CSEP 524: Assignment #1

(due prior to class, Tuesday January 15th)

0) **Survey:** Fill out class survey at: https://catalyst.uw.edu/webq/survey/bradc/189097

1) **Taking Stock:** Determine what parallel computing resources you already have available to you, as follows. Turn in a paragraph or two summarizing what you find.

    a. *Everyone:* Figure out what processor type(s) you have available on your personal desktop/laptop computer, how many cores/sockets it has, and what its coarse block diagram would look like (similar to the "Some Hardware Terminology" slide from lecture). List the parallel programming models that you are aware you have available on this system (without having to install anything additional).

    b. *PMP students:* Determine whether there are parallel computing resources available for your use at your company—clusters, cloud computing resources, cycle servers, etc. Describe the hardware resources and list the programming models that are readily available on it. Feel free to work with coworkers to gather this information collectively when appropriate.

2) **Reading:** Read the following selections and submit 1-3 questions for potential discussion in class to the staff mailing alias by 9pm Monday, January 14th).

    a. **The Free Lunch is Over: A Fundamental Turn Toward Concurrency in Software,** Herb Sutter, originally published in Dr. Dobb's Journal, 30(3), March 2005 (http://www.gotw.ca/publications/concurrency-ddj.htm).

    b. **Lin & Snyder:**
        - Preface and Ch 1 (pp. 1-29)
        - Initial section of Ch 11 (pp. 305-311)

    c. **A Brief Overview of Chapel,** Chamberlain, pp. 1-10 (stop at *Task Parallel Features*) (http://chapel.cray.com/papers/BriefOverviewChapel.pdf).

3) **SW Installation:** Download and install the official course software on your system. You are welcome to work from your native OS as well/instead, but the course staff only commits to supporting this setup:

a. If you don't already have the UW CSE Fedora 17 home VM image installed on your computer, get it. See the *Software* section of the course web for the link.

b. Configure the number of virtual cores in your VM. To do this, power off the VM, go to VM Settings -> Hardware tab -> processors -> change 'Number of processor cores'. It is recommended that you not set this higher than the number of physical cores on your host computer.

c. Download the pre-built Chapel 1.6.1.1 tarball from the *Software* section of the course website and unpack it within your VM's home directory as described on the course web.

4) **Work distribution:** Implement the following utility functions in C and Chapel to compute block and cyclic distributions of a set of items amongst a set of tasks. Functions like this are commonplace in parallel computing.

a. For C, assume we'll distribute a 0-based set of items (0..numItems-1):

```
void computeMyBlockPart(int numItems,     // # items to distribute
                        int numTasks,     // # of tasks
                        int myTaskID,     // my ID: 0..numTasks-1
                        int* myLo,        // my low bound
                        int* myHi) {…}    // my high bound

void computeMyCyclicPart(…same arguments as above…) {…}
```

b. For Chapel, assume we want to distribute a non-strided range and to return the result as a range (non-strided for block, stridable for cyclic):

```
proc computeMyBlockPart(items: range,     // items to distribute
                        numTasks: int,    // # of tasks
                        myTask: int       // my ID: 0..numTasks-1
                        ): range {…}      // my subset of items

proc computeMyCyclicPart(…same arguments as above…
                        ): range(stridable=true) {…}
```

5) **Embarrassingly Parallel Performance Study:** Write embarrassingly parallel programs in C+Pthreads *and* Chapel as described below. Each program should be parameterized by *(i)* a problem size and *(ii)* a number of tasks. Each program should compute a simple operation on every element of a 1D input array of integers.

The rough pattern for the code should be as follows:

startTimer
createTasks
computeOperation
joinTasks
stopTimer

Explore the following space of approaches within your code:

Operations: negate the array element **vs.** compute its factorial
Input Decks: fill array w/ random values **vs.** a *value ramp* (see below)
Distributions: block **vs.** cyclic work distributions
Numbers of tasks: explore the space from 1 to #cores (or more?)

Before collecting experimental results, predict which configurations will result in the best performance. Submit the following experimental write-up:

a. Write up a short description (1-3 paragraphs) of your predictions for which configurations will work best/worst, and your rationale for them.

b. Report your best speedup for each configuration, computed vs. the 1-task version; make sure to note the # of tasks and problem size for which you measured it.

c. In cases where your observed results didn't match your predictions, postulate why the results differed.

d. Did you achieve perfect speedup in any cases? If not, hypothesize why not.

Notes:

- When doing performance studies, try to minimize the number of other processes running on your system in order to avoid competition for the processors and other resources.

- A simple array "value ramp" is to initialize the array elements such that A[i] = i. For a factorial computation on a large array, this approach will likely take a ridiculous amount of time, however. So our recommendation would be to parameterize the program by the maximum value to put into the ramp and interpolate between the values. E.g., a 7-element array with a maximum value of 3 might store the ramp: 0, 0, 1, 1, 1, 2, 2 (you could even re-use your block distribution to compute which indices store which value). This same value could also be used to bound the random number generator's values.

- The timing and random number generator portions of this exercise are not intended to be part of the challenge, so if they don't fall out trivially for you, speak up. If someone wishes to contribute their approach to the class, that would be fine.

- Once you have your Chapel program working correctly, remember to compile it with ––fast before doing performance runs (to turn off runtime checks).

- Try to evaluate configurations in which the 1-task execution time is on the order of a few seconds. This should be long enough to be meaningful w.r.t. your timer's granularity while being short enough to make gathering results go quickly.

- One way to change the execution time is to vary the problem size. Many HPC benchmarks are performed using a problem size that consumes ¼ - ½ of the available physical memory on a system, but you're welcome to use anything that is substantially bigger than your cache.

- If your program is running faster than you'd like, another way to inflate the execution time is to wrap a loop around all the code inside your timing routines to execute a number of "trials" (which could be another argument to the program).

- If comparing Chapel vs. Pthreads performance, be sure that you are using the same size integers (most C compilers default to 32-bits and Chapel defaults to 64-bits).

- At the time of this writing, we're still gaining experience with getting performance results on the VM ourselves. We'll share notes as we learn more.