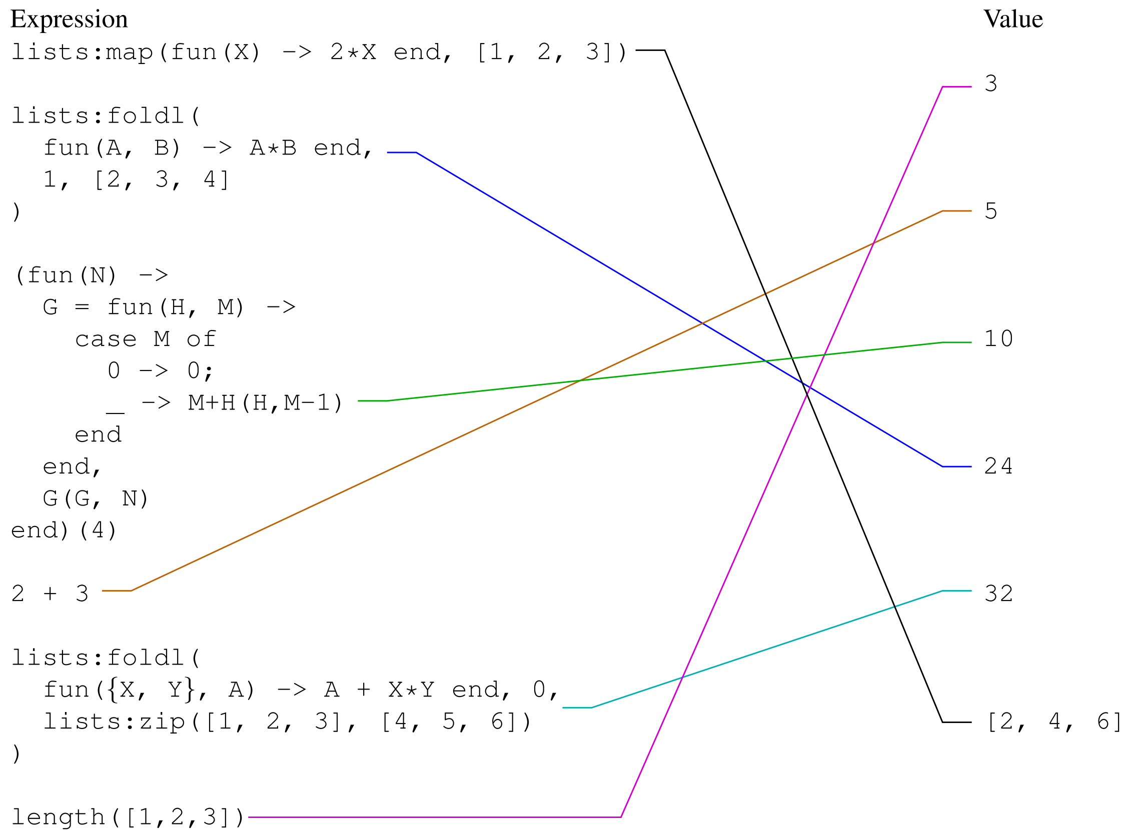
CpSc 418 **Midterm** October 18, 2012

Midterm

1. **Erlang (15 points + 3 extra credit).**
Draw a line joining each Erlang expression on the left to the value obtained by evaluating that expression on the right.
Grading: 3 points for each correct line, -1 point for each incorrect line.



2. **DLS Talk (10 points).**
What is a Craig Interpolant (check one):

- ☐ Craig Interpolants are a method for optimizing functional programs. The compiler computes the interval between when a variable is declared and when it is last read. If a new "copy" is made following the last read in order to modify a field, the compiler eliminates the copy and just modifies the existing value.
- ☒ If P and Q are boolean-valued formulas, and there is no way to satisfy both P and Q, then a Craig Interpolant is a boolean-valued formula, I, such that: $P \Rightarrow I$; $I \Rightarrow \neg Q$; and all variables that appear in I appear *both* P and Q.
- ☐ Given a list of points (x_1, y_1), (x_2, y_2), $\ldots (x_n, y_n)$ with $x_1 < x_2 < \cdots < x_n$, a Craig Interpolant is sequence of polynomials, P_1, P_2, $\ldots$, P_{N-1} for interpolating between values of the data. They are similar to cubic-splines but guarantee that interpolated values are bounded above and below by the values of the original data.
- ☐ Given an old API for Windows, or Linux, or Java, or Erlang, etc., and given a new version of the API, a Craig Interpolant is a structured way of writing a interpretation layer that allows programs that were written for the old API to run on a system with the new version.

3. **Synchronization (25 points)**

A mutual exclusion algorithm for two threads is *safe* if the two threads cannot be in their critical regions at the same time. An algorithm is *live* if any thread that tries to enter the critical region eventually does so.

Dekker's algorithm was designed to work with a shared memory where no processor is guaranteed uninterrupted access to the memory for a sequence of two or more reads or writes. For example, if a processor reads a memory location and then writes a new value to that location, other processors may access the same memory location between the read and the write. We say that a memory location is shared if it can be read and/or written by both threads.

In this problem, you will show that any mutual exclusion algorithm that is safe and live must use at least two shared memory locations. You will do this by showing that an algorithm that uses only one shared memory location cannot guarantee safety.

(a) **(8 points)** Consider the part of execution between when a thread leaves its non-critical code and when it enters its critical section. I will say that a thread in such a state is "trying to enter." Show that each thread must read the shared location at least once when trying to enter.

Assume otherwise. In particular, assume that thread 0 go through its "trying to enter" code without reading the shared location. Consider some exeuction that has reached a state where both threads are in their non-critical regions. I will examine two executions that continue from this state:

i. Thread 0 enters its critical section while thread 1 remains in its non-critical code. Let this be an execution where thread 0 does not read the shared location as assumed above. Thus, the actions of thread 0 while trying to enter only depend on its local state.

ii. Thread 1 enters its critical section. While thread 1 is still in its critical section, thread 0 tries to enter. As noted in the previous case, the actions of thread 0 only depend on its local state, and the local state of thread 0 is the same as for the previous case. Thus, thread 0 enters its critical section while thread 1 is in its critical section. Mutual exclusion is violated.

Grading: most solutions seem to have assume that the algorithm was based on using some kind of "lock." I'll show how to write a lock-free implementation that guarantees mutual exclusion but has a somewhat tricky problem with liveness at the end of my solution for this problem. However, my solution is too long to fit in the space provided. So, I will give full credit for a valid argument that assumes that the shared location is a lock. On the other hand, the problem statement did say *any* algorithm, not just one that uses a lock. Two points of extra credit will be given for any correct solutions that apply to *any* algorithm.

(b) **(5 points)** Show that each thread must write the shared location at least once when trying to enter.

Again, I will assume otherwise and consider two executions starting from a state where both threads are in their non-critical code.

i. Thread 0 enters its critical section while thread 1 remains in its non-critical code. Thread 0 may read and/or write its local state **and**/or the shared location.

ii. Thread 1 enters its critical section. Let this be an execution where thread 1 does not write the shared location while trying to enter. While thread 1 is in its critical section, thread 0 tries to enter. Thread 0 may read and/or write its local locations and/or the shared location is for the previous case. Because these locations hold the same values as for the previous case, thread 0 will enter its critical section. Thus, mutual exclusion is violated.

Grading: as for the previous part, full-credit will be given for a solution that assumes that the shared-location is used as a lock. One point of extra credit will be awarded for correct solutions that aapply to *any* algorithm.

(c) **(12 points)** Consider a scenario when both threads try to enter at the same time. Note that each thread must have its first write of the shared location and its last read. Using this (or you can come up with your own clever approach), describe an ordering of events that leads to a violation of mutual exclusion (assuming liveness).

As with the previous two parts, I will consider three executions starting from a state where both threads are in their non-critical code.

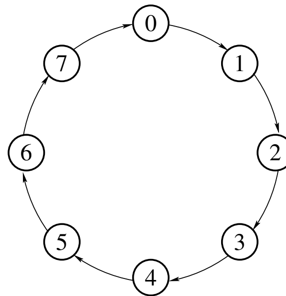
i. Thread 0 enters its critical section while thread 1 remains in its non-critical code. Thread 0 may read and/or write its local state and/or the shared location.

ii. Thread 1 enters its critical section while thread 0 remains in its and/or the shared location.

iii. Thread 0 tries to enter and makes pauses just before its first write to the shared location. At this point, thread 1 tries to enter its critical section. Because thread 0 has not written to the shared location, thread 1 sees the same values for its local locations and for the shared location as in the second case. Thus, thread 1 enters its critical section. Now, thread 0 continues. It immediately writes to the shared location, overwriting anything that thread 1 had written. Thus, the local locations for thread 0 and the shared location have exactly the same values as they did at this point of the first case, and thread 0 continues and enters its critical section. Thus, mutual exclusion is violated.

Grading: as for the previous part, full-credit will be given for a solution that assumes that the shared-location is used as a lock. Two points of extra credit will be awarded for correct solutions that aapply to *any* algorithm.

Here's an example of a lock-free algorithm as I promised above. The shared location is initially set to 0. When thread i is trying to enter its critical section, uses a while loop to spin until the shared location has the value i. Thread i performs its critical section, sets the shared location to $1 - i$ (i.e. sets the turn to the "other" thread), and continues executing non-critical code. This algorithm forces alternation between the two threads. For example, if thread 0 tries to enter it's critical section two times while thread 1 does not make such an attempt, then thread 0 will block. It is this property that one thread can be blocked from entering its critical section becasue the other thread *didn't even try*, makes this lock-free version fail the liveness requirement. It does guarantee mutual exclusion.



4. **Message Passing (25 points)**

(a) (10 points) Consider the simple ring network shown in the figure above. For simplicity, messages only travel clock-wise in the ring. Assume that each link can transfer a message in one unit of time. Now consider the program:

```
0. parallel_for(int i = 0; i < 8; i++) {
1.   while(true) {
2.     for(int j = 0; j < 8; j++) {
3.       if(i ≠ j) {
4.         node i sends a message to node j;
5.       }
6.     }
7.     node i receives any messages that have arrived.
8.   }
9. }
```

In other words, each node repeatedly sends messages to all of the other nodes. The receive at line 7 is just to make sure that all messages are delivered so that the network is never blocked waiting for a node to accept a message. The nodes have a enough buffering that they can be constantly offering traffic to the network. Assume that the computations are performed arbitrarily fast; so, the performace is determined only by how fast the network can deliver messages.

Show that in the long run, each of the eight nodes completes the while loop that spans lines 1–8 once every 28 time units.

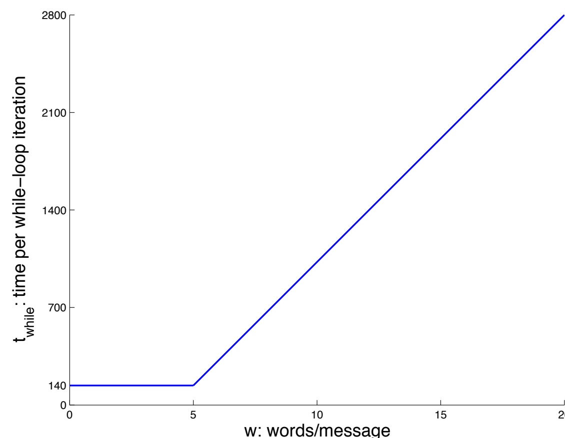
> Consider the link from node 7 to node 0. Each time all eight nodes complete the while loop, seven messages destined for node 0 must traverse this link (i.e. messages from nodes 1…7); six messages destined for node 1 must traverse the link (i.e. messages from nodes 2…7); and 7-i messages destined for node i must traverse this link (i.e. those from nodes $(i+1)\ldots 7$. The total number of messages traversing the link is
>
> $$\sum_{k=0}^{7}(7-k) \quad = \quad \sum_{k=0}^{7} k \quad = \quad 28$$
>
> The computations are "arbitrarily fast;" so, the time is determined by the network performance. Thus, each node completes the while loop once every 28 time units.

(b) (10 points) Now, assume that each node takes time t_0 total to send one message and receive one message (regardless of the message length). Assume that a link can transfer a message of length w in w time units. Derive a formula per iteration of the while loop as a function of t_0, w.

> Each node sends and receives seven messages for each iteration of the while loop. This takes time $7t_0$. By a calculation like that for the previous part, the time to send the messages through the network is $28w$. The throughput will be determined by whichever time is greater. Thus, the time to compelte each iteration of the while loop is:
>
> $$t_{while} \quad = \quad \max(7t_0, 28w)$$

(c) (5 points) Make a rough plot of the value of your formula for the case that $t_0 = 20$, and $0 \leq w \leq 20$. If you got stuck on (b), you can draw the shape that you think the plot should have and label critical points on the plot.

> If w is small enough such that $7t_0 > 28w$ then the time will be $7t_0$. Otherwise the time will be $28w$, The critical value for w is $w = \frac{1}{4}t_0$. So, for $w \leq 5$ the time per iteration is 140 time units. For $w \geq 5$, the time per iteration is $28w$. Here's the figure:



5. **Performance Measurement and Loss (25 points)** Give a short definition of each term below.

- If there is a mathematical formula associated with the term, write the formula **and** write a one or two sentence explaination of its significance.
- Otherwise, give a brief (one or two sentences) definition **and** a simple example (one sentence).

(a) Amdahl's Law

Amdahl's Law describes how the non-parallelizable part of a computation limits the performance of a parallel implementation. The formula is:

$$T_{parallel} = T_{sequential}\left(s + \frac{1-s}{P}\right)$$

where s is the fraction of the code that is inherently sequential, and P is the number of processors (assuming perfect parallelism for the parallelizable part).
Another way of writing the same thing is

$$\text{Speed_Up} = \frac{T_{sequential}}{T_{parallel}} = \frac{P}{s(P-1)+1}$$

Of course, algebra will let you write these formulas in a bunch of other, equivalent, forms.

(b) Computation overhead

This refers to extra computation performed by a parallel program compared with a sequential solution of the same problem. For example, our parallel version of the Sieve of Eratosthenes' (see the Sept. 18 slides), has each process compute the primes up to $\sqrt{N}$ to avoid the communication overhead of having one process compute these primes and then send it to all of the others.

(c) Communication overhead

This refers to time spent in a parallel computation for communication between parallel processes (or threads). For example, our count 3's program had time spent sending the counts from each worker processes to the root process for the final accumulation.

(d) Non-parallelizable code

Some computations seem to be unavoidably sequential, such as counting the number of elements in a linked-list. The portions of a program that are unavoidably sequential are refered to as *non-parallelizable code*.

(e) Speed-up

Speed-up is the measure of how much faster a parallel program is than it's sequential counterpart. Quantitatively,

$$\text{Speed_Up} = \frac{T_{sequential}}{T_{parallel}}$$

where $T_{sequential}$ is the time to execute the sequential program and $T_{parallel}$ is the time to execute the parallel one. (Note: a speed-up that is less than one indicates that the parallel version is *slower* than the sequential program).