DAT470/DIT065
Computational techniques for large-scale data

Assignment 1
Deadline: 2024-04-22 23:59

Problem 1: Blackjack (14 pts)

Blackjack is a very popular casino game. The game is played with the 52-card French-suited deck of playing cards, with potentially multiple decks. In the game, the *player* plays against the *dealer*, trying to get as close to the value of 21 in their hand, without going over. A concise set of rules is as follows:

1. Face cards (Jack, Queen, King) are worth 10 points.
2. The Ace is worth 1 or 11 points.
3. All other cards are worth their number.
4. The value of the hand is the sum of the values of the cards; if the hand contains an ace, the first ace is worth 11, unless this would make the total exceed 21, in which case it is worth 1.
5. The player is dealt two cards.
6. The dealer is dealt one card face up (the *dealer's card*), and one card face down (the *hole card*).
7. If the initial hand is worth exactly 21 (that is, a 10 + Ace), it is said to be a *Blackjack*.
    - If the dealer might have a Blackjack (the dealer's card is a 10 or an Ace), they peek at their hole card.
    - If the player has a Blackjack, they win immediately 3:2 (that is, if the player wins 1.5 times their bet).
    - If the dealer has a Blackjack, the player loses immediately.
    - If both the dealer and the player have a Blackjack, it's a tie and the player's bet is returned.
8. In other cases, the player may *hit* (ask for more cards) until their total is 21 or more, or they may *stand* (take no more cards).
9. If the player's hand exceeds 21, they *bust* and lose immediately.
10. If the player does not bust, the dealer plays a deterministic strategy: they take cards until their hand is worth 17 or more.
11. If the dealer's hand exceeds 21, they bust and the player wins immediately 1:1.
12. Otherwise, the winner is determined by the values of the hands:

- If the player's hand has greater value than that of the dealer's, they win 1:1.
- If the dealer's hand has greater value than that of the player's, the player loses.
- If the values are equal, it's a *push* and the player's bet is returned.

The rules here omit some more advanced rules, such as *doubling down*, *splitting*, *surrender*, and *insurance*. As with all casino games, the house has an *edge*: even if the player plays a perfect strategy, the house will always win in the long run. However, Blackjack is known for the fact that the house edge is very small in perfect play.

The file `blackjack.py` contains a very simple simulator that can be used to simulate playing Blackjack using a given strategy. The simulation plays a given number of deals, and estimates the expected amount of wins and loses. The simulation works with the rules laid out here, assuming one deck, and omits doubling and splitting. Furthermore, the simulation is rather inefficient, as it very accurately aims to replicate the actual play. As such, it is a useful *baseline*: we can very well understand what is going on in the simulation, so a more efficient simulation should behave in the same way. Your task will be to improve, parallelize, and evaluate the scalability of the simulation.

Note: a simulation like this that performs random sampling to obtain numerical results is called a *Monte Carlo simulation*. Such methodology is ubiquitous in many areas of science, including physics, engineering, biology, and economics, as they can be used to simulate phenomena whose mathematics can be intractable and cannot be solved in a closed form.

(a) Make the simulation reproducible by implementing the `--seed` parameter. Every time the same seed is used, the same results should be produced. (2 pt)

(b) Look at the function `simulate`. Describe in your report, which parts of the function need to be executed serially, and which could be parallelized. (2 pt)

(c) Using sufficiently large (but constant-sized) input, measure the fraction of time spent in the sequential and parallelizable part of the program. Report the size of input, the fraction and absolute time spent in sequential part of the program, the fraction and absolute time spent in the parallelizable part, and the total running time. (2 pt)

(d) Using Amdahl's law, what is the upper bound on speedup that can be achieved? (1 pt)

(e) Use the `multiprocessing` module to parallelize the `simulate` function, that is, implement the `-w` command-line parameter. (4 pt)

(f) Evaluate your parallel implementation on the cluster empirically. Produce a plot that contains the following:

- The observed speedup as a function of the number of CPU cores,
- The theoretical speedup as predicted by Amdahl's law as a function of the number of CPU cores,

- The upper bound on the speedup as predicted by Amdahl's law.

(3 pt)

Problem 2: UFO sightings (10 pts)

The file `/data/2024-DAT470-DIT065/ufo.csv` contains information about UFO sightings between the years 1949 and 2013[1]. The data has been cleaned and preprocessed somewhat and contains the following fields:

- datetime: date and time of the sighting,
- location: location of the sighting,
- latitude: latitude of the sighting (in degrees),
- longitude: longitude of the sighting (in degrees).

The accompanying file `ufo.py` contains a Python script that reads data from the file and reads *queries* from *standard input*. Each line of standard input is assumed to contain a *latitude longitude* pair, separated by whitespace, given in degrees. The program performs a *linear scan* of the dataset and determines the closest UFO sighting to each query, in terms of Haversine distance. Haversine distance is also known as *great circle distance*: it is the distance d that one has to travel on the surface of a sphere with radius r along the shortest path from latitude φ_1 and longitude λ_1 to latitude φ_2 and longitude λ_2, given by

$$d = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos\varphi_1 \cos\varphi_2 \sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right).$$

While the Earth is not exactly spherical, this is a very good approximation under most circumstances. As output, the program produces the location of the closest sighting for each query. Your task will be to parallelize the program.

(a) Change the queues into multiprocessing queues, and adjust the structure of the program such that you parallelize the program by creating a desired number of processes. (4 pts)

(b) Furthermore, change the way the dataset is handled: if you pass the NumPy array as an argument to the process, it is copied. We want to avoid this, so use a multiprocessing array to *share memory* between the processes; you *must not* make a copy for the processes, but the processes must access the original data. (2 pts)

(c) Like in Problem 1, determine the fraction of sequential and parallelizable code in the main function. Report the fractions and the maximal speedup as predicted by Amdahl's law in your report. (2 pts)

(d) Evaluate the speedup of the program empirically on the cluster, and produce a similar plot that shows the observed speedup, the speedup predicted by Amdahl's law, and the theoretical upper bound for the speedup. (2 pts)

[1] Data comes from `https://www.kaggle.com/datasets/NUFORC/ufo-sightings`

Hints

- Take care in how to set the seeds in Problem 1: it is incorrect to produce the same (or substantially the same) values in all processes.
- Look up the documentation of the `Random` class.
- Using a process pool is probably wise in Problem 1.
- In Problem 2, you will want to create `Process` objects.
- You will need to first allocate shared memory and then create a shallow view into that memory as a NumPy array; the array should point to the pre-existing memory and not allocate its own memory.
- Make sure you understand how to handle the queues: you will need to signal the end of data somehow because trying to read from a queue will result in it blocking infinitely.
- Also, it might be wise to start reading from the result queue ASAP and not wait for all the processes to finish.
- Processes need to be handled properly: they must be created, started, and ended gracefully, or otherwise you may get weird errors.
- The files `queries.txt`, `queries_large.txt`, and `queries_verylarge.txt` contain test queries; the corresponding correct output is given in `locations.txt`, `locations_large.txt`, and `locations_verylarge.txt`. Make sure you get the correct output.

Returning your assignment

Return your assignment on Canvas. Your submission should consist of a report that answers all questions as PDF file (preferably typeset in LaTeX) called `assignment2.pdf`. In addition, you should provide the code you used in Problems 1 and 2 as `assignment2_problem1.py` and `assignment2_problem2.py`, respectively. The code must match the interfaces of `blackjack.py` and `ufo.py`; the command line parameters must not be changed, and output must be correct. Do *not* deviate from the requested filenames and do *not* produce the plots in these files; these files will be used for evaluating the quality of your implementations automatically.