Full Name:
Andrew Id:

15-418/618
Exercise 4
SOLUTION

Problem 1: Lock Implementations

A. Direct.

This approach scales very poorly and is much slower than the others. The threads repeatedly contend with each other with CAS operations, requiring many iterations and many bus operations.

B. Test.

This approach does better than the direct method, but it still has serious scaling and performance problems. When the lock is released, multiple threads will detect this and attempt to acquire it through a CAS operation. At most one will succeed.

C. Backoff.

This approach does extremely well, even outperforming fetch-and-add. Its performance is almost independent of the number of threads. As before, the threads repeatedly test the lock status, but then they delay for widely varying amounts of time. These delays spread out the attempts to perform CAS operations, increasing their likelihood of success.

Problem 2: Memory Consistency

A. Explain why these two fence operations have such different relative performance.

With the global variable, all of the threads will be performing fetch-and-add on the same address. This will lead to lots of contention and bus traffic. With local variables, each thread will perform its fetch-and-add on a separate address, and this can be handled within the local cache of the executing thread.

B. Explain why the local version has performance that improves with the number of threads.

These fence operations all occur independently and so can be done in parallel. In fact, we can see that the program is getting almost perfect speedup (7.62× for 8 threads).

C. For each of the five functions, determine where fence operations must be inserted to guarantee the ordering properties. You should insert only a minimum set of fences. Justify why these particular fences are required and why no others are.

Many of the memory operations in these functions are performed by compare-and-swap operations, and these already act as barriers. The only direct operations are:

(a) Setting the lock to 0 in `cas_lock_init`. This operation should have a fence *after* it, in order to ensure that the lock initialization is completed before any thread can attempt to acquire it. Since other threads are spawned after the lock initialization, adding a fence here will ensure that the lock will have been initialized before they attempt to acquire the lock.

(b) Setting the lock to 0 in `cas_lock_unlock`. This operation should have a fence *before* it to prevent other threads from being able to acquire the lock before this thread's write operations have completed.

(c) Testing the lock in `cas_test_lock` and `cas_backoff_lock`. This does not require a fence, because the function will perform a compare-and-swap operation after it, providing the necessary barrier for the lock operation.

Problem 3: Interconnection Networks

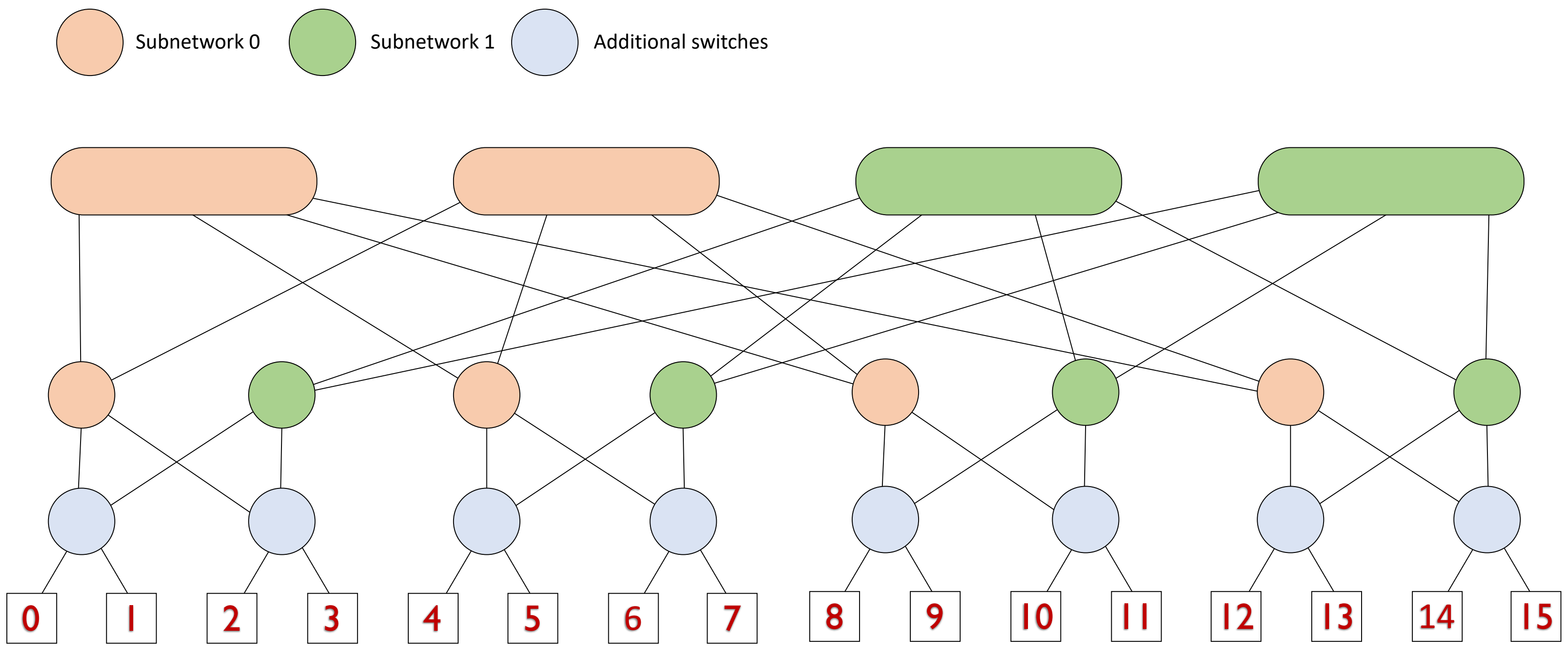

Figure 1: Fat-tree network, showing the recursive structure

A. Identify the $k/2$ subnetworks of type $N(k,2)$ for $k = 4$ in Figure 1. You can do this by modifying the diagram in Figure 1. Use different colors for the switches to indicate the different subnetworks and the additional switches.

B. Derive a closed-form formula for $P(k,l)$.

We can write a recurrence as follows:

$$\begin{aligned} P(k,1) &= k \\ P(k,l) &= k/2 \cdot P(k,l-1) \end{aligned}$$

The solution to this recurrence is $P(k,l) = 2(k/2)^l$.

C. Show that you could set up the eight links forming a *mirror permutation*, mapping port i to port $N-i-1$ for $0 \le i < N/2$. You can do this by modifying the diagram in Figure 2. Use different colors to illustrate the different links.

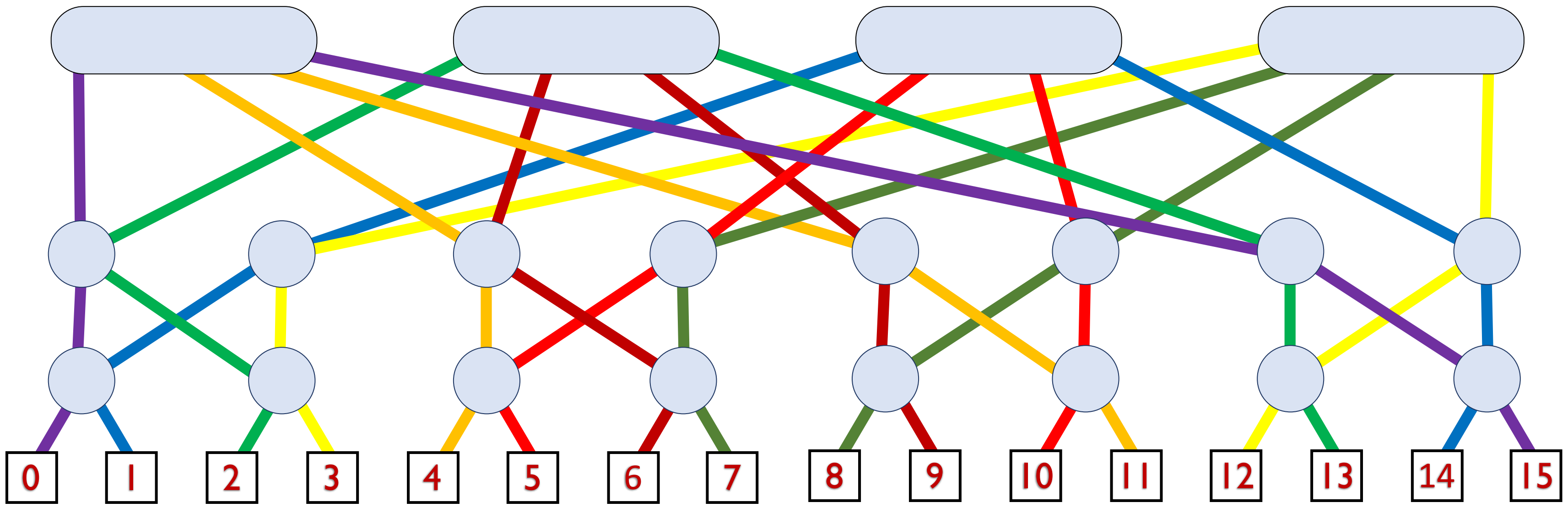

Figure 2: Fat-tree network with links for mirror permutation