CSEP 524: Assignment #4

(due prior to class, Tuesday February 5th)

1) **Reading:**
   - *MapReduce: Simplified Data Processing on Large Clusters*, Dean & Ghernawat, 2004.
   - Lin & Snyder, Chapter 5 – stop at end of "The Reduce and Scan Abstractions" (pp. 112-125) [**NOTE:** As stated in class, there is no need to become more familiar with Peril-L than you care to be/need to be to understand the core reduction/scan concepts described in this reading. Peril-L is defined in Chapter 4 and you should probably find it reasonably intuitive, but there's no need to become deeply familiar with it for the sake of this course].

   Submit 1 question per reading ***in a textfile format*** for consideration in class discussions by Monday evening, 9pm, Feb 4th .

2) **Impact of Load Imbalance:** In class, we discussed how the imbalanced block strategy defined by "give the first p-1 tasks floor(n/p) items; give the final task the remainder" could lead to ~2% load imbalance by applying it to multiple dimensions of a 3D array for a 16-core compute node. Defend or refute Brad's speculation that using this distribution to distribute to a larger machine could result in an even more significant load imbalance between the most unfortunate task and the typical one. In particular, distribute:

   - a 3D array of 64-bit reals consuming 25-50% of the system's physical memory
   - across 18,688 compute nodes, each containing 32 GB of memory and 16 cores

   You may either (a) distribute the array elements to the compute nodes themselves; (b) distribute them across all the cores; or (c) use one block distribution to distribute across the nodes and then a second to distribute across the cores (be sure to clearly state which case you are pursuing). For fun, feel encouraged to play the role of "adversary" by trying to find a pessimal/unluckiest problem size and use of the distribution, with the goal of maximizing the load imbalance (note: you will not be graded based on whether or not you find an absolute worst case; rather based on whether your argument is reasonable, accurate, and consistent). Show the math that backs up your argument, striving to be as clear as possible ("show your work").

   [Goal: analyze semi-realistic large-scale distribution/potential load imbalance case; start thinking about distributed memory computation and distributions]

3) **Proof of Deadlock-Freedom:** In class, we asserted that taking and releasing locks in a manner that respects a sorted order can prevent deadlock. Prove that this is the case

for two tasks using a proof by induction where P(*n*) = "Given *n* locks with a well-defined ordering 1..*n*; and 2 tasks, each of which needs to take a (potentially overlapping) subset of those locks; prove that if each task takes its locks in sorted order and releases them in the reverse order, deadlock cannot occur." Prove for all n > 0.

For those who need a refresher/tutorial on inductive proofs, I'll post a quick overview of them that should be sufficient for this problem on the discussion board soon – in the meantime, think intuitively about why this works (because you'll need that intuition in any case).

[*Goal:* formal reasoning about an important practical case]

4) **User-Defined Global-View Reduction:** Chapel supports the ability for users to create their own reductions by providing a class that (1) stores and initializes the state type, (2) accumulates input values into the state, (3) combines states, and (4) generates a result as we discussed in lecture. Using the provided template, familiarize yourself with how such reductions are created using the provided implementation of the min3 reduction (also described verbally in lecture). Then flesh out the template for a second reduction ('mostFrequent') that computes the most frequently occurring value in an array of integers. Show the output of your run for three problem sizes.

   [*Goal:* experience with the components required to create a generalized reduction]

5) **Implement a Collective Reduction:** As mentioned in lecture, Chapel does not yet support a general collective reduction feature (though it is planned for future work). Using the provided code framework, manually implement your own collective sum reduction routines using Chapel sync variables. In each case, each task will call into the routine, providing its task ID and contribution via arguments; the reduction routine should return the result in task 0 (other tasks can return a useful value or garbage as you wish). See comments in the code for further details.

   Provide the following variants:

   a) Implement a trivial "all-to-one" collective reduction which is computed via accumulation into a single variable in a coordinated manner;

   b) Implement a binary tree based collective reduction similar to the one described in lecture, assuming that the number of tasks participating is a power of 2;

   c) Create a variant of (b) in which the number of tasks may be arbitrary;

   d) Create a variant of (c) in which the tree's degree is 'd' rather than '2'/binary.

e) Supply sample output for your reductions for 3 different numbers of tasks (powers of two and non-)

[*Goal:* gain experience with the reduction pattern; further experience with interesting synchronization idioms]