Checkpointing in Python

Certain research and exploratory work may require running software applications for several days or weeks. Despite exempting such applications from the fair-use limits of the underlying shared computing platforms and allowing them to run for longer than normal durations, it is not uncommon to see the applications getting interrupted due to unplanned reasons. On certain occasions, we have seen such long-running applications crash due to unforeseen hardware issues that are triggered by the applications themselves. On some other occasions we had to urgently apply some firmware updates or do security related system patching due to which, we had to announce emergency maintenance and take the HPC system offline, thereby, terminating all the applications running on the system at that time. On yet another occasion, we saw some long-running applications getting terminated due to the actions of other customers on the shared HPC platform - just one customer can cause heavy, suboptimal IO that can bring down the filesystem for the entire customer base. Due to such reasons, it is furthermore important to make your applications write checkpoints at optimal frequencies so that you do not lose your entire progress and can restart your application from the latest checkpoint at a later stage.

Certain exploratory workflows/pipelines that require human in the loop involve repeating certain steps in the workflow selectively and iteratively, thereby also requiring the functionality of separately checkpointing each step in the workflow/pipeline.

In a previous article, we reviewed the basics of checkpointing and following is the link to that article: What is checkpointing? | LinkedIn . A sample C++ code demonstrating the checkpointing and restart capability was also shared through the following GitHub repository: bsswfellowship/checkpointing at main · ritua2/bsswfellowship (github.com)

In this article we will review the steps for adding the checkpointing - or save and restart - capabilities in Python code. Python supports object serialization and deserialization through its Pickle module and detailed information on Pickle is available at the following link: pickle — Python object serialization — Python 3.11.1 documentation. A simple Python code is shown below to demonstrate how Pickle can be used for implementing the checkpointing and restart (or the save and restart) functionality. This code is also available for download through the following GitHub repository:

```
import os
import time
import pickle
saved_dump = "ckptfile.pickle"
def main(start=0):
```

```
6.    #some useful code before this line
7.    global saved_dump
8.    a = start
9.    while 1:
10.     time.sleep(1)
11.     a += 1
12.     print(a)
13.     with open(saved_dump, 'wb') as f:
14.         pickle.dump(a, f)
15. if __name__ == '__main__':
16.   print("For testing, interrupt the running code by type ctrl+c")
17.   while 1:
18.     if os.path.exists(saved_dump):
19.     with open(saved_dump, "rb") as f:
20.         start = pickle.load(f)
21.     else:
22.     start = 0
23.     try:
24.       main(start=start)
25.     except KeyboardInterrupt:
26.       resume = raw_input('Would you like to continue running the code? Type the letter y
    for yes.')
27.       if resume != 'y':
28.         break
```

In line # 14 above, we use pickle.dump to covert the data - here the value of 'a' - into byte stream, and this byte stream is written to a file named "ckptfile.pickle" that was opened for writing in line # 13. Note that we are doing binary IO here.

In lines # 18-22 of the code above, we check if a file named "ckptfile.pickle" already exists or not. If the file exists - that is a checkpoint was written and the file in which the checkpoint was written is available - then, we initialize the value of 'start' to the value written in the file. However, if the file does not exist - because the checkpoint may have not been written or the checkpoint file may have been deleted - 'start' is set to 0 and the code begins executing from the beginning.

The steps shown below demonstrate how to run the code, interrupt it to generate the checkpoint file, and proceed either from the checkpoint or normally.

#To run the Python code shown above

$ python chkpt.py

For testing, interrupt the running code by type ctrl+c

```
1
2
3
4
5
6
7^CWould you like to continue running the code? Type the letter y for yes.
#After running the code for the first time, you will see the chptfile.pickle generated
$ ls chkpt.py ckptfile.pickle
#Let us remove the chptfile.pickle
$ rm ckptfile.pickle
$ ls chkpt.py
#Let us run the code again. It will start from the beginning as we removed the file # name chptfile.pickle. We will interrupt the code while it is running using ctrl+c, # and then restart
$ python chkpt.py
For testing, interrupt the running code by typing ctrl+c
1
2
3
4
5
6
```

```
7

8

9^CWould you like to continue running the code? Type the letter y for yes.y

10

11

12

13

14

15^CWould you like to continue running the code? Type the letter y for yes.y

16

17

18

19

20

21^CWould you like to continue running the code? Type the letter y for yes.

# As you notice from the values displayed above, instead of starting to print from # 1, the
code starts printing from the next number in the series that was printed # before
interruption

$ ls

chkpt.py ckptfile.pickle

# Now let us run the code again. Because the chptfile.pickle is present, the series # in the
example below begins from 22 and not 1. Before the code was # interrupted as shown
above, the last number that was printed was 21
```

```
$ python chkpt.py

For testing, interrupt the running code by typing ctrl+c

22

23

24

25

26

27^CWould you like to continue running the code? Type the letter y for yes.
```

References

1. pickle — Python object serialization — Python 3.11.1 documentation
2. pickle - How to "stop" and "resume" long time running Python script? - Stack Overflow