*discussed with Paul, Jack

Initial Design and Test Plan

Design

(1) Modules, Interfaces, Task Assignment to Threads, Invariants

Modules + Interfaces of the Program

We will break our code up into three responsibilities: reading input, running the algorithm, writing output.

Input Module

The goal of this module is to take in the input text file (as specified in the Homework 1 document) and construct an adjacency matrix that the algorithm can then process.

The input will specify that there is no edge from u to v if the corresponding edge weight is infinity, denoted by the character 'i'. We will parse as follows. We scan in the first line to know what N = |V| is, then we will proceed to fill out each row of a **2-dimensional int array** called adjacency matrix by reading in each subsequent line, where we split the line by the space delimiter. We can then use a string to int function like atoi to convert the weight for each edge we see for that particular line, or set it to an infinity value if we see the char 'i' (a discussion on what to choose for infinity value comes later)

Sample input.txt:
3
0 1000 2
3 0 i
5 6 0

Note that **we will not handle negative edge weights**. This will simplify our project to focus on parallelization. The max edge weight is 1000.

Algorithm (Serial) Module

```
let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
for each edge (u, v) do
    dist[u][v] ← w(u, v)  // The weight of the edge (u, v)
for each vertex v do
    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]
            end if
```

The goal of this module is to take in an adjacency matrix and return a 2 dimensional N x N array representing the shortest (directed) path to go from node u to v. We call this shortest path array **dist**.

Let us think about how we will implement the algorithm by understanding how it works. Floyd-Warshall (FW) operates on dynamic programming. We create a 2D integer array **data structure** which Wikipedia calls **dist**, which represents the currently known shortest path between two nodes for the current value of k we are iterating over. Initially, all the values in dist are infinity, except for the path weight of nodes to themselves (which is zero), or for one node to another with an existing connecting edge.

We then iterate on k, then recompute the shortest path from one node to another (these are the two inner for loops). Coding up the Serial version follows more or less from copying Wikipedia, and by the end, dist should contain the shortest distance to get from one node to any other. A discussion on the parallel implementation comes later.

How do we store infinity?

If the weight of directed edge u to v is infinity, it means there is no direct path from u to v. One way to represent this could be using a value like null. We would then code our algorithm so that whenever null is seen, we know that all integer values must be less than it. If dist[i][j] is null at the end, then that means the shortest path from i to j does not exist.

This approach would work, but it would be clunky to deal with the logic, plus having a dist array that is not ints.

Instead, we can be clever. Remember that the goal for our infinity value is to ultimately detect when there exists no path (whether a path of one edge or multiple) between u to v. Formally, if there exists no path from u to v in the input, then I should output that there is no path from u to v.

We can simply use an infinity value equal to 2 * (1000 * N^2). This works because an upper bound for the weight a path is the max edge weight (1000) times a bound on the max number of edges (N * 2). Clearly, 2 * (1000 * N^2) > (1000 * N^2). Thus, if we see a value of dist[i][j] = 2 * (1000 * N^2) at the end of the algorithm, then we know that we should output infinity as the path weight from i to j.

Output Module

We will generate the **output** by scanning through dist and writing its row values to an output file. Here's how it works. We scan one row of dist. We initialize an empty string to represent the output that should correspond to this row, i.e. the shortest path to get from the node of the current row to all other nodes. Call this string output_row_weights. Now, we iterate through the row of dist since it is an integer array, and convert the integer weight to a string with the itoa library function. We then append this onto the end of output_row_weights. If we ever encounter a value equal to INFINITY=2 * (1000 * N^2), we will simply write 'i' to represent that there is no path.

Algorithm (Parallel) Module

Now, let us consider how to parallelize our code. First, note that you can parallelize the initialization of the dist matrix, as we are reading in from an adjacency matrix into the dist program variable. We could have each thread read a row. I'll skip implementing this because we are really trying to parallelize the triple for loop.

We have the following recurrence:

$$\mathrm{shortestPath}(i, j, 0) = w(i, j)$$

and the recursive case is

$$\mathrm{shortestPath}(i, j, k) = \min\Big(\mathrm{shortestPath}(i, j, k-1), \mathrm{shortestPath}(i, k, k-1) + \mathrm{shortestPath}(k, j, k-1)\Big).$$

Parallelize on k? No we can't

We can't parallelize on k due to the nature of the algorithm. The way that this dynamic programming problem works is to say "for a given k, what is the shortest path from node i to j using nodes {0, 1, ... k}?". The answer is then based on the answer to a smaller k, as seen in the recurrence relation. What this means is that if you were to parallelize computation on different k's, then the thread t_2 with k = 2 would have to wait on the work of the thread t_1 with k = 1.

Concretely in code, parallelizing on k would be dangerous. Suppose we have the same case as above. Then if the thread t2 were executing an instruction and doing the comparison

```
if dist[i][j] > dist[i][k] + dist[k][j]:
        dist[i][j] = dist[i][k] + dist[k][j]
```

Then this presupposes that dist[i][k] on the RHS indeed represents shortestPath(i, k, k - 1) as written in the recurrence relation. But if t1 has not gotten there, then clearly the cell dist[i][k] does not do that, thus we fail the recurrence relation.

Parallelize on i? Yes we can

We know we must have all threads operate on the same fixed k=K, so how else can we parallelize the work? I claim you can parallelize on i. Suppose you have two threads t_1 with i = 1, t_2 with i = 2. Now, what subproblems does each thread tackle and do they overlap?

t_1 tackles k=K, i = 1 for j = 1 to N.
t_2 tackles k=K, i = 2 for j = 1 to N.

Looking at the lines where we may have a read/write overlap,

```
if dist[i][j] > dist[i][k] + dist[k][j]:
        dist[i][j] = dist[i][k] + dist[k][j]
```

It seems that we may be safe. After all, here's what these two lines look like for each thread

t_1	t_2
if dist[1][j] > dist[1][K] + dist[K][j]: dist[1][j] = dist[1][K] + dist[K][j]	if dist[2][j] > dist[2][K] + dist[K][j]: dist[2][j] = dist[2][K] + dist[K][j]

Note how even though both threads access dist[K][j], this is allowed because they are both reading the value, not writing it.

The potentially problematic part comes up for thread k, i.e.
t_k tackles k=K, i = k, j = 1 to N.

t_k
if dist[K][j] > dist[K][K] + dist[K][j]:

dist[K][j] = dist[K][K] + dist[K][j]

Here, you can see that t_k could potentially be writing to dist[K][j] **while other threads are reading from it!** This would be bad because it means that the other threads would be looking up a value that they don't want. The other threads want dist[K][j], which mathematically is shortestPath(K, j, K - 1). But by writing to the cell, you update that cell to represent shortestPath(K, j, K), which is not the recurrence.

However, if you look closely, you will see that t_k will never overwrite dist[K][j]. This is because the conditional that triggers the write only happens when dist[K][K] **is negative**. Since we assume no negative edge weight cycles, this never happens! Thus, we are safe to parallelize on i.

Parallelize on j? Yes we can

By a similar argument, we are safe to parallelize on j. This is basically dividing up the work of going through the ith row. Suppose we have the following for some fixed K, I.

t_1 tackles k=K, i = I for j = 1 to N / 2.
t_2 tackles k=K, i = I for j = N / 2 + 1 to N.

t_1	t_2
if dist[I][j] > dist[I][K] + dist[K][j]: dist[I][j] = dist[I][K] + dist[K][j] for j in [1, N/2]	if dist[I][j] > dist[I][K] + dist[K][j]: dist[I][j] = dist[I][K] + dist[K][j] for j in [N/2 + 1, N]

Again, both threads read dist[K][j] but never write to it, sowe're safe. The program variable dist[I][j] they write to is neatly separate for these two threads.

Again, the potentially problematic part comes up for the thread that writes to j = K, call it t_k.

t_k
at one point, the conditional will look like if dist[I][K] > dist[I][K] + dist[K][K]: dist[I][K] = dist[I][K] + dist[K][K]

You can see that dist[I][K] may be written to in t_k, even though it is read from in the other threads (see t_1, t_2). Again, this is a false alarm because this write only happens if dist[K][K] is negative, which is never true.

Implementation Details

In terms of the final program, I will choose to parallelize on i, not j. This is because parallelizing on j divides the problem into smaller chunks (divide into the rows then divide the rows) as opposed to parallelizing on i which just divides the problem into rows. In most cases, the number of threads is less than the number of nodes, so dividing by rows would already ensure that no thread is left idle. Eg. N = 1024, T = 6. However, if we had N = 32, T=64, then dividing only on i would leave 32 threads idle. My implementation will not code for this case because it's not too common.

Thus, we will solve the problem as follows. For a given k, spawn as many threads as we can that take care of a particular i, iterating through 1 to N to find the shortest path from i to nodes 1 to N for that given k. Then, wait until all the threads are done to move on to the next k (i.e. call join).

To divide up the N rows among T threads, we can use a counter. Eg. when a thread is done with its row, it checks the shared counter variable, atomically reads the value and increments, then continues onto its next assigned row. We could theoretically just pre-assign all the threads eg. if T = 64 and N = 1024, we could assign t_0 to rows 0, 64, 128, ... Since the work should be even among all threads, I see no issue with this, but I will stick to the counter pattern because that seems easier to implement.

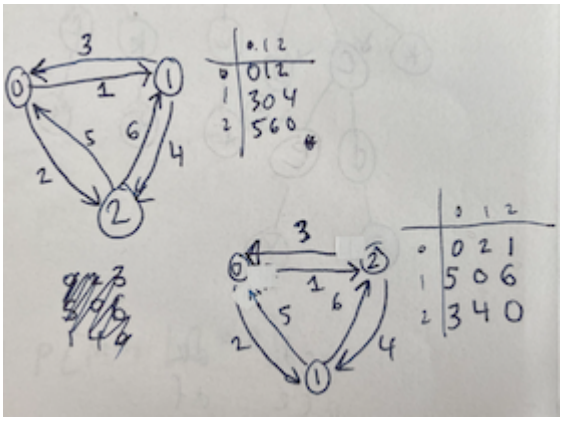
(2) Correctness Testing Plan

To test for correctness, I will compare my parallel results to my serial results. I will do this for all sorts of graph inputs, eg. sparse vs not sparsely-connected, or low node-count to high node-count. Of course, this relies on my serial implementation being correct, but the point is that serial correctness should be easier to check + reason about than the parallel version, so I will start there.

I plan to (manually) come up with three N=3 test inputs that I can manually verify for my serial implementation. One of them will contain an isolated node, another will contain some infinity edges, and the last one will contain non-infinity edges. If that passes, I then plan to run my serial algorithm on a decently-sized input where hand-checking wouldn't make sense, say N = 10. I would generate this graph via an RNG.

In this case, I wouldn't be able to manually verify the results, but I could do a sanity check with the following trick. I will take two nodes v_1 and v_2, and swap their labels. This corresponds to swapping rows v_1 and v_2, then swapping columns v_1 and v_2.

For example, if I were to re-label nodes 1 and 2 in the following (0-indexed) graph:



You can see that the structure of the graph does not change, so our output should not meaningfully differ except that the rows and columns of 1 and 2 be swapped.

output
0 1 2
3 0 4
5 6 0

output (swapped)
0 2 1
5 0 6
3 4 0

I will do this row swapping technique for every pair of rows for this 10 x 10 input. ensure that all is in order, I'll run the same procedure on another 10 x 10 input but with some infinite edges sprinkled in, say 40 of them. This should stress that my handling of infinity is correct.

(3) Performance Hypotheses and Test Plan

Parallel overhead. For T = 1, N = 16, ..., 1024.

The ratio of serial to parallel will start above 1 and converge to 1. This is because you are essentially running the same program in both cases, just that in the parallel case you have the overhead of spawning a thread which makes it take longer. However, this overhead is fixed since you are only spawning one thread. Thus, the proportion of this spawn time compared to the work running the algorithm gets smaller as the work of the algorithm gets larger with N. Thus, we expect parallel runtime to get closer to serial run time.

Parallel Speedup. For T = 2, ..., 64. For N = 16, ..., 1024.

Fix N. I predict the program will run faster as T increases, up to the point that increasing T doesn't put more threads into use (N = T), at which point the time should decrease because you are just increasing the work with doing thread spawns without getting additional efficiency.

The time decrease should be proportional to the number of workers. Eg. I have double the workers, so my time should decrease by half since an additional worker evenly divides up the work more. This is because I don't anticipate any one worker having a much harder time than the others (unlike the primality testing exercise) because the worker's operations are all the same, no matter what slice of the (k, i, j) problem space they get. Thus, your speed up is directly proportional to the number of workers you have.

I do wonder if the overhead cost may outweigh spawning additional threads. Eg. let T = 32 or 64 and N = 128. Then even though each thread can get assigned one row of work, it is not clear that taking the time to spawn a new thread + managing it with joins will be faster than just letting another thread chip at the work.