Wikiprint Book

Title: Loading the OpenCHK module

Subject: DEEP - Public/User_Guide/OpenCHK

Version: 3

Date: 14.05.2024 11:15:04

Table of Contents

What is OpenCHK?

The OpenCHK model is a pragma-based checkpointing model developed by the Programming Models group at the Barcelona Supercomputing Center.

The aim of this model is to provide a generic and portable way to checkpoint and recover data in C/C++ and Fortran High Performance Computing applications.

The prototype implementation of the model is based on two software components:

Mercurium source-to-source compiler Transparent Checkpoint Library

Loading the OpenCHK module

Firstly, it is required to append some paths to your "MODULEPATH" environment variable:

```
modulepath="/usr/local/software/skylake/Stages/2018b/modules/all/Core:$modulepath"
modulepath="/usr/local/software/skylake/Stages/2018b/modules/all/Compiler/mpi/intel/2019.0.117-GCC-7.3.0:$modulepath"
modulepath="/usr/local/software/skylake/Stages/2018b/modules/all/MPI/intel/2019.0.117-GCC-7.3.0/psmpi/5.2.1-1-mt:$modulepa
export MODULEPATH="$modulepath:$MODULEPATH"
```

Once this is done, simply load the following modules:

```
module load Intel/2019.0.117-GCC-7.3.0
module load ParaStationMPI/5.2.1-1-mt
module load OpenCHK/1.0
```

Documentation and User guide

Manual: ?https://github.com/bsc-pm/OpenCHK-model

Quick Start Guide

Before the Execution

TCL provides support for three different backends: FTI, SCR and VeloC. The backend library must be chosen using the environment variable "TCL_BACKEND".
NOTE: In the current installation, the only enabled backend is FTI.

```
export TCL_BACKEND=FTI
```

When using FTI as backend library, the user needs to provide an FTI configuration file using the environment variable "FTI_CONF_FILE". (see attachments: config.fti)

```
export FTI_CONF_FILE=config.fti
```

OpenCHK syntax

- The *init* construct defines the initialization of a checkpoint context. None of the other constructs must be used without initializing a checkpoint context.
- The *shutdown* construct defines the finalization of a checkpoint context.
- The 'store' construct specifies that some variables and memory regions are going to be saved in a new checkpoint.
- The 'load' construct specifies that some variables and memory regions are goind to be updated with the stored values that we have previously checkpointed (if any).

Example C/C++

Compile the example:

```
mpicxx -cxx=mcxx  --checkpoint -o test_scalar-cpp test_scalar.cpp
```

Run the example:
NOTE: Remember that you should have set TCL_BACKEND=FTI and a valid FTI_CONF_FILE.

```
mpirun -np 8 ./test_scalar-cpp [step_to_inject_error]
```

```
#include <iostream>
#include <cassert>
#include <stdlib.h>

void error_handler(int err_code)
{
   std::cout << "Error inside CheckpointLib. Error code: " << err_code << "." << std::endl;
   exit(-1);
}

int main(int argc, char **argv)
{
   int err = MPI_Init(&argc, &argv);
   assert(err == MPI_SUCCESS);
   MPI_Comm comm = MPI_COMM_WORLD;

   int inject_error = -1;
   if(argc == 2) {
       inject_error = std::atoi(argv[1]);
       std::cout << "Inject error at step " << inject_error << "." << std::endl;
   }

   #pragma chk init comm(comm)
   {
       int data, i = 0;
       bool restored = false;

       #pragma chk load(i, data)
       if(i != 0) {
           std::cout << "Restored data from iteration " << i << ". data = " << data << "." << std::endl;
           restored = true;
       }
       for(i; i < 10; i++) {
           data = i;
           #pragma chk store(i, data) kind(CHK_FULL) id(i) level((i%4)+1) if(1) handler(error_handler)
           if(i == inject_error && !restored) {
               std::cout << "Injected error." << std::endl;
               exit(-1);
           }
           std::cout << "Completed step " << i << std::endl;
       }
   }
   #pragma chk shutdown

   MPI_Finalize();
}
```

Example Fortran

Compile the example:

```
mpif90 -fc=ifort-mfc  --checkpoint -o test_scalar-fortran test_scalar.f90
```

Run the example:
NOTE: Remember that you should have set TCL_BACKEND=FTI and a valid FTI_CONF_FILE.

```
mpirun -np 8 ./test_scalar-fortran [step_to_inject_error]
```

```
PROGRAM T1
  include 'mpif.h'
  integer rank, size, ierror, tag, comm, status(MPI_STATUS_SIZE)
  integer actual_data = 0, restored_i = 0, restored = 0, inject_error = -1, i
   CHARACTER(len=32) :: arg

  call MPI_INIT(ierror)
  comm = MPI_COMM_WORLD

  if (iargc() == 1) then
      call getarg(1, arg)
      read (arg,'(I10)') inject_error
      print *, 'Inject error at step ', inject_error, '.'
  endif

!$chk init comm(comm)
!$chk load(restored_i, actual_data)
  if (restored_i .ne. 0) then
      print *, 'Restored data from iteration ', restored_i , '. data = ', actual_data, '.'
      restored = 1
  end if

  do i = restored_i, 10
      actual_data = i
!$chk store(i, actual_data) kind(0) id(i) level(mod(i,4)+1) if(i .ge. 0)
      if (i .eq. inject_error .AND. restored .eq. 0) then
          print *, 'Injected error'
          call EXIT(-1)
      end if
      print *, 'Completed step ', i
  end do
!$chk shutdown

  call MPI_FINALIZE(ierror)
END PROGRAM
```