DAT470/DIT065 Assignment 2

Xinyan Liu
liuxinya@chalmers.se

Leah Wanja Ndirangu
leahw@chalmers.se

2025-04-27

Problem 1

(a) Expected Speedup with Different Numbers of Processors

The expected speedup $S(n)$ when using n processors is given by:

$$S(n) = \frac{1}{(1-P) + \frac{P}{n}}$$

where $P = 0.7$.

- For $n = 2$:

$$S(2) = \frac{1}{0.3 + \frac{0.7}{2}} = \frac{1}{0.3 + 0.35} = \frac{1}{0.65} \approx 1.54$$

- For $n = 4$:

$$S(4) = \frac{1}{0.3 + \frac{0.7}{4}} = \frac{1}{0.3 + 0.175} = \frac{1}{0.475} \approx 2.11$$

- For $n = 8$:

$$S(8) = \frac{1}{0.3 + \frac{0.7}{8}} = \frac{1}{0.3 + 0.0875} = \frac{1}{0.3875} \approx 2.58$$

- For $n = 16$:

$$S(16) = \frac{1}{0.3 + \frac{0.7}{16}} = \frac{1}{0.3 + 0.04375} = \frac{1}{0.34375} \approx 2.91$$

- For $n = 32$:

$$S(32) = \frac{1}{0.3 + \frac{0.7}{32}} = \frac{1}{0.3 + 0.021875} = \frac{1}{0.321875} \approx 3.11$$

(b) Maximum Theoretical Speedup

The maximum theoretical speedup $S(\infty)$ is:

$$S(\infty) = \frac{1}{(1-P)} = \frac{1}{0.3} \approx 3.33$$

(c) Real World Speedup Expectations

In a real-world setting, the speedup may be less than the theoretical values due to factors such as:

- Overheads in managing parallel processes.
- Communication delays between processors.
- Imperfections in parallelization, such as data dependencies.

These factors can reduce the efficiency of parallel execution, leading to lower actual speedup compared to the theoretical maximum.

Problem 2

(a) Implement the functions compute checksum and get top10

The following picture 1 and picture 2 show our implementation for these two functions:

```python
def compute_checksum(counts):
    """
    Computes the checksum for the counts as follows:
    The checksum is the sum of products of the length of the word and its count

    Parameters:
    - counts, dictionary : word to count dictionary

    Return value:
    The checksum (int)
    """
    return sum(len(word) * word_count for word,word_count in counts.items())
```

Figure 1: Implementation for checksum

```python
def get_top10(counts):
    """
    Determines the 10 words with the most occurrences.
    Ties can be solved arbitrarily.

    Parameters:
    - counts, dictionary : a mapping from words (str) to counts (int)

    Return value:
    A list of (count,word) pairs (int,str)
    """
    return sorted(((word_count,word) for word,word_count in counts.items()),reverse=True)[:10]
```

Figure 2: Implementation for get top 10

Here is our.sh file to display the world count for these different datasets.

```
#! /bin/bash

#SBATCH --cpus-per-task=2
#SBATCH --mem=4G
#SBATCH -t 0:30:00

source /data/users/leahw/lab2/.venv/bin/activate

echo output from assignment2_problem2_skeleton.py
echo this is tiny
python3 assignment2_problem2a.py /data/courses/2025_dat470_dit066/gutenberg/tiny
echo this is small
python3 assignment2_problem2a.py /data/courses/2025_dat470_dit066/gutenberg/small
echo this is medium
python3 assignment2_problem2a.py /data/courses/2025_dat470_dit066/gutenberg/medium
echo this is big
python3 assignment2_problem2a.py /data/courses/2025_dat470_dit066/gutenberg/big
echo this is huge
python3 assignment2_problem2a.py /data/courses/2025_dat470_dit066/gutenberg/huge
```

Figure 3: Code for problem 1

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

We can use the get top 10 method to extract the 10 most common words in the huge dataset

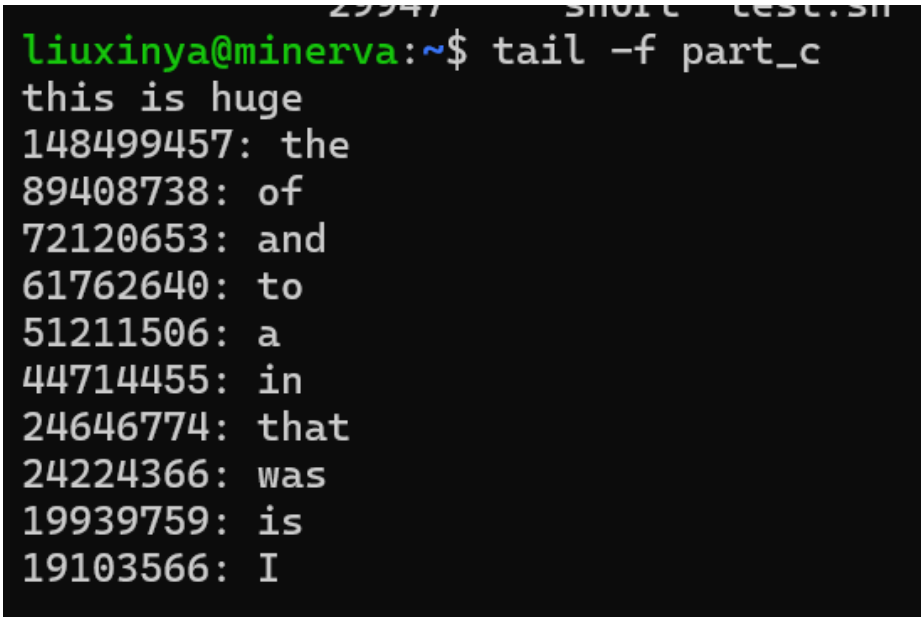

Figure 4: 10 most common words in the huge dataset

(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

This program consists of the following major parts and the following blocks

1. **Read data:**
    - *Purpose:* Iterate through all files in a directory tree, such as iterate all the file in huge dataset.
    - *Parallelization:* Generally cannot be parallelized easily due to I/O operations and directory traversal, which are typically sequential.
2. **Word Counting:**
    - *Purpose:* Count occurrences of words in each file.
    - *Parallelization:* Can be parallelized using a multiprocessing Pool. Each file's word counting can be done independently and then sum the result from all processors, allowing multiple processes to handle different files simultaneously.
3. **Checksum Calculation:**
    - *Purpose:* Compute a checksum for the word count.
    - *Parallelization:* Can also be parallelized since the calculation for each word is independent.

The following is a summary on which of those can be parallelized easily using a multiprocessing Pool, which cannot

- **Parallelizable Blocks:** Word counting across files is the most parallelizable part using a multiprocessing Pool.
- **Non-Parallelizable Blocks:** Data reading is typically not parallelizable due to their sequential nature and dependency on aggregated data.

(d) Measure the running time of the different blocks. Determine the parallelizable fraction f of the running time.

The figure 4 shows our results..

Table 1: Runtime Analysis and Parallelization Information

Block	**Time (seconds)**	**Fraction of Total Time**
Read Data	45.8063	0.0304
Count Words	1255.6444	0.8334
Merge Word Count	205.2176	0.1362
Total Execution Time	1506.6683	1.0000
Sequential Time	969.2270	0.6433
Parallel Time	537.4413	0.3567

- **Parallelizable Fraction** f: 0.3567
- **Upper Bound Speedup using Amdahl's Law**:

$$S = \frac{1}{(1-f) + \frac{f}{N}}$$

  where N is the number of processors.

- Fraction of Parallelizable Part: 0.3567
- Upper Bound Speedup: 1.5545 seconds

(e) Using multiprocessing.Pool, parallelize the loop that counts the words. Run your code against the huge dataset using 1, 2, 4, ,, 64 workers

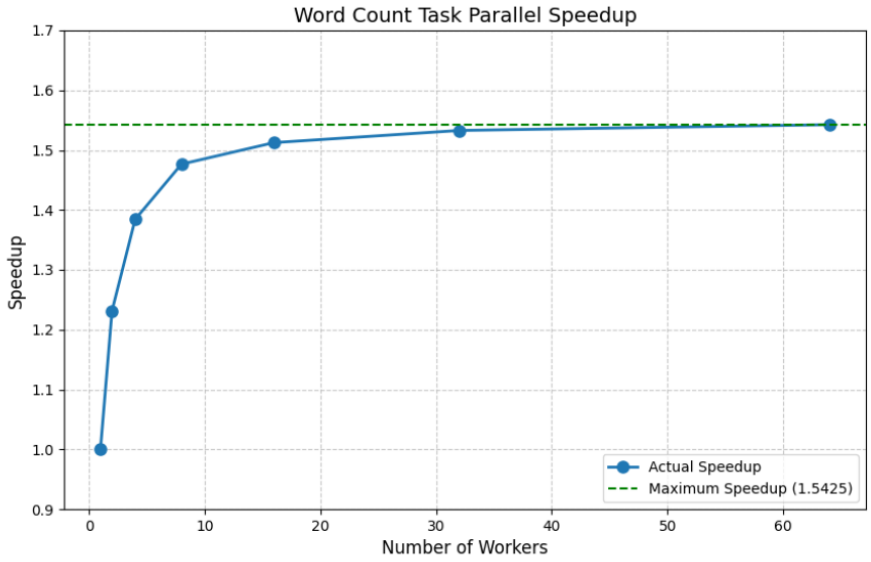

Figure 5: Running Results

The following is the total absolute running time with 64 cores is 1071s

(f) 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.

The figure 6 shows the speed up

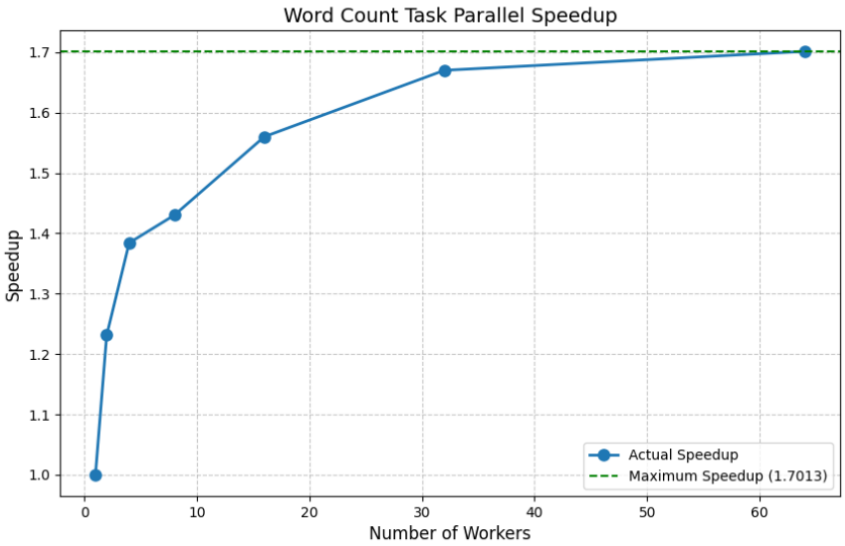

Figure 6: Running Results

The following is the total absolute running time with 64 cores: 881.6878

(g) Perform a more advanced parallelization attempt using two kinds of worker processes and three queues

Our batch size is 1000
The figure 7 shows the speed up

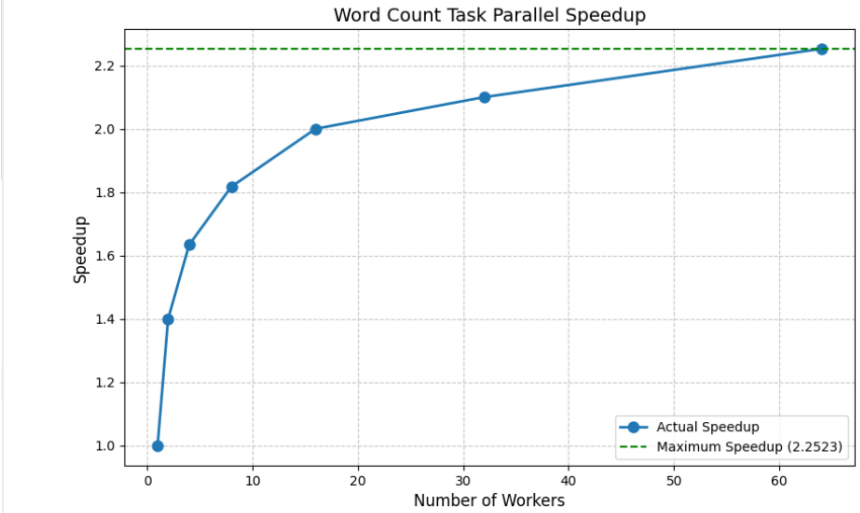

Figure 7: Running Results

The following is the total absolute running time with 64 cores: 673.2345