Chapel Tutorial Exercise: The Mandelbrot Set

The Mandelbrot Set is the set of points c in the complex plane for which the orbit of zero under iteration of the complex quadratic polynomial $z_{n+1} = z_n^2 + c$ 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 a pixel in a Mandelbrot set rendering, there are two main steps:

1. Convert the pixel's location to a value c in the complex plane. The typical 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 interpolation from a pixel's column and row coordinates within the image bounds to a real and imaginary value (respectively) within these complex bounds.

2. Compute the iterative process described above to see whether or not z diverges, and in how many steps. As pseudocode, this algorithm is:

```
start with complex value z equal to zero
do N steps
  replace z with z-squared plus c
  if the absolute value of z exceeds 2.0 then
    store the number of steps in the pixel value
    stop iterating
store zero in the pixel value if we never exceeded 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—15 by default (4 bits). The image itself is written out as PPM/PGM/PBM files using a helper module named MPlot. Compile and execute this program, and view 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.** Using the concepts presented in the *Base Language* lecture, 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.** Using the concepts presented in the *Data Parallelism* lecture, parallelize your serial Chapel program using forall loops or promoted functions/operators.

4. **A Task-Parallel Version.** Using the concepts presented in the *Task Parallelism* lecture, 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.

5. **A Multi-Locale Task-Parallel Version.** Using the concepts presented in the *Locales* lecture, extend the task-parallel version from the previous step to execute using multiple locales.

6. **A Multi-Locale Data-Parallel Version.** Using the concepts presented in the *Domain Maps* lecture, extend the data parallel version from step 3 to run using multiple locales. Compare the differences between this code and your single-locale data-parallel code with the differences between your single-locale and multi-locale task-parallel codes.

7. **Examine performance.** Measure the computational speedup achieved by your parallel versions. Check how performance varies with the number of tasks per locale and/or the number of locales. (Tip: see the other side of this document for notes on performance timings).

Notes on Performance Timings

- When doing performance runs of Chapel, always compile with the `--fast` flag. It turns off a number of runtime safety checks and turns on optimizations for the generated C code (see the **chpl** man page for details)
- For timing-related calls, use the Chapel Time module (`use Time;`). The simplest way to do a short-lived timing is to use the `getCurrentTime()` routine which by default returns the number of seconds that have passed since midnight as a `real(64)`. For example:

```
const startTime = getCurrentTime();
timeThis();
const stopTime = getCurrentTime();
writeln("Elapsed time was: ", stopTime - startTime);
```

- For further documentation on timing routines, refer to the Chapel language specification's description of the `Time` module in the *Optional Modules* section of the *Standard Modules* chapter.
- When using data parallelism, recall that the `dataParTasksPerLocale` configuration variable can be used to vary the number of tasks used per forall loop.

Variations

- **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?
- **Explore different regions** near the boundary of the Mandelbrot set: Change the boundaries of the region in 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 QIO file writing improvements.** If you are using the QIO preview release, explore modifying the image writing routines to use parallel I/O and/or binary I/O (see the Wikipedia article on PPM file formats to learn about the binary variation).