Introduction to Checkpointing

Ritu Arora, Wayne State University / Venra Tech Inc.

Email: ritu@wayne.edu / ritu@venratech.com

1. What is Checkpointing?

Checkpointing is the process of periodically saving (or writing) the execution state of an application such that in the event of an interruption in the execution of an application, this saved state can be used to continue the execution at a later time. Typically, the execution state is written to a file. Resuming the execution of an application using a previously saved state or checkpoint (instead of starting it from scratch) is referred to as the Restart phase.

2. What are the advantages of checkpointing?

Checkpointing not only saves time by offering the capability to resume the execution of an application in case of a hardware failure in the underlying computing platform (e.g., network interconnect failure) or if the computing platform becomes unavailable due to emergency maintenance, but it also helps in overcoming the time-limits associated with the different job queues/partitions.

3. What are the different types of checkpointing?

The different types of checkpointing include system-level checkpointing, application-level checkpointing, and user-level or library-level checkpointing.

- **System-Level checkpointing** involves taking core-dumps of the computational state of the machine or system on which the application is running.
    - Pros: It is convenient to use, no code changes needed, user only specifies the checkpointing frequency.
    - Cons: It involves large memory-footprint of checkpoints as the entire execution state of the application and the operating system processes are saved during checkpointing, and system administrator level privileges are needed for installation of additional code.
    - Example: Berkeley Lab Checkpointing and Restart (BLCR)

- **Library-Level or User-Level Checkpointing** involves the use of libraries for taking checkpoints while being agnostic to kernel-level information such as process IDs.
    - Pros: It is useful for checkpointing applications without requiring any changes to the source-code or the operating system kernel.

- Cons: The users may need to load the checkpointing library before starting their applications, and then, would need to dynamically link the loaded library to their applications. The checkpoints can have a large memory-footprint.
- Example: DMTCP

- **Application-Level Checkpointing** involves implementing the checkpoint-and-restart mechanism within the application itself. An efficient implementation of application-level checkpointing would require saving and reading the state of only those variables or data that are necessary for recreating the state of the entire application. Such variables or data are referred to as critical variables/data. As an example, consider the C code below.

```
int main(){
 int x = 4;
 int y = sqrt(x);
 int z, i; j = x*y;
 for (i =0; i< 100; i++){
     z += j* myFct(randomNumber * i);
 }
 return 0;
 }
```

In this code, "`i`" and "`z`" are critical variables as their values are updated and cannot be derived easily to recreate the execution state of the code once it is interrupted.

- Pros: Application-level checkpointing does not rely on the availability of any external libraries or tools, and hence, is useful for writing portable applications

- Cons: While an efficient implementation of this technique will generate checkpoints with smaller memory footprint and incur lesser I/O overheads as compared to other types of checkpointing, the onus is on the user (or the developer) to manually implement it on a per application basis, and therefore, the users should understand the code of the applications that they are checkpointing to manually reengineer the code for inserting checkpoint-restart logic

In case of distributed (message passing or MPI applications), a checkpoint can be written as a "central checkpoint" involving a single process (typically, the root or master or manager process in the MPI world) or a distributed checkpoint (involving multiple processes and an appropriate parallel I/O strategy).

4. What are the side-effects of checkpointing?

Writing and reading the application states or checkpoints introduces additional I/O overheads. Depending upon the frequency of checkpointing and the size of the checkpoint files, the IO overheads can add noticeable increase in the run-time and storage needs of an application.