Chapel Hands-on Exercise: The Mandelbrot Set

The Mandelbrot Set is the set of points c in the complex plane for which the sequence defined by the iteration $z_{n+1} = z_n^2 + c$ remains bounded. It can be proven that points outside of $|c| = 2.0$ will cause z_n to escape to infinity, so we can focus on the interior of that circle.

A given c is likely to be *in* the set if the magnitude of z_n remains less than 2.0 for a reasonably large number of iterations N. We can stop iterating early if z_n appears to diverge before n reaches N.

Pretty pictures you have seen of the Mandelbrot Set assign each pixel (representing a given c) a color corresponding to the number of iterations required to exceed the bound. Those with the color corresponding to N are likely to be in the set; those with other colors are probably not.

To compute each pixel in a Mandelbrot set rendering, use the following two steps:

1. Convert the pixel's location to a value c in the complex plane. The traditional Mandelbrot set image lies within the space whose corners are defined by $(-1.5, -1.0i)$ and $(0.5, 1.0i)$; so this conversion is simply a linear mapping from integer column and row coordinates to a complex number.

   We have provided a function `mapImg2CPlane` that takes in two arguments: an x value (colunmn) and a y value (row) and returns the complex value c to be used in the next step.

2. Compute the iterative process described above to see whether or setting c to that particular value causes the sequence z_n to diverge, and in how many steps. As pseudocode, this algorithm is:

```
// compute whether or not c is in the mandelbrot set
// input: c, a complex number
// output: an image value indicating the number of iterations

start with complex value z equal to zero
do N steps (in practice a value like 50 produces a nice image)
  replace z with z-squared plus c
  if the absolute value of z exceeds 2.0 then
    store the current number of steps in the image value
    stop iterating
store zero in the image value if we never exceed 2.0
```

To get experience with Chapel, try the following steps:

1. **Try the Basic Framework.** The file `mandelbrot.chpl` provides framework code which simply fills the image array with values corresponding to the sum of the row and column coordinates, scaled to match the image's color depth. The image itself is written out as BMP file using a helper module named MPlot. You can control the file format, image type, and color depth using configuration constants to your program if it uses the MPlot module. BMP files can be viewed by most image viewers or web browsers.

   Compile `mandelbrot.chpl` and execute it to create an image you can view. Have a look at the resulting image file. It should be filled with a diagonal gradient. Browse the sources or run with the --help option to see available configuration options.

2. **A Serial Variant.** Modify the program to fill the image with pixel values using the algorithm described above to draw the Mandelbrot set.

3. **A Data-Parallel Version.** Parallelize your serial Chapel program using data parallel features such as forall loops or promoted functions/operators.

4. **A Multi-Locale Data-Parallel Version.** Extend your data parallel version to run across multiple locales using distributions (domain maps that target multiple locales). See $CHPL_HOME/doc/README.multilocale for a guide to getting started with multiple locales if using your own Chapel installation; if using a pre-installed version provided for the hands-on session, refer to instructor-provided directions on using it. For a brief introduction to distributions beyond what was covered in lecture, refer to $CHPL_HOME/examples/primers/distributions.chpl.

5. **A Task-Parallel Version.** Explicitly parallelize the original serial Chapel program using multiple tasks. Compare this code to your data-parallel implementation in terms of effort to write, read, and maintain.

6. **A Multi-Locale Task-Parallel Version (for the particularly brave).** Use Chapel's locales concepts to extend your task-parallel implementation to execute using multiple locales.

Explorations

- **Explore load balancing.** When decomposed across a large number of tasks/processors, the Mandelbrot set can result in load imbalance because contiguous regions of the plane will typically require a similar number of iterations, some very large, others very small. What load-balancing techniques can you use with your various versions to avoid having some tasks/locales complete long before others? You can either write your own techniques or look at some standard advanced iterators on ranges provided in the `AdvancedIters` module (documented in the *Chapel Standard Library Documentation* - see `http://chapel.cray.com/docs/latest/modules/standard/AdvancedIters.html`)

- **Explore different regions** near the boundary of the Mandelbrot set: Change the boundaries of the subset of the complex plane where the computation is carried out. If you have coded the mapping between image coordinates and coordinates in the complex plane as a separate procedure or using constants, this should be easy.

- **Alternate color maps.** In the color images, the solutions use only the red plane. How would you apply a pseudocolor map to the output?

- **Expanded dynamic range.** Most of the values which are not in the set escape to infinity very rapidly, so the iteration counts and resulting color values are all close to zero. At low resolutions, it might yield a more interesting picture if one displayed the log of the iteration count. Can this be done efficiently within the computation routine?

- **Explore parallel I/O.** By default, applying write[ln]() to an array will result in its elements being written out serially. Parallelizing the I/O may make the entire program execution time more scalable (depending on your OS/file system). See the *IO* and/or the *Regexp* standard module documentation for more information on Chapel's I/O capabilities: `http://chapel.cray.com/docs/latest/modules/standard/Regexp.html` and `http://chapel.cray.com/docs/latest/modules/standard/IO.html`.