CSC726 Parallel Algorithms Fall 2016

Project 1: Getting started with OpenMP

The Mandelbrot Set

Consider the iteration in the complex plane:

$$\begin{aligned} z_0 &= 0 \\ z_{i+1} &= z_i^2 + c \end{aligned} \tag{1}$$

If $|c| \geq 2$, then the sequence defined in equation (1) diverges. Further, if we find $|z_j| \geq 2$ for some j (and some c), then we also know that the sequence diverges.

The smallest k such that $|z_k| \geq 2$ is known as the **escape time**. For different values of c, the escape time varies. If we compute the escape time over a region in the complex plane, and assign different colors according to the value of k mod m for some small m, e.g., $m = 7$, we obtain the following representation of the Mandelbrot computation in the complex plane.

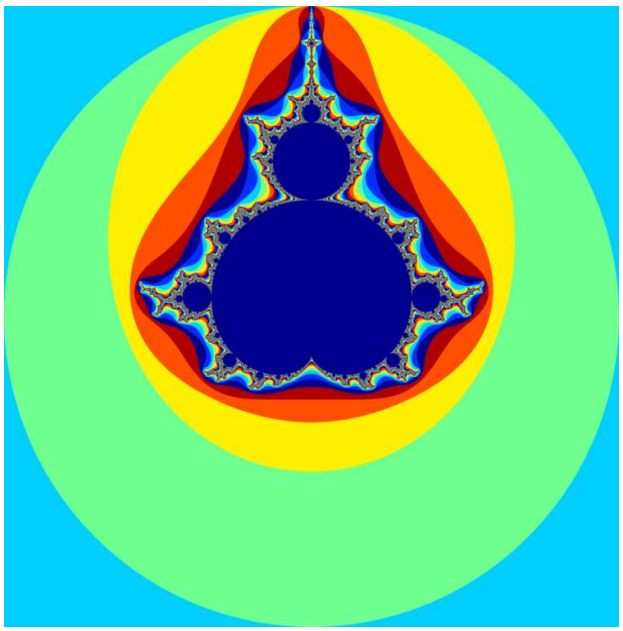

Figure 1:

Region of Interest and Resolution

The region of interest for the Mandelbrot fractal is the complex numbers $c = a + bi$ where $-2 \leq a \leq 2$ and $-2 \leq b \leq 2$. The resolution for our project is 4001×4001. Writing a sequential program to produce the image in Figure 1 is very easy. Start by allocating an matrix M size 4001×4001 with integer entries. Generate equally spaced points (a, b) in the set $[-2, 2] \times [-2, 2]$ with a spacing of 0.001. Form the complex number $c = a + bi$, and evaluate the escape time for that value of c. Store your result in the appropriate position in matrix M.

Some points in the complex plane never escape. We can say $k = \infty$. But, that poses a problem for a computer program that is looping until the escape time is discovered. A practical solution is to place an upper limit, e.g., $L = 1000$ on the number of iterations. If escape has not been achieved within L iterations, assign the "escape time" of zero for that value of c. This uniquely identifies those points which do not escape within L iterations and improves visualization of the escape times (see Figure (1)).

After computing all entries in the matrix M, write it to a binary file named **mandelbrot.dat**. The data format for our binary file assumes 4-byte integers[1]. The file format is as follows:

- The first 4 bytes in the file is the number of rows in the matrix, i.e., 4001.
- The next 4 bytes in the file is the number of columns in the matrix, i.e., also 4001.
- The next 64032004 bytes in the file are the 16008001 entries in the matrix in row major order.

To visualize your result, we will use **octave** to convert your binary file to a PNG image. An **octave** script to do the conversion is found on gottlieb in `/usr/local/octave_examples/make_png.m` To run the code, copy it to the directory containing your **mandelbrot.dat** file and type:

```
octave < make_png.m
```

The script will read a file named "mandelbrot.dat" and create a file named "mandelbrot.png". Use the `display` command (on gottlieb) to visualize your **PNG** image. The image is larger than the resolution of most laptop screens. Whenever the image is larger than the screen resolution, the `display` program provides a small pan icon on screen. Use your mouse (or track point, touchpad, etc) and left button to move the rectangle within the pan icon ; the `display` program will show the region of the image indicated by the rectangle. An example pan icon is shown below.

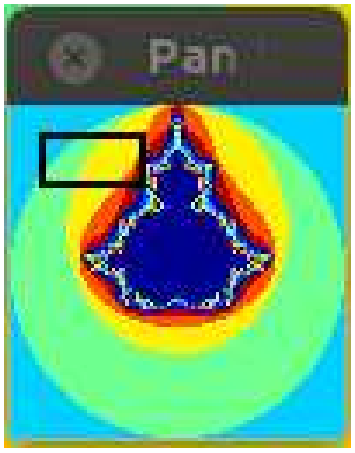

Figure 2:

[1]This assumption is true on gottlieb for C/C++ data type **int**

Parallel Computing – Part I

The Mandelbrot computation is "embarrassingly parallel" because computing the escape time is independent for each value of c. You can convert your sequential code to a parallel code by adding just a few pragmas. Use `#pragma omp for` to parallelize the outer loop of your sequential code.

However, load balancing is an issue. In this exercise, We will explore four alternate scheduling options, their effect on load balancing and consequently their effect on speedup. Run your parallel code with $1, 2, 3, ..., 32$ threads and plot a speedup curve for each run. An example using **octave** to plot a speedup curve is found on gottlieb in `/usr/local/octave_examples/speedup.m`.

Generate a speedup curve for each of the following four schedule types:

- static schedule with `chunksize` $= N/p$, with array size $N \times N$ and p threads.
- static schedule with `chunksize` $= 1$
- dynamic schedule with `chunksize` $= 10$
- guided schedule with `chunksize` $= 10$

Which scheduling type produces the best speedup curve ?

Parallel Computing – Part II

An interesting load balancing alternative is to use a technique called "work stealing". Instead of having the workload managed by a master thread with a single work queue, each thread has its own work queue. When a thread runs out of work to do, it takes work off of the queue of another thread. How the other thread is chosen is a topic for creative thinking, but however you choose to do it, all threads should be examined in the search for work. Examining the threads in some order is similar to choosing a probe sequence for open addressing hashing.

One possible way: The threads are numbered $\{0, 1, 2, ..., (p-1)\}$. We choose an increment r which is relatively prime to p. Then, for thread t we define the sequence:

$$t_i \;=\; (\, t \,+\, jr \,) \text{ modulo } p \tag{2}$$

for $0 \leq j < p$. A familiar theorem from introductory abstract algebra guarantees that the sequence t_i will contain the numbers $\{0, 1, 2, ..., (p-1)\}$ in some order provided r is relatively prime to p. Notice that different threads could possibly use different values of r.

Start the computation with consecutive rows assigned to each thread. Distribute the number of rows as evenly as possible (see example handout for computing the start row and the number of rows). Devise and implement a "work stealing" scheme by which work can be taken from another thread whenever a thread runs out of work. Consider carefully the number of rows you would like to choose to define a "chunk" of work stolen from another process.

You must also devise a logical way for the threads to detect that there is no more work to perform.

Run your code with $1, 2, 3, ..., 32$ threads and plot a speedup curve. Compare this speedup curve to the other scheduling methods.

Discussion: Each thread should have its own queue of work and that queue should be accessible by all threads. Taking work off a queue must be done atomically to ensure an orderly distribution of work. Ideally, each queue should have its own lock variable, so that different queues can be accessed in parallel. This would avoid contention on a single lock variable. Unfortunately, the ideal scheme can not be implemented in OpenMP using `#pragma` statements alone. You have two options:

Option 1 Use Openmp critical section for all queue updates. I.e., use `#pragma omp critical`

Option 2 Use OpenMP locks to implement atomic updates with a unique lock for each queue.

Part III – the Math Part

Prove the claim that the sequence defined by equation (2) actually contains all of the thread numbers in some order. *Hint: Use the fact that the prime factorization of any integer is unique.*