ECE 420 Lab 2
Parallel Floyd-Warshall Algorithm Implementation

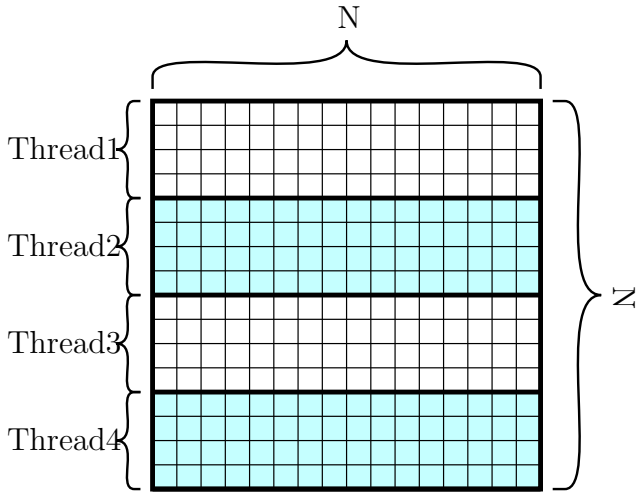
Mark Langen - 1293728

2015-02-10

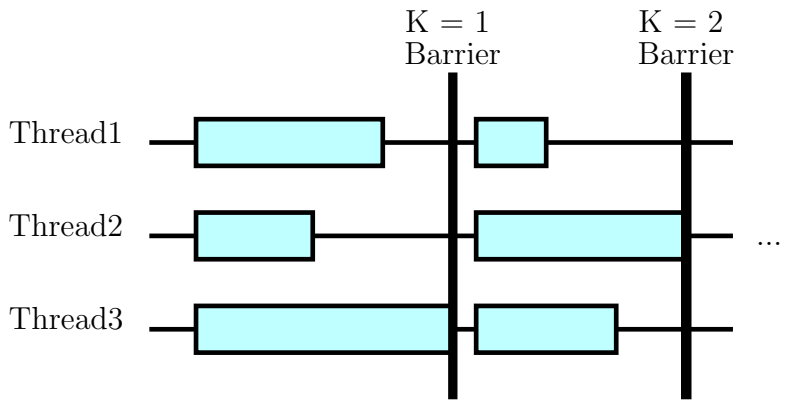
1 Description of Implementation

In this lab we wrote parallel implementations of the Floyd-Warshall algorithm for finding the all-pair shortest distances associated with travelling between a set of cities. Given the volume of calculation necessary to solve this problem, it is a good candidate for parallelization.

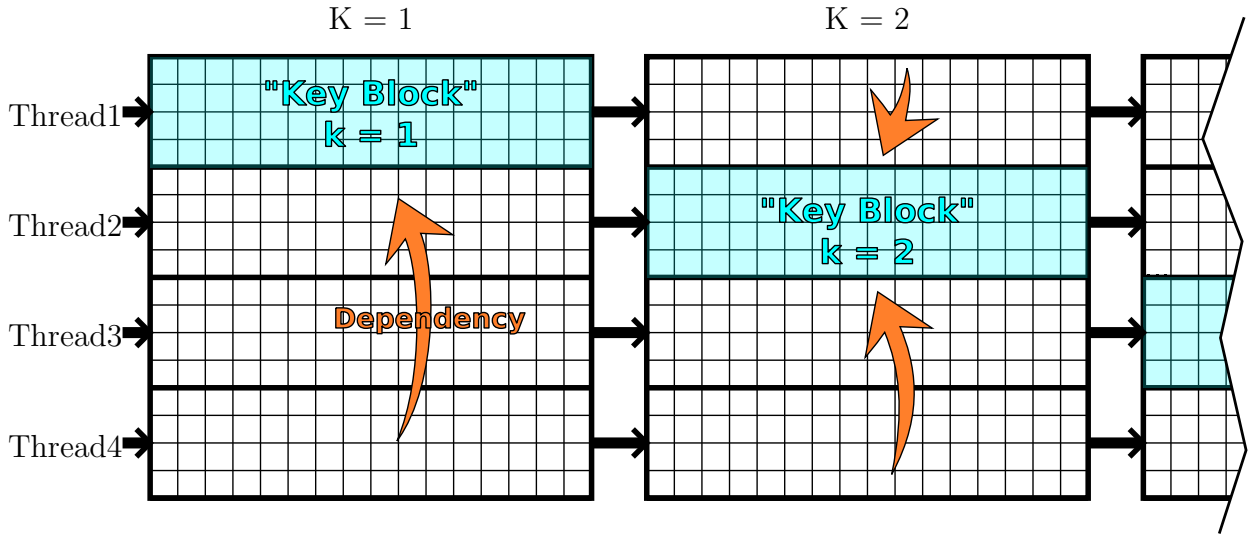
We developed two separate implementations of the parallel algorithm, each with a different approach. At their core, both implementations consisted of dividing up the $N \times N$ Floyd-Warshall matrix evenly into several blocks of rows, with a separate thread assigned to each of the blocks. The two implementations differed in the synchronization method used between the threads. Below we will use k to refer to the main outer loop iteration number, with $0 < k < N$ as per the typical Floyd-Warshall algorithm.



The first, *synchronous* implementation, kept all of the threads in sync with one and other for a given k^{th} iteration step over the data. That is, for the synchronous implementation, every thread was kept at the exact same value of k. This was done through the use of a global `pthread_barrier`. Each thread waited on the pthread barrier after completing each of its iterations. Since every thread was kept on the same iteration for this implementation, we could flatten the data needed, into only storing the distance values for the current k and destructively updating those values in the calculation. Since the dependency during iteration k is on the data in the k^{th} row and column, and the k^{th} row and column never change during iteration k, there is no conflict between the threads when destructively updating a single $N \times N$ distance matrix.



The second, *asynchronous* implementation, allowed each of the threads processing different parts of the matrix to run at their own pace, only halting for synchronization when the data that they depended on was not yet ready. Since our implementation broke the matrix into blocks of rows, the dependency in question is that during each thread's k^{th} iteration, it depends on the block containing the k^{th} row being complete up to the $(k-1)^{th}$ iteration. We'll call this $(k-1)^{th}$ iteration of the k^{th} block the *"key block"* for that k. In order to synchronize the threads, we created a completed flag and condition variable for each of the key blocks. When any thread wanted to begin processing its k^{th} iteration, it would lock the condition variable, and check the completed flag for the key block for that k. If the key block was not completed yet, it would wait on the condition variable to halt until it was complete. When finished with its processing for the k^{th} iteration, each thread would check if it had just completed the key block for that k, and signal on the condition variable if it had. For the asynchronous implementation we used an $N \times N \times N$ matrix to store the distances. Since the threads proceeded at their own pace, we needed to keep all of the distance data for each k^{th} iteration. If we destructively updated the matrices like we did for the synchronous implementation then it might cause problems if one of the threads had not yet used that data we were overwriting.



Both the asynchronous and synchronous versions of the program also made use of a set of sets of `pthread_join`s in order for the main thread to wait for all of the worker threads to complete the final result.

2 Testing and Verification

In doing our implementation, the first thing that we did was set up a test harness in order to verify the correctness of our implementation. We developed a Makefile that would generate a test set of data, and run the program on it with varying thread counts verifying the correctness of the output each time against a provided known good single threaded implementation.

Using this test harness we could verify that our changes were valid as we worked on developing the asynchronous and synchronous implementations of the program. We attempted to make an alternative version of the asynchronous implementation, which used a different block structure and synchronization strategy. We believed that this would be faster, but the test harness uncovered that that alternative version was not correct, and would fail under some circumstances.

Finally, after completing both versions of the program, we ran them over an exhaustive test of the lower end of the problem space to help uncover any subtle edge case related problems. We generated the cases of 1 to 10 cities and tested them with the thread counts 1 to 8 wherever the constraint of $city\_count \pmod{thread\_count} = 0$ was satisfied. We also tested the upper boundary with an input size of 1024 cities and 16 threads. We uncovered no issues with either implementation.

3 Performance Discussion

We were successful in creating a multi-threaded implementation of the Floyd-Warshall algorithm that performed significantly better than the single-threaded version. Our synchronous version of the algorithm was able to achieve an efficiency close to 100% for 1-4 threads. However, the asynchronous version was unable to perform as well as even the single threaded version, running slower than it. For 256 cities:

$$t_{single\,threaded} = 82ms$$

$$t_{2\,threads,synchronous} = 44ms$$

$$t_{4\,threads,synchronous} = 24ms$$

$$t_{2\,threads,asynchronous} = 105ms$$

$$speedup_{2\,threads,synchronous} = \frac{82ms}{44ms} \simeq 3.4\times$$

$$speedup_{4\,threads,synchronous} = \frac{82ms}{24ms} \simeq 1.9\times$$

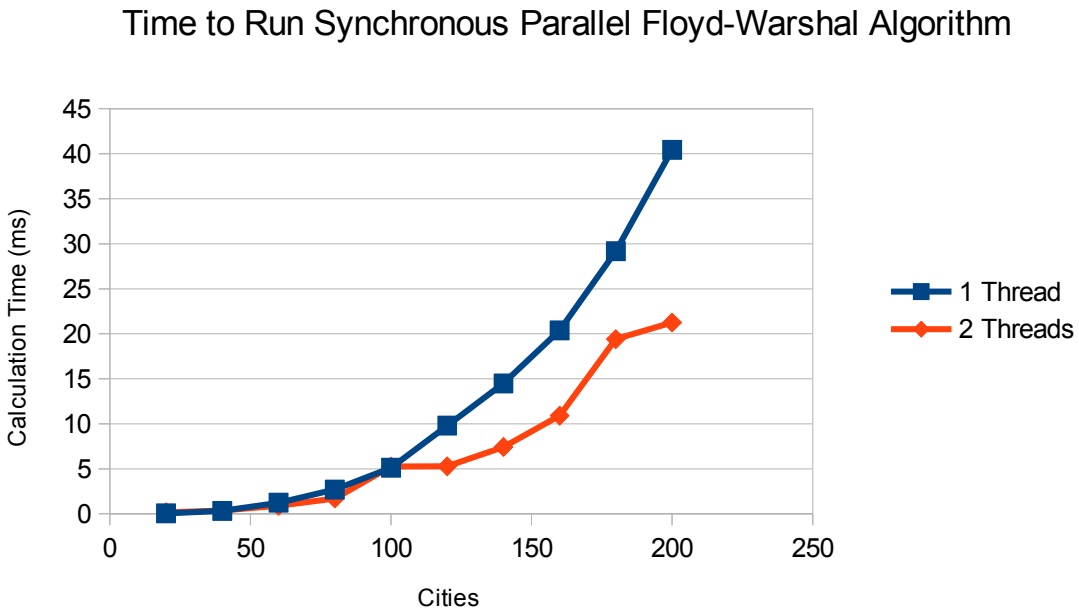
$$\text{efficiency}_{4\,threads,synchronous} = \frac{82ms}{4 \times 24ms} \times 100\% \simeq 85\%$$

$$\text{efficiency}_{2\,threads,asynchronous} = \frac{82ms}{2 \times 105ms} \times 100\% \simeq 39\%$$

As you can see, the asynchronous performance is quite bad at $< 50\%$ for two threads. It turns out that the synchronization overhead for the asynchronous version dominates over the gains that might be had by allowing each thread to work at its own pace. The asynchronous version is also requires us to allocate and use much more memory, putting more strain on the system and cache.

Even if we could fix these issues with the asynchronous version, it is unlikely that we would be able to get a significant performance gain. If we are to map out the dependencies between various $k+$ block pairs, we find that the synchronous version is pretty close to optimal. If you take a given pair of cities in the final result, and step back a few k iterations, you will find that the set of distances that that one distance depends on explodes quite chaotically with even a few k steps. Even just 4 iterations back, one value in the final result depends on > 80 values spread fairly evenly across the distances matrix. This is evidence that the threads doing the calculation have to be fairly close to being synchronized to avoid being halted on unfulfilled dependencies anyways, so the theoretical gains from an asynchronous version must be small at best over a synchronous one.

As for the performance characteristics with input size, up to around 100 cities, the synchronous version of the algorithm does not perform significantly better than the single-threaded version. After 100 cities the synchronous version approaches the calculated $\simeq 80\%$ efficiency.



4 Conclusion and Experiences

In implementing the synchronous and asynchronous versions of the Floyd-Warshall algorithm we found that there is significant room for parallelization in this area. We found that the synchronous approach is much better than the asynchronous approach for implementing this algorithm. we found that the synchronization overheads on the asynchronous implementation are much too large over the simple barrier used for synchronization in the synchronous version, making the asynchronous implementation a bad choice. We also found that even theoretically there is not much to gain by using an asynchronous implementation.