CMSC 23010: Homework 1 Design Documentation

Will Thomas

October 6, 2022

Introduction

The goal of the following design documentation is to outline a cohesive plan for the design, correctness testing, and performance analysis of a program that will compute the Floyd-Warshall algorithm using explicit parallelism. We will outline design of both a serial and parallel version of the program, although most of the rest of the document will pertain mostly to the parallel version.

Assumptions

We will make the following assumptions:

1. All edge weights will be positive integers.

   We choose to do this to obviate the need to check test graphs for negative cycles.

2. The maximum edge weight will be 1000.

   We do this to keep paths from becoming extremely large with sufficiently sized graphs.

3. Infinity will be represented by $1000|V|$.

   Given a maximum edge weight of 1000, the longest path in a graph with $|V|$ vertices in the worst case in which node 1 is $|V| - 1$ edges away from the last node $|V|$ and all edges have weight 1000 would be $1000(|V| - 1)$. As such, $1000|V|$ will be substantially greater than the longest path possible for any graph with $|V|$ vertices, meaning that it will serve as infinity well.

Design

We will begin by outlining our design for the serial program.

Serial:

The main datastructure that we will use is a $|V|$ by $|V|$ matrix M where V is the set of vertices in the input graph G. As with all dynamic programming algorithms, Floyd-Warshall requires memoization of previously computed subproblems, and our matrix M will serve that purpose.

We will now lay out the program in three phases. Phase 1 corresponds to reading the input text file containing the adjacency matrix into memory, Phase 2 corresponds to computing the Floyd-Warshall algorithm, and Phase 3 corresponds to outputting the computed solution.

- Phase 1:

  We will create a function called input that will read the values held in the textfile adjacency matrix into a $|V|$ by $|V|$ matrix *in*. The function will take as a parameter the file name, the value $|V|$, and the empty *in* matrix with space already allocated. As such, we can keep all allocation and freeing within the function that calls input.

- Phase 2:

  From there, we will create our matrix M by renaming in to M and then changing every cell in M with value 0 that is not on the diagonal to $1000|V|$. That is because there is no direct edge between these two vertices, meaning that initially, considering paths of length 0, these vertices are unreachable from eachother.

  From here, we will start the timer and then compute the Floyd-Warshall algorithm via a triply nested loop with variables, in order from outermost to innermost loop, of k, i, j, each upper bounded by $|V|$. For each iteration of the innermost loop, we will compute the following:

  $$M[i][j] = \min(M[i][j], M[i][k] + M[k][j]).$$

  Once every loop terminates, we will have the minimum distances between every possible combination of vertices in G stored in our matrix M.

- Phase 3:

  Finally, we will call a function called output that, given the matrix M as well as a filename, will convert M to text and write the text to the file. Then, we will free the space allocated for in (now M).

Before moving on to the parallel version of the program, it is important to consider invariants of the serial program as they will be crucial in informing our testing strategies for the parallel program.

1. The only cells that should ever have a value of 0 in the minimum distance table are the ones that lie on the diagonal, i.e. the ones where the column index is equal to the row index.
2. Consider two iterations of the outermost loop. Call the value of a cell after the first iteration $M_0[i][j]$ and the cell after the second iteration $M_1[i][j]$. It should always be the case that:

   $$M_0[i][j] \geq M_1[i][j].$$

   That is to say that the values stored in each cell should never increase between iterations.
3. For any iteration of the outermost loop k that has been completed, it should always be the case that:

   $$M[i][j] = \min(M[i][j], M[i][k] + M[k][j]).$$

   Effectively, this just means that the recurrence relation must always be computed properly, although this may become a more present issue when altering the program to compute in parallel.

Now, we must outline our design for the parallel version of the program:

Parallel

The parallel version of the program will function much the same despite a few key differences in Phase 2 of the design. It is important to note that everything done between each iteration of the outermost loop can be done completely in parallel.

To show this, one must consider that on iteration l such that $1 < l < k$, the computation of the recurrence relation at every cell $M[i][j]$ needs the value at $M[i][j]$ (which will be unchanged up to this point in this iteration of the loop), and then one value from the lth column ($M[i][l]$) and one value from the lth row ($M[l][j]$). This computation can be done in parallel for all i, j because obviously $M[i][j]$ will not be changed until the recurrence relation is run and because in the lth iteration of the outermost loop the lth column and lth row stay the same.

To show this, consider computing the recurrence relation for every cell in the lth row. We would have a situation as follows:

$$M[l][i] = \min(M[l][i], M[l][l] + M[l][i]).$$

Because $M[l][l]$ is always 0, $M[l][i] = M[l][i]$ always for iteration l. The same logic applies for computations in the lth row. Consider again the lth iteration of the outermost loop:

$$M[i][l] = \min(M[i][l], M[i][l] + M[l][l]).$$

Once again, because $M[l][l]$ is always 0, $M[i][l] = M[i][l]$ for iteration l. As such, because the values needed to update the matrix for a certain iteration are constant for that iteration, everything within the outermost loop can be completed in parallel.

With this in mind, we can consider how we will divide the work up for each iteration of the outermost loop. Given t pthreads, we allocate $\lceil \frac{|V|}{t} \rceil$ contiguous rows to each phtread. In the case of $|V| < t$, we will do the same thing, accepting that the remainder of the pthreads will simply be unused. Also, in order to ensure that no pthread has started on the next iteration of the loop before the rest, we will enforce a pthread barrier at the end of each pthread's computation that will not let any pass until all of them have finished. All of this is to ensure that the most important invariant, that $M[i][j] = \min(M[i][j], M[i][k] + M[k][j])$, is always upheld. For extra security, we will add in an assert statement to make sure that no cells have the value of 0 besides the ones wherein the row is equal to the column. After all iterations of the outermost loop have completed, we will join all the pthreads.

Correctness Testing Plan

The testing plan is as follows:

1. Make a collection of small(er) graphs that are easily computable by hand. Include a few test graphs that are edge cases such as being empty (which should result in an error), being very sparse, and being disconnected. Solve each one, and then run each one with the serial version of the program. If they are correct, run the parallel program on the same collection of test graphs multiple times with different thread counts. If the answers match up, continue.

2. Next, make a collection of larger test graphs by using a random number generator with constraints to give edge values that will be blindly placed in an adjacency matrix. Then, run back through each matrix and, depending on the value of another call to an RNG, set certain edges equal to $1000|V|$. That is, given a range of RNG outcomes, make it so that if it gives a value within a certain range then the edge is 'deleted' by having the weight set to infinity. Change the percentage of missing edges for each graph to emulate different levels of sparseness.

3. Run the serial program on these graphs and record the output, and then run the parallel program on the graphs and check for equality. Once again, make sure to run the parallel program with a variety of different thread counts. If all of the outputs are the same between the parallel and the serial programs, we can assume with a high degree of certainty that both programs are correct.

Performance Hypothesis and Testing Plan

The performance testing plan will follow the one laid out in the Homework 1 sheet. The test graphs that I will use will be created via the same method as the correctness testing graphs. My hypotheses are as follows:

1. **Parallel Overhead:**

   When independently running Serial and Parallel with $T = 1$ and $N \in \{16, 32, 64, 128, 256, 512, 1024\}$, we will see, for each value of N, that Parallel is a constant amount of time slower than Serial and that the ratio of Serial to Parallel will approach 1 as N grows larger (the constant overhead to create and join 1 thread is getting smaller in reference to the overall runtime as the inputs get larger).

2. **Parallel Speedup**:

   I hypothesize that for small values of N such as 16 or 32 then the speedup will be very small and that it will plateau once the number of threads becomes larger than the number of nodes. However, I believe that the speedup will become considerably more pronounced on larger input sizes. For the largest example of $N = 1024$, I suspect that the speedup will resemble a straight line.