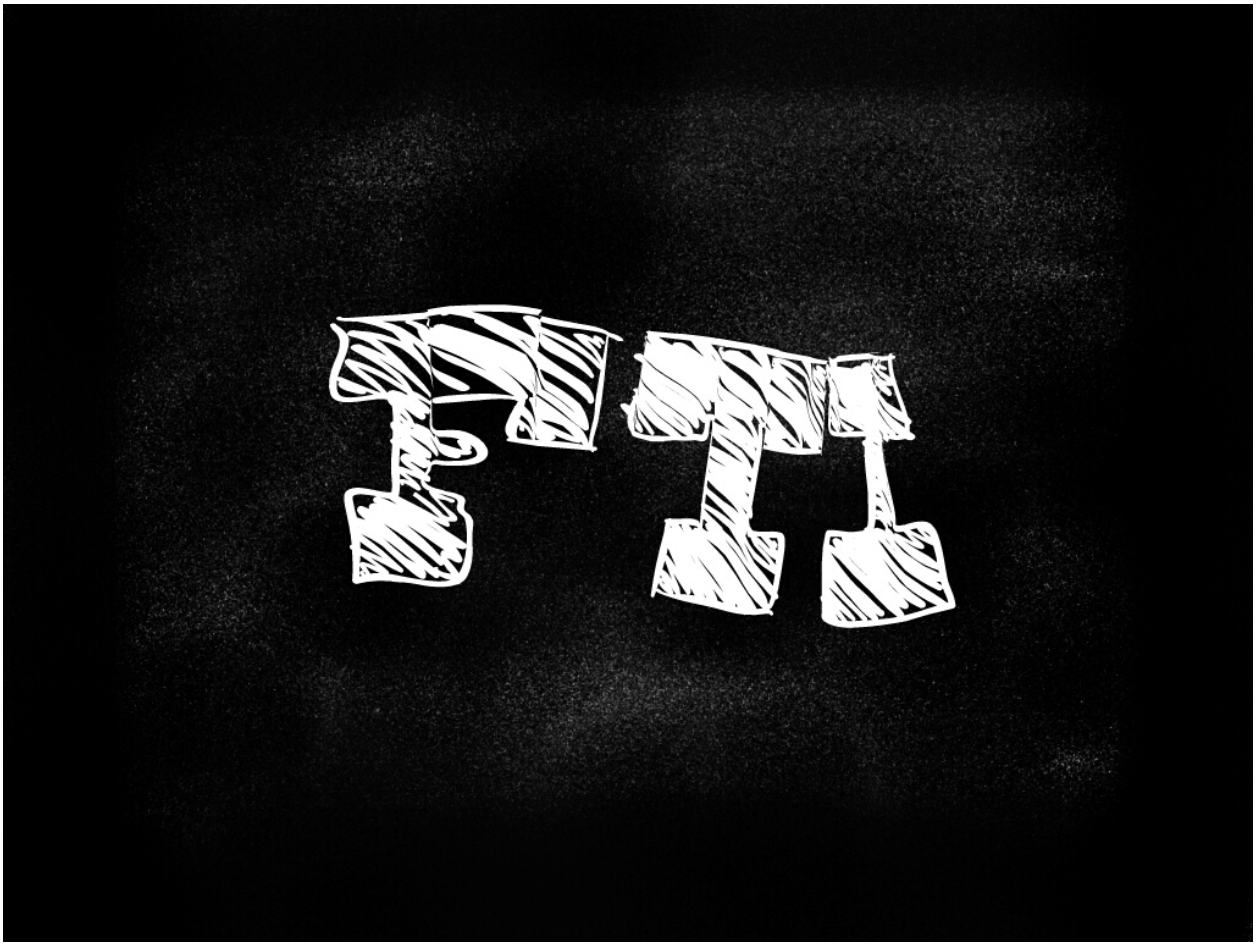
FTI - Fault Tolerance Interface

Contact:
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



Authors: Kai Keller, Tomasz Paluszkiewicz

Release: 0.9.9

Introduction

In high performance computing (HPC), systems are built from highly reliable components. However, the overall failure rate of supercomputers increases with component count. Nowadays, petascale machines have a mean time between failures (MTBF) measured in hours or days 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 will have a significantly larger number of components and an important amount of memory. This will have 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. Although the overall failure rate of future post-petascale systems is a common factor to study when designing FT-techniques, another important point to take into account is the pattern of the failures. Indeed, when moving from 90nm to 16nm technology, the soft error rate (SER) is likely to increase significantly, as shown in a recent study from Intel. A recent study by Dong et al. explains how this provides an opportunity for local/global hybrid checkpoint using new technologies such as phase change memories (PCM). Moreover, some hard failures can be tolerated using solid-state-drives (SSD) and cross-node redundancy schemes, such as checkpoint replication or XOR encoding which allows to leverage multi-level checkpointing, as proposed by Moody et al.. Furthermore, Cheng et al. demonstrated that more complex erasure codes such as Reed-Solomon (RS) encoding can be used to further increase the percentage of hard failures tolerated without stressing the PFS.

FTI is a multi-level checkpointing interface. It provides an api which is easy to apply and offers a flexible configuration to enable the user to select the checkpointing strategy which fits best to the problem.

Multilevel-Checkpointing

L1

L1 denotes the first safety level in the multilevel checkpointing strategy of FTI. The checkpoint of each process is written on the local SSD of the respective node. This is fast but possesses the drawback, that in case of a data loss and corrupted checkpoint data even in only one node, the execution cannot successfully restarted.

L2

L2 denotes the second safety level of checkpointing. On initialisation, FTI creates a virtual ring for each group of nodes with user defined size (see group_size). The first step of L2 is just a L1 checkpoint. In the second step, the checkpoints are duplicated and the copies stored on the neighbouring node in the group.

That means, in case of a failure and data loss in the nodes, the execution still can be successfully restarted, as long as the data loss does not happen on two neighbouring nodes at the same time.

L3

L3 denotes the third safety level of checkpointing. In this level, the check- point data trunks from each node getting encoded via the Reed-Solomon (RS) erasure code. The implementation in FTI can tolerate the breakdown and data loss in half of the nodes.

In contrast to the safety level L2, in level L3 it is irrelevant which of nodes encounters the failure. The missing data can get reconstructed from the remaining RS-encoded data files.

L4

L4 denotes the fourth safety level of checkpointing. All the checkpoint files are flushed to the parallel file system (PFS).

Compilation

FTI uses Cmake to configure the installation. The recommended way to perform the installation is to create a build directory within the base directory of FTI and perform the cmake command in there. In the following you will find configuration examples. The commands are performed in the build directory within the FTI base directory.

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

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

> 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
```

> 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.

FTI Datatypes and Constants

FTI Datatypes

`FTI_CHAR` : FTI data type for chars
`FTI_SHRT` : FTI data type for short integers.
`FTI_INTG` : FTI data type for integers.
`FTI_LONG` : FTI data type for long integers.
`FTI_UCHR` : FTI data type for unsigned chars.
`FTI_USHT` : FTI data type for unsigned short integers.
`FTI_UINT` : FTI data type for unsigned integers.
`FTI_ULNG` : FTI data type for unsigned long integers.
`FTI_SFLT` : FTI data type for single floating point.
`FTI_DBLE` : FTI data type for double floating point.
`FTI_LDBE` : FTI data type for long double floating point.

FTI Constants

`FTI_BUFS` : 256
`FTI_DONE` : 1
`FTI_SCES` : 0
`FTI_NSCS` : -1

FTI_Init

> - Reads configuration file.
> - Creates checkpoint directories.
> - Detects topology of the system.
> - Regenerates data upon recovery.

DEFINITION

```
int FTI_Init ( char * configFile , MPI_Comm globalComm )
```

INPUT

Variable	What for?
char * configFile	Path to the config file
MPI_Comm globalComm	MPI communicator used for the execution

OUTPUT

Value	Reason
FTI_SCES	Success
FTI_NSCS	No Success

DESCRIPTION

This function initializes the FTI context. It should be called before other FTI functions, right after MPI initialization.

EXAMPLE

```
int main ( int argc , char **argv ) {
    MPI_Init (&argc , &argv );
    char *path = "config.fti"; // config file path
    FTI_Init ( path , MPI_COMM_WORLD );
.
.
.
    return 0;
}
```

FTI_InitType

> - Initializes a data type.

DEFINITION

```
int FTI_InitType ( FTIT_type *type , int size )
```

INPUT

Variable	What for?
FTIT_type * type	The data type to be initialized
int size	The size of the data type to be initialized

OUTPUT

Value	Reason
FTI_SCES	Success

DESCRIPTION

This function initializes a data type. A variable's type which isn't defined by default by FTI (see FTI Datatypes) should be added using this function before adding this variable to protected variables.

EXAMPLE

```
typedef struct A {
    int a;
    int b;
} A;
FTIT_type structAinfo ;
FTI_InitType (& structAinfo , 2 * sizeof ( int ));
```

FTI_Protect

> - Stores metadata concerning the variable to protect.

DEFINITION

```
int FTI_Protect ( int id, void *ptr, long count, FTIT_type type )
```

INPUT

Variable	What for?
int id	Unique ID of the variable to protect
void * ptr	Pointer to memory address of variable
long count	Number of elements at memory address
FTIT_type type	FTI data type of variable to protect

OUTPUT

Value	Reason
FTI_SCES	Success
exit(1)	Number of protected variables is > FTI_BUFS

DESCRIPTION

This function should be used to add data structure to the list of protected variables. This list of structures is the data that will be stored during a checkpoint and loaded during a recovery. It resets the dataset with given id if it was already previously registered. When size of a variable changes during execution it should be updated using this function before next checkpoint to properly store data.

EXAMPLE

```
int A;
float *B = malloc (sizeof(float) * 10) ;
FTI_Protect(1, &A, 1, FTI_INTG );
FTI_Protect(2, B, 10, FTI_SFLT );
// changing B size
B = realloc(B, sizeof(float) * 20) ;
// updating B size in protected list
FTI_Protect(2, B, 20, FTI_SFLT);
```

FTI_Checkpoint

> - Writes values of protected runtime variables to a checkpoint file of requested level.

DEFINITION

```
int FTI_Checkpoint( int id, int level )
```

INPUT

Variable	What for?
int id	Unique checkpoint ID
int level	Checkpoint level (1=L1, 2=L2, 3=L3, 4=L4)

OUTPUT

Value	Reason
FTI_DONE	Success
FTI_NSCS	Failure

DESCRIPTION

This function is used to store current values of protected variables into a checkpoint file. Depending on the checkpoint level file is stored in local, partner node or global directory. Checkpoint's id must be different from 0.

EXAMPLE

```
int i;
for (i = 0; i < 100; i ++) {
    if (i % 10 == 0) {
        FTI_Checkpoint ( i /10 + 1, 1) ;
    }
.
. // some computations
.
}
```

FTI_Status

> - Returns the current status of the recovery flag.

DEFINITION

```
int FTI_Status()
```

OUTPUT

Value	Reason
int 0	No checkpoints taken yet or recovered successfully
int 1	At least one checkpoint is taken. If execution fails, the next start will be a restart
int 2	The execution is a restart from checkpoint level L4 and `keep_last_checkpoint` was enabled during the last execution

DESCRIPTION

This function returns the current status of the recovery flag.

EXAMPLE

```
if ( FTI_Status () != 0) {
    .
    . // this section will be executed during restart
    .
}
```

FTI_Recover

> - Loads checkpoint data from the checkpoint file and initializes the runtime variables of the execution.

DEFINITION

```
int FTI_Recover()
```

OUTPUT

Value	Reason
FTI_SCES	Success
FTI_NSCS	Failure

DESCRIPTION

This function loads the checkpoint data from the checkpoint file and it up- dates some basic checkpoint information. It should be called after initial- ization of protected variables after a failure. If a variable changes it's size during execution it must have the latest size before Recover. The easiest way to do so is to add size of variable as another variable to protected list, and then call Recover twice. First to recover size of variable. Second to recover variable's data (after an update of protected list).

EXAMPLE

Basic example:

```
if ( FTI_Status() == 1 ) {
   Recover() ;
}
```

Example if a variable changes its size during execution:

```
int *A;
int Asize ;
.
.
.
if ( FTI_Status() != 0 ) {
   FTI_Recover(); // to recover size of variable
   A = realloc( A, sizeof(int)*Asize ) ;
   // updating protected list
   FTI_Protect( 2, buf, Asize, FTI_INTG );
   FTI_Recover(); // to recover variable A
}
```

FTI_Snapshot

> - Loads checkpoint data and initializes runtime variables upon recovery.
> - Writes multilevel checkpoints regarding their requested frequencies.

DEFINITION

```
int FTI_Snapshot()
```

OUTPUT

Value	Reason
FTI_SCES	Successfull call (without checkpointing) or if recovery successful
FTI_NSCS	Failure of `FTI_Checkpoint`
FTI_DONE	Success of `FTI_Checkpoint`
exit(1)	Failure on recovery

DESCRIPTION

This function loads the checkpoint data from the checkpoint file in case of restart. Otherwise, it checks if the current iteration requires checkpointing (see e.g.: ckpt_L1) and performs a checkpoint if needed (internal call to `FTI_Checkpoint`). Should be called after initialization of protected variables.

EXAMPLE

```
int res = Snapshot();
if ( res == FTI_SCES ) {
    .
    . // executed after successful recover
    . // or when checkpoint is not required
}
else { // res == FTI_DONE
    .
    . // executed after successful checkpointing
    .
}
```

FTI_Finalize

> - Frees the allocated memory.
> - Communicates the end of the execution to dedicated threads.
> - Cleans checkpoints and metadata.

DEFINITION

```
int FTI_Finalize()
```

OUTPUT

Value	Reason
FTI_SCES	For application process
exit(0)	For FTI process

DESCRIPTION

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 end of the program the FTI processes will never finish (deadlock). Should be called before `MPI_Finalize()`.

EXAMPLE

```
int main ( int argc , char ** argv ) {
    .
    .
    .
    FTI_Finalize () ;
    MPI_Finalize () ;
    return 0;
}
```

[Basic]

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*)

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*)

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/*)

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/*)

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*)

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*)

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*)

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*)

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*)

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*)

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*)

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*)

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`	After `FTI_Finalize()` , the meta files and checkpoints will be removed. No checkpoint data will be kept on the PFS or on the local hard drives of the nodes
`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*)

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*)

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*)

ckpt_io

> Sets the I/O mode.

Value	Meaning
1	POSIX I/O mode
2	MPI-IO I/O mode
3	SIONLib I/O mode

(*default = 1*)

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*)

[Restart]

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*)

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*)

[Advanced]

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

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*)

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*)

mpi_tag

> FTI uses a certain tag for the MPI messages. This tag can be set here.

Value	Meaning
int	Tag, used for MPI messages within FTI

(*default = 2612*)

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*)

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*)

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*)

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*)

Default Configuration

```
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
failure                 = 0
exec_id                   = NULL
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 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

```
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
failure                 = 0
exec_id                   = NULL
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.

Using only selected ckpt level with FTI_Snapshot

```
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
failure                 = 0
exec_id                   = NULL
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).

Keeping last checkpoint

```
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          = 1
group_size              = 4
max_sync_intv           = 0
ckpt_io                 = 1
verbosity               = 2
failure                 = 0
exec_id                 = NULL
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.

Using different IO mode

For instance MPI-I/O:

```
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
failure                 = 0
exec_id                 = NULL
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 IO mode is set to MPI IO (`ckpt_io = 2`). Third option is SIONlib IO mode (`ckpt_io = 3`).

Restart after a failure

```
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
failure                 = 1
exec_id                   = 2017-07-26_13-22-11
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()`.

Examples

Using FTI_Snapshot

```
#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).

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.