Comparison between sequential and concurrent programming on n-gram counting with asyncio

Francesco Todino
E-mail address
francesco.todino@edu.unifi.it

Abstract

The aim of this paper is to compare the performance of sequential and parallel n-gram counting algorithms using the asyncio concurrent library.

1. Introduction

In this paper, we will analyze the performance obtained by sequential and parallel implementations of finding bigrams and trigrams inside a text.

Our main goal is to understand the capabilities of the asyncio library by comparing it with sequential algorithms. We will cover different tests, including benchmarking the execution time, measuring speedup and efficiency, and examining scalability across different input sizes.

1.1. N-gram counting

To perform these tests we must first take a look at the theory behind n-grams. An n-gram is a sequence of n adjacent symbols in particular order. The symbols may be n adjacent letters (including punctuation marks and blanks), syllables, or rarely whole words found in a language dataset. In this case, we will consider only bigrams (n=2) and trigrams (n=3). N-grams are so important in the literature for their many purposes:

- **Natural Language Processing (NLP):** n-grams are widely used in NLP for language recognition, spelling correction, text prediction, etc.
- **Language Models:** They are used to build probabilistic models of language, where the probability of a word depends on previous words.
- **Text Analysis:** They are useful in text analysis to identify patterns of word usage, for sentiment analysis, and for recognizing recurring themes.
- **Computer Security:** In computer security, n-grams can be used to detect unusual patterns in network traffic or to identify malware based on code.

1.2. Asynchronous I/O

Asyncio, or Asynchronous I/O, is a Python library used for writing concurrent code using the async/await syntax. Introduced in Python 3.4, it provides a framework for dealing with asynchronous operations, allowing for the efficient handling of I/O and other high-level network protocols. Asyncio is particularly useful in applications that require high performance and scalability, such as web servers, database queries, and network connections.

It works by enabling the code to be paused and resumed at certain points, which are typically I/O operations. This approach allows for multiple tasks to be executed in a non-blocking manner, making the most out of the system's resources.

1.3. Setup

All tests were developed in Python and performed on a MacBook Pro (mid 2015) with an Intel Core i7-4770HQ with 4 cores and 8 threads. The main text was Moby Dick, repeated several times based on the number of megabytes of text needed to run the tests.

2. Methods

In this section we will see all the code involved into this comparison. The goal was to find all the occurences of bigrams and trigrams inside a text. To do this, we will follow these steps:

- We pick a text from a selected file. The one we selected for these tests is Moby Dick, but this is not an important parameter for our results;
- We iterate the whole text, letter by letter, counting every bigram/trigram found in a dictionary;

- We stop the iteration at the end of the text, focusing only on the time elapsed between the beginning and the end.

If the text is particularly large, this task can be very challenging, especially if the size exceeds 300/500 MB.

To perform these analysis, we obtain text from a file with the function in Listing 1 and repeat it several times to check performance with increasing text length.

```
def textFromFile(filename):
  with open(os.path.join(
      os.path.dirname(__file__),
      'DATA/'+filename), 'r',
      encoding='utf-8') as file:
    text = file.read()
    text = text.replace('\n', '')
    text = text.replace(' ', '')
    text = text.lower()
  return text
```

Listing 1. Collect text from a file

2.1. Sequential implementation

As mentioned before, we iterate over the entire text by counting the occurrences of the n-grams within the dictionary *ngrams*. The function used for the sequential version is the one showed in Listing 2.

```
def get_ngrams(text, n):
  ngrams = {}
  for i in range(len(text) - n + 1):
    ngram = text[i:i+n]
    if ngram not in ngrams:
      ngrams[ngram] = 0
    ngrams[ngram] += 1
  return ngrams
```

Listing 2. Sequential implementation

2.2. Parallel implementation

Here we can see the parallel implementation. This version uses a revised version of the *get_ngrams* function, to select the range of elements to be counted:

```
def get_ngrams_interval(text, n, i_from,
    i_to):
  ngrams = {}
  for i in range(i_from, i_to - n + 1):
    ngram = text[i:i+n]
    if ngram not in ngrams:
      ngrams[ngram] = 0
    ngrams[ngram] += 1
  return ngrams
```

Listing 3. Parallel implementation

Regarding asyncio, we define an async function with inputs the text, the n of n-grams and the number of threads used.

We split the text in equal parts, giving each part to each thread involved. In order to divide the items correctly in case the division between threads has some rest, we decided to give the last thread its items plus the rest.

Then, a pool of processes is created, each process is assigned to its own range of elements to be enumerated, executed and, at the end, waited for to have produced their results.

In the end, we combine all the results into a single dictionary, which is used as the output of the function.

```
async def get_ngrams_parallel(text, n,
    nThreads):
  with ProcessPoolExecutor() as
      process_pool:
    loop: AbstractEventLoop =
        asyncio.get_event_loop()
    chunks = [0] +
        [int(len(text)/nThreads) * i for i
        in range(1, nThreads)] +
        [len(text)]
    calls: List[partial[int]] =
        [partial(get_ngrams_interval,
        text, n, chunks[i], chunks[i+1])
        for i in range(nThreads)]
    call_coros = []
    for call in calls:
      call_coros.append(
      loop.run_in_executor(process_pool,
          call)
      )
    results = await
        asyncio.gather(*call_coros)

    n_grams = {}
    for result in results:
      for key in result:
        if key not in n_grams:
          n_grams[key] = 0
        n_grams[key] += result[key]

    return n_grams
```

Listing 4. Parallel implementation

3. Results

To evaluate the performance of the parallel solution, we repeated the tests by doubling the text size each time. The results are shown in Figure 1 and Figure 2 (the second one is used to compare smaller sizes more simply).

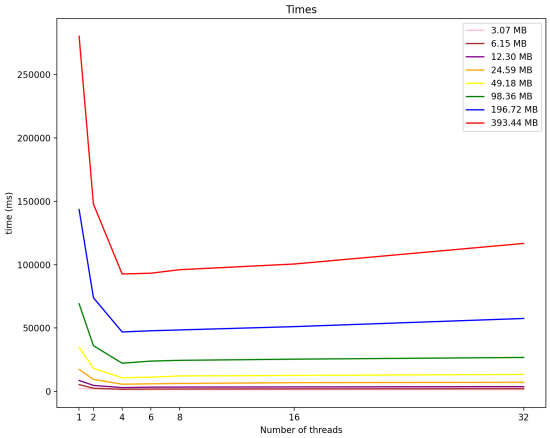

Figure 1. Times increasing threads and dimension.

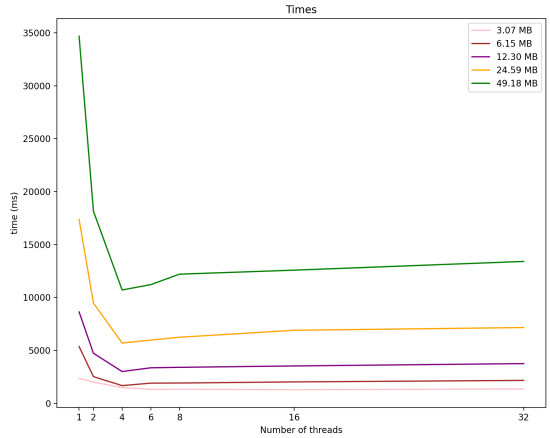

Figure 2. Times increasing threads and dimension detailed.

As can be seen, all the time lines follow the same path, going down in time to 4 threads and then increasing slightly as the number of threads increases. This graph may mean that asyncio performs best with the number of physical cores available (4 in this case) and not with the number of virtual processor cores (8).

Then we can see related speedups shown in Figure 3.

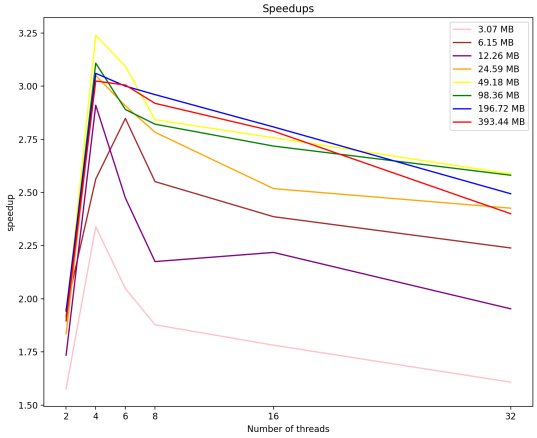

Figure 3. Speedups increasing threads and dimension.

Going by the times, the speedups achieved are promising, having achieved speedup up to 3.239x on 4 threads with 49.18 MB.

As can be seen, by increasing the text size, the parallel version increases its performance, peaking with more data than less.

4. Conclusion

In conclusion, the comparison between sequential and parallel programming in the context of the N-grams counting using asyncio confirms that the best approach is always the parallel one, having obtained increasingly better performance in all text dimensions.