Reducing the Barriers to PDES Checkpointing

Elsa Gonsiorowski (gonsie@rpi.edu) and Christopher Carothers (chrisc@cs.rpi.edu)
Rensselaer Polytechnic Institute, Troy, NY

Background

Parallel discrete-event simulation (PDES) is a common application in high performance computing. For any large scale simulation within an HPC system, running time can be a limiting constraint. Through checkpointing, a single simulation can be performed across many independent HPC "jobs." A PDES checkpoint requires saving the state of the model, in-flight events, and the settings of the simulation system itself. Here we use Rensselaer's Optimistic Simulation System, ROSS [2], and its YAWNS-based conservative synchronization algorithm [4].

ROSS Checkpointing: RIO

ROSS I/O, or RIO [1], is an integrated checkpointing system. It is able to take advantage of both the regularity of a PDES system (a model made up of LPs and events partitioned across nodes) and the efficiency of MPI [3] to quickly create compact checkpoints. RIO adds to the ROSS API and allows model developers to create checkpoints at the conclusion of their serial or conservative simulations. These checkpoints can then be used to restart and continue the simulation. RIO is dynamic and can allow for a change in parallel configuration across executions of the same simulation.

A RIO Checkpoint

Read-Me for Humans

To ensure data legacy, it is important to capture details about the software used to create the checkpoint. Both a ROSS and RIO git hash are recorded, as well any command line options and global variables specified during simulation.

Formatted Metadata

A checkpoint of any parallel system will consist of multiple partitions. For a PDES system, a partition will contain both LPs and events. Information about the location, size, and content of each partition's checkpoint data is recorded in a separate metadata file.

Checkpoint Data

Each partition's checkpoint data is stored in binary format. For increased parallelism, multiple files can be used for data within the same checkpoint. Since the metadata is separated from the data, each partition can be read in parallel.

The RIO API

Global Variables

RIO has several variables which the model developer can tune for a simulation. Command line arguments specify the number of partitions in a checkpoint and the number of files used for data. Another variable specifies the way in which a RIO checkpoint is loaded into a simulation. The model developer can also specify a model version string which is written to the checkpoint's read-me file.

LP Serialization

Model developers need to implement a serialize and de-serialize methods for each LP type in the simulation. These methods can be as simple as copying an LP's state into a provided buffer or as complex as needed. To build a compact checkpoint, LP sizing functions must also be defined.

Implementation

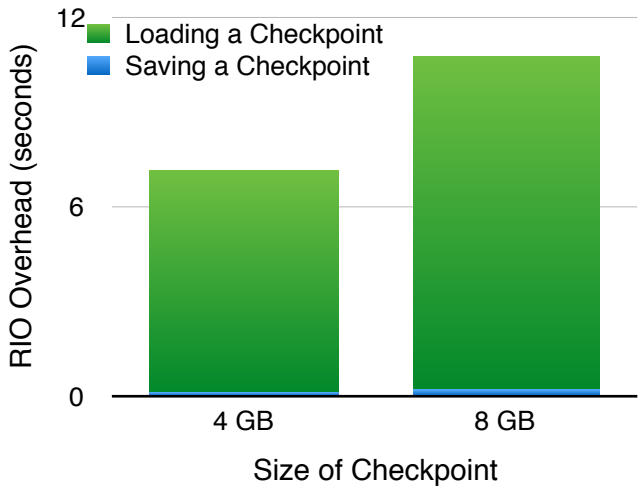
RIO is based on MPI file collective operations including:

- `MPI_File_write_at_all`
- `MPI_File_read_at_all`

These operations are blocking and require an explicit offset into the file [3].

Performance

As with any software reliant on file I/O, RIO performance is linked with file system performance. The graph on the right shows the overhead caused by RIO on 8 nodes of an IBM Blue Gene/Q supercomputer. Overall, RIO is able to take advantage of a fast I/O subsystem. The high overhead of loading a checkpoint is due to the time taken by the initialization of each LP.



[1] RIO: ROSS I/O. http://github.com/gonsie/RIO
[2] ROSS: Rensselaer's Optimistic Simulation System. http://github.com/carothersc/ROSS
[3] MPI: A Message-Passing Interface Standard. Version 3.1 http://www.mpi-forum.org/
[4] D. Nicol. The Cost of Conservative Synchronization in Parallel Discrete Event Simulations. *Journal of the ACM*, 40(2): 304–333, 1993.