3. NOVEMBER 2023

PROBLEM SET 1

Computing at Scale in Machine Learning: Distributed computing and algorithmic approaches
WS_23/24

Submitted by:

Group ABA consisting of:

Kalson, Akshay

Biswal, Anuj

Hakari, Bhavanishankar Ramesh

Problem 2

2a)

In the provided code in the file pi-montecarlo.py, we look at the code step by step and see whether the program is executed sequentially or there's some level of parallelisation in the code, Here's a function-wise report on whether the processing within each function is sequential:

-sample_pi(n):

```python
import argparse # See https://docs.python.org/3/library/argparse.html
import random
from math import pi

def sample_pi(n):
    """ Perform n steps of Monte Carlo simulation for estimating Pi/4.
        Returns the number of sucesses."""
    s = 0
    for i in range(n):
        x = random.random()
        y = random.random()
        if x**2 + y**2 <= 1.0:
            s += 1
    return s
```

Function Purpose: This function performs a Monte Carlo simulation for estimating π/4 by generating random points and counting how many fall within a quarter of the unit circle.
Sequential Processing: The code inside this function is entirely sequential. It generates random points one at a time within a loop and checks whether each point is inside the quarter of the unit circle. This process is inherently sequential.

-compute_pi(args):

```
def compute_pi(args):
    random.seed(1)

    n_total  = args.steps
    s_total = sample_pi(n_total)
    pi_est = (4.0*s_total)/n_total
    print(" Steps\tSuccess\tPi est.\tError")
    print("%6d\t%7d\t%1.5f\t%1.5f" % (n_total, s_total, pi_est, pi-pi_est))

```

Function Purpose: The compute_pi function is responsible for simulating the Monte Carlo simulation and estimating π.
Sequential Processing: This function is sequential. It sets the random seed, obtains the number of steps for the simulation (n_total) from the command-line arguments, and then calls the sample_pi function. The sample_pi function, as mentioned, is inherently sequential. The rest of the code within compute_pi, including the calculation of the estimated value of π and the printing of results, is executed sequentially.

-main(args):

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='Compute Pi using Monte Carlo simulation.')
    parser.add_argument('--steps', '-s',
                        default='1000',
                        type = int,
                        help='Number of steps in the Monte Carlo simulation')
    args = parser.parse_args()
    compute_pi(args)

```

Function Purpose: This function acts as the entry point for the script, parsing command-line arguments and initiating the Monte Carlo simulation.
Sequential Processing: The main function is also sequential in the original code. It sets the random seed, obtains the number of steps for the simulation (n_total) from the command-line arguments, and then calls the compute_pi function, which in

turn calls the sequential sample_pi function. The remaining code in main is also executed sequentially, including any output or result printing.
In summary, the code does not incorporate parallelism. All the key computations and operations within the functions are performed sequentially.

Measuring time for each of the section

Processor used: Google Colab's Intel(R) Xeon(R) CPU @ 2.20GHz, Dual Core

Method: Time measurement in the modified code was achieved using the built-in Python time module. The code recorded timestamps at the beginning and end of specific sections, such as the Monte Carlo simulation and result printing. By subtracting the start time from the end time, it calculated the elapsed time for each section. The measured times were then displayed in seconds. This straightforward approach allowed for accurate tracking of execution times without the need for external libraries or tools.

Output:

```
 Steps  Success Pi est. Error
  1000      778 3.11200 0.02959
Time for Monte Carlo simulation: 0.003999233245849609 seconds
Time for printing results: 0.00013303756713867188 seconds
```

Execution:

```python
import random
from math import pi
import time

def sample_pi(n):
    s = 0
    for i in range(n):
        x = random.random()
        y = random.random()
        if x**2 + y**2 <= 1.0:
            s += 1
    return s

def compute_pi(n_total):
    random.seed(1)

    # Measure the time for the Monte Carlo simulation
    start_time = time.time()
    s_total = sample_pi(n_total)
    end_time = time.time()
    monte_carlo_time = end_time - start_time

    pi_est = (4.0 * s_total) / n_total

    # Measure the time for printing the results
    start_time = time.time()
    print(" Steps\tSuccess\tPi est.\tError")
    print("%6d\t%7d\t%1.5f\t%1.5f" % (n_total, s_total, pi_est, pi - pi_est))
    end_time = time.time()
    print_time = end_time - start_time

    print(f"Time for Monte Carlo simulation: {monte_carlo_time} seconds")
    print(f"Time for printing results: {print_time} seconds")

# Specify the number of steps as an argument
steps = 1000
compute_pi(steps)

```

Compute the amount of serial computation that does not benefit from parallelization

To compute this, in the modified code, we introduced time measurements to assess the execution times of distinct sections. The Monte Carlo simulation time and result printing time were measured by capturing timestamps at the beginning and end of these sections. To compute the proportion of inherently sequential computation, we timed a specific sequential section, 'sample_pi', and compared its duration to the total execution time. This yielded the proportion of the code's execution that is inherently sequential and not conducive to parallelization, expressed as a percentage.

```python
import random
from math import pi
import time

def sample_pi(n):
    s = 0
    for i in range(n):
        x = random.random()
        y = random.random()
        if x**2 + y**2 <= 1.0:
            s += 1
    return s

def compute_pi(n_total):
    random.seed(1)

    # Measure the time for the Monte Carlo simulation
    start_time = time.time()
    s_total = sample_pi(n_total)
    end_time = time.time()
    monte_carlo_time = end_time - start_time

    pi_est = (4.0 * s_total) / n_total

    # Measure the time for printing the results
    start_time = time.time()
    print(" Steps\tSuccess\tPi est.\tError")
    print("%6d\t%7d\t%1.5f\t%1.5f" % (n_total, s_total, pi_est, pi - pi_est))
    end_time = time.time()
    print_time = end_time - start_time

    print(f"Time for Monte Carlo simulation: {monte_carlo_time} seconds")
    print(f"Time for printing results: {print_time} seconds")

    # Calculate the amount of serial computation
    start_time_total = time.time()
    start_time_sequential = time.time()
    sample_pi(n_total)  # Call the function that contains sequential computation
    end_time_sequential = time.time()
    end_time_total = time.time()

    sequential_time = end_time_sequential - start_time_sequential
    proportion_serial = sequential_time / (end_time_total - start_time_total)

    print(f"Proportion of serial computation: {proportion_serial * 100:.2f}%")

# Set the number of steps directly in the notebook
steps = 1000
compute_pi(steps)
```

```
 Steps  Success Pi est. Error
  1000      778 3.11200 0.02959
Time for Monte Carlo simulation: 0.0008859634399414062 seconds
Time for printing results: 0.0013654232025146484 seconds
Proportion of serial computation: 99.94%
```

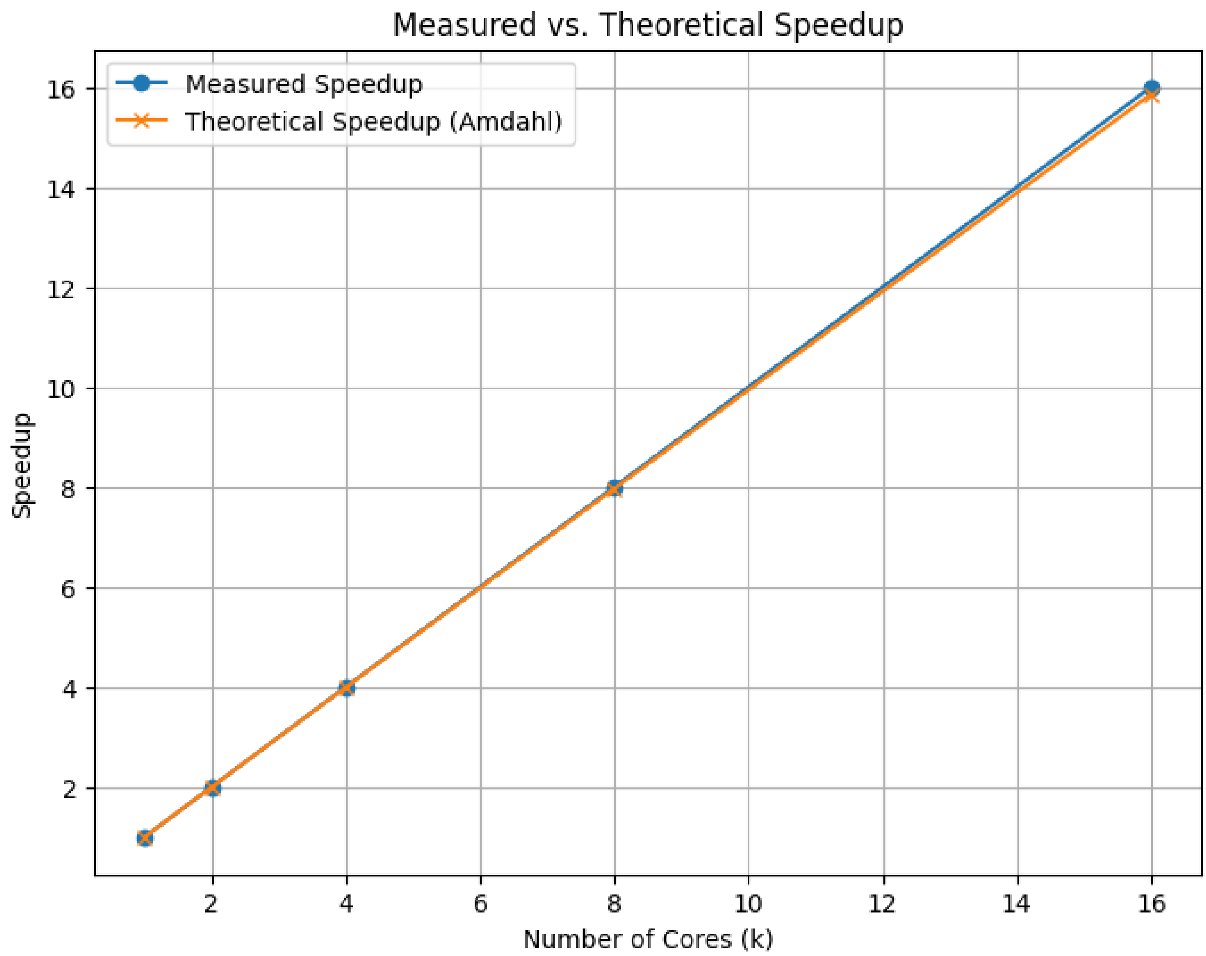
2b)

Now we will display measured and the theoretical speedup for k = 1, 2, 4, . . . cores using Amdahl's law

Speedup = 1 / ((1 - P) + P/k)
Where, Where P is the proportion of computation that is inherently serial. In our case its 99.94%.

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

# Measured execution time for serial execution (1 core)
serial_time = 0.0008859634399414062

# Proportion of the program that is sequential
sequential_proportion = 0.9994  # 99.94%

# Calculate the theoretical speedup using Amdahl's law
def theoretical_speedup(P, k):
    return 1 / ((1 - P) + P / k)

# Number of cores (k)
cores = [1, 2, 4, 8, 16]

# Calculate the measured speedup
measured_speedup = [serial_time / (serial_time / core) for core in cores]

# Calculate the theoretical speedup based on Amdahl's law
theoretical_speedup_values = [theoretical_speedup(sequential_proportion, core) for core in cores]

# Create a plot
plt.figure(figsize=(8, 6))
plt.plot(cores, measured_speedup, marker='o', label='Measured Speedup')
plt.plot(cores, theoretical_speedup_values, marker='x', label='Theoretical Speedup (Amdahl)')
plt.xlabel('Number of Cores (k)')
plt.ylabel('Speedup')
plt.title('Measured vs. Theoretical Speedup')
plt.legend()
plt.grid(True)

# Show the plot
plt.show()
```

In this script, we use the provided measured serial execution time and proportion of sequential computation (99.94%) to calculate the theoretical speedup based on Amdahl's law. The script then generates a plot comparing both the measured and theoretical speedup for various core counts.



As we can observe here, the degree of our code that is inherently sequential is 99.94% , hence parallelisation will not help speed up the execution.4

2.c

This Python code performs a Monte Carlo simulation to estimate the value of pi by randomly sampling points in a unit square and counting how many of them fall inside a unit circle. The estimated pi value is calculated, and the program measures and prints the time taken for the simulation and result printing.

Import necessary modules:

random: This module is used for generating random numbers.
math.pi: It's used to obtain the value of pi (π) from the math library.
time: Used to measure the time taken by different parts of the program.
Define the sample_pi function:

This function takes two arguments: n (the number of samples to take) and seed (the random number generator seed).
It initializes a random number generator with the given seed.
It samples n random points (x, y) within a unit square (0 <= x, y < 1).
It counts the number of points that fall inside the unit circle (x^2 + y^2 <= 1).
The function returns the count of points inside the circle.
Define the compute_pi function:

This function allows the user to input the number of steps (samples) in the Monte Carlo simulation and the random number generator seed.
It measures the time it takes to run the simulation and to print the results.
Inside the function:

It calculates the estimated value of pi using the formula: pi_est = (4.0 * s_total) / n_total, where s_total is the count of points inside the circle, and n_total is the total number of samples.
It prints a table with columns: Steps, Success (count of points inside the circle), Pi est. (the estimated value of pi), and Error (the absolute difference between the estimated pi and the math library's pi value).
It also prints the time taken for the Monte Carlo simulation and the time taken for printing the results.
The program checks if it's being run as the main script using if __name__ == "__main__":. If it is, it calls the compute_pi function.

2d)

When setting explicit seeds for a multiprocessing solution, it's essential to ensure that each process uses a different seed to achieve independent random number generation. This prevents any potential interference or correlation between the random numbers generated by different processes.