FTI - User Manual

(Release 0.9.8)

Authors: Kai Keller, Tomasz Paluszkiewicz

July, 2017

Contents

1 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 systemâĂŹs 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.

2 Multilevel Checkpointing

2.1 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.

2.2 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 4.1.14). 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.

2.3 L3

L3 denotes the third safety level of checkpointing. In this level, the checkpoint 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.

2.4 L4

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

3 API Reference

3.1 FTI Datatypes and FTI Constants

3.1.1 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 doble floating point.

3.1.2 FTI Constants

FTI_BUFS : 256

FTI_DONE : 1

FTI_SCES : 0

FTI_NSCS : -1

3.2 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)
```

On entry

```
char* configFile
MPI_Comm globalComm
```

configFile : Path to the config file

globalComm : MPI communicator used for the execution

On return

```
FTI_SCES
FTI_NSCS
```

FTI_SCES : On success

FTI_NSCS : On failure

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;
}
```

3.3 FTI_InitType(...)

– initializes a data type.

Definition

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

On entry

```
FTIT_type* type
int size
```

type : The data type to be initialized.

size : The size of the data type to be initialized.

On return

```
FTI_SCES
```

FTI_SCES : On success.

Description

This function initalizes a data type. A variable's type which isn't defined by default by FTI (see: 3.1.1) 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));
```

3.4 FTI_Protect(...)

– Stores metadata concerning the variable to protect.

Definition

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

On entry

```
int id
void* ptr
long count
FTIT_type type
```

id : Unique ID of the variable to protect

ptr : Pointer to memory address of variable

count : Number of elements at memory address

type : FTI datatype of variable to protect

On return

```
FTI_SCES
exit(1)
```

FTI_SCES : On 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 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);
```

3.5 FTI_Checkpoint(...)

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

Definition

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

On entry

```
int id
int level
```

id : Unique checkpoint ID

level : Checkpoint level (1=L1, 2=L2, 3=L3, 4=L4)

On return

```
FTI_DONE
FTI_NSCS
```

FTI_DONE : On success

FTI_NSCS : On 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
    .
}
```

3.6 FTI_Status()

– Returns the current status of the recovery flag.

Definition

```
int FTI_Status()
```

On return

```
int
```

0 : No checkpoints taken yet or recovered successfully.

1 : At least one checkpoint is taken. If execution fails, the next start will be a restart.

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() == 1) {
    .
    . //this section will be executed during restart
    .
}
```

3.7 FTI_Recover()

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

Definition

```
int FTI_Recover()
```

On return

```
FTI_SCES
FTI_NSCS
```

FTI_SCES : Checkpoint data is successfully restored.

FTI_NSCS : on failure.

Description

This function loads the checkpoint data from the checkpoint file and it updates some basic checkpoint information. It should be called after initialization of protected variables after a failure. If a variable changes it's size during execution Recover should be called twice. First to recover size of variable (must be added to protected list, see: 3.4). Second to recover variable's data (after an update of protected list).

Example

```
if (FTI_Status() == 1) {
    Recover();
}
//when variable changed it's size during execution
int* A;
int Asize;
.
.
.
if (FTI_Status() == 1) {
    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
}
```

3.8 FTI_Snapshot()

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

Definition

```
int FTI_Snapshot()
```

On return

```
FTI_SCES
FTI_DONE
exit(1)
```

FTI_SCES : On successful call / recovery.

FTI_DONE : If checkpoint was taken successfully.

exit(1) : If recovery unsuccessful.

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: 4.1.6) and performs a checkpoint if needed. Should be called after initialization of protected variables.

Example

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

3.9 FTI_Abort()

– Aborts the application after cleaning the file system.

Definition

```
int FTI_Abort()
```

On return

```
exit(1)
```

exit(1) : After call.

Description

This function aborts the application after cleaning the file system.

Example

```
int fail = 0;
.
. //some computations
.
if (fail == 1) {
    FTI_Abort();
}
```

3.10 FTI_Finalize()

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

Definition

```
int FTI_Finalize()
```

On return

```
FTI_SCES
exit(0)
```

FTI_SCES : For an application process.

exit(0) : For a head 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;
}
```

4 Configuration

4.1 [Basic]

4.1.1 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*)

4.1.2 Node_size

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

Value	**Meaning**
npp (int $>$ 0)	Number of processing units within each node (notice: The total number of processes must be a multiple of group size $\times$ node size)

(*default = 2*)

4.1.3 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 = _BLANK_*)

4.1.4 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 (notice: The directory has to be created before execution).

(*default = /path/to/global/storage/*)

4.1.5 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 informations about the topology of the execution.

Value	Meaning
string	Path to the meta files directory. (notice: The directory has to be created before execution).

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

4.1.6 Ckpt_L1

Here, the user sets the checkpoint frequency of L1 checkpoints.

Value	Meaning
L1 freq. (int ≥ 0)	L1 checkpointing frequency in min^{-1}. If the value is equal to 0, L1 checkpointing is disabled.

(*default = 3*)

4.1.7 Ckpt_L2

Here, the user sets the checkpoint frequency of L2 checkpoints.

Value	Meaning
L2 freq. (int ≥ 0)	L2 checkpointing frequency in min^{-1}. If the value is equal to 0, L2 checkpointing is disabled.

(*default = 5*)

4.1.8 Ckpt_L3

Here, the user sets the checkpoint frequency of L3 checkpoints.

Value	Meaning
L3 freq. (int ≥ 0)	L3 checkpointing frequency in min^{-1}. If the value is equal to 0, L3 checkpointing is disabled.

(*default = 7*)

4.1.9 Ckpt_L4

Here, the user sets the checkpoint frequency of L4 checkpoints.

Value	Meaning
L4 freq. (int ≥ 0)	L4 checkpointing frequency in min^{-1}. If the value is equal to 0, L4 checkpointing is disabled.

(*default = 11*)

4.1.10 Inline_L2

In this entry, the user decides, whether the post-processing work on the L2 checkpoints is done by the HEAD or by the application processes.

Value	Meaning
0	The post-processing work of the L2 checkpoints is done by the HEAD (notice: This setting is only allowed if Head = 1).
1	The post-processing work of the L2 checkpoints is done by the application processes.

(*default = 1*)

4.1.11 Inline_L3

In this entry, the user decides, whether the post-processing work on the L3 checkpoints is done by the HEAD or by the application processes.

Value	Meaning
0	The post-processing work of the L3 checkpoints is done by the HEAD (notice: This setting is only allowed if Head = 1).
1	The post-processing work of the L3 checkpoints is done by the application processes.

(*default = 1*)

4.1.12 Inline_L4

In this entry, the user decides, whether the post-processing work on the L4 checkpoints is done by the HEAD or by the application processes.

Value	Meaning
0	The post-processing work of the L4 checkpoints is done by the HEAD (notice: This setting is only allowed if Head = 1).
1	The post-processing work of the L4 checkpoints is done by the application processes.

(*default = 1*)

4.1.13 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*)

4.1.14 Group_size

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

Value	Meaning
int i ($2 \leq i \leq 32$)	Number of nodes contained in a group (notice: The total number of processes must be a multiple of group size $\times$ node size).

(*default = 4*)

4.1.15 Verbosity

Sets the verbosity level

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

(*default = 2*)

4.1.16 Ckpt_io

Sets the IO mode

Value	Meaning
1	POSIX IO mode.
2	MPI IO mode.
3	SIONlib IO mode.

(*default = 1*)

4.2 [Restart]

4.2.1 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*)

4.2.2 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 = XXXX-XX-XX_XX-XX-XX*)

4.3 [Advanced]

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

4.3.1 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 checkpopoint levels L2 and L3.

(*default = 1024*)

4.3.2 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*)

4.3.3 Mpi_tag

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

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

(*default = 2612*)

4.3.4 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 cofiguration.
1	Local test is enabled (notice: FTI will check if the settings are correct on initialization and if necessary stop the execution).

(*default = 1*)

5 Example

```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;
}
```