Implementing Multi-threading And Multi-processing In Data Science

By

Anik Bhaumik(21MCA1134)

ABSTRACT

Nowadays, Data science is widely used to extract actionable insights from the large and ever-increasing volumes of data collected and prepared by today's organizations. It also has the capability of preparing data for analysis, processing, performing advanced data analysis, and presenting the results with easy understandable patterns and enabling stakeholders to draw informed conclusions. To make these happen in real scenarios we always need a dataset to be trained upon. Certainly, It's very time consuming to train those datasets which consist of gigantic data.

Multi-Processing: Multiprocessing refers to the ability of a system to support more than one processor at the same time. It works in parallel and doesn't share memory resources.

MUlti-Threading: Threads are components of a process, which can run sequentially. In multithreading multiple threads to be created within a process, executing independently but concurrently sharing process resources.

This paper proposes the implementation of parallelization techniques like multiprocessing and multithreading that can reduce the training time of large datasets for a data science problem. While dealing with larger implementations of machine learning, the time complexity is a major concern. Through this paper we are gonna optimize the process of analyzing the dataset by having a best case time-complexity in it.

MULTITHREADING

Threads are components of a process, which can run parallely. There can be multiple threads in a process, and they share the same memory space, i.e. the memory space of the parent process. This would mean the code to be executed as well as all the variables declared in the program would be shared by all threads.

Multithreading is a model of program execution that allows for multiple threads to be created within a process, executing independently but concurrently sharing process resources. Depending on the hardware, threads can run fully parallel if they are distributed to their own CPU core.

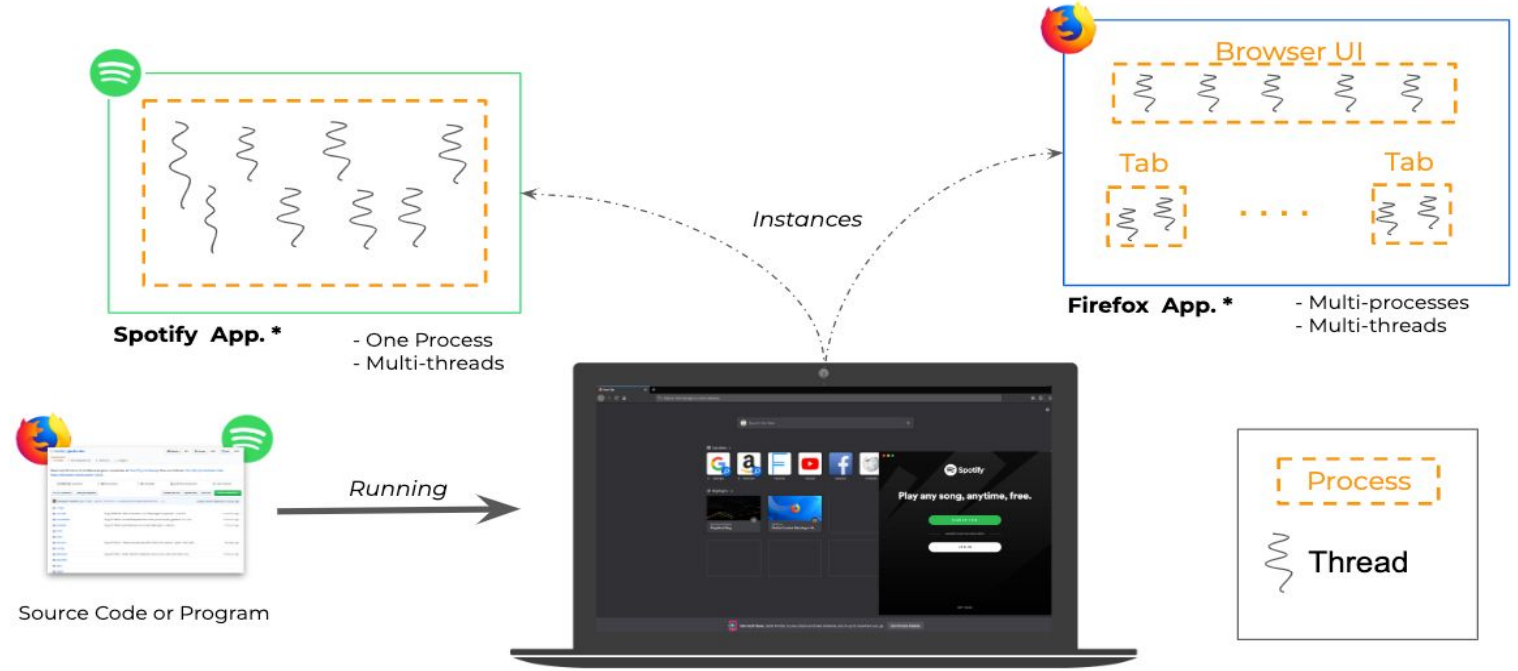
MULTIPROCESSING

A process is defined as an entity which represents the basic unit of work to be implemented in the system and is basically a program in execution.

multiprocessing, in computing, a mode of operation in which two or more processors in a computer simultaneously process two or more different portions of the same program (set of instructions). Multiprocessing is typically carried out by two or more microprocessors.

PARALLEL COMPUTING

Programs, Apps, Processes & Threads



** this image may not reflect the reality for the show-cased apps*

PITFALLS OF PARALLEL COMPUTING

Introducing parallelism to a program is not always a positive-sum game; there are some pitfalls to be aware of. The most important ones are as follows.

- **Race Condition**: As we already discussed, threads have a shared memory space, and therefore they can have access to shared variables. A race condition occurs when multiple threads try to change the same variable simultaneously. The thread scheduler can arbitrarily swap between threads, so we have no way of knowing the order in which the threads will try to change the data. This can result in incorrect behavior in either of the threads, particularly if the threads decide to do something based on the value of the variable. To prevent this from happening, a mutual exclusion (or mutex) *lock* can be placed around the piece of the code that modifies the variable so that only one thread can write to the variable at a time.

- **Starvation**: Starvation occurs when a thread is denied access to a particular resource for longer periods of time, and as a result, the overall program slows down. This can happen as an unintended side effect of a poorly designed thread-scheduling algorithm.
- **Deadlock**: Overusing mutex locks also has a downside - it can introduce deadlocks in the program. A deadlock is a state when a thread is waiting for another thread to release a lock, but that other thread needs a resource to finish that the first thread is holding onto. This way, both of the threads come to a standstill and the program halts. Deadlock can be thought of as an extreme case of starvation. To avoid this, we have to be careful not to introduce too many locks that are interdependent.
- **Livelock** : Livelock is when threads keep running in a loop but don't make any progress. This also arises out of poor design and improper use of mutex locks.

AVOIDING THE PITFALLS

- **The Global Interpreter Lock**

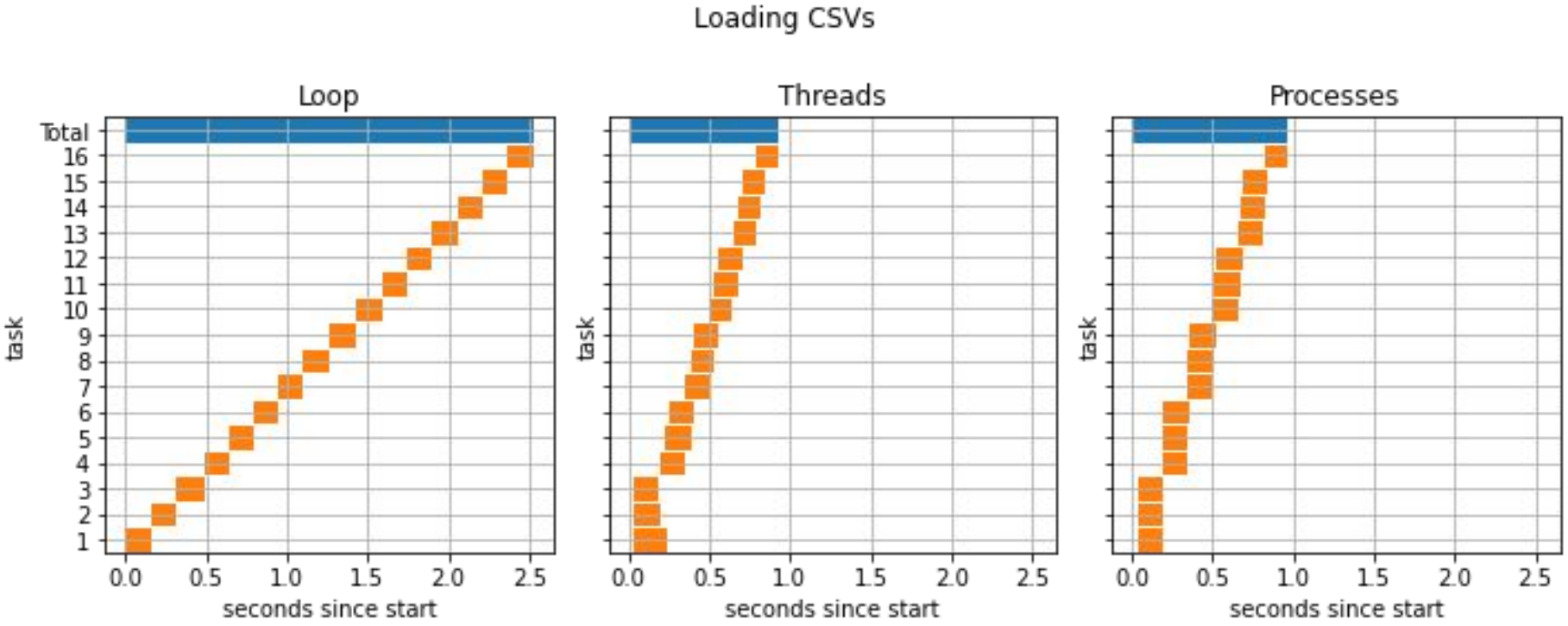
  When it comes to Python, there are some oddities to keep in mind. We know that threads share the same memory space, so special precautions must be taken so that two threads don’t write to the same memory location. The CPython interpreter handles this using a mechanism called GIL, or the Global Interpreter LocK.

  In CPython, the **global interpreter lock**, or **GIL**, is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes at once. This lock is necessary mainly because CPython's memory management is not thread-safe.

MULTITHREADING & MULTIPROCESSING IN PYTHON

- With Python threading, concurrency is achieved using multiple threads, but due to the GIL only one thread can be running at a time. In Python multiprocessing, the original process is forked process into multiple child processes bypassing the GIL. Each child process will have a copy of the entire program's memory.
- Both multithreading and multiprocessing allow Python code to run concurrently. Only multiprocessing will allow your code to be truly parallel. However, if your code is IO-heavy (like HTTP requests), then multithreading will still probably speed up your code.

EXECUTION TIME DIFFERENCE



Multithreading vs. Multiprocessing in Python

Using the Python program with a traditional approach can consume a lot of time to solve a problem. Multiprocessing and multithreading techniques optimize the process by reducing the training time of big data sets.

In a data science course, we can do a practical experiment with the normal approach as well as with the multiprocessing and multithreading approach.

The difference between these techniques can be calculated by running a simple task on Python. For instance, if a task takes 18.01 secs using the traditional approach in Python, the computational time reduces to 10.04 secs using the pool technique. The multithreading process can reduce the time taken to mere 0.013 secs. Both multiprocessing and multithreading have great computational speed.

Choosing between multithreading and multiprocessing

- **Use multithreading to make user interaction (UI) programs responsive.** These programs have to wait for the user to interact with them, so using threads provides enough computing power. For example, you might choose to use multithreading if you create an online writing program so that one thread tracks the user's keystrokes, a second thread displays the text for the user to read and a third thread proofreads the text to identify spelling and grammar errors.
- **Use multithreading to create I/O-bound or network bound applications.** Threads can provide you with all the computing power you need to access web servers and download content from the internet. For example, many data scientists use multithreading to create web scraping applications.
- **Use multiprocessing to create computation-intensive programs.** Multiprocessing can help you analyze large volumes of data quickly.
- **Use multiprocessing to develop programs that are CPU intensive.** Multiprocessing can help you speed up processes and provide reliable solutions for programs that involve several CPU tasks.

Conclusion

So, we can conclude that multiprocessing and threading have great computational speed. As the trend of increasing parallelism will continue to rise in future, these techniques will become more and more important in providing solutions to a data science problem in a much lesser time.

Thank You