CMSC 23010 Homework 1 Design

Ruolin Zheng

April 3, 2019

1 Phase 1

The program will read in a text file representing an adjacency matrix and fill in the `dist` array. `dist` has 0's along the main diagonal; integer values in $[-1000, 1000]$ everywhere else for connected vertices i, j; and 10000000 for disconnected ones. Run with `./shortest_paths -f FILE` for Serial; and `./shortest_paths -t NUM_THREADS -f FILE` for Parallel.
Precondition: The text file should be correctly formatted. The input graph should not contain negative cycles (as specified by the Floyd-Warshall algorithm.) For the Parallel version, we may assume that the number of vertices is a multiple of the number of threads.

2 Phase 2

2.1 Serial Implementation

Algorithm 1: Serial

```
for k ← 1 to N do
    for i ← 1 to N do
        for j ← 1 to N do
            if dist[i][j] > dist[i][k] + dist[k][j] then
                dist[i][j] ← dist[i][k] + dist[k][j];
```

2.2 Parallel Implementation

The version illustrated below is very easy to implement, yet may incur an overhead as the main thread executes the k loop and spawns T threads for a total of N times.

Algorithm 2: Parallel

```
1 for k ← 1 to N do
      // spawn T threads and assign a chunk of size N/T to each
2     for t ← 1 to T do
3         for i ← (t − 1)N/T to tN/T do
4             for j ← 1 to N do
5                 if dist[i][j] > dist[i][k] + dist[k][j] then
6                     dist[i][j] ← dist[i][k] + dist[k][j];
```

2.3 Design Questions

- **Module:** `pthread.h` library for parallelism; `shortest_paths.c` contains all functions.
- **Assignment of Tasks to Threads:** The Parallel version fixes the intermediate node k and assigns N/T source nodes to each thread.
- **Invariant:** The invariant for the **Serial** version is in the outermost loop: for every i, j, `dist[i][j]` holds the minimum distance among the distances of all possible paths which use only the intermediate vertices up to k. **Parallel** relies on the same invariant.
- **Data Structure:** A 2D-array `dist` for both Serial and Parallel.
- **Data and Synchronization:** For Parallel, `dist` must be a shared variable among all threads. All threads must wait until all source nodes have been processed as in line 3-6 before returning to the main thread to increment k.

2.4 Correctness Testing

2.4.1 Hypothesis

For both Serial and Parallel, the following invariant holds: for every i, j, `dist[i][j]` contains the minimum distance among the distances of all possible paths which use only the intermediate nodes up to k. The **Proof** follows from the fact that k is in the outermost loop; thus, all i, j pairs would have been computed (and the minimum recorded) for a fixed k before we increment k to $k + 1$. In addition, we update `dist[i][j]` as soon as we identify a shorter path which uses only the intermediate nodes up to k.

2.4.2 Testing Plan

We generate random test input files with `test.py`. We stress the invariant by testing on:

- A graph where all vertices are disconnected; we expect no update to any `dist[i][j]`, all entries remain 10000000 (∞) save for the 0's along the main diagonal.

- (Test During Development Phase) An arbitrary graph; we check that for each iteration of k, the updated entries `dist[i][j]` must be smaller than that in iteration $k-1$.
- (Test After Development) A small, arbitrary graph; we compute with brute force $O(N^N)$ all possible path costs for each i, j, take the min, and test against our program output.

For error handling, we note that **Floyd-Warshall** produces wrong output on a graph with negative cycles; hence we test:

- A graph with negative cycles; we should expect negative values instead of 0's along the diagonal of the output. This indicates invalid input as noted in **Precondition**.

For **Parallel**, we require the threads be synchronized such that all i's must have been processed before k is incremented.

2.5 Performance Testing

2.5.1 Hypothesis

The time complexity of **Serial** is $O(N^3)$. For **Parallel**, we leave the outermost k loop unchanged and apply parallelism to the two inner loop. Suppose **Serial** runs the two loops in N^2 amount of time, then **Parallel** requires N^2/T amount of time by distributing approximately N/T sources i's to each of its thread. Thus the time complexity of **Parallel** is $O(N * N^2/T) = O(N^3/T)$. The inner loop speedup by Amdahl's Law is $S = \frac{1}{1-1+1/T} = T$. Suppose $N = 16, T = 2$, then $S = 2$, meaning that **Parallel** is 2x faster than **Serial**.
To conclude, the performance speedup of **Parallel** is proportional to T but may grow at a decreasing rate for large N.

2.5.2 Testing Plan and Experiment Hypotheses

- **Parallel Overhead:** With $T = 1$, **Parallel** incurs a constant overhead cost of spawning a single thread for N times, and thus should be slower by an amount proportional to N than **Serial** for all N.
- **Parallel Speedup:** From the **Hypothesis** above, **Parallel** should be T factors faster than **Serial** for all N. However, it incurs a total overhead of spawning TN threads; thus, for larger N, **Parallel**'s speedup may be less significant.

3 Phase 3

The program will write the updated `dist` array to a text file.