Thomas Waterman
ICS 440 - Programming Assignment 4

A Concurrent Approach to the Floyd-Warshall Algorithm

(and the recognized pitfalls of my approaches)

When determining the shortest distance between two paths in the Floyd Warshall algorithm, the distance is compared from the direct path of two point (i, j) and the distance between the two points when stopping at an intermediate point along the way (k). In the Floyd Warshall algorithm, this is represented using nested loops as so (in light pseudo code):

```
For each |V| k {
    For each |V| i {
        For each |V| j {
            The distance of (i,j) = min((i,j), (i,k) +(k,j)
```

We can see because of this, that k not only represents a stopping point along the way, but can also be thought of the entire previous iteration working through the matrix of points. Because of this, determining the distance between (i, j) is fully dependent on the value of k, and the data returned from the previous iteration, as we continuously are updating shorter distances when they are discovered. Therefore, the k level of the nested loop cannot be executed concurrently. However, so long as k remains the same, all distances between any two point (i, j) can be discovered in any random order. This presents a huge opportunity for concurrency.

In my initial design, my thought was because the distance between any two points could be calculated at random, perhaps it would be best to let all threads execute on the matrix and find the value between two points, as if the processors were completing an excel spreadsheet. So long as all cells were submitted to a task queue, the threads could work together to fill out the matrix:

For each k (where i is rows, j is columns):

T1 computes (1,1)			
		T2 computes (2,3)	
	T3 computes (3,2)		
			T(x) computes (i, j)

and so on, throughout the 5000 x 5000... matrix
by the use of maximum of 10 threads.

While this approach certainly recognizes the opportunity for concurrency, this was far too demanding on the available overhead. In my original approach, I had created an object of type <Cell>, which held variables i, j, and distance. The idea behind this was every cell could be added to a list of futures, which then as the list was iterated through, the values for i and j could be called, and the distance could be updated in the matrix at points (i, j). The pitfall of this approach is that ultimately 25,000 callable objects are created, and return 25,0000 futures of type <Cell> to be added to a list of futures. This is not something that my computer could handle from a hardware standpoint. When attempting to run this against a matrix of 1000 x 1000, my approach would actually run significantly slower than a sequential approach. When attempting to run this approach for a matrix of the required size (5000 x 5000), Java would run out of physical memory, and throw an out of memory error.

Clearly, this needed to be redesigned, and approached from a new standpoint, which is what the remainder of this paper is on.

Partitioning:

I wanted to come up with a larger approach, that still took advantage of the small tasks that this could be divided into. My next thought was to have each thread determine all values for each row throughout the matrix. This takes the number of tasks to be executed from a que from 25,000 down to 5,000, and seemed like a next best step. This is domain decomposition. For each k (where i is rows, j is columns):

T1 computes each distance to j from i	→	→	→
T2 computes each distance to j from i+1	→	→	→
T3 computes each distance to j from i+2	→	→	→
T4 computes each distance to j from i+3	→	→	→

and so on, throughout the 5000 x 5000... matrix
by use of maximum of 10 threads.

Communication:

Each thread will need to communicate any updated distances within the matrix so that for the next iteration of k, the values can be referenced. A Callable or Runnable could be used by the thread pool, that is created as an inner class so that the matrix is directly available for manipulation. This minimizes the volume of comparisons, object variable assignments, and overall overhead.

Agglomeration:

While a Runnable is more desirable, as we do not need to necessarily access the data from the thread (now that the Runnable or Callable is a class created as an inner class) a

Callable will be more ideal for overall control. All threads will need to join before entering the next iteration of k. By creating a Future (that holds nothing, as the matrix is directly updated), we can create a list of these futures, and execute the method future.get() for the entire list of futures, ensuring all threads join. By approaching the problem in this manner, the overall work to be done is now fully divided into iterations of k.

Mapping:

Tasks will be mapped to processors by use of thread pools, and Callables to submit tasks to a queue of tasks for completion. Callables are desired over Runnables, as we can create a null variable used to control the thread pool, calling future.get() on all threads to control the iterations of k (ensuring all threads have finished all necessary work before entering next iteration of k).

Testing will be conducted on thread pools of size 1 -10, assigning each thread a series of Callable tasks. When all threads have completed after each iteration of k, the shortest distance between all pairs will be computed.

Results:

Below are the results times for running a sequential approach vs. a concurrent approach on the same input matrix, at increasing counts of threads in the thread pool. In all tests, the result matrices of both the concurrent and sequential execution contained the same result distances. Testing was done on a Mac machine running a 1.1 GHz intel quad-core i5 (4 core processors).

Number of Threads in Threadpool	Sequential Execution Time (in seconds)	Concurrent Execution Time (in seconds)
1	79.0706222	109.725455925

2	73.760598549	52.58330732
3	75.756305182	38.61266982
4	78.978202634	38.543702174
5	77.347055738	35.238257578
6	74.009865582	35.673082791
7	75.968148045	33.954728169
8	76.934721066	39.548439576
9	78.692250957	40.902827191
10	75.396552287	39.592041124

An interesting observation to note is that after 3 threads were uses in thread pool, the overall speedup has seemed to have topped off using this approach at close to a 2x speedup. The use of more threads is not giving additional return, and therefore they are unnecessary as far as resources are concerned. Because of this, I feel that there must be a better way to partition, and agglomerate the overall work to be done, to maximize on the use of 10 threads in a pool.

Perhaps it would be best to take the partitioning and agglomeration farther. Instead of assigning a thread pool to a given row, what if the overall matrix was to be divided up into smaller matrices the size of the thread pool themselves? A single thread would execute per usual on a matrix that mirrors the input matrix, however, a thread pool of 10 threads could each concurrently tackle 10 matrices that are subdivisions of the origin matrix itself. This is an interesting idea to explore, but exceeds the scope of remaining available time for this project.