leftarrow g.V$
  - Here, we must note that the serial version of Floyd -Warshall operates by calculating the shortest distnace between all $(i, j)$ pairs for increasing values of $k$. We can say then that while the innner loops of the serialized verion are dependent on the outer-most loop, they themselves are independent of each other. Therefore, this section of the code is ripe for parallelization.
  - Therefore, I propose the following parallelized loop structure to compute shortest paths using Floyd-Warshall: $\forall k \leq g.V$, compute all the shortest path lengths bewtween all $(i, j)$ pairs in parallel. For a single thread, this could look like, for a fixed value of $k', i', \forall j \leq g.V$, computing $res[i'][j] \leftarrow MIN(res[i'][j], res[i'][k'] + res[k'][j])$. In other words, we should turn the inner for-loop in the serialized version into a parallel for-loop in which a pool of threads computes shortest path lengths bewtween all $(i, j)$ pairs in parallel for a given value of $k$
  - As an invariant, we know that we will always have generated the shortest path lengths of all $(i, j)$ pairs for a value of $k$ before we generate such lengths for a value of $k + 1$, maintaining data dependencies within the algorithm.
  - The maximum number of threads we can use at one time is equivalent to $g.V$, since each row of the martix needs to operated on independently. These threads will need to be synchronized such that all $(i, j)$ pairs for a given value of $k$ are calculated before moving on to $k + 1$
  - Each thread must know the following information in order to operate properly; a pointer to $res$ (the array being operated on), what row it is currently computing in the matrix, as well as for what value of $k$

# Test plan:

Initially, I will verify that my serial implementation of Floyd-Warshall is correct by reviewing my implementation directly as well as testing trivially small example matricies and verifying results. Considering that the implementation of a serial Floyd-Warshall algorithm is a reasonably reproducable and simple dynamic program, this should be enough to verify correctness.
I will then use the validity of my serialized implementation in order to verify results produced by my parallel

implementation of Floyd-Warshal. Because my serial implementation is verifiable, I can compare results of $fw\_serial$ and $fw\_parallel$ in order to infer the correctness of $fw\_parallel$.
In regards to what examples of adjacency matrix I will use when testing the correctness; Python has a library for creating representations of DiGraphs. I plan on using this library to create a pool of randomly generated graphs. I can use this pool to test my serial and parallel implementations of Floyd-Warshall. Additionally, I plan on hard-coding examples of edge cases as they occur to me while testing; currently I have in mind to hard code a fully connected graph with no negative cycles as well as a graph with only one vertex.

## Expected Performance (Performance Hypotheses):

In general I expect both serial and parallel implementation to have similar memory and cache performance since they both iterate through rows in the Matrix; the paralell implementation only does so concurrently.
My hypothesis is that for graphs with trivial amounts of vertices, the overhead of parallelizing row computations will outweigh the benefits of multiple threads; for example, I expect the serialized implementation to outperform the parallel implementation for a graph with fewer than 32 vertices.
For larger graphs, I expect the parallel version to experience linear speedup over the serial version by a factor of the number of threads that can be in use at any one time; this is bounded by the number of vertices in a graph (or the number of rows in the adjacency matrix). I expect each utilized thread to demonstrate similar performance, though some may experience more or less writes to the array than others. Therefore I expect to see speedup vary as a function of total number of vertices in a given graph; bigger graphs with more rows can experience greater total speedup because the work can be broken up into more, row-based, components.