DAT470/DIT066
Computational techniques for large-scale data

Assignment 2

Deadline: 2025-04-27 23:59

Problem 1 (4 pts)

A data processing pipeline consists of two main stages:

- Data loading and preprocessing, accounting for 30% of the execution time, and cannot be parallelized.
- Computation-heavy analysis, accounting for 70% of the execution time, which is fully parallelizable.

Answer the following questions using *Amdahl's law*.

(a) Compute the expected speedup when using $n = 2, 4, 8, 16, 32$ processors. (1 pt)

(b) What is the maximum theoretical speedup if an infinite number of processors were available? (1 pt)

(c) What kind of speedup would we expect to see in a real world setting, in comparison to the values computed above? Why? (2 pts)

Problem 2 (20 pts)

You are given the skeleton file `assignment2_problem2_skeleton.py`. The file contains a program that iterates through all `.txt` files in a directory tree, and computes the number of occurrences of *words*. Here, by word we mean a string of consecutive non-whitespace characters. Whitespaces are ignored and work as separators for words. We will try to parallelize the program using different strategies with the Python multiprocessing module.

There are five datasets, consisting of subsets of Project Gutenberg[1], located on Minerva in the directory `/data/courses/2025_dat470_dit066/gutenberg/`. For verifying correctness, the checksums for the subsets are

- `tiny`: 1885973
- `small`: 11520967
- `medium`: 152806641
- `big`: 1520444169
- `huge`: 14880058115

[1] `https://www.gutenberg.org/`

(a) Implement the functions `compute_checksum` and `get_top10`. The function `compute_checksum` shall compute a checksum for the word count. Suppose there are n words w_i and denote the associated counts by c_i (that is, the word w_i occurs c_i times). Denote the length of the word by $|w_i|$. Then, we define the checksum to be

$$\sum_{i=1}^{n} |w_i| \cdot c_i \,,$$

that is, the checksum is the sum of lengths and counts of words.

The function `get_top10` shall return a list of word-count pairs of the 10 most common words, in descending order, as a list. (2 pts)

(b) List the 10 most common words in the `huge` dataset, together with their counts. (1 pt)

(c) Explain which are the major parts of the program (at the level of the main function). Which blocks does the program consist of? Which of those can be parallelized easily using a multiprocessing Pool, which cannot? (2 pt)

(d) Measure the running time of the different blocks. Determine the parallelizable fraction f of the running time. Report f for the `huge` dataset, the total running time, and also the fraction of time spent in each of the blocks identified. Using Amdahl's law, what is the upper bound on speedup that we could hope to achieve? (2 pts)

(e) Using `multiprocessing.Pool`, parallelize the loop that counts the words. Run your code against the `huge` dataset using $1, 2, 4, \ldots, 64$ workers. Plot the speedup as function of the workers. Include in your plot the maximum speedup from previous subproblem as a dashed horizontal line. Also report the total absolute running time with 64 cores. (4 pts)

(f) It is reasonable to believe that reading files takes a considerable amount of time, so let's bake this into parallelization. Modify your code in such a way that, instead of reading files before counting words, the function `count_words_in_file` takes a filename as input, reads the content of the file within the function, but otherwise works the same. Then, plot the speedup using the `huge` dataset as in the previous subproblem, and report the total absolute running time with 64 cores. (3 pts)

(g) Finally, we will perform a more advanced parallelization attempt using two kinds of worker processes and three queues. Use `assignment2_problem2_-skeleton2.py` as your starting point.

We shall have $w + 2$ processes:

- w workers that read filenames from a queue, read the file, count the words, and then feed the word count dictionaries into a `wordcount_queue`; when no more filenames are available, signals end of input by putting a `None` into the queue
- One *merger* process that reads word count dictionaries from the `word-count_queue`, merges them into a global word count dictionary; when no more input is available, computes the checksum and top 10, puts them into the `out_queue`

- The main process feeds filenames into the `filename_queue`, concludes by signalling end of input with appropriate `None` sentinel values, and then reads the checksum and top10 from the `out_queue`.

Furthermore, since the I/O between processes easily forms a bottleneck, use the parameter `batch_size` to specify how many files the workers should process before they copy their intermediate word count dictionary into `wordcount_queue`.

Figure 1 displays the intended data flow. The idea here is to hide latency of the different potential bottlenecks: Merging the dictionaries takes time, so we try to do this while the workers are busy reading files and counting words. Finally, we want to only move the results we care about back to the main process, as moving large dictionaries is likely very expensive.

As before, plot the speedup as a function of the number of workers at $w = 1, 2, 4, \ldots, 64$, and report the running time on the `huge` set with 64 workers. Choose the batch size yourself, and remember to mention it in your report (and how you selected the value). (6 pts)

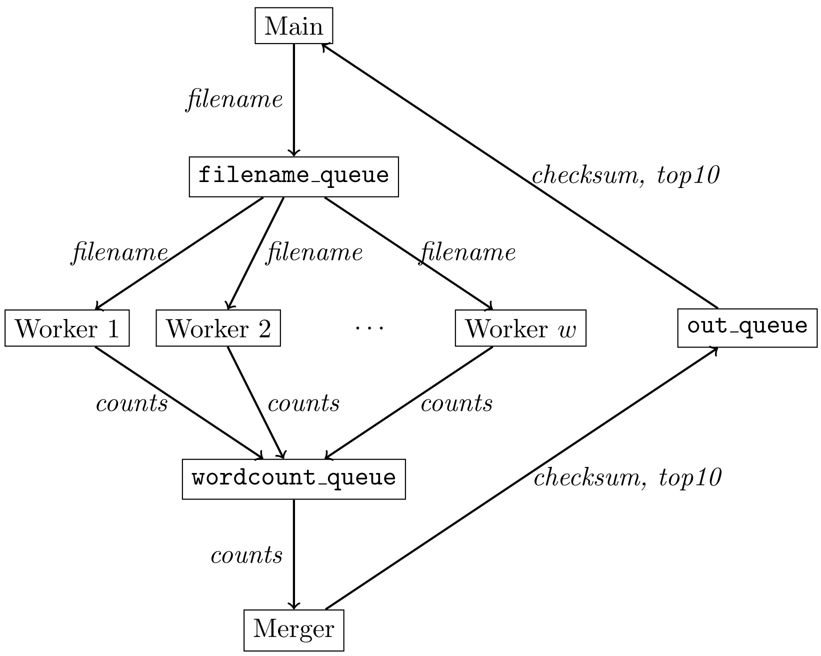

Figure 1: The parallelization strategy for Problem 2g.

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 Problem 2 as

- `assignment2_problem2a.py`,
- `assignment2_problem2d.py`,
- `assignment2_problem2e.py`,
- `assignment2_problem2f.py`, and
- `assignment2_problem2g.py`.

Do *not* deviate from the requested filenames.