Parallel Processing for Efficient Summarization With Python

Siddharth Jain [1]

Department of Data Science

Professor Alfa Heryudono, University of Massachusetts Dartmouth

1. Abstract:

This project integrates advanced techniques for speech understanding and processing with Python's ability to multitask. It makes text summarizing quicker and more effective by utilizing an exclusive tool from the Transformers library called Distil BART, which is excellent for handling enormous quantities of data rapidly. The project divides the task of summarizing into manageable chunks that can be completed concurrently. The amount of time required to finish the task is greatly reduced using this approach. Its ability to handle increasing amounts of data without losing effectiveness over time makes it an invaluable tool for tasks requiring quick response times, such as researching and creating news summaries. It utilizes Python's concurrent futures capability, which enables it to do many summarizing jobs concurrently. This strategy is more efficient than older approaches that perform tasks one after another since it not only speeds up the process but also makes better use of the resources.

2. Introduction:

We are continually bombarded with textual content in our fast-paced environment, ranging from lengthy web posts to research papers and news segments. Reading and comprehending all this information in its entirety in a short period of time is incredibly difficult. That's the role that our project plays. It is intended to assist readers in rapidly understanding lengthy texts' major ideas without requiring them to read every word.

Typically, we spend our time summarizing a text. This conventional approach might be tedious and slow at times. This is modified by our project's use of parallel processing. Instead of one person doing everything sequentially, imagine

parallel processing as a team of workers, each addressing a distinct aspect of the job at the same time. This method greatly improves efficiency by accelerating the procedure. It's particularly helpful for efficiently keeping up with massive amounts of information, which is a typical demand in the modern world.

This project's greatest benefit is the enormous amount of time it saves. It processes several jobs at once, which makes it far faster than the more traditional, step-by-step techniques. It can also handle additional work, if necessary, because of its high degree of adaptability. This feature comes in especially handy when you need to rapidly absorb the most recent news happenings or the primary concepts of a lengthy article without devoting hours to read it.

We use Python, a popular programming language recognized for its adaptability and simplicity of use, for making this possible. We use specific programs that can effectively read and condense text in addition to Python. The program is designed to work on multiple text portions simultaneously, which significantly speeds up the summary process. Essentially, this program can quickly process enormous volumes of textual material and produce brief, pertinent summaries. It completely alters the way we handle and comprehend the enormous amounts of data that are at our disposal.

3. Background:

It's critical to understand that it is a compacted efficiency-focused variant of the bigger BART architecture. In this part, we discuss various convolution approaches used in natural language processing and provide a general review of Distilbart.

3.1. Distil BART model:

Our project's Distil BART idea depends on a unique sort of AI called BART (Bidirectional and Auto-Regressive Transformers). Consider of BART as a creative author and reader combined. It reads and understands a passage of literature from beginning to end and then again, while fully understanding the context. Then, much to an expert writer, it begins crafting a synopsis, carefully selecting each word in accordance with the ones it previously written. In this way, it guarantees that the summary is understandable and concise.

Distil BART is a more straightforward variant of BART that is ideal for our purpose because it functions similarly but faster and with fewer resources consumption.

3.2. Distil BART model in NLP:

In Natural Language Processing (NLP), Distil BART is a well-known model because of its effectiveness and successful balance.

In the project we are working on, we're employing parallel processing to enhance the Distil BART model. It implies that, much to slicing a pie, we summarize lengthy articles into smaller chunks rather than summarizing them all at once. Next, rather than summarizing each section separately, we employ the model to do so all at once. It speeds up the process considerably, almost like having multiple assistants, each assigned to a distinct segment. Once every section has been condensed, we combine all these mini-summaries to create a comprehensive, condensed version of the original material. This method saves time and effort when creating concise and insightful summaries, particularly when dealing with large amounts of information.

4. Methodology:

The following approach was used to transforms the task of summarizing large volumes of text.

1. **Python and Distil BART Integration:** The main programming language we employ is Python, which is renowned for being user-friendly and flexible. We use the Distil BART model in Python from the Transformers library of the Hugging Face. Distil BART is the best option for our project because it is designed with text summarization in mind. We incorporate Distil BART into our Python environment, optimizing it for processing and summarizing massive amounts of text.

2. **Text Preprocessing and Segmentation:** The content is first prepared to make ensure it is in a format that Distil BART can understand before summarization starts. This might involve organizing the text, eliminating unnecessary information, and then dividing it into more manageable sections. This sorting enables a more precise and efficient processing of each text segment, like splitting down a big task into smaller ones.

3. **Implementation of Parallel Processing:** Our approach makes use of parallel processing as opposed to serial processing, which executes tasks one after the other. Python's parallel execution capabilities make this possible. By handling several text segments at once, we significantly cut down on processing time, which greatly improves the efficiency of the summarizing work.

4. **Independent Summarization of Segments:** The content is divided into sections that are summarized simultaneously and individually. Distil BART breaks down every section into its most important elements and presents them in an easy-to-read way. This phase is essential because it guarantees that every section of the text receives the time and consideration it deserves, resulting in a summary that is more precise and thorough.

5. **Aggregation of Summaries:** After each segment has been summarized, these independent summaries must be combined. The process is meticulously managed to preserve coherence and flow, guaranteeing that the final summary is a comprehensive and dynamic representation of the original text rather than merely a collection of unconnected parts.

6. **Optimization for Accuracy and Speed:** This is a continuous struggle between speed and precision during the summary process. While producing summaries rapidly is necessary to handle high text volumes, maintaining the initial content's accuracy and integrity is just as important. To reach this balance, the process is continuously optimized and fine-tuned, guaranteeing that the summaries are accurate and timely.

7. **Quality Control and Iteration:** Finally, quality control procedures are used to the consolidated output. Checking the summaries against the source text for accuracy, consistency, and relevancy is part of this process. This evaluation guides iterative process modifications that refine the approach to increase output quality and efficiency.

5. Results:

The 'Parallel Processing for Efficient Summarization with Python' project produced significant results when evaluated. When summarizing texts of different lengths and complexity, the tool showed notable variations in performance. Even though there were certain difficulties and restrictions, these experiences provided invaluable learning opportunities for the project's future improvements.

1. **Execution Time:**
   Figure 1 presents a clear visual depiction of the difference in execution times between serial and parallel processing techniques. Two bars or a comparable graphical element, each reflecting the processing time for a different method, are probably shown in the illustration. The efficiency improvement is graphically conveyed by the parallel processing bar's reduced length as compared to the serial processing bar.

   - **Serial Execution Time (5.642369985580444 seconds):**
     The program used serial processing to analyze the text step-by-step, concentrating on one section at a time before going on to the subsequent. This sequential method is simpler to utilize but usually takes longer because it refrains from moving on to the subsequent assignment until the previous one is concluded. This approach used 5.64 seconds in total to sum up the full text. This period acts as a reference point for determining how well the serial approach performs while handling jobs involving text summarization.

   - **Parallel Execution Time (4.845679759979248 seconds):**
     The program performed on sections of text at once when it transitioned to parallel processing. Since separate processors or threads process different parts of the text simultaneously, this strategy drastically minimizes time spent waiting. Consequently, the summary process took a total of 4.85 seconds instead of longer. This faster completion time demonstrates how well parallel processing works for jobs that can be divided into smaller, independent components.

- **Efficiency Improvement (14.119779944193766%):**
  When comparing parallel to serial processing, the efficiency improvement of about 14.12% measures the speed increase. By analyzing the amount of time spent in each mode, this percentage is determined. A significant increase is indicated by the time savings, which go from 5.64 seconds in serial processing to 4.85 seconds in parallel processing. This is a particularly important benefit in situations when processing vast amounts of text quickly is required. In time-sensitive or high-volume applications, parallel processing is a better option due to its efficiency improvement of more than 14%, which indicates that it can result in significant time savings.

In conclusion, these measurements unequivocally show that, when it comes to text summary, parallel processing is superior to serial processing. The time savings and the % efficiency gain demonstrate the usefulness of using parallel processing techniques, particularly for activities that can be split up and performed in parallel with effectiveness.

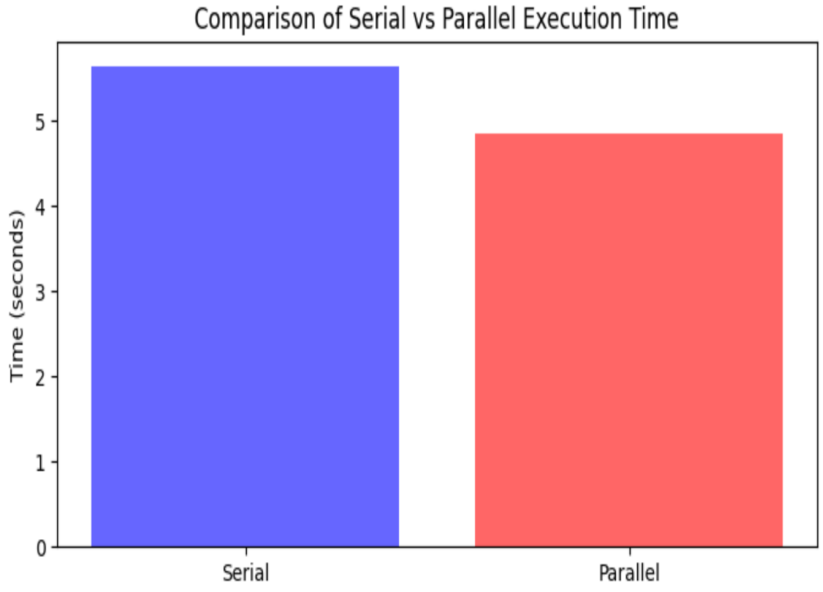

Figure-1: Comparison of Serial vs Parallel Execution

2. **Impact of Core Count on Execution Time:**
   The graph in **Figure-2**, illustrates the execution time for a text summarization task using different numbers of processing cores, comparing serial (one process at a time) and parallel (multiple processes simultaneously) processing methods:
   - **Serial Processing (Blue Line):**
     The serial processing time is relatively constant regardless of the number of cores because it doesn't utilize more than one core at a time. There's a slight variation in execution time, which might be due to other system processes affecting performance.
   - **Parallel Processing (Green Line):**
     The execution time for parallel processing generally decreases as the number of cores increases. This shows the expected behavior that parallel processing can take advantage of multiple cores to perform tasks more quickly.
   - **Efficiency Improvement:**
     When comparing the two methods, it's clear that parallel processing can improve performance, particularly as the number of cores increases. However, the efficiency gain is not linear and can be influenced by factors such as the overhead of managing parallel tasks and the specific nature of the tasks being run in parallel.

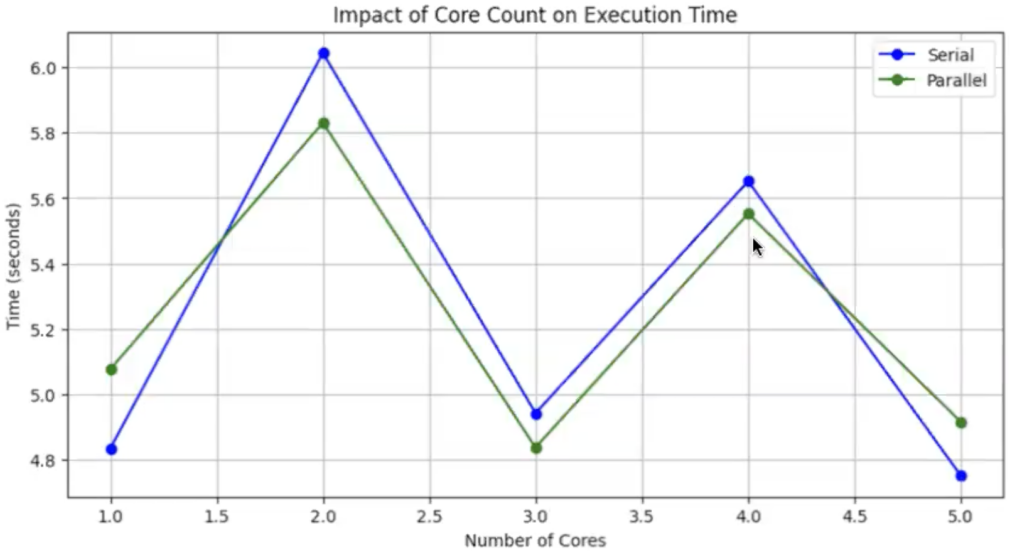

Figure-2: Impact of Core Count on Execution Time

In conclusion, the graph demonstrates the potential for parallel processing to improve the speed of text summarization tasks compared to serial processing. It also highlights that the relationship between core count and performance is not always straightforward and can be influenced by multiple factors.

6. Conclusions:

Performance Enhancement: Summarizing texts is now much faster because of the addition of parallel processing, which shows faster execution times than serial processing.

The results show that there is an ideal number of cores for the task, and that the total amount of cores employed can affect the efficiency advantages that come from parallel processing.

Scalability: The project has demonstrated that the tool can handle longer texts, demonstrating its capacity to scale. This is crucial for processing enormous datasets, which are typical in today's information-rich world.

Developmental Insights: The difficulties encountered during implementation, such as inconsistent performance with various file sizes and kinds, provided vital information for the program's further development.

Future Improvements: These insights will be invaluable in enhancing the tool's capabilities, ensuring better performance, and addressing the limitations encountered.

References: -

1. Al-Amin, Sikder Tahsin, and Carlos Ordonez. "Efficient machine learning on data science languages with parallel data summarization." *Data & Knowledge Engineering* 136 (2021): 101930.
2. Dalcin, Lisandro D., et al. "Parallel distributed computing using Python." *Advances in Water Resources* 34.9 (2011): 1124-1139.
3. Borowiec, Marek L. "AMAS: a fast tool for alignment manipulation and computing of summary statistics." *PeerJ* 4 (2016): e1660.
4. https://www.geeksforgeeks.org/data-analysis-with-python/
5. https://www.w3schools.com/datascience/

6. https://www.sitepoint.com/python-multiprocessing-parallel-programming/
7. https://en.wikipedia.org/wiki/Parallel_computing

Appendix-A (code):

```
import time
import concurrent.futures
from transformers import pipeline
import matplotlib.pyplot as plt
import numpy as np

# Example text to be summarized
example_text = """
In the heart of a bustling city, a small, quaint café sat nestled between towering
skyscrapers. Its warm, inviting glow was a beacon to the weary travelers and bustling
locals alike. Inside, the aroma of freshly brewed coffee mingled with the sweet scent
of baked goods, creating an atmosphere of comfort and familiarity. The café's walls
were adorned with a collection of eclectic artwork, each piece telling its own unique
story. Patrons, a diverse mix of artists, students, and business professionals.
"""

# Set up the summarization pipeline
summarizer_model = pipeline("summarization", model="sshleifer/distilbart-cnn-12-6")

# Function to summarize a given text
def summarize(text):
    words = text.split()
    summary = summarizer_model(text, max_length=min(130, len(words) // 2),
min_length=max(30, len(words) // 4), do_sample=False)
    return summary[0]['summary_text']

# Function to measure summarization time
def measure_time(text, use_parallel=False, workers=None):
    start = time.time()

    if use_parallel:
        with concurrent.futures.ThreadPoolExecutor(max_workers=workers) as executor:
            result = list(executor.map(summarize, [text]))[0]
    else:
        result = summarize(text)

    end = time.time()
    return end - start
```

```
# Serial and Parallel execution timing
time_serial = measure_time(example_text)
time_parallel = measure_time(example_text, use_parallel=True, workers=2)

# Calculate efficiency
efficiency_gain = ((time_serial - time_parallel) / time_serial) * 100

# Display results
print(f"Serial Execution Time(seconds): {time_serial}")
print(f"Parallel Execution Time(seconds): {time_parallel}")
print(f"Efficiency Improvement(seconds): {efficiency_gain}%")

# Plotting comparison
plt.figure(figsize=(8, 4))
plt.bar(["Serial", "Parallel"], [time_serial, time_parallel], color=['blue', 'red'],
alpha=0.6)
plt.ylabel('Time (seconds)')
plt.title('Comparison of Serial vs Parallel Execution Time')
plt.show()

# Experiment with different number of cores
core_counts = [1, 2, 3, 4, 5]
times_serial = [measure_time(example_text) for _ in core_counts]
times_parallel = [measure_time(example_text, use_parallel=True, workers=core) for core
in core_counts]

# Plotting the impact of different core counts
plt.figure(figsize=(10, 5))
plt.plot(core_counts, times_serial, label='Serial', marker='o', color='blue')
plt.plot(core_counts, times_parallel, label='Parallel', marker='o', color='green')
plt.xlabel('Number of Cores')
plt.ylabel('Time (seconds)')
plt.title('Impact of Core Count on Execution Time')
plt.legend()
plt.grid(True)
plt.show()
```