Assignment 1 : Design and Test Document

Rachel Hwang

January 14, 2014

1 Serial Program

This implementation of the Floyd-Warshall algorithm will be quite similar to standard matrix multiplication. The algorithm requires three loops, the nested inner loops i an j iterate through, for each cell in the matrix, the kth row and the column, summing entries pairwise and updating the cell appropriately (that is, with a shorter distance if found). The outermost loop k, performs these updates over the whole matrix n times. Note that the first row and first column will never have their values updated.

Since this program will be iterating through the same matrix along two dimensions, this implmentation with create two copies of the original matrix, each an array, and transposing (effectively swapping content across the top-left to bottom-right diagonal) one copy in order to take advantage of data locality. The data structures that this implementation will require are thus simply two arrays each indexed $[n * i + j]$. Both will be updated simulanteously as the algorithm is executed.

While the transposition operation does add some overhead time because it involves n reads and writes, for large input sizes, it is negligible compared to the benefits of being able to access both matrices in sequential order. More pressing is the fact that using two matrices requires a write to each seperate matrix per update. I am unsure about the advantages of transposition given the additional cost of performing updates on two matrices, so I plan to compare the performance with and without.

In summary, this serial implementation will read in an adjacency matrix from a file and store two copies of the matrix in two flat arrays a and b. b will be passed to a function for transposition. The algorithm itself runs using the i, j and k loops described above. After exiting these loops, the program returns a, which holds the final results.

2 Parallel Program

This implementation will use the same basic structure of the serial implementation, but with the additions of Posix threads, mutexes, a simple wrapper data structure for passing arguments to threads and a counter which will keep track of current position in the matrix.

Given the i, j, and k loops, only the i loop seems suitable for threading. The k loop cannot be threaded since each k iteration depends on the results from the previous iteration. The j loop should not be threaded because the work which each thread would perform in parallel, just a few operations, would probably not compensate for the overhead of spawning the additional threads. Thus, the i loop is distributed among threads and the j loop is packed into a function which will be passed to each of the threads. Each thread will be given this function, and pointers to the counter and the arrays a and b. This will allow the program to perform the algorithm on 16 seperate rows in parallel.

To summarize this implementation, the setup is same as the serial program, with the additions of initializing the counter, the mutexes and the threads. Within each k loop, threads are spawned. Because each thread is operating on a seperate row, the number of the next available row must be stored in the counter (which is initialized to zero). Each thread operates as follows: it checks the meta-data in the counter for the next unclaimed row, locking the counter with a mutex as it checks it. If the counter number is greater than n, the thread exits. If not, the thread increments the counter, unlocks it and proceeds to perform the work of the j loop for its assigned row. Once finished with that row, the thread function loops, again checking the counter and proceeding to the next available row until they have all been updated. After all of the threads have finished their work and exited (exit status monitored using the pthread join function), that iteration of the k loop is concluded and the program proceeds to the next k iteration.

Since threads are divided by rows, each thread performing the work of the j loop on one row at a time, it is necessary to keep the information on which row is the next unclaimed centralized and synchronized, which is accomplished using the counter and the mutexes. Because the counter is locked when viewed and/or updated, each thread will only see the latest counter value and no two threads will ever attempt to work on the same row simultaneously. Once assigned a row, threads will only be updating disparate values, so the counter only imformation that must be communicated between threads. Only the counter and the arrays a and b will be kept synchronized since both are centralized.

3 Testing

This section focuses on testing for correctness, returning the expected results) more than perfomance, which is a focus in the Experiments section. I plan to use unit tests for each my serial and parallel implmentations, applied consecutively. Most tests should and will be applied mid-development. The serial tests will also be applied to the parallel program.

For Serial:

1. Read-in: compare matrix printed from an input array to the original?
2. Transposition: compare matrix printed from b to the expected result.
3. i and j loops: check results of matrix after one k loop iteration. Small inputs. $n = 2, 10, 100$
4. Correctness: run complete program with small inputs $n = 10, 100$
5. Extreme values: run with dummy matrices will all zero values, all infinite. The program should actually check for any unexpected values, or in the case of zeros, avoid unnecessary computation.

For Parallel:

1. Soundoff: Are all threads working? Print pID in the thread function to ensure all threads execute correctly.
2. Exit: Do all threads exit correctly at the end of the k loop? Print exit status.
3. Lock: Check Mutex operation. See counter.
4. Counter: is the counter initialized to 0? Does the thread function check the counter and exit correctly if the counter value is greater than n? Print the counter to ensure it is read each time.
5. Correctness: run complete program with small inputs $n = 10$, 100

4 Experiments

1. Transposition: My experiences with matrix multiplcation proved the value of optimizing for cache usage, but as mentioned, I am unsure of the value of transposition. I plan to compare the performance of the program with and without. Without transposition, there is also no need for a a second copy of the matrix, which saves both update overations and overall complexity of the program. I hypothesize that transposition will provide a small, but noticable speedup thanks to the better cache hit rate.
2. Thread Count: Unexperienced with parallel programming, I have little idea of what is the optimal number of threads to use. This will of course depend on the input size, but I hypothesize that using the maximum number of threads possible will become optimal above some threshold of n size, perhaps $n > 256$. $n < threads$ will of course be unusually slow. I may also experiement with dymanically assigning the number of threads based on the input size which requires minimal additional code. I hypothesize that the optimal number of threads will no greater than $n/2$, since each thread should do at least the work of two rows in order to compensate for thread overhead cost.
3. Serial vs. Parallel: Ideally the parallel program should provide about a (thread-count)x speed-up, but given the overhead of creating threads, context switching and managing meta-data, I would be surprised to see that much improvement. I hypothesize that the increase in performance will be very small (or even negative) for small input sizes or in any case where $n < threads$, then increase sharply with n, then flatline somewhere before t times improvement (graphing this increase should yield a log shape). These improvements will hinge upon using the optimal number of threads, as discussed above.