Fault Tolerance Library

Release 1.4

Leonardo Bautista-Gomez

Oct 01, 2020

USER GUIDE

CHAPTER

ONE

INTRODUCTION

In high-performance computing (HPC), systems are built from highly reliable components. However, the overall failure rate of supercomputers increases with the component count. Nowadays, petascale machines have a mean time between failures (MTBF) measured in hours and fault tolerance (FT) is a well-known issue. Long-running large applications rely on FT techniques to successfully finish their long executions. Checkpoint/Restart (CR) is a popular technique in which the applications save their state in stable storage, frequently a parallel file system (PFS). Upon a failure, the application restarts from the last saved checkpoint. CR is a relatively inexpensive technique in comparison with the process-replication scheme that imposes over 100% of overhead. However, when a large application is checkpointed, tens of thousands of processes will each write several GBs of data and the total checkpoint size will be in the order of several tens of TBs. Since the I/O bandwidth of supercomputers does not increase at the same speed as computational capabilities, large checkpoints can lead to an I/O bottleneck, which causes up to 25% of overhead in current petascale systems. Post-petascale systems with a significantly larger number of components and an important amount of memory will observe an impact on the system's reliability. With a shorter MTBF, those systems may require a higher checkpoint frequency and at the same time, they will have significantly larger amounts of data to save. At the same time, we observe deeper storage hierarchies, with high-bandwidth memories (HBM), non-volatile memories (NVM), solid-state drives (SSD) among others. Such a deep storage hierarchy can be complemented with cross-node redundancy schemes, such as checkpoint replication, XOR encoding or even more complex erasure codes such as Reed-Solomon (RS) encoding which is the basis of multi-level checkpointing. FTI is a multi-level checkpointing library with a simple API easy to use and offers a flexible configuration to enable the user to select the checkpointing strategy which fits best to the problem.

CHAPTER

TWO

TUTORIAL

In this page we present a tutorial of FTI. The purpose of the practice section is for you to get familiar with the FTI API as well as with the configuration file. Therefore there is limited information on how you should proceed.

For a detailed step-by-step guide for installing FTI:

A demonstration of the multi-level checkpointing is explained here:

2.1 Installation

2.2 Preparation

1. Create FTI directory

```
mkdir FTI
cd FTI
```

2. Create Installation Directory

```
mkdir install-fti
```

3. Set environmental variable to installation path

```
export FTI_INSTALL_DIR=$PWD/install-fti
```

4. Download FTI.

```
git clone https://github.com/leobago/fti
```

5. Change into base directory

```
cd fti
```

6. Set Environment Variable to FTI root

```
export FTI_ROOT=$PWD
```

2.3 Configure and Install

1. Create build directory and change into it

```
mkdir build
cd build
```

2. Build FTI

```
cmake -DCMAKE_INSTALL_PREFIX:PATH=$FTI_INSTALL_DIR -DENABLE_TUTORIAL=1 ..
make
make install
```

The flag -DENABLE_TUTORIAL=1 besides building FTI, will also build the tutorial files

2.4 Executables, tutorial source code, and fti library files

The library is installed at the $FTI_INSTALL_DIR the source code of the FTI library is in ${FTI_ROOT}/src and the source code of the tutorial is under ${FTI_ROOT}/tutorial, the executables of the tutorial are under ${FTI_ROOT}/build/tutorial/. For conveniency on the rest of the tutorial set also the following variables:

```
export TUTORIAL_EXEC=${FTI_ROOT}/build/tutorial/
export TUTORIAL_SRC=${FTI_ROOT}/tutorial/
```

You should always export this variables every time you try to start/continue the tutorial. Under the ${TUTORIAL_SRC} directory you can find various directories, each directory corresponds to a step presented in the tutorial.

2.5 Demonstration of FTI

To demonstrate the various safety levels of FTI, we will execute an example which uses the API function 'FTI_Snapshot()'. Run the example in each case for at least one minute and interrupt the execution after that time by pressing 'ctrl+c'. In some systems 'ctrl+c' does not kill all executing MPI processes, to kill all processes just killall 'executable'.

2.6 L1 - Local checkpoint on the nodes

Change into folder ${TUTORIAL_EXEC}/L1 and run the execution with 'make hdl1'. While the program is running, you may follow the events by observing the contents in the 'local' folder. In order to do that you can use the commands:

```
watch -n 1 $(find local)
watch -n 1 $(du -kh local)
```

or

```
cd local; watch -n 1 $(ls -lR)
```

(It may be illuminating to open the files in the '${TUTORIAL_EXEC}/L1/meta' folder, using a text editor. What kind of information do you think is kept in these files?)

After interrupting the execution, run again 'make hdl1'. The execution will (hopefully) resume from where the checkpoint was taken.

After the successful restart, interrupt the execution and delete one of the checkpoint files. The files are stored as (you can also simply delete the whole node directory): ${TUTORIAL_EXEC}/L1//local///l1/ckpt-Rank.fti.. You will notice, that in that case the program won't be able to resume the execution.

2.7 L2 – local checkpoint on the nodes + copy to the neighbor node:

Change into folder ${TUTORIAL_EXEC}/L2 and run the execution with 'make hdl2'. While the program is running, you may follow the events by observing the contents in the 'local' folder.

After interrupting the execution, run again 'make hdl2'. The execution will also in this case (hopefully) resume from where the checkpoint was taken.

After the successful restart, interrupt the execution and delete one of the checkpoint files. You will notice that now the program (hopefully) will be able to resume the execution. Try to delete more then one file.

2.7.1 Questions: In order to keep the execution able to resume:

1. How many files you can delete?
2. Which files can you delete?

L3 – local checkpoint on the nodes + copy to the neighbor node + RS encoding:

Change into folder ${TUTORIAL_EXEC}/L3 and run the execution with 'make hdl3'. While the program is running, you may follow the events by observing the contents in the 'local' folder.

After interrupting the execution, run again 'make hd3'. The execution will (surprisingly) also in this case resume from where the checkpoint was taken.

After the successful restart, interrupt the execution and delete one of the checkpoint files, the program will be able to resume.

2.7.2 Questions: In order to keep the execution able to resume:

1. How many files you can delete?
2. Which files can you delete?

2.8 L4 – flush of the checkpoints to the parallel file system:

Change into folder ${TUTORIAL_EXEC}/L4 and run the execution with 'make hdl4'. While the program is running, you may follow the events by observing the contents in the 'global' folder. After interrupting the execution, run again 'make hdl4'. The execution will resume from where the checkpoint was taken.

2.9 L4 – Differential Checkpoint:

Change into folder ${TUTORIAL_EXEC}/DCP/ and run the execution with 'make hdDCP'. While the progam is running you may follow the "blue" messages in the terminal. What is actually happening? After a couple of checkpoints, you can kill the application and restart it.

Delete all files under ./local, ./global/ ./meta/ and open file config.DCP.fti with your favorite text editor. Change the following parameters :

1. ckpt_io = 3 to ckpt_io = 1
2. failure = "x" to failure = 0

The first option changes the file format and the second option indicates that we will do a fresh run (not a recovery). Run the execution with 'make hdDCP', do you observe any difference in the timings of the checkpoints?

2.10 Practice

1. In the '${TUTORIAL_SRC}/practice' folder you will find the source code of the program we used to demonstrate the FTI features. In this case without FTI being implemented. Try to implement FTI. You can use either the 'FTI_Snapshot' or 'FTI_Checkpoint' function to cause FTI taking a checkpoint. To build the code changes you implemented you can :

```
cd $FTI_ROOT/build
make
```

To execute your implementation change directory to ${TUTORIAL_EXEC}/practice and execute the binary hdp.exe.

Besides implementing the source code you need also to create an appropriate configuration file. Information about the options in the configuration file can be found in the *Configuration* page and example configuration files can be found in the *APIs* page.

```
cd $TUTORIAL_EXEC/practice
make
mpirun -n 4 ./hdp.exe GRID_SIZE
```

GRID_SIZE is an integer number defining the size of the grid to be solved in Mb.

2. Change into the folder '${TUTORIAL_EXEC}/tutorial/experiment' and play with the settings of the configuration file. To run the program, type: 'mpirun -n 8 hdex.exe <*GRIDSIZE*> config.fti'. Perform executions with 'Head=0' and 'Head=1', do you notice any difference in the execution duration? (Note: You may take frequent L3 checkpointing and a gridsize of 256 or higher. In that case you will most likely see a difference). (Remark: denotes the dynamic memory of each mpi process in MB)

CHAPTER

THREE

COMPILATION

FTI uses CMake as the build manager to configure and perform the installation. We also provide an installation script, **install.sh**, in the root directory to faciliate this process. Both processes are identical, the script merely wraps some common cases and configurations as options. If your preffered way to build FTI is to use CMake, we recommend to do an out-of-source build. The following bash code snippets showcase how to build FTI with a given prefix path.

Default The default configuration builds the FTI library with Fortran and MPI-IO support for GNU compilers:

```
mkdir build && cd build
cmake -DCMAKE_INSTALL_PREFIX:PATH=/install/here/fti ..
make all install
```

Note: Notice: THE TWO DOTS AT THE END INVOKE CMAKE IN THE TOP LEVEL DIRECTORY.

Intel compilers Fortran and MPI-IO support for Intel compilers:

```
cmake -C ../intel.cmake -DCMAKE_INSTALL_PREFIX:PATH=/install/here/fti ..
make all install
```

Disable Fortran Only build FTI C library:

```
cmake -DCMAKE_INSTALL_PREFIX:PATH=/install/here/fti -DENABLE_FORTRAN=OFF ..
make all install
```

Lustre For Lustre user who want to use MPI-IO, it is strongly recommended to configure with Lustre support:

```
cmake -DCMAKE_INSTALL_PREFIX:PATH=/install/here/fti -DENABLE_LUSTRE=ON ..
make all install
```

Cray For Cray systems, make sure that the modules craype/* and PrgEnv* are loaded (if available). The configuration should be done as:

```
export CRAY_CPU_TARGET=x86-64
export CRAYPE_LINK_TYPE=dynamic
cmake -DCMAKE_INSTALL_PREFIX:PATH=/install/here/fti -DCMAKE_SYSTEM_
↪NAME=CrayLinuxEnvironment ..
make all install
```

Note: Notice: MODIFY x86-64 IF YOU ARE USING A DIFFERENT ARCHITECTURE. ALSO, THE OPTION CMAKE_SYSTEM_NAME=CrayLinuxEnvironment IS AVAILABLE ONLY FOR CMAKE VERSIONS 3.5.2 AND ABOVE.

CHAPTER

FOUR

INSTALLING ADDITIONAL IO LIBRARIES

FTI can work alongside other IO libraries when creating checkpoint files. Currently, FTI has support for HDF5 and SIONLib. These libraries can be linked to FTI through CMake options.

4.1 HDF5

FTI can use the HDF5 library and format to generate checkpoint files. FTI is compatible with the parallel version of the HDF5 library. Usually package managers have both both HDF5 versions available, so make sure the one installed is the correct one. Moreover, if you need to compile HDF5 from source, make sure to supply the following option: **–enable-parallel**. HDF5 support is enabled passing the *ENABLE_HDF5* option to CMake. Alternatively, the **–enable-hdf5** option can be informed to the install script.

```
mkdir build && cd build
      cmake -DENABLE_HDF5=1 ..
      make all install
# Or, alternatively
./install.sh --enable-hdf5
```

4.2 SIONLib

FTI also supports SIONLib as the IO library. As it is with HDF5, FTI must be linked with the parallel version of SIONLib. Inform the *ENABLE_SIONLIB* option to CMake in order to link FTI with SIONLib. If necessary, use the *SIONLIBBASE* CMake option to assist the linker in finding the library. The bash script snippet below showcase the commands for a build where SIONLib is installed at */opt/sionlib*.

```
mkdir build && cd build
      cmake -DENABLE_SIONLIB=1 DSIONLIBBASE=/opt/sionlib ..
      make all install
# Or, alternatively
./install.sh --enable-sionlib --sionlib-path=/opt/sionlib
```

CHAPTER

FIVE

APIS

int **FTI_Init**(**const** char *_configFile_, MPI_Comm _globalComm_)
Initializes FTI.

This function initializes the FTI context and prepares the heads to wait for checkpoints. FTI processes should never get out of this function. In case of a restart, checkpoint files should be recovered and in place at the end of this function.

Return integer FTI_SCES if successful.

Parameters

- `configFile`: FTI configuration file.
- `globalComm`: Main MPI communicator of the application.

int **FTI_Status**()
It returns the current status of the recovery flag.

This function returns the current status of the recovery flag.

Return integer FTI_Exec.reco.

int **FTI_InitType**(FTIT_type *_type_, int _size_)
It initializes a data type.

This function initalizes a data type. The only information needed is the size of the data type, the rest is black box for FTI. Types saved as byte array in case of HDF5 format.

Return integer FTI_SCES if successful.

Parameters

- `type`: The data type to be intialized.
- `size`: The size of the data type to be intialized.

int **FTI_InitComplexType**(FTIT_type *_newType_, FTIT_complexType *_typeDefinition_, int _length_, size_t _size_, char *_name_, FTIT_H5Group *_h5group_)
It initializes a complex data type.

This function initalizes a simple data type. New type can only consists fields of flat FTI types (no arrays). Type definition must include:

- length => number of fields in the new type
- field[].type => types of the field in the new type
- field[].name => name of the field in the new type
- field[].rank => number of dimentions of the field
- field[].dimLength[] => length of each dimention of the field

Return integer FTI_SCES if successful.

Parameters

- `newType`: The data type to be intialized.
- `typeDefinition`: Structure definition of the new type.
- `length`: Number of fields in structure
- `size`: Size of the structure.
- `name`: Name of the structure.
- `h5group`: Group of the type.

void **FTI_AddSimpleField**(FTIT_complexType **typeDefinition*, FTIT_type **ftiType*, size_t *offset*, int *id*, char **name*)

It adds a simple field in complex data type.

This function adds a field to the complex datatype. Use offsetof macro to set offset. First ID must be 0, next one must be +1. If name is NULL FTI will set "T${id}" name. Sets rank and dimLength to 1.

Return integer FTI_SCES if successful.

Parameters

- `typeDefinition`: Structure definition of the complex data type.
- `ftiType`: Type of the field
- `offset`: Offset of the field (use offsetof)
- `id`: Id of the field (start with 0)
- `name`: Name of the field (put NULL if want default)

void **FTI_AddComplexField**(FTIT_complexType **typeDefinition*, FTIT_type **ftiType*, size_t *offset*, int *rank*, int **dimLength*, int *id*, char **name*)

It adds a simple field in complex data type.

This function adds a field to the complex datatype. Use offsetof macro to set offset. First ID must be 0, next one must be +1. If name is NULL FTI will set "T${id}" name.

Return integer FTI_SCES if successful.

Parameters

- `typeDefinition`: Structure definition of the complex data type.
- `ftiType`: Type of the field
- `offset`: Offset of the field (use offsetof)
- `rank`: Rank of the array
- `dimLength`: Dimention length for each rank
- `id`: Id of the field (start with 0)
- `name`: Name of the field (put NULL if want default)

int **FTI_GetStageDir**(char **stageDir*, int *maxLen*)

Places the FTI staging directory path into 'stageDir'.

This function places the FTI staging directory path in 'stageDir'. If allocation size is not sufficiant, no action is perfoprmed and FTI_NSCS is returned.

Return integer FTI_SCES if successful, FTI_NSCS else.

Parameters

- `stageDir`: pointer to allocated memory region.
- `maxLen`: size of allocated memory region in bytes.

int **FTI_GetStageStatus**(int *ID*)
Returns status of staging request.

This function returns the status of the staging request corresponding to ID. The ID is returned by the function 'FTI_SendFile'. The status may be one of the five possible statuses:

Return integer Status of staging request on success, FTI_NSCS else.

Parameters

- `ID`: ID of staging request.

FTI_SI_FAIL - Stage request failed FTI_SI_SCES - Stage request succeed FTI_SI_ACTV - Stage request is currently processed FTI_SI_PEND - Stage request is pending FTI_SI_NINI - There is no stage request with this ID

Note If the status is FTI_SI_NINI, the ID is either invalid or the request was finished (succeeded or failed). In the latter case, 'FTI_GetStageStatus' returns FTI_SI_FAIL or FTI_SI_SCES and frees the stage request ressources. In the consecutive call it will then return FTI_SI_NINI.

int **FTI_SendFile**(char **lpath*, char **rpath*)
Copies file asynchronously from 'lpath' to 'rpath'.

This function may be used to copy a file local on the nodes via the FTI head process asynchronously to the PFS. The file will not be removed after successful transfer, however, if stored in the directory returned by 'FTI_GetStageDir' it will be removed during 'FTI_Finalize'.

Return integer Request handle (ID) on success, FTI_NSCS else.

Parameters

- `lpath`: absolute path local file.
- `rpath`: absolute path remote file.

If staging is enabled but no head process, the staging will be performed synchronously (i.e. by the calling rank).

int **FTI_InitGroup**(FTIT_H5Group **h5group*, char **name*, FTIT_H5Group **parent*)
It initialize a HDF5 group.

Initialize group defined by user. If parent is NULL this mean parent will be set to root group.

Return integer FTI_SCES if successful.

Parameters

- `h5group`: H5 group that we want to initialize
- `name`: Name of the H5 group
- `parent`: Parent H5 group

int **FTI_setIDFromString**(char **name*)
Searches in the protected variables for a name. If not found it allocates and returns the ID.

This function searches for a given name in the protected variables and returns the respective id for it.

Return integer id of the variable.

Parameters

- `name`: Name of the protected variable to search

int **`FTI_getIDFromString`**(char *_name_)

Searches in the protected variables for a name. If not found it allocates and returns the ID.

This function searches for a given name in the protected variables and returns the respective id for it.

Return integer id of the variable.

Parameters

- `name`: Name of the protected variable to search

int **`FTI_RenameGroup`**(FTIT_H5Group *_h5group_, char *_name_)

Renames a HDF5 group.

This function renames HDF5 group defined by user.

Return integer FTI_SCES if successful.

Parameters

- `h5group`: H5 group that we want to rename
- `name`: New name of the H5 group

int **`FTI_Protect`**(int _id_, void *_ptr_, int32_t _count_, FTIT_type _type_)

It sets/resets the pointer and type to a protected variable.

This function stores a pointer to a data structure, its size, its ID, its number of elements and the type of the elements. This list of structures is the data that will be stored during a checkpoint and loaded during a recovery. It resets the pointer to a data structure, its size, its number of elements and the type of the elements if the dataset was already previously registered.

Return integer FTI_SCES if successful.

Parameters

- `id`: ID for searches and update.
- `ptr`: Pointer to the data structure.
- `count`: Number of elements in the data structure.
- `type`: Type of elements in the data structure.

int **`FTI_DefineGlobalDataset`**(int _id_, int _rank_, FTIT_hsize_t *_dimLength_, **`const`** char *_name_, FTIT_H5Group *_h5group_, FTIT_type _type_)

Defines a global dataset (shared among application processes)

This function defines a global dataset which is shared among all ranks. In order to assign sub sets to the dataset the user has to call the function 'FTI_AddSubset'. The parameter 'did' of that function, corres- ponds to the global dataset id define here.

Return integer FTI_SCES if successful.

Parameters

- `id`: ID of the dataset.
- `rank`: Rank of the dataset.
- `dimLength`: Dimention length for each rank.
- `name`: Name of the dataset in HDF5 file.
- `h5group`: Group of the dataset. If Null then "/".

- `type`: FTI type of the dataset.

int **`FTI_AddSubset`** (int *id*, int *rank*, FTIT_hsize_t **offset*, FTIT_hsize_t **count*, int *did*)
Assigns a FTI protected variable to a global dataset.

This function assigns the protected dataset with ID 'id' to a global data- set with ID 'did'. The parameters 'offset' and 'count' specify the selec- tion of the sub-set inside the global dataset ('offset' and 'count' correspond to 'start' and 'count' in the HDF5 function 'H5Sselect_hyperslab' For questions on what they define, please consult the HDF5 documentation.)

Return integer FTI_SCES if successful.

Parameters

- `id`: Corresponding variable ID.
- `rank`: Rank of the dataset.
- `offset`: Starting coordinates in global dataset.
- `count`: number of elements for each coordinate.
- `did`: Corresponding global dataset ID.

int **`FTI_UpdateGlobalDataset`** (int *id*, int *rank*, FTIT_hsize_t **dimLength*)
Updates global dataset (shared among application processes)

updates only the rank and number of elements for each coordinate direction.

Parameters

- `id`: ID of the dataset.
- `rank`: Rank of the dataset.
- `dimLength`: Dimention length for each rank.

int **`FTI_GetDatasetRank`** (int *did*)
returns rank of shared dataset

Return integer rank of dataset.

Parameters

- `id`: ID of the dataset.

FTIT_hsize_t ***`FTI_GetDatasetSpan`** (int *did*, int *rank*)
returns static array of dataset dimensions

Parameters

- `id`: ID of the dataset.
- `rank`: Rank of the dataset.

int **`FTI_RecoverDatasetDimension`** (int *did*)
loads dataset dimension from ckpt file to dataset 'did'

Parameters

- `id`: ID of the dataset.

int **FTI_DefineDataset** (int *id*, int *rank*, int **dimLength*, char **name*, FTIT_H5Group **h5group*)
Defines the dataset.

This function gives FTI all information needed by HDF5 to correctly save the dataset in the checkpoint file.

Return integer FTI_SCES if successful.

Parameters

- `id`: ID for searches and update.
- `rank`: Rank of the array
- `dimLength`: Dimention length for each rank
- `name`: Name of the dataset in HDF5 file.
- `h5group`: Group of the dataset. If Null then "/"

int32_t **FTI_GetStoredSize** (int *id*)
Returns size saved in metadata of variable.

This function returns size of variable of given ID that is saved in metadata. This may be different from size of variable that is in the program. If this function it's called when recovery it returns size from metadata file, if it's called after checkpoint it returns size saved in temporary metadata. If there is no size saved in metadata it returns 0.

Return int32_t Returns size of variable or 0 if size not saved.

Parameters

- `id`: Variable ID.

void ***FTI_Realloc** (int *id*, void **ptr*)
Reallocates dataset to last checkpoint size.

Return ptr Pointer if successful, NULL otherwise This function loads the checkpoint data size from the metadata file, reallacates memory and updates data size information.

Parameters

- `id`: Variable ID.
- `ptr`: Pointer to the variable.

int **FTI_BitFlip** (int *datasetID*)
Bit-flip injection following the injection instructions.

This function injects the given number of bit-flips, at the given frequency and in the given location (rank, dataset, bit position).

Return integer FTI_SCES if successful.

Parameters

- `datasetID`: ID of the dataset where to inject.

int **FTI_Checkpoint** (int *id*, int *level*)
It takes the checkpoint and triggers the post-ckpt. work.

This function starts by blocking on a receive if the previous ckpt. was offline. Then, it updates the ckpt. information. It writes down the ckpt. data, creates the metadata and the post-processing work. This function is complementary with the FTI_Listen function in terms of communications.

Return integer FTI_SCES if successful.

Parameters

- `id`: Checkpoint ID.
- `level`: Checkpoint level.

int **FTI_InitICP** (int *id*, int *level*, bool *activate*)
Initialize an incremental checkpoint.

This function defines the environment for the incremental checkpointing mechanism. The iCP mechanism consists of three functions: FTI_InitICP, FTI_AddVarICP and FTI_FinalizeICP. The two functions FTI_InitICP and FTI_FinalizeICP define the iCP region within the user may write the protected variables in any order. The iCP region is active, when the expression passed through 'activate' evaluates to TRUE.

Return integer FTI_SCES if successful.

Parameters

- `id`: Checkpoint ID.
- `level`: Checkpoint level.
- `activate`: Boolean expression.

Note This function is not blocking for POSIX, FTI-FF and HDF5, but, blocking for MPI-IO. This is due to the collective open call in MPI_IO

int **FTI_AddVarICP** (int *varID*)
Write variable into the CP file.

With this function, the user may write the protected datasets in any order into the checkpoint file. However, before the call to FTI_FinalizeICP, all protected variables must have been written into the file.

Return integer FTI_SCES if successful.

Parameters

- `id`: Protected variable ID.

int **FTI_FinalizeICP** ()
Finalize an incremental checkpoint.

This function finalizes an incremental checkpoint. In contrast to InitICP, this function is collective on the communicator FTI_COMM_WORLD and blocking.

Return integer FTI_SCES if successful.

int **FTI_Recover** ()
It loads the checkpoint data.

This function loads the checkpoint data from the checkpoint file and it updates some basic checkpoint information.

Return integer FTI_SCES if successful.

int **FTI_Snapshot** ()
Takes an FTI snapshot or recovers the data if it is a restart.

This function loads the checkpoint data from the checkpoint file in case of restart. Otherwise, it checks if the current iteration requires checkpointing, if it does it checks which checkpoint level, write the data in the files and it communicates with the head of the node to inform that a checkpoint has been taken. Checkpoint ID and counters are updated.

Return integer FTI_SCES if successful.

int **FTI_Finalize**()

It closes FTI properly on the application processes.

This function notifies the FTI processes that the execution is over, frees some data structures and it closes. If this function is not called on the application processes the FTI processes will never finish (deadlock).

Return integer FTI_SCES if successful.

int **FTI_RecoverVar**(int *id*)

Recovers given variable.

Return integer FTI_SCES if successful.

Parameters

- `integer`: id of variable to be recovered

Warning: doxygenfunction: Cannot find function "FTI_Print" in doxygen xml output for project "Fault Tolerance Library" from directory: ../Doxygen/xml

CHAPTER

SIX

CONFIGURATION

6.1 [Basic]

6.1.1 head

The checkpointing safety levels L2, L3 and L4 produce additional overhead due to the necessary postprocessing work on the checkpoints. FTI offers the possibility to create an MPI process, called HEAD, in which this postprocessing will be accomplished. This allows it for the application processes to continue the execution immediately after the checkpointing.

Value	Meaning
0	The checkpoint postprocessing work is covered by the application processes
1	The HEAD process accomplishes the checkpoint postprocessing work (notice: In this case, the number of application processes will be (n-1)/node)

(*default = 0*)

6.1.2 node_size

Lets FTI know, how many processes will run on each node (ppn). In most cases this will be the amount of processing units within the node (e.g. 2 CPU's/node and 8 cores/CPU ! 16 processes/node).

Value	Meaning
ppn (int > 0)	Number of processing units within each node (notice: The total number of processes must be a multiple of `group_size*node_size`)

(*default = 2*)

6.1.3 ckpt_dir

This entry defines the path to the local hard drive on the nodes.

Value	Meaning
string	Path to the local hard drive on the nodes

(*default = /scratch/username/*)

6.1.4 glbl_dir

This entry defines the path to the checkpoint folder on the PFS (L4 checkpoints).

Value	Meaning
string	Path to the checkpoint directory on the PFS

(*default = /work/project/*)

6.1.5 meta_dir

This entry defines the path to the meta files directory. The directory has to be accessible from each node. It keeps files with information about the topology of the execution.

Value	Meaning
string	Path to the meta files directory

(*default = /home/user/.fti*)

6.1.6 ckpt_L1

Here, the user sets the checkpoint frequency of L1 checkpoints when using `FTI_Snapshot()`.

Value	Meaning
L1 intv. (int >= 0)	L1 checkpointing interval in minutes
0	Disable L1 checkpointing

(*default = 3*)

6.1.7 ckpt_L2

Here, the user sets the checkpoint frequency of L2 checkpoints when using `FTI_Snapshot()`.

Value	Meaning
L2 intv. (int >= 0)	L2 checkpointing interval in minutes
0	Disable L2 checkpointing

(*default = 5*)

6.1.8 ckpt_L3

Here, the user sets the checkpoint frequency of L3 checkpoints when using `FTI_Snapshot()`.

Value	Meaning
L3 intv. (int >= 0)	L3 checkpointing interval in minutes
0	Disable L3 checkpointing

(*default = 7*)

6.1.9 ckpt_L4

Here, the user sets the checkpoint frequency of L4 checkpoints when using `FTI_Snapshot()`.

Value	Meaning
L4 intv. (int >= 0)	L4 checkpointing interval in minutes
0	Disable L4 checkpointing

(*default = 11*)

6.1.10 dcp_L4

Here, the user sets the checkpoint frequency of L4 differential checkpoints when using `FTI_Snapshot()`.

Value	Meaning
L4 dCP intv. (int >= 0)	L4 dCP checkpointing interval in minutes
0	Disable L4 dCP checkpointing

(*default = 0*)

6.1.11 inline_L2

In this setting, the user chose whether the post-processing work on the L2 checkpoints is done by an FTI process or by the application process.

Value	Meaning
0	The post-processing work of the L2 checkpoints is done by an FTI process (notice: This setting is only alowed if head = 1)
1	The post-processing work of the L2 checkpoints is done by the application process

(*default = 1*)

6.1.12 inline_L3

In this setting, the user chose whether the post-processing work on the L3 checkpoints is done by an FTI process or by the application process.

Value	Meaning
0	The post-processing work of the L3 checkpoints is done by an FTI process (notice: This setting is only alowed if head = 1)
1	The post-processing work of the L3 checkpoints is done by the application process

(*default = 1*)

6.1.13 inline_L4

In this setting, the user chose whether the post-processing work on the L4 checkpoints is done by an FTI process or by the application process.

Value	Meaning
0	The post-processing work of the L4 checkpoints is done by an FTI process (notice: This setting is only alowed if head = 1)
1	The post-processing work of the L4 checkpoints is done by the application process

(*default = 1*)

6.1.14 keep_last_ckpt

This setting tells FTI whether the last checkpoint taken during the execution will be kept in the case of a successful run or not.

Value	Meaning
0	During `FTI_Finalize()`, all checkpoints will be removed (except case 'keep_l4_ckpt=1')
1	After `FTI_Finalize()`, the last checkpoint will be kept and stored on the PFS as a L4 checkpoint (notice: Additionally, the setting failure in the configuration file is set to 2. This will lead to a restart from the last checkpoint if the application is executed again)

(*default = 0*)

6.1.15 keep_l4_ckpt

This setting triggers FTI to keep all level 4 checkpoints taken during the execution. The checkpoint files will be saved in glbl_dir/l4_archive.

Value	Meaning
0	During `FTI_Finalize()`, all checkpoints will be removed (except case 'keep_last_ckpt=1')
1	All level 4 checkpoints taken during the execution, will be stored under `glbl_dir/l4_archive`. This folder will not be deleted during the `FTI_Finalize()` call.

(*default = 0*)

6.1.16 group_size

The group size entry sets, how many nodes (members) forming a group.

Value	Meaning
int i (2 <= i <= 32)	Number of nodes contained in a group (notice: The total number of processes must be a multiple of `group_size*node_size`)

(*default = 4*)

6.1.17 max_sync_intv

Sets the maximum number of iterations between synchronisations of the iteration length (used for `FTI_Snapshot()`). Internally the value will be rounded to the next lower value which is a power of 2.

Value	Meaning
int i (0 <= i <= INT_MAX)	maximum number of iterations between measurements of the global mean iteration time (`MPI_Allreduce` call)
0	Sets the value to 512, the default value for FTI

(*default = 0*)

6.1.18 ckpt_io

Sets the I/O mode.

Value	Meaning
1	POSIX I/O mode
2	MPI-IO I/O mode
3	FTI-FF I/O mode
4	SIONLib I/O mode
5	HDF5 I/O mode

(*default = 1*)

6.1.19 enable_staging

Enable the staging feature. This feature allows to stage files asynchronously from local (e.g. node local NVMe storage) to the PFS. FTI offers the API functions FTI_SendFile, FTI_GetStageDir and FTI_GetStageStatus for that.

Value	Meaning
0	Staging disabled
1	Stagin enabled (creation of the staging directory in folde 'ckpt_dir')

(*default = 0*)

6.1.20 enable_dcp

Enable differential checkpointing. In order to use this feature, ckpt_io has to be set to 3 (FTI-FF). To trigger differential checkpoints, use either level `FTI_L4_DCP` in FTI_Checkpoint or set the interval in dcp_L4 for usage in FTI_Snapshot.

Value	Meaning
0	dCP disabled
1	dCP enabled

6.1.21 dcp_mode

Set the hash algorithm used for differential checkpointing.

Value	Meaning
0	MD5
1	CRC32

(*default = 0*)

6.1.22 dcp_block_size

Set the desired partition block size for differential checkpointing in bytes. The block size must be within 512 .. `USHRT_MAX` (65535 on most systems).

Value	Meaning
b (512 <= i <= USHRT_MAX)	block size for dataset partition for dCP

(*default = 16384*)

6.1.23 verbosity

Sets the level of verbosity.

Value	Meaning
1	Debug sensitive. Beside warnings, errors and information, FTI debugging information will be printed
2	Information sensitive. FTI prints warnings, errors and information
3	FTI prints only warnings and errors
4	FTI prints only errors

(*default = 2*)

6.2 [Restart]

6.2.1 failure

This setting should mainly set by FTI itself. The behaviour within FTI is the following:

- Within `FTI_Init()`, it remains on it initial value.
- After the first checkpoint is taken, it is set to 1.
- After `FTI_Finalize()` and `keep_last_ckpt` = 0, it is set to 0.
- After `FTI_Finalize()` and `keep_last_ckpt` = 1, it is set to 2.

Value	Meaning
0	The application starts with its initial conditions (notice: In order to force a clean start, the value may be set to 0 manually. In this case the user has to take care about removing the checkpoint data from the last execution)
1	FTI is searching for checkpoints and starts from the highest checkpoint level (notice: If no readable checkpoints are found, the execution stops)
2	FTI is searching for the last L4 checkpoint and restarts the execution from there (notice: If checkpoint is not L4 or checkpoint is not readable, the execution stops)

(*default = 0*)

6.2.2 exec_id

This setting should mainly set by FTI itself. During `FTI_Init()` the execution ID is set if the application starts for the first time (failure = 0) or the execution ID is used by FTI in order to find the checkpoint files for the case of a restart (`failure` = 1,2)

Value	Meaning
yyyy-mm-dd_hh-mm-ss	Execution ID (notice: If variate checkpoint data is available, the execution ID may set by the user to assign the desired starting point)

(*default = NULL*)

6.3 [Advanced]

The settings in this section, should **ONLY** be changed by advanced users.

6.3.1 block_size

FTI temporarily copies small blocks of the L2 and L3 checkpoints to send them through MPI. The size of the data blocks can be set here.

Value	Meaning
int	Size in KB of the data blocks send by FTI through MPI for the checkpoint levels L2 and L3

(*default = 1024*)

6.3.2 transfer_size

FTI transfers in chunks local checkpoint files to PFS. The size of the chunk can be set here.

Value	Meaning
int	Size in MB of the chunks send by FTI from local to PFS

(*default = 16*)

6.3.3 general_tag

FTI uses a certain tags for the MPI messages. The tag for general messages can be set here.

Value	Meaning
int	Tag, used for general MPI messages within FTI

(*default = 2612*)

6.3.4 ckpt_tag

FTI uses a certain tags for the MPI messages. The tag for messages related to checkpoint communication can be set here.

Value	Meaning
int	Tag, used for MPI messages related to a checkpoint context within FTI

(*default = 711*)

6.3.5 stage_tag

FTI uses a certain tags for the MPI messages. The tag for messages related to staging communication can be set here.

Value	Meaning
int	Tag, used for MPI messages related to a staging context within FTI

(*default = 406*)

6.3.6 final_tag

FTI uses a certain tags for the MPI messages. The tag for the message to the heads to trigger the end of the execution can be set here.

Value	Meaning
int	Tag, used for the MPI message that marks the end of the execution send from application processes to the heads within FTI

(*default = 3107*)

6.3.7 lustre_striping_unit

This option only impacts if `-DENABLE_LUSTRE` was added to the Cmake command. It sets the striping unit for the MPI-IO file.

Value	Meaning
int i (0 <= i <= INT_MAX)	Striping size in Bytes. The default in Lustre systems is 1MB (1048576 Bytes), FTI uses 4MB (4194304 Bytes) as the dafault value
0	Assigns the Lustre default value

(*default = 4194304*)

6.3.8 lustre_striping_factor

This option only impacts if `-DENABLE_LUSTRE` was added to the Cmake command. It sets the striping factor for the MPI-IO file.

Value	Meaning
int i (0 <= i <= INT_MAX)	Striping factor. The striping factor determines the number of OST's to use for striping.
-1	Stripe over all available OST's. This is the default in FTI.
0	Assigns the Lustre default value

(*default = -1*)

6.3.9 lustre_striping_offset

This option only impacts if `-DENABLE_LUSTRE` was added to the Cmake command. It sets the striping offset for the MPI-IO file.

Value	Meaning
int i (0 <= i <= INT_MAX)	Striping offset. The striping offset selects a particular OST to begin striping at.
-1	Assigns the Lustre default value

(*default = -1*)

6.3.10 local_test

FTI is building the topology of the execution, by determining the hostnames of the nodes on which each process runs. Depending on the settings for `group_size`, `node_size` and `head`, FTI assigns each particular process to a group and decides which process will be Head or Application dedicated. This is meant to be a local test. In certain situations (e.g. to run FTI on a local machine) it is necessary to disable this function.

Value	Meaning
0	Local test is disabled. FTI will simulate the situation set in the configuration
1	Local test is enabled (notice: FTI will check if the settings are correct on initialization and if necessary stop the execution)

(*default = 1*)

CHAPTER

SEVEN

CONFIGURATION EXAMPLES

7.1 Default Configuration

```
[basic]
head                            = 0
node_size                       = 2
ckpt_dir                        = ./Local
glbl_dir                        = ./Global
meta_dir                        = ./Meta
ckpt_l1                         = 3
ckpt_l2                         = 5
ckpt_l3                         = 7
ckpt_l4                         = 11
dcp_l4                          = 0
inline_l2                       = 1
inline_l3                       = 1
inline_l4                       = 1
keep_last_ckpt                  = 0
keep_l4_ckpt                    = 0
group_size                      = 4
max_sync_intv                   = 0
ckpt_io                         = 1
enable_staging                  = 0
enable_dcp                      = 0
dcp_mode                        = 0
dcp_block_size                  = 16384
verbosity                       = 2

[restart]
failure                         = 0
exec_id                         = 2018-09-17_09-50-30

[injection]
rank                            = 0
number                          = 0
position                        = 0
frequency                       = 0

[advanced]
block_size                      = 1024
transfer_size                   = 16
general_tag                     = 2612
ckpt_tag                        = 711
```

(continues on next page)

(continued from previous page)

```
stage_tag                       = 406
final_tag                       = 3107
local_test                      = 1
lustre_striping_unit            = 4194304
lustre_striping_factor          = -1
lustre_striping_offset          = -1
```

DESCRIPTION

This configuration is made of default values (see: 5). FTI processes are not created (`head = 0`, notice: if there is no FTI processes, all post-checkpoints must be done by application processes, thus `inline_L2`, `inline_L3` and `inline_L4` are set to 1), last checkpoint won't be kept (`keep_last_ckpt = 0`), `FTI_Snapshot()` will take L1 checkpoint every 3 min,L2 - every 5 min, L3 - every 7 min and L4 - every 11 min, FTI will print errors and some few important information (`verbosity = 2`) and IO mode is set to POSIX (`ckpt_io = 1`). This is a normal launch of a job, because failure is set to 0 and `exec_id` is `NULL`. `local_test = 1` makes this a local test.

Using FTI Processes

```
[ Basic ]
head                            = 1
node_size                       = 2
ckpt_dir                        = /scratch/username/
glbl_dir                        = /work/project/
meta_dir                        = /home/username/.fti/
ckpt_L1                         = 3
ckpt_L2                         = 5
ckpt_L3                         = 7
ckpt_L4                         = 11
inline_L2                       = 0
inline_L3                       = 0
inline_L4                       = 0
keep_last_ckpt                  = 0
group_size                      = 4
max_sync_intv                   = 0
ckpt_io                         = 1
verbosity                       = 2
[ Restart ]
failure                         = 0
exec_id                         = NULL
[ Advanced ]
block_size                      = 1024
transfer_size                   = 16
mpi_tag                         = 2612
lustre_striping_unit            = 4194304
lustre_striping_factor          = -1
lustre_striping_offset          = -1
local_test                      = 1
```

DESCRIPTION

FTI processes are created (`head = 1`) and all post-checkpointing is done by them, thus `inline_L2`, `inline_L3` and `inline_L4` are set to 0. Note that it is possible to select which checkpoint levels should be post-processed by heads and which by application processes (e.g. `inline_L2 = 1`, `inline_L3 = 0`, `inline_L4 = 0`). L1 post-checkpoint is always done by application processes,

because it's a local checkpoint. Be aware, when `head = 1`, and `inline_L2`, `inline_L3` and `inline_L4` are set to 1 all post-checkpoint is still made by application processes.

7.2 Using only selected ckpt level with FTI_Snapshot

```
[ Basic ]
head                        = 0
node_size                   = 2
ckpt_dir                    = /scratch/username/
glbl_dir                    = /work/project/
meta_dir                    = /home/username/.fti/
ckpt_L1                     = 0
ckpt_L2                     = 5
ckpt_L3                     = 0
ckpt_L4                     = 0
inline_L2                   = 1
inline_L3                   = 1
inline_L4                   = 1
keep_last_ckpt              = 0
group_size                  = 4
max_sync_intv               = 0
ckpt_io                     = 1
verbosity                   = 2
[ Restart ]
failure                     = 0
exec_id                     = NULL
[ Advanced ]
block_size                  = 1024
transfer_size               = 16
mpi_tag                     = 2612
lustre_striping_unit        = 4194304
lustre_striping_factor      = -1
lustre_striping_offset      = -1
local_test                  = 1
```

DESCRIPTION

`FTI_Snapshot()` will take only L2 checkpoint every 5 min Notice that other configurations are also possible (e.g. take L1 ckpt every 5 min and L4 ckpt every 30 min).

7.3 Keeping last checkpoint

```
[ Basic ]
head                        = 0
node_size                   = 2
ckpt_dir                    = /scratch/username/
glbl_dir                    = /work/project/
meta_dir                    = /home/username/.fti/
ckpt_L1                     = 3
ckpt_L2                     = 5
ckpt_L3                     = 7
ckpt_L4                     = 11
inline_L2                   = 1
```

(continues on next page)

(continued from previous page)

```
inline_L3                   = 1
inline_L4                   = 1
keep_last_ckpt              = 1
group_size                  = 4
max_sync_intv               = 0
ckpt_io                     = 1
verbosity                   = 2
[ Restart ]
failure                     = 0
exec_id                     = NULL
[ Advanced ]
block_size                  = 1024
transfer_size               = 16
mpi_tag                     = 2612
lustre_striping_unit        = 4194304
lustre_striping_factor      = -1
lustre_striping_offset      = -1
local_test                  = 1
```

DESCRIPTION

FTI will keep last checkpoint (`Keep_last_ckpt = 1`), thus after finishing the job Failure will be set to 2.

7.4 Using different IO mode

For instance MPI-I/O:

```
[ Basic ]
head                        = 0
node_size                   = 2
ckpt_dir                    = /scratch/username/
glbl_dir                    = /work/project/
meta_dir                    = /home/username/.fti/
ckpt_L1                     = 3
ckpt_L2                     = 5
ckpt_L3                     = 7
ckpt_L4                     = 11
inline_L2                   = 1
inline_L3                   = 1
inline_L4                   = 1
keep_last_ckpt              = 0
group_size                  = 4
max_sync_intv               = 0
ckpt_io                     = 2
verbosity                   = 2
[ Restart ]
failure                     = 0
exec_id                     = NULL
[ Advanced ]
block_size                  = 1024
transfer_size               = 16
mpi_tag                     = 2612
lustre_striping_unit        = 4194304
lustre_striping_factor      = -1
```

(continues on next page)

(continued from previous page)

```
lustre_striping_offset      = -1
local_test                  = 1
```

DESCRIPTION

FTI IO mode is set to MPI IO (`ckpt_io = 2`). Third option is SIONlib IO mode (`ckpt_io = 3`).

7.5 Restart after a failure

```
[ Basic ]
head                        = 0
node_size                   = 2
ckpt_dir                    = /scratch/username/
glbl_dir                    = /work/project/
meta_dir                    = /home/username/.fti/
ckpt_L1                     = 3
ckpt_L2                     = 5
ckpt_L3                     = 7
ckpt_L4                     = 11
inline_L2                   = 1
inline_L3                   = 1
inline_L4                   = 1
keep_last_ckpt              = 0
group_size                  = 4
max_sync_intv               = 0
ckpt_io                     = 1
verbosity                   = 2
[ Restart ]
failure                     = 1
exec_id                     = 2017-07-26_13-22-11
[ Advanced ]
block_size                  = 1024
transfer_size               = 16
mpi_tag                     = 2612
lustre_striping_unit        = 4194304
lustre_striping_factor      = -1
lustre_striping_offset      = -1
local_test                  = 1
```

DESCRIPTION

This config tells FTI that this job is a restart after a failure (`failure` set to 1 and `exec_id` is some date in a format `YYYY-MM-DD_HH-mm-ss`, where `YYYY` - year, `MM` - month, `DD` - day, `HH` - hours, `mm` - minutes, `ss` - seconds). When recovery is not possible, FTI will abort the job (when using `FTI_Snapshot()`) and/or signal failed recovery by `FTI_Status()`.

CHAPTER

EIGHT

FTI EXAMPLES

8.1 Using FTI_Snapshot

```c
#include <stdlib.h>
#include <fti.h>

int main(int argc, char** argv){
    MPI_Init(&argc, &argv);
    char* path = "config.fti"; //config file path
    FTI_Init(path, MPI_COMM_WORLD);
    int world_rank, world_size; //FTI_COMM rank & size
    MPI_Comm_rank(FTI_COMM_WORLD, &world_rank);
    MPI_Comm_size(FTI_COMM_WORLD, &world_size);

    int *array = malloc(sizeof(int) * world_size);
    int number = world_rank;
    int i = 0;
    //adding variables to protect
    FTI_Protect(1, &i, 1, FTI_INTG);
    FTI_Protect(2, &number, 1, FTI_INTG);
    for (; i < 100; i++) {
        FTI_Snapshot();
        MPI_Allgather(&number, 1, MPI_INT, array,
              1, MPI_INT, FTI_COMM_WORLD);
    number += 1;
    }
    free(array);
    FTI_Finalize();
    MPI_Finalize();
    return 0;
}
```

DESCRIPTION

`FTI_Snapshot()` makes a checkpoint by given time and also recovers data after a failure, thus makes the code shorter. Checkpoints intervals can be set in configuration file (see: ckpt_L1 - ckpt_L4).

8.2 Using FTI_Checkpoint

```c
#include <stdlib.h>
#include <fti.h>
#define ITER_CHECK 10

int main(int argc, char** argv){
    MPI_Init(&argc, &argv);
    char* path = "config.fti"; //config file path
    FTI_Init(path, MPI_COMM_WORLD);
    int world_rank, world_size; //FTI_COMM rank & size
    MPI_Comm_rank(FTI_COMM_WORLD, &world_rank);
    MPI_Comm_size(FTI_COMM_WORLD, &world_size);

    int *array = malloc(sizeof(int) * world_size);
    int number = world_rank;
    int i = 0;
    //adding variables to protect
    FTI_Protect(1, &i, 1, FTI_INTG);
    FTI_Protect(2, &number, 1, FTI_INTG);
    if (FTI_Status() != 0) {
        FTI_Recover();
    }
    for (; i < 100; i++) {
        if (i % ITER_CHECK == 0) {
            FTI_Checkpoint(i / ITER_CHECK + 1, 2);
        }
        MPI_Allgather(&number, 1, MPI_INT, array,
              1, MPI_INT, FTI_COMM_WORLD);
    number += 1;
    }
    free(array);
    FTI_Finalize();
    MPI_Finalize();
    return 0;
}
```

DESCRIPTION

`FTI_Checkpoint()` allows to checkpoint at precise application intervals. Note that when using `FTI_Checkpoint()`, `ckpt_L1`, `ckpt_L2`, `ckpt_L3` and `ckpt_L4` are not taken into account.

8.3 Using FTI_Realloc with Fortran and Intrinsic Types

```fortran
program test_fti_realloc
    use fti
    use iso_c_binding
    implicit none
    include 'mpif.h'

    integer, parameter          :: dp=kind(1.0d0)
    integer, parameter          :: N1=128*1024*25  !> 25 MB / Process
    integer, parameter          :: N2=128*1024*50  !> 50 MB / Process
    integer, parameter          :: N11 = 128
    integer, parameter          :: N12 = 1024
```

(continues on next page)

(continued from previous page)

```
    integer, parameter          :: N13 = 25
    integer, parameter          :: N21 = 128
    integer, parameter          :: N22 = 1024
    integer, parameter          :: N23 = 50
    integer, target             :: FTI_COMM_WORLD
    integer                     :: ierr, status

    real(dp), dimension(:,:,:), pointer :: arr
    type(c_ptr)                 :: arr_c_ptr
    real(dp), dimension(:,:,:), pointer :: tmp
    integer(4), dimension(:), pointer   :: shape

    allocate(arr(N11,N12,N13))
    allocate(shape(3))

    !> INITIALIZE MPI AND FTI
    call MPI_Init(ierr)
    FTI_COMM_WORLD = MPI_COMM_WORLD
    call FTI_Init('config.fti', FTI_COMM_WORLD, ierr)

    !> PROTECT DATA AND ITS SHAPE
    call FTI_Protect(0, arr, ierr)
    call FTI_Protect(1, shape, ierr)

    call FTI_Status(status)

    !> EXECUTE ON RESTART
    if ( status .eq. 1 ) then
        !> REALLOCATE TO SIZE AT CHECKPOINT
        arr_c_ptr = c_loc(arr(1,1,1))
        call FTI_Realloc(0, arr_c_ptr, ierr)
        call FTI_recover(ierr)
        !> RESHAPE ARRAY
        call c_f_pointer(arr_c_ptr, arr, shape)
        call FTI_Finalize(ierr)
        call MPI_Finalize(ierr)
        STOP
    end if

    !> FIRST CHECKPOINT
    call FTI_Checkpoint(1, 1, ierr)

    !> CHANGE ARRAY DIMENSION
    !> AND STORE IN SHAPE ARRAY
    shape = [N21,N22,N23]
    allocate(tmp(N21,N22,N23))
    tmp(1:N11,1:N12,1:N13) = arr
    deallocate(arr)
    arr => tmp

    !> TELL FTI ABOUT THE NEW DIMENSION
    call FTI_Protect(0, arr, ierr)

    !> SECOND CHECKPOINT
    call FTI_Checkpoint(2,1, ierr)

    !> SIMULATE CRASH
```

(continues on next page)

(continued from previous page)

```
    call MPI_Abort(MPI_COMM_WORLD,-1,ierr)
end program
```

8.4 Using FTI_Realloc with Fortran and Derived Types

```
program test_fti_realloc
    use fti
    use iso_c_binding
    implicit none
    include 'mpif.h'

    !> DEFINE DERIVED TYPE
    type :: polar
        real :: radius
        real :: phi
    end type

    integer, parameter          :: dp=kind(1.0d0)
    integer, parameter          :: N1=128*1024*25  !> 25 MB / Process
    integer, parameter          :: N2=128*1024*50  !> 50 MB / Process
    integer, parameter          :: N11 = 128
    integer, parameter          :: N12 = 1024
    integer, parameter          :: N13 = 25
    integer, parameter          :: N21 = 128
    integer, parameter          :: N22 = 1024
    integer, parameter          :: N23 = 50
    integer, target             :: FTI_COMM_WORLD
    integer                     :: ierr, status
    type(FTI_type)              :: FTI_Polar

    type(c_ptr)                             :: cPtr
    type(polar), dimension(:,:,:), pointer :: arr
    type(polar), dimension(:,:,:), pointer :: tmp
    integer(4), dimension(:), pointer      :: shape

    !> INITIALIZE FTI TYPE 'FTI_POLAR'
    call FTI_InitType(FTI_Polar, 2*4, ierr)

    allocate(arr(N11,N12,N13))
    allocate(shape(3))

    !> INITIALIZE MPI AND FTI
    call MPI_Init(ierr)
    FTI_COMM_WORLD = MPI_COMM_WORLD
    call FTI_Init('config.fti', FTI_COMM_WORLD, ierr)

    !> PROTECT DATA AND ITS SHAPE
    call FTI_Protect(0, c_loc(arr), size(arr),FTI_Polar, ierr)
    call FTI_Protect(1, shape, ierr)

    call FTI_Status(status)

    !> EXECUTE ON RESTART
    if ( status .eq. 1 ) then
```

(continues on next page)

(continued from previous page)

```
        !> REALLOCATE TO DIMENSION AT LAST CHECKPOINT
        cPtr = c_loc(arr)
        call FTI_Realloc(0, cPtr, ierr) !> PASS DATA AS C-POINTER
        call FTI_recover(ierr)
        call c_f_pointer(cPtr, arr, shape) !> CAST BACK TO F-POINTER
        call FTI_Finalize(ierr)
        call MPI_Finalize(ierr)
        STOP
    end if

    !> FIRST CHECKPOINT
    call FTI_Checkpoint(1, 1, ierr)

    !> CHANGE ARRAY DIMENSION
    !> AND STORE IN SHAPE ARRAY
    shape = [N21,N22,N23]
    allocate(tmp(N21,N22,N23))
    tmp(1:N11,1:N12,1:N13) = arr
    deallocate(arr)
    arr => tmp

    !> TELL FTI ABOUT THE NEW DIMENSION
    call FTI_Protect(0, c_loc(arr), size(arr), FTI_Polar, ierr)

    !> SECOND CHECKPOINT
    call FTI_Checkpoint(2,1, ierr)

    !> SIMULATE CRASH
    call MPI_Abort(MPI_COMM_WORLD,-1,ierr)
end program
```

CHAPTER

NINE

DOXYGEN DOCUMENTATION

For a deeper dive into FTI's members, you can explore the documentation through Doxygen's auto-generated documentation.

9.1 Contributing

FTI uses Doxygen for generating its documentation. For a new function to be included in the documentation, it should be commented in the source code as follows:

```
/*-------------------------------------------------------------------------*/
/**
  @brief      Description of the function
  @param      arg1              arg1 meaning
  @param      arg2              arg2 meaning
  ...
  @return     integer           return value meaning

  Here goes an explanation of what the function does in details or how it
  relates to other functions in the project.

 **/
/*-------------------------------------------------------------------------*/
int myFunction(arg1, arg2...)
{
...
}
```

This is an example from FTI:

```
/*-------------------------------------------------------------------------*/
/**
  @brief      It sets/resets the pointer and type to a protected variable.
  @param      id              ID for searches and update.
  @param      ptr             Pointer to the data structure.
  @param      count           Number of elements in the data structure.
  @param      type            Type of elements in the data structure.
  @return     integer         FTI_SCES if successful.

  This function stores a pointer to a data structure, its size, its ID,
  its number of elements and the type of the elements. This list of
  structures is the data that will be stored during a checkpoint and
  loaded during a recovery. It resets the pointer to a data structure,
```

(continues on next page)

(continued from previous page)

```
  its size, its number of elements and the type of the elements if the
  dataset was already previously registered.

 **/
/*-------------------------------------------------------------------------*/
int FTI_Protect(int id, void* ptr, long count, FTIT_type type)
{
...
}
```

Upon contributing to FTI with a new API, it is recommended to have it show on the APIs pag. To do so, add the following lines `FTI_ROOT/docs/source/apireferences.rst` where `FTI_API` is the name of your function.

```
.. doxygenfunction:: FTI_API
:project: Fault Tolerance Library
```

For further details on how to include Doxygen's directives to ReadTheDocs page, visit the Breathe's guide.

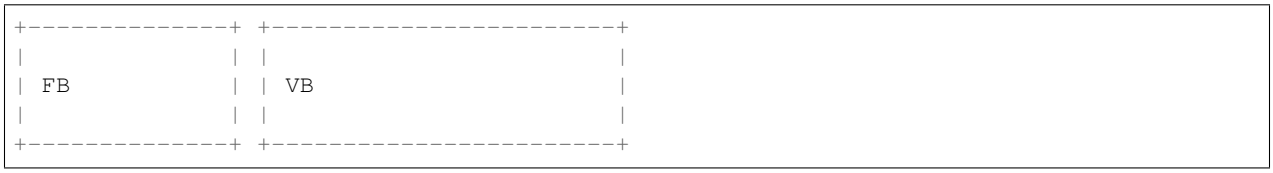
CHAPTER

TEN

FTI FILE FORMAT

10.1 FTI-FF Structure

The file format basic structure, consists of a meta block and a data block:

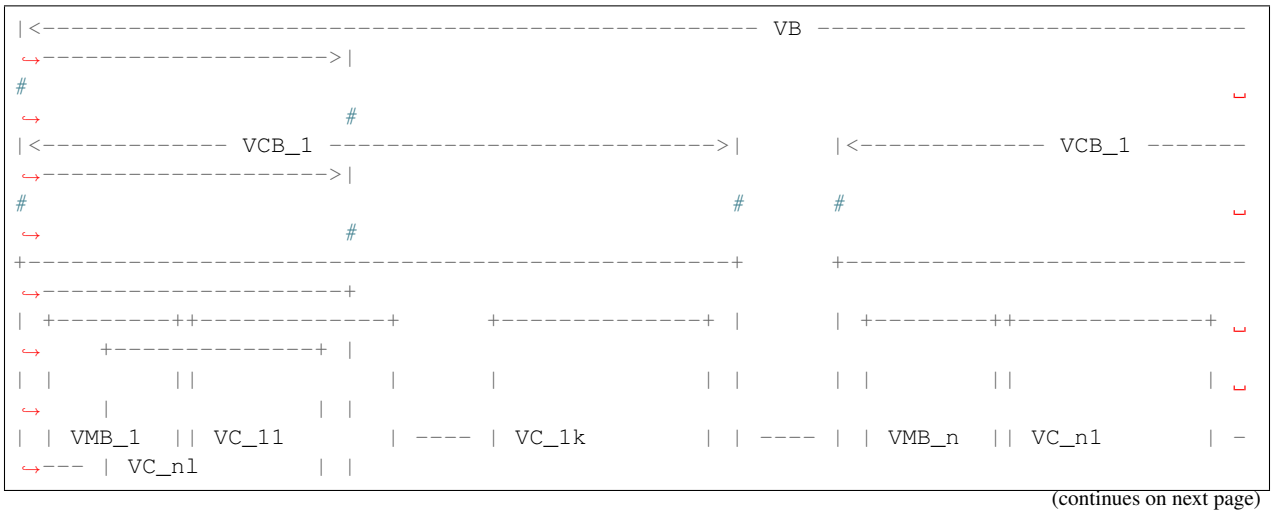


The FB (file block) holds meta data related to the file whereas the VB (variable block) holds meta and actual data of the variables protected by FTI.

The FB has the following structure:

```
FB {
        checksum        // Hash of the VDB block in hex representation (33 bytes)
        hash            // Hash of FB without 'hash' in unsigned char (16 bytes)
        ckptSize        // Size of actual data stored in file
        fs              // Size of FB + VB
        maxFs           // Maximum size of FB + VB in group
        ptFs            // Size of FB + VB of partner process
        timestamp       // Time in ns of FB block creation
}
```

The VB block possesses the following sub structure:



(continues on next page)

(continued from previous page)

```
| |         ||              |       |               | |       | |         ||              | ␣
↪    |               | |
| +--------++-------------+       +---------------+ |       | +--------++-------------+ ␣
↪    +---------------+ |
+-----------------------------------------------+       +-----------------------------
↪----------------------+
```

Where the `VMB_i` (variable meta data block) hold meta data related to the data chunk stored in `VDB_ij` (variable chunk). The number of data chunks (e.g. `k` and `l` in the scetch), generally may differ. We refer to the set `VMB_i`, `VC_i1`, ..., `VC_ik` as `VCB_i` (variable chunk block).

The `VMB_i` have the following sub structure:

```
|<--------------- VMB_i ------------->|
#                                      #
+-------++---------+       +----------+
|       ||         |       |          |
| BMD_i || VMD_i1  | ----  | VMD_ij   |
|       ||         |       |          |
+-------++---------+       +----------+
```

Where the `BMD_i` (block meta data) have the following structure:

```
BMD_i {
    numvars          // Number of variable chunks in data block
    dbsize           // Size of entire block VCB_i (meta + actual data)
}
```

The `VMD_ij` have the following structure:

```
VMD_ij {
    id               // Id of protected variable the data chunk belongs to
    idx              // Index of element in FTI_Data corresponding to protected␣
↪variable with id='id'
    containerid      // Id of container variable chunk is stored in
    hascontent       // Boolean value indicating if container holds data or not
    dptr             // Position of chunk in runtime-data (FTI_Data[idx].ptr + dptr)
    fptr             // Position of chunk in file
    chunksize        // Size of chunk stored in container
    containersize    // Total space in container
    hash             // Hash of 'VC_ij'
}
```

CHAPTER

ELEVEN

CODE FORMATTING

11.1 Code Checkers

To enhance the code quality of FTI, we use the following open source code checkers:

Language	Code Checker
C	cpplint
Fortran	fprettify
CMake	cmakelint

11.2 Coding Style

cpplint checks C/C++ files style issues following Google C/C++ style guide. Please visit this guide to understand how you should format your code to comply to FTI's style.

Fortran's and CMake style checkers have a plenty of formatting options, as the respective documentation lists. For FTI, we choose to adopt the following style rules:

Formatting options for Fortran Files

Options	Explanation
–indent 4	relative indentation width
–line-length 140	column after which a line should end
–whitespace 2	Presets for the amount of whitespace : 2
–strict-indent	strictly impose indentation even for nested loops

Formatting options for CMake Files

Options	Explanation
–line-width 80	How wide to allow formatted cmake files
–tab-size 4	How many spaces to tab for indent
–separate-ctrl-name-with-space	separate flow control names from their parentheses with a space
–separate-fn-name-with-space	separate function names from parentheses with a space

11.3 Implementation

Code checking is integrated in FTI through a script that traverses any added/modified code in FTI and checks if it conforms to the desired coding style. The script acts as a pre-commit hook that gets fired by a local commit.

Examples of the execution on FTI s code

```
bscuser@linux-uvh9:~/Documents/test_astyle/test_astyle/test> git commit -m "msg"
[Info] cpplint installed..
[Info] fprettify installed..
[Info] cmake-lint installed..

####################### C ######################
testC.c:0:  No copyright message found.  You should have a line: "Copyright [year] <Copyright Owner>"  [legal/copyright] [5]
testC.c:2:  Extra space before ( in function call  [whitespace/parens] [4]
testC.c:3:  Tab found; better to use spaces  [whitespace/tab] [1]
testC.c:4:  Weird number of spaces at line-start.  Are you using a 2-space indent?  [whitespace/indent] [3]
Done processing testC.c
Total errors found: 4

#################### FORTRAN ####################
[Error] testF.f90 does not respect the agreed coding style.
[Info] You can run fprettify on the file separately to see the expected formatting

#################### CMake ####################
testCMake.cmake
==============
testCMake.cmake:00: [C0303] Trailing whitespace
testCMake.cmake:04,02: [C0111] Missing docstring on function or macro declaration
testCMake.cmake:04,02: [C0305] too many newlines between statements
testCMake.cmake:04,11: [C0103] Invalid function name "APPEND_PROPERTY"
testCMake.cmake:04,27: [C0103] Invalid argument name "TYPE"
testCMake.cmake:05,07: [C0103] Invalid local variable name "APPEND_PROPERTY_TARGETS"
testCMake.cmake:06,06: [C0103] Invalid local variable name "APPEND_PROPERTY_PROPNAME"
testCMake.cmake:07,06: [C0103] Invalid local variable name "APPEND_PROPERTY_VALUES"
testCMake.cmake:08,04: [C0103] Invalid local variable name "APPEND_PROPERTY_CURRENT"
testCMake.cmake:09,08: [C0103] Invalid loopvar name "ARG"
testCMake.cmake:11,06: [C0103] Invalid local variable name "APPEND_PROPERTY_PROPNAME"
testCMake.cmake:12,07: [C0103] Invalid local variable name "APPEND_PROPERTY_CURRENT"
testCMake.cmake:14,07: [C0103] Invalid local variable name "APPEND_PROPERTY_CURRENT"
testCMake.cmake:19: [C0301] Line too long (107/80)
testCMake.cmake:19,09: [C0103] Invalid loopvar name "TARGET"
testCMake.cmake:20: [C0301] Line too long (137/80)

Summary
=======
files scanned: 1
found lint:
  Convention: 16




######################### Commit ####################
[Error] Some files need formatting. Please format them accordingly before trying to commit.
[Error] Failed to commit...
bscuser@linux-uvh9:~/Documents/test_astyle/test_astyle/test>
```

11.4 Contributing

Prerequisites

Before you will be able to contribute to FTI, you need to have the code checkers installed so that your code is checked prior to any commit. The checkers are easy to install if you have pip. For the latest installation steps, please visit the *Code Checkers*.

To make use of the pre-commit hook, after cloning the repository, one should initialize their branch through `git init` command.

This should port the pre-commit hook, along with the default git hooks, to your `GIT_DIR`

Note: Notice: For a temporary commit where the developer is aware that the code might still need formatting but still wants to commit, use the flag **–no-verify** along with the commit command.

CHAPTER

TWELVE

FTI INTEGRATED TESTS FRAMEWORK (FTI-ITF)

The FTI Integrated Test Framework (ITF) is a tool to develop black-box bash scripts to test FTI. ITF defines a standard procedure to develop FTI tests and allows a common design for FTI test cases. Given its nature, FTI tests interfaces with the operating system (e.g. to kill processes, delete, and corrupt files). ITF serves the purpose of abstracting these common operations into bash functions within a test engine.

ITF is the result of a refactoring effort to standardize FTI test scripts authored by multiple developers. As the original tests, the framework and its test cases are written in bash. ITF provides simple assertion features as in libraries like CUnit, JUnit, google test, and others. Moreover, ITF also contains FTI-specific functions to assist with configuration and checkpoint file manipulation.

This guide documents how to use ITF as well as how to develop and debug FTI test cases. The discussion is organized in a multi-level hierarchy of topics as follows: (i) FTI Test Design; (ii) User Guide; (iii) Test Developer Guide and (iv) ITF Internals. Topics 1 and 2 are recommended to any FTI contributor as to execute existing FTI tests. Topic 3 is recommended for FTI contributors working on new tests. Finally, topic 4 details and documents the ITF engine implementation. The latter is recommended for DevOps looking to enhance FTI integration and development pipelines.

12.1 FTI Test Design

FTI is a fault tolerance checkpointing library for distributed-memory applications. The library is used on MPI applications tailored to execute in high-performance environments. However, for testing purposes, application-specific computations can be ignored for the most part. Instead, FTI tests must guarantee the correct **interactions** among application, library, and system.

FTI uses black-box tests as the testing model to validate its features. In this model, FTI internal workings are abstracted. The tests focus on validating the system state between calls to an FTI-enhanced application. To implement this model, two major components are required, a **test application** and a **test script**. The **test application** is a software developed in C/C++, or Fortran, using FTI. The **test script** verifies the system state as well as set up and launch the test application.

12.1.1 Overview of Design Rules

FTI test applications are not yet subject to strict ruleset on how to be developed. However, each application must portray one target FTI feature to be tested under different configurations. On the other hand, the contents of test scripts follow the guidelines implemented by ITF. Among ITF goals, the framework attempts to reduce code duplication. Hence, ITF contains a library of standard procedures used when testing FTI used to: (i) register a test case; (ii) manage and launch MPI applications; (iii) manage FTI configuration files; (iv) disrupt FTI checkpoint files and (v) express assertions (i.e conditional and unconditional commands to pass/fail a test); The ITF standards are encapsulated in a software piece called **ITF engine**. Adhering to these standards creates a straightforward integration to CMake and continuous integration (CI) pipelines. An overview of the FTI test design is displayed in the Figure below.

12.1.2 Detailed Design Rules

ITF test scripts are also referred to as **ITF suite files**, or simply **suite files**. These scripts must, by convention, test a single FTI feature for better code quality. Each suite file contains one or more **test functions** and have the *.itf* filename extension. These test functions are bash equivalents of **parameterized test cases**. In other words, functions with a fixed behavior that generate unique test cases when subject to different *parameters*. In summary, **suite files** declare **test cases** by defining sets of *parameter values* for **test functions**.

Suite files are not meant to be executed directly from the terminal despite being bash scripts. They are similar to a library of test functions. Besides, they also contain a *declarative block* of bash code to register test cases. Ultimately, suite files depend on ITF code to execute and must be linked to its source.

A program called ITF test-driver puts every piece together so that tests can be executed. It creates a sandbox for each test case by orchestrating code from the ITF, its modules, and the suite file. In summary, the test-driver includes ITF modules and engine sources. Then, it calls the *declarative block* from the suite files and populates the engine with test cases. Next, it interacts with the *control interface* in the engine to execute the tests. Finally, the test-driver listens to engine events and collects the test results. A summary of these interactions is depicted in the Figure below.

12.2 ITF User guide

This section teaches FTI contributors to utilize the ITF test-driver program. The goal is that, by the end of the section, new contributors can: (i) execute the existing FTI test cases; (ii) cherry-pick test cases for execution from suites and (iii) filter out tests.

12.2.1 Executing FTI tests cases: ITF test-driver

ITF test-driver application is found in *build_dir/testing/itf/testdriver* after building FTI. The program has a *–help* argument to display all of its options and configurations. Its mandatory arguments are the paths to ITF suite files that contain the test cases to be executed. By convention, ITF suite files are located in the *build_dir/testing/suites* subdirectories. As an example, it is possible to execute the test cases for the **recoverVar** feature with the following script.

ITF test-driver can also execute multiple suite files at once and display accumulated statistics. The following command executes two suites, *recovervar* and *keepl4*.

The *testing/suites* directory is organized hierarchically. Each subdirectory represents a collection of ITF suites. The **compilation** collection is for tests targeting the compilation process of FTI. The **core** collection is for the main FTI features (i.e multi-level checkpointing). Finally, the **features** collection is for the additional FTI features. This organization allows for the use of Unix tools to execute groups of related tests. For instance, the following command is used to run the *core* suite collection.

12.2.2 Executing FTI tests cases: CTest

FTI uses CMake as the build manager for the library and its additional resources. CMake contains a tool, CTest, used to define tests to different CMake targets. As such, FTI tests can also be executed using the CMake *ctest* command. Suite files are populated with labels so that the *ctest* command is more appealing for users. For instance, the following command will execute all ITF suites in the *suites/core* folder.

The *-V* argument configures CTest to print the test-driver output to the terminal. By default, CTest suppresses all output from underlying commands. The *-L* argument limits the execution to tests that contains the **core** label. To check out all labels existing in FTI, use the *ctest –print-labels* command. As of now, there are individual tags for each suite plus the **core** and **features** for both suite collections.

CTest calls ITF test-driver program with pre-defined attributes under the hood. However, it also has its own set of features. For instance, the *–rerun-failed* argument will re-execute ITF suites that failed in a previous execution. This kind of functionality is out of the scope for the ITF test-driver due to its stateless design. Ultimately, we recommend using the test-driver for configuration flexibility and CTest for complete project validation.

12.2.3 Filtering test cases

An important part of debugging a program is reproducing the problem. However, by default, the test-driver executes all **test cases** in a given **suite file**. Imagine that a bug is found in a trait present only in a single **test function**. For instance, consider the *dCP* suite. It tests the differential checkpoint feature and contains two **test functions**: (i) *standard* and (ii) *corrupt_check*. Imagine now that a bug manifests only in the *corrupt_check* function. It is possible to invoke only this test function by using the following command.

The *–verbose* is optional and is used to see the application output as the **test cases** progress. The *–pick* argument for the test-driver program restricts the functions to be executed to the one supplied in the *format*. Moreover, the format, *suite_name:function_name*, is common to all filtering functions in ITF: (i) *suite_name* is the **suite file** name without the suffix and (ii) *function_name* is the test function name. In the previous example, the test-driver ignores all functions save for *corrupt_check*.

The *–pick* option can be passed multiple times to the test-driver. Each new instance appends the *format* to a list of allowed **test functions**. In other words, function names must be in at least one of the supplied formats to be executed. This behavior remains when the test-driver is processing multiple suites at once. For instance, both dCP and recovervar suites have test functions named *standard*. If we desire to run both standard functions but ignore *corrupt_check*, the following command must be used.

ITF also allows filtering out test functions in addition to cherry-picking by name. This function manifests as the *–ignore* option. Keep in mind that it is not possible to use both *–pick* and *–ignore* simultaneously. It is possible to rewrite the previous example using *–ignore* instead with the following commands.

Sometimes it is interesting to execute only one or a few **test cases** from a **test function** (i.e hunting a bug related to an IO library). In this case, these functions are available under the *–filter* and *–revfilter* options. The *filter* option is used to ignore **test cases** when a given argument value matches at least one value from a supplied set. The *revfilter* option is used to ignore **test cases** when a given argument value does not match any value from a supplied set. Both commands extend the *filter format* definition resulted in *suite_name:function_name:argument_name=value[,value2]...*. As an example, observe the following command using *–filter*.

The aforementioned command will execute the *recovervar* test suite. The suite is composed of a single function, *standard*. The *standard* test function has two parameters: (i) iolib and (ii) level. The *–filter* option signals the test-driver to ignore all test cases where the *iolib* parameter is assigned the values 1, 2, 3, or 4.

The *–revfilter* option works similarly, but it ignores test cases that **do not match** any of the values. It is possible to achieve the same results with both commands. For instance, we know that *recovervar:standard* contains test cases for iolib ranging from 1 through 5. Given the objective of executing test cases where iolib is 5, the following command can be used.

The filtering options for test cases can also be used multiple times in the same command. When applied to the same function, the filters are combined additively. That is, the conditions are added atop of each other. For instance, observe the following command.

The aforementioned filter will evaluate into a single test case being executed. That is, the *recovervar:standard* test case where *iolib* is 5 and *level* is 1.

The test-driver allows most filters to be combined. They have an order of precedence when being evaluated in the engine. First, the function-level filters are evaluated (i.e *–pick* or *–ignore*). Then, the *–revfilter* filter is applied, if present. Finally, the *–filter** option is evaluated, again, if present. Together, these options give the user complete control of which test cases are to be executed.

12.3 Test Developer Guide

The development of new tests stems from the creation of new features. A feature without full coverage can also be the reason behind a new test. Before developing the test, it is necessary to evaluate what needs to be tested. If the object of testing is a new feature, a whole new **suite file** is needed. Instead, if the object is another trait for an existing feature, a **test function** is needed. Sometimes, it is just necessary to accommodate another use case, which frequently fits in a new **test case** for an existing test. This section is dedicated to documenting the process of constructing a new suite file from scratch.

The public and private API regarding ITF functions are documented in-source. ITF source is scattered in different files, each containing its own set of functions. The core functionalities are found in *testing/tools/itf/src* while FTI-specific are located in *testing/tools/itf/modules*. This guide goes through the most simple functions but readers are advised to look into more advanced features.

12.3.1 Assertions API

There is a collection of example suites that can be found in *testing/suites/examples*. This suite collection is also installed in *build_dir/testing/suites/examples* if FTI is compiled with examples enabled. These suite files will serve as examples for the remainder of this section.

The **assertions.itf** example suite file demonstrates the assertion API exposed by ITF. The first two functions, *always_pass* and *always_fail* showcase unconditional assertion directives. Indeed, the *pass* directive marks the test as passed while the *fail* directive will mark the test as failed.

Every ITF test function **may** branch out to an assertion directive. If an assertion is not present at the last instruction, the test will be regarded as successful. Moreover, unconditional directives will terminate the test regardless of the resolution. This condition is not true for all conditional assertion directives. Examples of conditional assertions can be observed in the *pass_if_zero*, *fail_if_zero* and *check_assertion* functions.

ITF conditional directives are divided into two categories: (i) assertions and (ii) checks. Assertions will always finish the test regardless of the evaluation. Checks, on the other hand, will only terminate the test if the check fails. Besides, any directive that can cause the test to fail can be supplemented with an optional failure message.

All ITF assertions are implemented in the *testing/tools/itf/src/assertions* bash file. The source code is documented and fairly short. Nonetheless, a list of all available assertions and check functions is as follows.

Unconditional directives

- **pass**
  - Mark the test as passed and exit with 0;
- **fail**
  - Mark the test as failed and exit with 1;

Conditional assertion directives

- **assert_equals**
  - Evaluates two values and pass the test if both are equals, fail otherwise
- **assert_not_equals**
  - Same as assert_equals but pass the test if both values are different
- **assert_files_exists**
  - Pass the test if a given path represents a file, fail otherwise

Conditional check directives

- **check_equals**
  - Evaluates two values and fails the test if both are different
- **check_not_equals**
  - Evaluates two values and fails the test if both are equal
- **check_is_zero**
  - Fail the test if a given value is not zero
- **check_non_zero**
  - Fail the test if a given value is zero
- **check_files_exists**
  - Fail the test if a given path does not represent a file

12.3.2 Declaration Block API

The *declaration block* is a segment of Bash code to describe the suite file for the ITF engine. For instance, it combines bash function names and a set of parameters to create **test cases**. By convention, this segment of code is written at the end of the **suite file**.

The *ITF Declaration Block API* exposes commands to link bash names with actions in the ITF engine. The most simple action is to register a test case. This is done by using the *itf_case* command. It takes the test function name and a set of parameters as arguments to form the **test case**. For every call, ITF engine will register the function name as a **test function name**. Besides, the parameters are stored in a list of **test cases** associated with that test function.

One important aspect of testing is to guarantee that tests do not affect the results of others. This is usually done by creating a fixture, a pair of functions to execute before and after the test case. Fixtures guarantee that the starting conditions are set before a test. They also guarantee that everything created in the test case is destroyed before the next test. For FTI, it usually comes down to unsetting bash variables and destroying files/directories.

ITF has support for the declaration of fixture functions for test cases in the *Declaration Block API*. The *declaration_block.itf* example suite file showcases the use of this API. In the test suite, three test variants are exposed to perform a simple check. The behavior is to verify if a given string, passed by parameter, is present in a pattern file containing one entry per line.

The first test function in the **declaration_block** suite is *attempt_1*. This code creates the file within the scope of the function, makes the check, and never disposes of the file. The test cases for this function are registered in line 72 using the *itf_case* command. The system state before and after the execution is not the same, hence this test function is ill-written. The next attempt, *attempt_2*, uses *check* directives so that the test will not terminate on an assertion. It clears the pattern files when the test succeeds. However, the file is preserved if the test fails (i.e *check* directives terminate the test on failure).

The last attempt in the **declaration_block** suite, *attempt_3*, is a bit different. It uses a **setup function**, namely *create_patterns*, to create the pattern file and set up a bash variable with its path. The test is then executed normally. After the test is executed, a **teardown function** is called, namely *delete_aptterns*, to unset the bash variable and remove the file. These functions are registered per-function basis using the *itf_setup* and *itf_teardown* functions. Moreover, these two functions can be combined in a single command, *itf_fixture*. These commands associate a given **test function** to **fixture functions** that are called before and/or after each test case. Ultimately, these functions support the sandbox created to isolate one test from another regardless of their status.

ITF provide another declaration block function, *itf_suite_unload*. This command registers a function to be executed after **all test cases** for the suite has been executed. It is used to keep the sandbox environment among suites when using the test-driver to run multiple suite files. Use this function to unset variables, functions, and delete files created in the suite file outside the scope of the tests. Keep in mind that ITF tracks and unsets the **test functions**, so clearing these names are not necessary.

12.3.3 Parameters API

ITF provides a *parameter API* to describe a function's required parameters. It supports automatic parameter parsing, call validation, and bash variable creation procedures for **test functions**. The API supports code readability by shadowing the Bash naming convention for function parameters (i.e $1, $2, $3). Instead, it parses parameters out of order that follows the format: *–varname=value*.

The example suite **parameters.itf** showcase three different forms of validating parameters in **test functions**. The first method, *no_validation* is subject to wrong test function invocations. Line 23 attempts to register a test case without the second parameter, *password*. As expected, this test case fails when performing the assertion as there is no value assigned to *password*. The second test, *manual_validation* works as intended but adds extra behavior and complexity to the test. Besides, after executing the tests, the bash variables *username* and *password* will continue to exist.

The last attempt, *automatic_validation* makes use of *ITF Parameters API* function *param_parse*. This function enforces a standard on how to pass parameters to **test functions**. As noted, in lines 54-58, parameters are passed to the function following the aforementioned standard format. The **test function** redirect its parameters to *param_parse* using bash **$@** variable. Also, it exposes variable names that must be informed for the test using the *+varname* notation. In other words, line 47 states that the *automatic_validation* function requires the *username* and *password* variables assigned.

The *param_parse* command parses all arguments and automatically creates bash variables. If a required argument is not found, the test automatically fails (i.e Line 58). The test also fails if an argument is passed but not expected (i.e Line 56). Besides, these variables are marked to be managed by ITF. This means that they are automatically cleared after the execution of the test case, preventing them from leaking to other scripts.

12.3.4 FTI Module: Overview

ITF function library can be extended by linking modules using the *itf_load_module* command. An **ITF module** is similar to a **suite file** in the sense that it contains functions and a *declarative block*. However, instead of describing tests, the *declarative block* describes new ITF procedures and a method to unload them after processing the suite file. Modules are loaded per-suite, so, if multiple test suite requires a set of functions from a module, each suite must issue a call to *itf_load_module*. As of now, the only existing ITF module is called **FTI** and adds FTI common testing procedures.

The source code for ITF modules is located in *testing/tools/itf/modules*. Currently, the modules are configured by CMake and installed at *build_dir/testing/itf/modules*. ITF must be informed of the directory that holds its configured modules through an internal variable. This variable can be set through the test-driver using the *–path-modules [DIR]* argument. Moreover, the test-driver program located in FTI's build directory should have this argument correctly configured after compilation.

FTI module adds several procedures and bash variables to be used in **testing functions**. They are divided into three main categories: (i) FTI configuration file management API; (ii) MPI application management API and (iii) FTI checkpoint file query and disruption API.

12.3.5 FTI Module: Configuration File Management API

The example suite file **fti_cfg_api.itf** contains examples of how to use the FTI configuration file management API. In a nutshell, this API contains functions and variables to query and modify values to an FTI configuration file. When the FTI module is loaded, ITF creates a copy of an FTI configuration file with default settings for each test case in the suite. The filename for this copy can be accessed through the ITF configuration variable, **itf_cfg**, which is a bash associative array. To get the filename, query the *itf_cfg* variable for the key **fti:config**. This procedure is showcased in the *print_cfg_file* test function within **fti_cfg_api.itf** suite file.

The base configuration file used for ITF to create copies is located at *testing/tools/itf/resources/fti_template.cfg*. The copies can be modified during the test cases without affecting other tests. To do so, it is possible to use the API function *fti_config_set*. This function takes two parameters, the first in the field that will be modified and the second is the new value. A complete list of functions for this API is listed below and usage examples are found in the **fti_cfg_api.itf** suite file.

- **fti_config_set *field value*;**
  - Set a field in the FTI configuration file;
- **fti_config_get *field*;**
  - Get the value of a field from the FTI configuration file;
- **fti_config_set_inline;**
  - Sets all checkpoint levels to inline (i.e inline_l[1,2,3]=1).
- **fti_config_set_noinline;**
  - Sets all checkpoint levels to be carried out in parallel (i.e inline_l[1,2,3]=0).
- **fti_config_set_ckpts [l1] [l2] [l3] [l4];**
  - Set the checkpoint interval for different checkpoint levels (all parameters are optional).
- **fti_config_dupe *filename***
  - Creates a copy of the current FTI configuration file (the file is managed by the user).

12.3.6 FTI Module: MPI Application Management API

Another important procedure for testing FTI is launching MPI applications and parsing their logs. The FTI module contains functions to assist developers with this task. The list of functions for the MPI application management API, followed by a list of variables, is as follows.

Functions

- **fti_run *app* [app arguments]...**
  - Executes an MPI application with mpirun and stored its output in a temporary file.
- **fti_run_success *app* [app arguments]...**
  - Same as *fti_run* but will cause the test to fail if the application returns non-zero.
- **fti_check_in_log *msg***
  - Checks is a message exists in the last application log, fails the test if it does not exist.

- **fti_check_not_in_log** ***msg***
  - Same as *fti_check_in_log* but fails the test if the message **is present** in the log.
- **fti_assert_in_log** ***msg***
  - Same as *fti_check_in_log* but also passes the test if the message does exist.
- **fti_assert_not_in_log** ***msg***
  - Same as *fti_check_not_in_log* but also passes the test if the message does not exist.

Variables

All variables are keys contained in the *itf_cfg* associative bash array variable. * **fti:app_stdout**

- The full path to a file containing the stdout for the last application executed with *fti_run* or *fti_run_success*.
- **fti:nranks**
  - The rank count used to configure the mpirun command, resets to 16 every time the module is loaded.

The **mpi_api.itf** suite file contains examples for accessing and using these functions and variables. Other variables about verbosity, checkpoint file directory configuration, and others also exist. These were omitted for simplicity. For a complete list, check the fti module source code located in *testing/tools/itf/modules/fti*.

12.3.7 FTI Module: Checkpoint file query and disruption API

The checkpoint file query and disruption API provides high-level functions to interact with FTI checkpoint files. They rely on the ITF-managed configuration file to find the locations of the checkpoint files. The API methods can be used to assert that the files exist and to corrupt or delete them to set up a recovery scenario. A list with all the functions is displayed below.

- **ckpt_disrupt** ***disruption_type object_type level*** **[*id*]...**
  - Disrupt checkpoint objects from the application launched with *fti_run* or *fti_run_success*
- **ckpt_disrupt_all** ***disruption_type object_type level***
  - Disrupt all FTI checkpoint, partner files or node directories
- **ckpt_disrupt_first** ***disruption_type object_type level*** **[*node_id*]...**
  - Disrupt first checkpoint file in each node (sorted by name)
- **find_fti_objects** ***object_type level*** **[*id*]...**
  - Echo all checkpoints, partner file names, and node directories that match the rank ids.
- **find_fti_objects_all** ***object_type level***
  - Echo all FTI checkpoint, partner file names, or node directories in a given level.

Examples of how to use these methods can be found in suites from the features and core collections. The *features* collection contains the **diff_sizes.itf** suite, which applies this API in the *verify_log_disrupt* **test function**. The *core* collection contains the **standard.itf** suite, which applies this API in the *ckpt_disruption* **test function**.

12.3.8 Integrating test suites to CMake and CTest

Suite files must be integrated into CMake to be built alongside FTI. ITF facilitates this process through a CMake function called *DeclareITFSuite*. A usage example of this function can be observed in the *testing/suites/example/CMakeLists.txt* file.

The first parameter to the *DeclareITFSuite* CMake function is the **suite file name**. The other arguments are optional and denote the **CTest labels** associated with this suite of tests. Calling this function will copy the suite file to the build directory. Then, it creates a CTest check using the ITF test-driver and the suite annotated with the supplied labels.

As of now, the CI test-driver script executes all suites under the *suites/core* and *suites/features* folder. This means that the suite is automatically included in the CI pipeline depending on its target location. The CI test-driver program uses the ITF test-driver under the hood but has some more options itself. This script is located in *testing/tools/ci/testdriver* and is also written in bash.

12.4 ITF Internals

ITF is based on a modular design that, at its core, is implemented using the publisher/subscriber design pattern. The source files that configure ITF internal pillars are the *list*, *hooks*, *test*, and *itf* scripts in *testing/tools/itf/src*. From a design perspective, the ITF engine serves as a backend to process suite files and store state. The ITF test-driver application configures a front-end, or client, that starts the engine and subscribe to its events. Ultimately, ITF Modules (e.g FTI module) and core ITF features, like logging, work similarly as the test-driver.

ITF relies heavily on bash associative arrays to store information. This variable type is employed due to its similarity to C++ maps, Python dictionaries, or even objects in object-oriented languages. The *itf* source file glues every piece together and declare ITF main variable, *itf_cfg*. This variable stores global and specialized configurations using a standard naming convention for keys: *module:topic*.

ITF contains a custom implementation for lists that are represented using regular strings. This implementation can be found in the *list* source file. ITF string lists can be stored as the value counterpart of associative bash arrays key. As a result, it is possible to use bash associative arrays as indexed structures for organizing lists. This kind of structure is used to construct the notion of a *function callback*, known in ITF as **hooks**.

ITF hooks are described in the *hooks* source file. In this file, the variable *_itf_hooks* acts as the central queue for the publisher/subscriber design pattern. Each key to this variable represents a topic, or event, in ITF. On the other hand, the values are ITF lists which maps to bash function names to be called when those events are triggered.

ITF hooks are used in the engine to unify the different actions in ITF from executing tests to generating logs. These events can be subscribed to by ITF modules to add functionality. As an example, the FTI module creates and destroys FTI configuration files as a response to the events of starting and finishing a test case respectively. We believe this module will allow ITF to scale in feature count alongside FTI scope over time. A list of all current hook names, their description, and parameters passed to the functions can be found in the *test* file. Below is a list of the current ITF events alongside the parameters they yield to the callback methods.

- **onSuiteBegin** ***suitename***
  - Triggers when a suite is loaded
- **onSuiteEnd** ***suitename***
  - Triggers after all the tests in a suite have been executed
- **onTestLoad** ***testname ncases***
  - Triggers before starting the test cases for a given test function
- **onTestRunBegin** ***testname params***
  - Triggers before every test case setup function

- **onTestRunEnd** *testname params*
  - Triggers after every test case teardown function
- **onTestPass** *message*
  - Triggers wehenever a test passes
- **onTestFail** *message*
  - Triggers wehenever a test fails

ITF events are designed to link both front-end (i.e test-driver) and back-end (i.e ITF feature scripts and modules) components. They are not meant to be used in tests or suite files. The table below documents the actions were taken by each ITF component when a given event is raised.

Events	ITF Core	Log Feature	FTI Module	Test-driver
onSuiteBegin	•	Prepare new log file	•	Print feedback
onTestLoad	•	•	•	Print feedback
onTestRunBegin	•	Save test parameters	Create FTI config-file	Print feedback
onTestRunEnd	•	•	Destroy FTI config-file and ckpt dirs	•
onTestPass	•	Log parameters and output	•	Print feedback
onTestFail	•	Log parameters and output	•	Print feedback
onSuiteEnd	Unload suite modules	•	•	Print feedback

Apart from the events, ITF also has some internal variables to control the progress and configuration of the program. The remainder of this section is dedicated to describing the variables and their respective key semantics.

Variable itf_cfg

A unified point to contain all user-configurable variables.

- **core:dry_run: boolean string**
  - When set, only account for sets but do not execute them.
- **core:verbose: boolean string**
  - When set, configures ITF to print the test stdout to the terminal.
- **core:stdout: file path**
  - The temporary file populated by the test stdout.
- **core:module_path: directory path**
  - The directory where ITF will look for modules when prompted by *itf_load_module*.
- **core:assert_msg_buf: file path**
  - Temporary buffer to store the fail message raised by checks and asserts.
- **log:failed_cases: boolean string**

  - When set, create log files with the stdout of tests that failed.
- **log:passed_cases: boolean string**
  - When set, create log files with the stdout of tests that passed.
- **fti:nranks: integer**
  - The amount of MPI ranks to spawn when launching an application with *fti_run* and *fti_run_success*.
- **fti:verbose: boolean string**
  - When unset, suppress FTI module actions output to the terminal.
- **fti:verbose_log: boolean string**
  - When unset, suppress FTI module actions output to the test log.
- **fti:verbose_app: boolean string**
  - When set, the FTI module will pipe the MPI application stdout to the terminal.
- **fti:app_stdout: file path**
  - Temporary buffer for the last MPI application run output.
- **fti:config_template: file path**
  - Source FTI configuration file used to generate copies for ITF test cases.
- **fti:config: file path**
  - The FTI configuration file copy path for the current test case.
- **fti:keep_ckpt_dir: boolean string**
  - When set, the FTI module will not erase the checkpoint directories after executing a test case.

Variable itf_state

The internal state of the engine when processing and running a suite file. The values here are reset every time a suite is loaded.

- **ntests**
  - The number of tests in the current suite file.
- **failed**
  - The number of tests that failed in the current suite file.
- **suite_teardown**
  - The function to call when ITF is done executing the current suite.
- **suite_name**
  - The current suite file name without the extension and path.

Variable _itf_setup

A map that associates test function names with their respective setup functions.

Variable _itf_teardown

A map that associates test function names with their respective teardown functions.

Variable _itf_cases

A map that associates test function names with ITF string lists containing their parameters.

Variable itf_filter

The variable that holds filters for the current ITF engine execution.

- **blacklist**
  - ITF string list with function names that must be ignored by the engine.
  - If this list is empty, every test function is considered.
- **whitelist**
  - ITF string list with the function names that are allowed to be processed by the engine.
  - If this list is empty, every test function is considered.

CHAPTER

THIRTEEN

TEST SUITES

FTI is bundled with a set of testing programs and scripts to execute them. The combination of script and a testing program is called a **test suite**. The test suites validate, by design, a single FTI feature under different conditions. Moreover, FTI contains several test suites to achieve higher software quality.

The **test suites** scripts are developed using the **FTI Integration Test Framework (ITF)**. In this format, **test functions** define how to validate different aspects of the target feature. This is done by creating multiple **test cases** by variating parameters that might affect the feature's behavior. For more information about how to implement these concepts, we recommend reading the *ITF documentation*. This page is devoted to explaining which test suites exist and what they validate.

We divide the test suite in three **test categories**: (i) core functionalities; (ii) additional features and (iii) compilation/build. Indeed, the suites are grouped by the type of feature they validate. This organization is expressed in the testing folder hierarchy. Hence, the suites are bundled in hierarchical directories with the root in *testing/suites*. A brief description of every suite in each category can be found in the table below.

Suite Name	Focus on testing	Test Cases
Core Functionality Suite		
multiLevelCkpt	Multi-level checkpointing	640
cornerCases	Supported corner cases	56
useCases	Simplified use-case scenarios	18
syncIntv	Runtime-aware checkpoint interval	1
ckptDiffSizes	Different checkpoint sizes per rank	96
keepL4Ckpt	Archive checkpoint in PFS	4
Additional Features Suite		
dCP	Differential Checkpointing	10
vpr	Variate Processor Restart	8
recoverName	Recover Variable per Name	20
recoverVar	Recover Variable per Id	20
staging	Staging Feature	2
getConfig	Manipulate FTI configurations	5
hdf5	HDF5 support and sanity checks	12
Compilation and Build Suite		
cmake_versions	Build with different CMake versions	13

13.1 Core functionality test category

The *core* category is attributed to test suites that validate the main FTI behavior. In other words, this category applies mainly to the multi-level checkpoint feature. Furthermore, we added into this category other FTI behaviors that support the multi-level functionality.

13.1.1 Multi-level checkpoint

The multi-level checkpoint suite is located in the *testing/suites/core/multiLevelCkpt* folder. The ITF suite file is declared under the name *standard.itf*. The suite is composed of two testing functions: (i) **normal_run** and (ii) **ckpt_disruption**.

The *normal_run* function checks the expected behavior of FTI under normal circumstances. The function simulates an application crashing and re-starting to verify FTI checkpoint/restart behavior.

The *ckpt_disruption* function checks the expected behavior of FTI when the checkpoint files are disrupted (i.e erased or corrupted). It follows the same flow as *normal_run* but the checkpoint files are disrupted before the second application run. This function simulates scenarios where FTI is supposed to both fail and succeed in recovering the application state.

Warning: The tests which cause FTI to fail are currently disabled in the CI environment due to unexpected MPI hanging.

13.1.2 FTI corner cases

The corner cases suite is located in the *testing/suites/core/cornerCases* folder. The ITF suite file is declared under the name *corner_cases.itf*. The suite is composed of corner case scenarios regarding the consistency and hierarchy of checkpoint files. There are three scenarios regarding the consistency aspect represented as the test functions: (i) *ckpt_consistency*; (ii) *keep_last_consistency* and (iii) *double_fti_init*.

The *ckpt_consistency* tests check if FTI creates consistent checkpoint **and** partner files. The checks validate if FTI can recover from one group of files when the other is corrupted. Then, a new set of checkpoint files is created and another group is corrupted. The test validates if all application states are consistent, regardless of the recovery strategy.

The *keep_last_consistency* tests is similar to *ckpt_consistency*. However, instead of simulating crashes to test the recovery, the application finishes and stores its last checkpoint on the PFS. Then, the function asserts that FTI uses the last checkpoint from PFS when a re-run is issued with the same configuration file.

The *double_fti_init* asserts that FTI is capable of functioning if the initialization function is called twice. Moreover, this function mimics a live restart and/or protection of individual application segments.

The remainder test functions are related to the hierarchical relationship between checkpoint levels. There are two test functions targeting these corner cases: (i) *subsequent_checkpoints* and (ii) *subsequent_ckpts_restart*. FTI is expected to overwrite less recent files depending on the order the checkpoints are taken. Hence, the former function asserts that FTI maintains the most secure checkpoint after taking subsequent checkpoints. Finally, the *subsequent_ckpts_restart* function asserts that FTI restores from the most recent non-corrupted checkpoint.

13.1.3 FTI use cases

The use cases suite is located in the *testing/suites/core/useCases* folder. The ITF suite file is declared under the name *use_cases.itf*. The suite is composed of three applications that simulate a simplified use case for FTI. These tests can be considered as true integration tests given that they are based on mini-kernels. There are two test functions on this test case: (i) *nodeflag* and (ii) *simulated_use_cases*.

13.1.4 Synchronization interval

The synchronization interval suite is located in the *testing/suites/core/syncIntv* folder. The ITF suite file is declared under the name *sync_intv.itf*. It contains only one function, *checkpoint_interval*. This test executes a 3d heat distribution kernel. Furthermore, the function asserts that checkpoints are taken in the correct application iterations and time intervals.

13.1.5 Ranks with different checkpoint sizes

The *ckptDiffSizes* suite is located in the *testing/suites/core/ckptDiffSizes* folder. The ITF suite file is declared under the name *diff_sizes.itf*. This suite checks if FTI is capable of checkpointing ranks with different checkpoint sizes. It contains two test functions: (i) *verify_log* and (ii) *verify_log_disrupt*. Both functions use FTI logs to assert that all the data is being checkpointed regardless of the difference in size. The latter check also adds disruption to the checkpoint files between application runs.

13.1.6 Keep level 4 checkpoints

The *keepL4Ckpt* suite is located in the *testing/suites/core/keepL4Ckpt* folder. The ITF suite file is declared under the name *keepl4.itf*. It contains a single test function, *standard*. The function asserts that FTI pushes the L4 checkpoint into an archive when configured to do so.

13.2 Additional features test category

The *feature* test category applies to test suites that validate FTI features beyond the scope of the main checkpoint/restart feature. Those are variations for API functions, support for IO libraries, and other non-essential functionalities. Test suites that adhere to this category are located under the *testing/suites/features* folder.

13.2.1 Differential Checkpointing

The differential checkpoint suite is located in the *testing/suites/features/differentialCkpt* folder. The ITF suite file is declared under the name *dCP.itf*. It contains two test functions: (i) *standard* and (ii) corrupt_check;

The *standard* test function asserts the differential checkpoint encodes the correct amount of data. The *corrupt_check* function asserts that FTI can recover from corrupted differential checkpoint data.

Note: The *standard* function implements the checks for POSIX and FTI IO modes.

13.2.2 Variate Processor Restart

The variate processor restart suite is located in the *testing/suites/features/variateProcessorRestart* folder. The ITF suite file is declared under the name *vpr.itf*. It contains one test function, *standard*.

The *standard* function asserts that FTI is capable of restarting an application in a different number of ranks.

Note: The *standard* function only verifies the behavior for the HDF5 IO library.

13.2.3 Recover variable by name

The *recover-name* suite is located in the *testing/suites/features/recoverName* folder. The ITF suite file is declared under the name *recovername.itf*. It contains one test function, *standard*. The function asserts that FTI can correctly recover variables given their name.

Warning: This functionality is not enabled for FTI IO mode and is disabled in the CI environment.

13.2.4 Recover variable by id

The *recover-var* suite is located in the *testing/suites/features/recoverVar* folder. The ITF suite file is declared under the name *recovervar.itf*. It contains one test function, *standard*. The function asserts that FTI can correctly recover variables given a numeric id.

13.2.5 Staging API

The *staging* suite is located in the *testing/suites/features/staging* folder. The ITF suite file is declared under the name *staging.itf*. It contains one test function, *standard*. The function asserts the correct functioning of the staging functionality. In other words, it asserts that FTI can push files to the PFS in the background as requested by the application.

13.2.6 GetConfig API

The *GetConfig* suite is located in the *testing/suites/features/getConfig* folder. The ITF suite file is declared under the name *getconfig.itf*. It contains one test function, *standard*. This test asserts that FTI can retrieve the configuration file contents during runtime.

13.2.7 HDF5 support

The *hdf5* suite is located in the *testing/suites/features/hdf5* folder. The ITF suite file is declared under the name *hdf5.itf*. It contains onde test functions, *hdf5_test*. This test asserts that FTI yields correct HDF5 structures when issuing HDF5 checkpoint files.

13.3 Compilation test category

The *compilation* test category applies to test suites that validate the FTI build process. Test suites that adhere to this category are located under the *testing/suites/compilation* folder. As of now, there is only one test suite in this category: **cmake_versions**.

The *CMake versions* test suite is used to test FTI compilation under different CMake versions. It is used to guarantee the build process portability from the minimum CMake required version up to more recent ones. This test is tailored to function in the FTI CI environment. Thus, reproducibility will involve changing the behavior of the test so it can find the installed CMake binaries.

CHAPTER

FOURTEEN

FTI CONTINUOUS INTEGRATION ENVIRONMENT

FTI follows the guidelines of Continuous Integration (CI) development process. In this scheme, small new additions are included in Pull Requests (PR) to the *develop* branch on github. The additions must be automatically validated through the *test suites* prior to integration. This is achieved using Jenkins to define a development pipeline.

Jenkins pipeline is defined in the **JenkinsFile** in FTI root folder. This pipeline is executed by a Jenkins server configured to test every PR sent to FTI's github repository. In most cases, Jenkins will compile FTI with the GCC compiler and all IO libraries and execute all test suites. Furthermore, if the PR targets the master branch, this process is repeated for these additional compilers: (i) PGI; (ii) CLang and (iii) Intel.

The FTI CI environment is composed of all libraries and software tools involved in the build and testing pipeline. The CI environment is contained in a Docker image so that these software pieces can be easily managed and duplicated. Docker is a software to create a sandbox to execute software in the form of containers. The current image used in FTI is named **alexadrelimasssantana/fti-ci:latest**. Jenkins uses this image to instantiate a container and run the pipeline without conflicting with the host machine software. To donwload FTI docker image, make sure you have docker installed and perform the following command.

14.1 Docker Image software stack

The FTI Docker image is based on the official x86_64 ArchLinux image. The image is extended to contain the majority of software required to: (i) build FTI with all supported libraries; (ii) execute all FTI test suites and (iii) generate code coverage reports. A non-exhaustive list of software in the Docker image, annotated with its installation procedure, is as follows.

Libraries - OpenMPI-4.0.3 (pacman) - HDF5-1.12 (pacman) - SIONLib-1.7.6 (compiled from source and installed in /opt/sionlib)

Compilers - CLang-10 (pacman) - gcc-10.1 (pacman) - gccfortran-10.1 (pacman)

Tools - CMake-3.17 (pacman) - DiffUtils (pacman) - make-4.3 (pacman) - python-3.8 (pacman) - gcovr-4.2 (pacman) - git-2.27 (pacman)

We recommend getting to know a bit about Docker for further details on the CI environment. A good place to start is the Docker official tutorial page. Moreover, most commands used in this guide will be shortly explained. Granted that docker is already installed, the following command can be used to run the Docker image.

The command will create and run a container named **fti** based on FTI DockerHub image. The container will be executed in daemon mode and be available for connections. To connect to the container, issue the following command.

This command will connect to the **fti** container and execute the bash application. The *-it* flag informs docker that this is an interative session. Once inside the container, you can clone the fti repository using git and checkout to any specific branch.

It is possible to replicate the GCC and CLang stages of the CI pipeline with this current setup. This can be done by performing the same commands as depicted in those stages. For the sake of simplicity, we added the command in the

snippet below for the GCC compiler stage. Those steps are not required to be executed, they are included here for demonstration only.

As mentioned earlier, this environment is not able yet to run the **Intel**, **PGI** and **Compilation Checks** CI stages. This is due to the fact that these compilers are proprietary and require licenses to use. As such, it is not possible to redistribute them within containers. For the compilation checks, the stage lacks additional CMake installations. We chose to let those out of the docker as to minimize its size. These software pieces must be incorporated into the containers using Docker volumes.

14.2 Docker image: required volumes

Docker volumes are, in essence, filesystem bindings between the host machine and the container. Volumes can store persistent data in the host machine and allow the container to use it. FTI employs three volumes to fulfill its CI pipeline: (i) cmake-versions; (ii) pgi-compiler and (iii) intel-compiler.

In order to fully replicate the CI environment, these three volumes must be mounted in the docker container. We will go over the process of creating these volumes using your own licenses and CMake installations.

14.2.1 cmake-versions

The *cmake-versions* volume is a requirement of the **Compilation Checks** CI stage. This volume contains multiple CMake version installations that are used in the *cmake_versions* test suite. The CI pipeline expects this volume to be a filesystem with one folder for each CMake version that is supported by the test suite. Each folders must be named under the CMake version preppended with a 'v' character (e.g v3.10, v3.3.2). As of now, these are the CMake versions that must be present in the cmake-versions volume folders: (i) 3.3.2; (ii) 3.4; (iii) 3.5 (iv) 3.7; (v) 3.8; (vi) 3.9; (vii) 3.10; (viii) 3.11; (ix) 3.12; (x) 3.13; (xi) 3.14; (xii) 3.15 and (xiii) 3.16.

Docker volumes are managed by the Docker application. As such, they need to be populated by a container. To create the cmake-versions volume, run the FTI container with the following parameters.

This command will create a new container, **make-cmake**, and a new volume **cmake-versions** mounted in */opt/cmake*. Now, all we must do is to populate the container with the CMake installations. For that, you will need to connect to the container which can be done with the following command.

Now that you are inside the container, verify if the volume is mounted in */opt/cmake* with the *ls* command. You should see an empty folder, for now. Now, to populate the volume, we need to build and install the multiple CMake versions there. One of the ways to do this, is to clone the CMake github directory and build all versions from the source. Fortunately, the build part can be using CMake which is already installed in the Docker Image. The following script will install the first two required CMake versions (i.e 3.3.2 and 3.4) in the mounted volume.

The aforementioned script can be used to build and install all versions. To do that, simply append the other versions into the **versions** bash array. After running this script for all versions, the volume should be ready to use. To check if everything is in order, you can manually run the **cmake_versions** test case with the following command.

This script will run the compilation suite which will only succeed if all CMake versions where installed correctly. If everything went well, you can exit the container and the volume will persist in the host machine. It is important to remember that you need to launch the container with the volume mounted everytime you need to run this stage. The following command will do this.

14.2.2 pgi-compiler

The *pgi-compiler* volume is a requirement of the **PGI** CI stage. This volume should contain an installation of the PGI community edition compiler and license. To build this volume, you first need to download the PGI compiler in this link. After this step, get a docker running with the following command.

The command will create an image named **make-pgi** and a new volume, **pgi-compiler** in */opt/pgi*. We need to install the PGI compiler inside the volume just as we did with the CMake versions. However, this time we downloaded the compressed compiler in the host machine.

Note: The PGI website does not provide a link to download the compiler with wget.

To copy the tar file into the container, we can use the *docker cp* command. The following snippet exemplifies this.

Now we can connect to the docker, unpack the compiler and run its install script. Using the default options should be enough, just make sure to install the compiler at the volume in */opt/pgi*. To verify if the installation went out correctly, try to build FTI with PGI using the following command.

This should build FTI using the pgi compiler found in /opt/pgi. If the compilation fails, check if the paths in the *build.sh* matches the PGI compiler version. As of now, the script will rely on version *19.10* of the compiler. Once you are done, you can leave the container and the volume will persist in the host machine.

14.2.3 intel-compiler

The *intel-compiler* volume is a requirement of the **Intel** CI stage. This volume must contain an installation of the Intel C/C++ compiler. The volume is mounted at */opt/intel* and should contain a valid license in */opt/intel/licenses/*.

The first step to build the *intel-compiler* volume is to download and install the compiler in the host machine. Using the default options, the installation will install the package in */opt/intel*. After this step, get a docker running with a bind and a volume mounts with the following command.

The aforementioned command will create two mounts, a volume in */opt/intel* and a bind in */opt/intel-host*. The volume will be empty while the bing will contain the host's installation of the Intel tools. To populate the volume, simply copy all files from the bind mount to the volume using the following command.

To test the new volume, instantiate a new docker and mount only the *intel-compiler* volume.

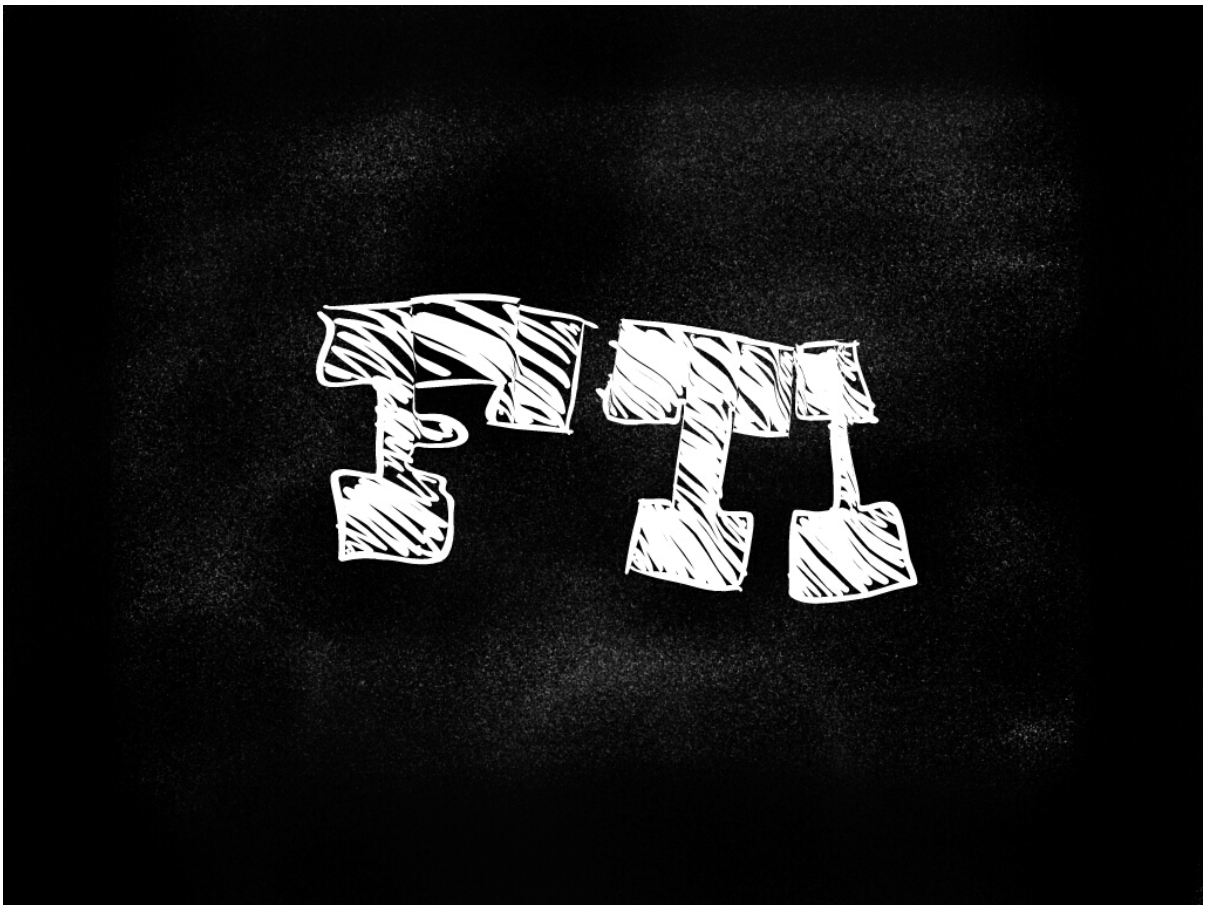
Connect to the docker container, pull fti from github and check if the build script is working.

This command is the same that is used in Jenkins and should be able to compile FTI using ICC. After this step, the container is ready to run the tests and check FTI's behavior.

Code author: Dr. Leonardo Bautista-Gomez (Leo) <*leonardo.bautista@bsc.es*>

Barcelona Supercomputing Center Carrer de Jordi Girona, 29-31, 08034 Barcelona, SPAIN

Phone : +34 934 13 77 16



INDEX

F