Exercises for Week 5 & 6

---

1. Amdahl's Law

1. **Autolab** Suppose that a specific task consists of two subtasks. The first subtask retrieves data from a file; it takes 20 seconds to execute on a given system, and it cannot be parallelized. The second subtask processes a large number of data records; it takes 100 seconds to execute sequentially. Each data record can be processed independently, so the second subtask is easily parallelized.

   a) What is the parallel fraction of the task consisting of the two subtasks?

   b) What is the theoretical speedup for $p = 10$ processors?

   c) What is the theoretical maximum speedup?

   d) Imagine we currently have 4 processors available for parallization. To improve performance, we have two options:

      i. Optimize the first subtask so we can retrieve the data from the file in 5 seconds instead of 20.

      ii. Purchase more processors so we have 8 processors available instead of 4.

      Which option will provide the shortest runtime?

2. Parallel π

The goal of this exercise is to use parallelization to improve the performance of our code. We will use the Monte Carlo approximation of pi from Advanced Python Programming, Chapter 7. Let's take a look at the three different implementations from Advanced Python Programming:

- Implementation 1: Fully serial

```
import random

samples = 1000000
hits = 0

for i in range(samples):
    x = random.uniform(-1.0, 1.0)
    y = random.uniform(-1.0, 1.0)
    if x**2 + y**2 <= 1:
```

```python
            hits += 1

pi = 4.0 * hits/samples
```

- Implementation 2: Fully parallel

```python
import random
import multiprocessing

def sample():
    x = random.uniform(-1.0, 1.0)
    y = random.uniform(-1.0, 1.0)
    if x**2 + y**2 <= 1:
        return 1
    else:
        return 0

if __name__ == '__main__':
    samples = 1000000
    hits = 0

    n_proc = 10
    pool = multiprocessing.Pool(n_proc)
    results_async = [pool.apply_async(sample) for i in range(
        samples)]
    hits = sum(r.get() for r in results_async)
    pi = 4.0 * hits/samples
```

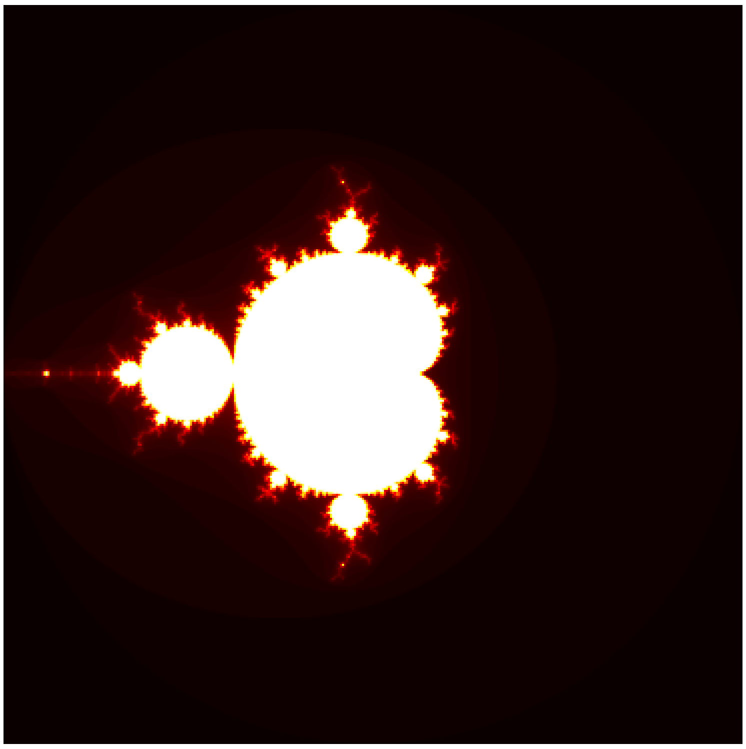
- Implementation 3: Chunked parallel

```python
import random
import multiprocessing

def sample():
    x = random.uniform(-1.0, 1.0)
    y = random.uniform(-1.0, 1.0)
    if x**2 + y**2 <= 1:
        return 1
    else:
        return 0

def sample_multiple(samples_partial):
    return sum(sample() for i in range(samples_partial))

if __name__ == '__main__':
    samples = 1000000
    hits = 0

    n_proc = 10
    chunk_size = samples//n_proc
    pool = multiprocessing.Pool(n_proc)
    results_async = [pool.apply_async(sample_multiple, (chunk_size
        ,))
                     for i in range(n_proc)]
    hits = sum(r.get() for r in results_async)
    pi = 4.0 * hits/samples
```

1. **Autolab** Run the three implementations as a batch job and measure their run time using the `time` command. Remember to select a CPU model for repeatable results and to select multiple cores.
2. **Autolab** Which implementation was fastest? Did some perform slower than expected? Why? Can you relate it to the output of the 'time' command?
3. Run the fastest parallel implementation for `n_proc` varying from 1 to the maximum number of threads on your CPU. Plot the speedup as a function of the number of processes.

   Hint: Use the `lscpu` command to check the number of threads available.
4. Estimate the parallel fraction of the program by fitting the theoretical speedup curve from Amdahl's law to your results. Nothing fancy, just play with the numbers until it looks okay. What is the estimated parallal fraction? Given this, what is your theoretical maximum speedup?
5. (Optional) Rewrite the code using `pool.map()` instead of `pool.apply_async()` to perform the computation of π. What do you observe? What are pros and cons of `pool.map()`?

3. The Mandelbrot Set



The Mandelbrot Set is a famous mathematical set of complex numbers defined by iterating a simple mathematical function. Given a complex number, c, the iteration is defined by the formula:

$$z_{n+1} = z_n^2 + c.$$

The Mandelbrot Set is the set of complex numbers for which the iteration remains bounded as n approaches infinity. Points outside the set exhibit chaotic and complex behavior when iterated.

In this exercise, we'll explore the computation of the Mandelbrot Set using parallel processing. The task is to generate an image of the Mandelbrot Set by calculating the escape time for each point in the complex plane. The escape time is the number of iterations it takes for a point to achieve an absolute value over 2 - if that happens, it's been shown that it must eventually escape to infinity.

Use the following template for your code:

```
import multiprocessing
import numpy as np
import matplotlib.pyplot as plt

def mandelbrot_escape_time(c):
    z = 0
    for i in range(100):
        z = z**2 + c
        if np.abs(z) > 2.0:
            return i
    return 100

def generate_mandelbrot_set(points, num_processes):
    ########################
    # YOUR CODE HERE #
    ########################
    return escape_times

def plot_mandelbrot(escape_times):
    plt.imshow(escape_times, cmap='hot', extent=(-2, 2, -2, 2))
    plt.axis('off')
    plt.savefig('mandelbrot.png', bbox_inches='tight', pad_inches=0)

if __name__ == "__main__":
    width = 800
    height = 800
    xmin, xmax = -2, 2
    ymin, ymax = -2, 2
    num_proc = 4

    # Precompute points
    x_values = np.linspace(xmin, xmax, width)
    y_values = np.linspace(ymin, ymax, height)
    points = np.array([complex(x, y) for x in x_values for y in
        y_values])

    # Compute set
    mandelbrot_set = generate_mandelbrot_set(points, num_proc)

    # Save set as image
    mandelbrot_set = mandelbrot_set.reshape((height, width))
    plot_mandelbrot(mandelbrot_set)
```

1. **Autolab** Implement the function `generate_mandelbrot_set` to compute the escape times for each complex number in parallel. The function must return a NumPy array. Distribute the points equally between all the processors. Hint: adapt the chunked parallel implementation from the previous exercise.

2. **Autolab** As before, run the program as a batch job where you vary the number of processes `num_proc`. Make a speedup plot.

3. **Autolab** Implement a new function `generate_mandelbrot_set_chunks`, that distributes the points in smaller chunks to the workers. Make sure there are more chunks than workers. Hint: adapt the chunk parallel implementation by setting `chunk_size` to a fixed number.

4. Run the new function for varying processes, make a speedup plot and estimate the parallel

fraction with Amdahl's law. What do you see? Why does the chunked version achieve a higher speedup? Hint: does all parts of the Mandelbrot set require the same amount of computation?