ECE 420 Parallel and Distributed Programming
Lab 2: Shortest Paths between Cities

Winter 2015

Suppose we are given a number of cities and the cost of traveling between any pair of these cities. For each pair of cities, what is the smallest cost of traveling between them? More generally speaking, given a directed graph and the weights of all the arcs, what is the length of the shortest path between any two vertices?

1 Background: Floyd-Warshall Algorithm

The Floyd-Warshall algorithm is one of the algorithms that can elegantly calculate all-pair shortest paths on a directed graph. Suppose we have a *directed graph* $G(V, A)$, namely, a graph with the vertex set V and the arc set A, with $A(i, j)$ representing the cost (weight) of arc (i, j). And there are N vertices in total. The main idea of Floyd-Warshall algorithm is to decompose the all-pair shortest path problem into a series of sub-problems, which are solved recursively. Consider the shortest path between any two vertices i and j *that only uses the first k vertices as possible intermediate points*, and denote its length by $w^k(i, j)$. There is a recursive relationship between the length of the shortest path of iteration k and that of iteration $k - 1$, i.e., we have

$$w^k(i,j) = \min \{w^{k-1}(i,j), w^{k-1}(i,k) + w^{k-1}(k,j)\}, \quad \text{for all } i, j. \tag{1}$$

To make the notations consistent, in the initial iteration $k = 0$, we let

$$w^0(i,j) = A(i,j), \quad \text{for all } i, j. \tag{2}$$

By calculating (1) recursively for $k = 1$ to N, we can finally get the lengths of shortest paths between all pairs of vertices. Algorithm 1 shows the pseudo code for a serial implementation of the Floyd-Warshall algorithm.

Algorithm 1 The (Seiral) Floyd-Warshall Algorithm

```
**Input:** A graph $G(V, A)$ with $N$ vertices; the cost $A(i, j)$ of each arc $(i, j)$.
**Output:** A weight matrix $W = (w_{ij})$ representing the lengths of all the pairwise shortest paths.
for $i = 1$ to $N$ do
    for $j = 1$ to $N$ do
        $w_{ij} \leftarrow A(i, j)$
for $k = 1$ to $N$ do
    for $i = 1$ to $N$ do
        for $j = 1$ to $N$ do
            if $w_{ik} + w_{kj} < w_{ij}$ then
                $w_{ij} \leftarrow w_{ik} + w_{kj}$
```

Note that in this pseudo code, there is only one weight matrix. In each iteration k, the weight matrix is updated to store the latest results.

2 Parallel Implementation of the Floyd-Warshall Algorithm

We can implement Algorithm 1 with multiple threads by letting each thread handle a block (or a row/column) of the weight matrix. Similar to the serial version, we keep a matrix to store all the weights and let each thread update its part of the matrix in each iteration. However, in the parallel version, some threads may run faster so that when it updates $w^k(i, j)$ by (1) in the k^{th} iteration, the related $w^{k-1}(i, k)$ and $w^{k-1}(k, j)$ might not have been updated to the $(k-1)^{th}$ iteration yet. In this case, the procedure will fail to produce the right results.

Synchronous Implementation. To avoid such concurrency issues, one idea is that we can synchronize all the threads at the end of each iteration. Recall that one way to do this is to insert a barrier at the end of each iteration inside each thread. Another way to do this is to use the join function. (Different methods may be different in terms of run times). In this approach, every thread is "synchronized" in each iteration to guarantee that all the weights in $(k-1)^{th}$ iteration are available to be used by the calculations in the k^{th} iteration.

Asynchronous Implementation. Another idea is to implement the algorithm in an "asynchronous" manner. When a thread is about to update $w^k(i,j)$ in its k^{th} iteration, it can check whether $w^{k-1}(i,k)$ and $w^{k-1}(k,j)$ are already available ($w^{k-1}(i,j)$ should be available too). It will be blocked if any of these is not available. The thread(s) updating $w(i,k)$ and $w(k,j)$ will signal the blocked thread after finishing the updates in the $(k-1)^{th}$ iteration.

3 Tasks and Requirements

Tasks: implement multi-threaded versions of the Floyd-Warshall algorithm using Pthreads. Implement a "synchronous" version and then an "asynchronous" version as discussed in the previous section. (You need to finish two programs.) Make sure your program is correct.

Requirements and Remarks:

- You can use the "datagen.c" to generate uniformly *random* inputs. Your results should be saved in the same format as the data generated by "datagen.c" using the file name "data_output". Some IO functions are also provided. However, it is not mandatory to use the provided IO functions.
- Your program should be able to run with different numbers of threads. However, you only need to consider the case in which the input size is divisable by the number of threads.
- Measure and compare the run times (or speedups) of different implementations under different numbers of threads.
- In the lab report, describe your implementation choices clearly.
- In the lab report, compare the performance (speedup or efficiency) of your implementations under different numbers of threads. You ONLY need to consider *one or a couple of* fixed input sizes in this lab. Explain your results.

- Your results will not be marked competitively based on performance. The problem is open-ended in nature. The focus is on explaining the performance.

Please have a TA check your work when you finish your implementations with your marking sheet filled with names and IDs. You will be asked to run your programs with some specific inputs and numbers of threads. The results will be checked against those produced by the serial program "serialtester.c". For the lab report, please also refer to the Lab Report Guide for general guidelines or formatting issues. The report should be submitted to the assignment box in ECERF by **4:00 pm on February** 11th.

4 Hints and Tips

Synchronous Implementation. Synchronization at the end of each iteration can be achieved by barriers or join(). However, in the case of join, new threads must be launched again to continue updating the weights. Therefore, barrier could be more efficient. You can reuse the sample code for barrier posted on eClass.

Asynchronous Implementation. For asynchronous implementation, to update $w^k(i,j)$ in the k^{th} iteration, we must have $w^{k-1}(i,k)$ and $w^{k-1}(k,j)$ available. ($w^{k-1}(i,j)$ should be available too, but it should have been calculated by the same thread in a previous iteration). You can use semaphores or condition variables to implement such block-waiting and signalling mechanisms. Think about how many condition variables or semaphores you need to declare for this purpose. How to coordinate the events of blocking and waking up? Can an asynchronous implementation be more efficient or not? Why or why not?

What will happen if the $w(i,k)$ or $w(k,j)$ are updated with more iteration than $w(i,j)$? For example when one thread is updating $w^k(i,j)$ at the k^{th} iteration, $w^{k+1}(i,k)$ or $w^{k+1}(k,j)$ is already updated. Does it matter for producing the right result?

Advanced open questions: Could read-write locks be helpful in this scenario? Does the input affect the performance?

Tips: The problem is open-ended in nature and could lead to open-ended discussions. However, this lab is not a competition and the requirement is minimum. You will meet the requirements as long as your programs are correct and you have explained the observations. You can reuse any sample code provided to you on eClass or in lectures.

ECE420 Lab 2: Marking Sheet

Names and IDs:______________________________

In-Lab (40):

Synchronized Approach. 15	__________
Asynchronous Approach. 20	__________
Time Measurement. 5	__________

TA Signature:____________________

Lab Report (60):

Description of Implementation 20	__________
Testing and Verification 5	__________
Performance Discussion 20	__________
Conclusion and Experiences 5	__________
Presentation 5	__________
Coding Style 5	__________

TA Signature:____________________

Total (100): __________