Assignment #1

Will Tekulve

September 11, 2019

1 Shared code

[ ]:
```
import os
import time
import numpy as np
import multiprocessing
from pylab import imshow, show
from timeit import default_timer as timer
```

[ ]:
```
def mandel(x, y, max_iters):
    """
    Given the real and imaginary parts of a complex number,
    determine if it is a candidate for membership in the Mandelbrot
    set given a fixed number of iterations.
    """
    c = complex(x, y)
    z = 0.0j
    for i in range(max_iters):
        z = z*z + c
        if (z.real*z.real + z.imag*z.imag) >= 4:
            return i

    return max_iters
```

2 Sequential code

[ ]:
```
def create_fractal(min_x, max_x, min_y, max_y, image, iters):
    height = image.shape[0]
    width = image.shape[1]

    pixel_size_x = (max_x - min_x) / width
    pixel_size_y = (max_y - min_y) / height

    for x in range(width):
```

```
        real = min_x + x * pixel_size_x
        for y in range(height):
            imag = min_y + y * pixel_size_y
            color = mandel(real, imag, iters)
            image[y, x] = color
```

[ ]:
```
def run_sequential(height, width):
    image = np.zeros((height, width), dtype = np.uint8)
    start = timer()
    create_fractal(-2.0, 1.0, -1.0, 1.0, image, 20)
    dt = timer() - start

    print(f"Mandelbrot created in {dt} s")
    imshow(image)
    show()
```

3 Parallel code

[ ]:
```
def create_fractal(min_x, max_x, min_y, max_y, image, width, start_idx,␣
 ↪stop_idx, iters, event):
    event.wait()
    pid = os.getpid()

    print(f"PID {pid} starting with {stop_idx - start_idx} points")

    pixel_size_x = (max_x - min_x) / width
    pixel_size_y = (max_y - min_y) / (len(image) // width)

    for i in range(start_idx, stop_idx):
        x = i % width
        y = i // width

        real = min_x + x * pixel_size_x
        imag = min_y + y * pixel_size_y
        color = mandel(real, imag, iters)
        image[x + y * width] = color

    print(f"PID {pid} complete")
```

[ ]:
```
def run_parallel(height, width, num_cores):
    image = multiprocessing.Array('B', height * width, lock=False)
    pixels_per_core = len(image) // num_cores
    start_event = multiprocessing.Event()
    procs = []
    for i in range(num_cores):
```

```
        start = i * pixels_per_core
        stop = start + pixels_per_core

        p = multiprocessing.Process(target=create_fractal, args=(-2.0, 1.0, -1.
↪0, 1.0, image, width, start, stop, 20, start_event))
        p.start()
        procs.append(p)

    time.sleep(5)  # Give some time for all processes to start

    start = timer()
    start_event.set()
    for p in procs:
        p.join()
    dt = timer() - start

    print(f"Mandelbrot created in {dt} s")

    image = np.reshape(image, (height, width))
    imshow(image)
    show()
```

4 Analysis

The sequential and parallel versions were both run on a Pitzer Jupyter instance with 32 cores and 188GB ram. Each code version was tested with Mandelbrot resolutions of 1024x1536 and 10240x15360. As seen below, there is a marked improvement in run times when comparing sequential to parallel. The parallel code was initally run with 40 processes, since the system monitor displayed 40 available cores. However, when the process count was reduced to 32 (the number of cores indicated by the OSC instance launcher), the calculation time was actually reduced by 0.5 seconds for the high resolution test.

Type	Height	Width	Pixel Count	Worker Count	Run Time	Improvement
Sequential	1024	1536	1,572,864	1	3.837	–
Parallel	1024	1536	1,572,864	32	0.247	1,553%
Sequential	10240	15360	157,286,400	1	368.441	–
Parallel	10240	15360	157,286,400	32	19.457	1,894%

To parallelize this problem, I allocated a 1D array that represented the full resolution image in shared memory and each process was spawned with a reference to this memory. Each process was given a subset of the full array to work on such that no process had any overlapping work with any other. The first time I ran this code I actually saw an almost 2 minute increase in the parallel version over the sequential version. As it turns out, Python's `multiprocessing` module's shared memory `Array` has a rw lock by default to help prevent accidental parallel data accesses.

After disabling that lock, the time to process in parallel dropped from >490 seconds to the final result of approximately 19.5 seconds.