CSEP 524: Assignment #3

(due prior to class, Tuesday January 29th)

1) **Reading:**
   a) *Threads Cannot Be Implemented as a Library*, Hans Boehm, 2005.
   b) *A Brief Overview of Chapel,* Sec 9.3.2, Task Parallel Features (pp. 10-13)
   c) Lin & Snyder, remainder of Chapter 6 (pp. 174-200)

   Submit 1 question per reading for consideration in class discussions by Monday evening, 9pm, January 26th.

2) **Concurrency vs. Parallelism:** Using the definitions given in lecture (*parallelism* suggests a computation that uses multiple tasks to accelerate a computation, but which would still be correct if executed with only a single task; *concurrency* suggests a computation that inherently uses multiple tasks as part of its algorithm), classify the following computations as being parallel or concurrent and explain your reasoning in a sentence or two.
   a) Implementing a graphics pipeline using multiple tasks
   b) Implementing the divide and conquer steps of Quicksort using multiple tasks
   c) Ray tracing using a task per pixel
   d) Implementing an event handler (like a GUI) using multiple tasks
   e) The Successive Over-relaxation example from Chapter 6

3) **Parallel Histogram:** A colleague writes the following Chapel code to compute a histogram from an array using your (bug-free) computeMyBlock() implementation:

```
config const n = 1000, numBuckets = 10, numTasks=4;
var A: [1..n] int = …;
var histogram: [0..#numBuckets] int;
coforall tid in 0..#numTasks {
  const myInds = computeMyBlockPart(1..n, numTasks, tid);
  for i in myInds {
    const bucketNum = computeBucketNum(i);
    histogram[bucketNum] += 1;
  }
}
var total = 0;
for hv in histogram do
  total += hv;
assert(total == n);
```

   Upon executing it s/he is surprised that the program runs to completion but that the assertion fails. What did s/he do wrong and how would you suggest fixing it?

4) **Multiple producer, multiple consumer bounded buffer:** Implement a bounded buffer that may be accessed by any number of consumer tasks and any number of producer tasks.

   Follow these guidelines:
   - Your solution **must not contain any data races** to be correct.
   - Your solution must, in principle, be relatively scalable to multiple producers and consumers. For example, the `produce` and `consume` procedures **should not** be implemented with a single lock over the entire buffer.

   a) Implement the buffer in C+Pthreads. The interface and test code is located in hw3/BoundedBuffer.h and hw3/BoundedBuffer.c. The README describes how to build the executable. Head and tail pointers may not be protected with the same lock.

   b) Implement the buffer in Chapel. The interface and test code is located in hw3/BoundedBuffer.chpl.

   c) Basic test code is provided in hw3/BoundedBuffer.c and hw3/BoundedBuffer.chpl. P producers produce a total of N elements. C consumers keep consuming until they consume a TERM element.

      A working solution:
      - Always terminates (no deadlock)
      - Prints producer and consumer counts that each add up to N.

      Include brief test output to show that your program works for a few representative inputs, such as when C and P differ.

   d) Add a timer to the **Chapel** version and report performance for a fixed N and at least 8 different combinations of C, P, and capacity.

Notes:
- Debugging may be aided by including assertions that buffer invariants hold at the start and end of `produce()` and `consume()` (as well as elsewhere depending on your design).

5) **Dynamic work distribution:** In the embarrassingly parallel study (Assignment 1 & 2), you wrote functions to statically distribute the work between tasks. When using this method of distribution, an imbalance of work between tasks, such as in the (ramp, factorial) case, cannot be corrected.

Devise a scheme (in either C+Pthreads **OR** Chapel) to dynamically distribute iterations. The particular approach is up to you. Note that we cannot prescribe an interface since the arguments and return types will depend on the approach you take; however, the invocation of the routine should take the following approximate form (in that it will likely need to call your distribution routine within a loop until no more work remains).

```
do {
  const myInds = dynamicDistribution(…);
  for i in myInds do
    …A[i]…
} while (myInds.size > 0);
```

a) Briefly discuss the scalability of your approach.

b) Write your new program and run it on (ramp, factorial), varying numTasks, and compare the performance to that of the static distributions from Assignment 1.

c) For one value of numTasks ≥ 8, provide brief output of this program that reports how many iterations each task executed.