CSEP 524: Assignment #5

(due prior to class, Tuesday February 12th)

1) **Reading:**
   a) *LogP: Towards a Realistic Model of Parallel Computation,* Culler et al., 1993
   b) A Brief Overview of Chapel, Sec 9.3.3, Data Parallelism (pp. 13-15)
   c) Lin & Snyder, Chapter 2

   Submit 1 question per reading ***in a textfile format*** for consideration in class discussions by Monday evening, 9pm, Feb 11th .

2) **Full Scan Algorithm:** Given your newfound knowledge of computing scans (parallel prefixes) on n-element 1D arrays using p tasks in $O(n/p + \log p)$ time, derive a full scan algorithm for a 2D n x n array distributed in a block-block manner to p x p tasks arranged in a virtual 2D grid. Recall from lecture that a full scan on a 2D array wraps around the array dimensions as shown here:

1 1 1		1 2 3
1 1 1	should produce:	4 5 6
1 1 1		7 8 9

   Your solution should run in $O(n^2/p^2 + n/p * \log p)$ time. Hint: The solution involves applying the 1D scan algorithm that you're familiar with multiple times. Describe your approach and give a brief argument for why it works.

   [Goal: get some firsthand experience with the scan algorithm]

3) **Atomic Operations:** Having heard about these "cool new atomic operations" supported by C/C++/Chapel, your boss would like you to convert the:
   head = (head + 1) % buffSize
   idiom in your company's proprietary bounded buffer implementation to store 'head' as an atomic int (and ditto for 'tail'). You don't have the heart to tell him/her that there is no atomic "increment-and-mod" operator available in these languages, so decide to implement one yourself using the existing atomic operations.

   Your approach must be correct (implement the intended operation) and atomic w.r.t. other threads' reads/writes of 'head', though it is acceptable for it to be susceptible to livelock. How do you implement the operation? If you'd like to think in terms of a prototype, the Chapel procedure might be something like:

   proc incrementAndMod(**ref** head: **atomic int**, numItems **int**);

But please note that there's no requirement to code up this problem unless you want to – it's intended as a thought exercise rather than a coding exercise.

[*Goal:* experience thinking about using existing atomic operations to implement new ones]

4) **Nine-Point Stencil:** Given the provided code framework in C (stencil9.c), write a parallel 9-point stencil computation. Your implementation should compute the value of point $Y_{i,j}$ as a weighted average of its 9 nearest neighbors in X (i.e., elements i +/- 1 and j +/-1 of X). As weights, use 0.25 as the central element, 0.125 for the adjacent neighbors and 0.0625 for the diagonal neighbors. In the framework, the input array is initialized with four nonzero entries for you. Treat the boundary values as being fixed at 0.0. Complete the implementation via the following steps:

   a) *Complete the Sequential Implementation*
      i) Write the loop that computes Y's elements as the weighted average of the corresponding neighbors in X. Note that the 0^{th} and $(n+1)^{st}$ rows/cols in each array are designed to serve as boundary values to guard against out-of-bounds indexing.
      ii) Compute 'delta' – the absolute value of the largest change in any single element between X and Y for the current application of the stencil.
      iii) Copy Y back into X for the next iteration.

   b) *Parallelize the Implementation Using OpenMP:* Use appropriate OpenMP pragmas and features to parallelize the computation. Submit this code.

   c) *Report on your findings:* Write a brief summary of the performance improvement you are able to achieve with OpenMP over the serial solution. Note that you should probably run with a larger problem size and smaller epsilon than the framework ships with, and to disable the printing of the array as the problem size grows.

   [*Goal:* gain experience with OpenMP and stencil computations]

5) **Mandelbrot Set Computation:** Given the provided code frameworks, in C+OpenMP (mandelbrot.c) ***OR*** Chapel (mandelbrot.chpl and MPlot.chpl), create a program that computes the Mandelbrot set in parallel.

   a) *Complete the Sequential Implementation:* All the hard code to compute the Mandelbrot set is provided for you. You simply need to create an array that stores the number of time steps required for each pixel to converge using the supplied coordToNumSteps() routine -- it takes a *(row, column)* coordinate as input and returns the number of steps required to converge as an integer (this will in turn be used by the plot routine to choose a shade of red to draw). Upon successful completion, mandelbrot.ppm should contain the Mandelbrot set image.

b) Convert the Sequential Implementation into a Parallel Implementation: Using OpenMP directives (for the C version) ***OR*** Chapel data parallel concepts (for Chapel), parallelize your program. Submit this code.

c) *Try a Dynamic Schedule:* Because different pixels take a different amount of time to converge, the Mandelbrot set computation can be difficult to load balance as the number of threads/tasks grows. Using either the dynamic schedule features available in OpenMP ***OR*** Chapel or the dynamic work distribution algorithm you wrote yourself in a previous assignment, see whether you can find an improved load balance as compared to a simple static blocking. Submit this code and report on your block vs. dynamic performance.

[*Goal:* Additional data parallelism experience; application of dynamic work scheduling concepts]