SEVENTH FRAMEWORK PROGRAMME

FP7-ICT-2013-10

DEEP-ER

DEEP Extended Reach

Grant Agreement Number: 610476

D5.2

Interface abstraction layer

Approved

Version:	2.0
Author(s):	A. Galonska (JUELICH)
Contributor(s):	K. Thust (JUELICH), W. Frings (JUELICH) S. Breuner (FHG-ITWM), V. Beltran, (BSC), C. Clauss (ParTec), N. Eicker (JUELICH)
Date:	09.12.2015

Project and Deliverable Information Sheet

DEEP-ER Project	**Project Ref. №:** 610476	
	Project Title: DEEP Extended Reach	
	Project Web Site: http://www.deep-er.eu	
	Deliverable ID: D5.2	
	Deliverable Nature: Report	
	Deliverable Level: PU PU*	**Contractual Date of Delivery**: 31 / March / 2015
		Actual Date of Delivery: 31 / March / 2015
	EC Project Officer: Panagiotis Tsarchopoulos	

* - The dissemination level are indicated as follows: **PU** – Public, **PP** – Restricted to other participants (including the Commission Services), **RE** – Restricted to a group specified by the consortium (including the Commission Services). **CO** – Confidential, only for members of the consortium (including the Commission Services).

Document Control Sheet

Document	**Title:** Interface abstraction layer	
	ID: D5.2	
	Version: 2.0	**Status:**Approved
	Available at: http://www.deep-er.eu	
	Software Tool: Microsoft Word	
	File(s): DEEP-ER_D5.2_Interface_abstraction_layer_v2.0-ECapproved	
Authorship	**Written by:**	A. Galonska (JUELICH)
	Contributors:	K. Thust (JUELICH), W. Frings (JUELICH) S. Breuner (FHG-ITWM), V. Beltran, (BSC), C. Clauss (ParTec), N. Eicker (JUELICH)
	Reviewed by:	J.Schmidt (UniHD), J.Kreutz (JUELICH)
	Approved by:	BoP/PMT

Document Status Sheet

Version	Date	Status	Comments
1.0	31/March/2015	Final	EC submission
2.0	09/December/2015	Approved	EC approved

Document Keywords

Keywords:	DEEP-ER, HPC, Exascale

Table of Contents

List of Figures

List of Tables

Executive Summary

With a growing amount of components in future Exascale systems the Mean Time Between Failures (MTBF) becomes smaller and smaller. It will become more likely that a critical component like an interconnect, a core or even a whole node gets corrupted during runtime of a compute job.

The design of the DEEP-ER system architecture intends to address the need of high resiliency at Exascale. It foresees a multi-staged storage facility to save data needed for a certain job to restart from.

While the hardware – in this case, the storage devices – is only one part of that solution, a more important one from the programmer's point of view is an appropriate software, which manages the data flows between the application and storage layers, as well as the data transfer between the layers.

The focus has to lay on easy usability and easy configurability plus making the underlying resiliency software transparent to the programmer.

The project also decided not only to develop the resiliency software in form of a checkpoint/restart approach, but also to provide an interface, which allows the programmer to easily enable resiliency features without the need for drastic changes within the code.

The interface abstraction layer presented in this document consists of only 6 functions the end-user needs to integrate in his code at prominent places to make use of the resiliency features provided by the system.

This report at hand is intended for developers of user-codes which want to use the resiliency software developed in task 5.3.

1 Introduction

This deliverable describes the interface abstraction layer of the multi-level checkpoint/restart capabilities developed in the DEEP-ER project. It will be the connecting point between the user and the underlying resiliency software to be implemented in Task 5.3. However, during the design phase it turned out that the Scalable Checkpoint Restart (SCR) library from Lawrence Livermore National Laboratory (LLNL) already constitutes a suitable environment for the DEEP-ER architecture and can serve as a first starting point in the implementation of an application-based checkpoint/restart software. So this document not only gives a definition of the interface, but also a short overview about SCR and the necessary extensions for DEEP-ER. A complete description of the implementation details and changes taken in the final software will be presented in D5.3.

2 SCR – Scalable Checkpoint Restart

2.1 Main features

The Scalable Checkpoint Restart library (SCR) was chosen to be the basis of our application-based checkpoint/restart approach to be developed in Task 5.3. The software allows multi-level checkpointing to different stages which is suitable for the DEEP-ER system architecture [1].

The main task of this library is book-keeping of task-local checkpoint files including meta-data handling and transfer of checkpoint files between different stages of storage levels. Checkpoint files are written by the user-application itself to file-paths provided by the library. Any file-based I/O method can be used making this approach highly flexible.

SCR ensures that the file will be stored at the correct storage location. The file transfers can be carried out synchronously by the application itself or asynchronously via background transfer by a special software daemon. It provides also buddy-checkpointing and is also able to compute and restore XOR-sets of checkpoint data. Buddy-checkpointing in this context means that checkpointing data of one node is also saved on another node – the buddy node. With XOR-sets the data is distributed not on one another node but on a set of participating nodes similar to RAID5.

Policies can be defined to determine the number and frequency of checkpoints and their storage locations.

Valid checkpoint files are found during a restart by checking available meta-data and storage locations. These files are automatically transferred from background to node local storage if necessary.

As the user only needs to add 6 function calls to gain all of the above mentioned resiliency features for an application, whereas the underlying complexity is completely transparent, SCR was chosen in the context of the DEEP-ER project as the application-based checkpoint/restart approach. However, modifications are necessary to fully exploit the hardware and foreseen software layers of the architecture and are described in Section 5.

2.2 API

This section gives a brief overview of the functions the interface abstraction layer exposes to the user. The functions are derived from the ones originally implemented in SCR but the underlying functionality will be adapted to the DEEP-ER system architecture. The interface described in the sections below is the same as in SCR, except in the case of `SCR_Init`, where in DEEP-ER the desired I/O method (i.e. parallel I/O or task-I/O) can be selected.

2.2.1 int SCR_Init(int mode)

in: `int mode`

out: -

description: shall be called after `MPI_Init()`. Initializes the SCR environment, sets up communicators, directory paths, reads existing meta-data and searches for former checkpoints to eventually restart from. With the help of `mode` the user can select if he intends to use parallel I/O (e.g. via SIONlib), task local I/O and also if he wants to use the BeeGFS cache capabilities. For this purpose the three makros `SCR_MODE_PIO`, `SCR_MODE_DEFAULT` and `SCR_MODE_BEEGFS` were defined and can be used as values for `mode`. `SCR_MODE_BEEGFS` can be connected via bitwise OR with either `SCR_MODE_DEFAULT` or `SCR_MODE_PIO`.

*2.2.2 int SCR_Need_Checkpoint(int *flag)*

in: -

out: `int *flag`

description: the user hands over a pointer of a pre-allocated integer to this function. The value is set to 1 if a checkpoint is necessary now, or 0 if it is not. Whether it is necessary or not depends on a policy, which needs to be defined for the platform the application runs on.

2.2.3 int SCR_Start_Checkpoint()

in: -

out: -

description: this function signals SCR, that the application is going to write a new checkpoint. The user has to call it prior to `SCR_Route_Filename()` and before it starts to write a checkpoint.

*2.2.4 int SCR_Route_Filename(char *name, *char *file)*

in: `char *name`

out: `char *file`

description: if this function is called **without** calling `SCR_Start_Checkpoint()` before, it returns the file name of the last valid checkpoint. If it is called **after** `SCR_Start_Checkpoint()`, the path to a new checkpoint is returned. The user hands over a string *name* to the function and gets the path *file* from it, which is derived from *name.*

2.2.5 int SCR_Complete_Checkpoint(int valid)

in: `int valid`

out: -

description: the user has to call this function after writing a checkpoint. The input integer *valid* has to be 1, if writing the checkpoint data was successful, or 0 if it was not.

2.2.6 int SCR_Finalize()

in: -

out: -

description: this function has to be called at the end of the application before calling `MPI_Finalize()`. It frees all structures and destroys all MPI communicators. It also marks the run to be successful, so that it will not restart from the available, and now outdated, checkpoint data the next time.

2.3 Configuration

To control the behaviour of the resiliency software the user might specify options via

1. Environment Variable
2. User configuration file
3. System configuration file
4. Compile-time constants

Configurations parameters are searched in the above order and SCR will take the first value it finds.

2.3.1 Environment variables

The easiest way to configure SCR is to use environment variables, which is sufficient for users desiring only one node-local cache and one back-end storage location. It is worth mentioning that in DEEP-ER caching can be done into the storage internal to the node and also into the NVM devices connected to the Booster Nodes.

Name	Default	Description
SCR_CACHE_BASE	/tmp	Base directory to cache checkpoints on node local storage
SCR_CNTL_BASE	/tmp	Base directory for metadata storage
SCR_PREFIX		Base directory for checkpoints on the background-storage
SCR_CACHE_SIZE	2	Number of checkpoints to keep in cache directory
SCR_FLUSH	10	Number of checkpoints between periodic flushes to the parallel file system
SCR_CHECKPOINT_INTERVAL		Number of times

		SCR_Need_Checkpoint(*flag) needs to be called between checkpoints
SCR_CHECKPOINT_SECONDS		Minimum number of seconds between checkpoints
SCR_CHECKPOINT_OVERHEAD		Maximum overhead allowed for checkpointing in %

Table 1: Environment Variables

2.3.2 Configuration file

The configuration of SCR via a configuration file allows the user a more fine grained definition of what they desire. All options, which can be set via environment variables (see Section 2.3.1) can also be defined within the configuration file. While the users are able to edit their own configuration files, a system-wide configuration file can also be used. The syntax of the different configuration parameters is described in this section. All indices mentioned here are starting with 0 and are incrementing with every entry of a single type.

2.3.2.1 Store Descriptors

First of all, one needs to specify so called store descriptors for every directory which will be used, either for caching or storing checkpoints:

```
STOREDESC=k BASE=<directory> COUNT=s
```

Here `k` is the index and `s` checkpoints should be kept in `<directory>`.

2.3.2.2 Checkpoint Descriptors

After defining the store descriptors one can define checkpointing policies for each store in the following way:

```
CKPTDESC=k BASE=<directory> INTERVAL=i TYPE=SINGLE
```

Here `k` is the index, and `i` is the checkpoint interval. The directory is specified in `BASE`. Besides `SINGLE` one can also specify `PARTNER` as `TYPE` to enable buddy-checkpointing in SCR. However, in DEEP-ER we follow a different approach by letting SIONlib do the buddy-checkpointing.

2.4 Examples

2.4.1 Configuration file

```
// where to store meta data
SCR_PREFIX=/work/deep06/checkpoints

// flush every checkpoint after creating the first one
SCR_FLUSH=1

// define the store descriptors, keep 3 checkpoints on every background storage and 1 in the
// cache
STOREDESC=0 BASE=/tmp COUNT=1
STOREDESC=1 BASE=/work/deep06/checkpoints2 COUNT=3
STOREDESC=2 BASE=/work/deep06/checkpoints4 COUNT=3
```

// save every checkpoint to /tmp
```
CKPTDESC=0 BASE=/tmp INTERVAL=1 TYPE=SINGLE
```

// save every 2nd checkpoint to /work/deep06/checkpoints2
```
CKPTDESC=1 BASE=/work/deep06/checkpoints2 INTERVAL=2 TYPE=SINGLE
```

// save every 4th checkpoint to /work/deep06/checkpoints4
```
CKPTDESC=2 BASE=/work/deep06/checkpoints4 INTERVAL=4 TYPE=SINGLE
```

2.4.2 Example of how to write a checkpoint

```
SCR_Need_checkpoint(&perform_checkpoint);
if (perform_checkpoint == 1) {
      SCR_Start_checkpoint();
      sprintf(name, "solve_nbody-%dx%d_r%d.ckpt", nb,
      BLOCK_SIZE,rank);
      if(SCR_Route_file(name, path)==SCR_SUCCESS){
            FILE* fd = fopen(path, "wb");
            assert(fd != NULL);
            saved_data = write(&current_tstep,sizeof(int),1,fd);
            saved_data += write(particles,
            sizeof(particles_block_t), nb,fd);
            assert(fclose(fd)==0);
      }
      int is_valid = saved_data == (nb + 1);
      SCR_Complete_checkpoint(is_valid);
```

2.4.3 Example of how to restart from a checkpoint

```
if (SCR_Route_file(name, path) == SCR_SUCCESS) {
            FILE* fd = fopen(path, "rb");
            // Open, read and close file
            assert(fd != NULL);
            num_read = fread(&temp_tstep,sizeof(int),1,fd);
            num_read += fread(local, sizeof(particles_block_t),
            nb,fd);
            assert(fclose(fd)==0);
            if(num_read == (nb+1)){
                  found_cp = 1;
            }else{
                  found_cp = -1;
            }
      }
```

3 SIONlib

SIONlib [2] is a scalable I/O library developed by JSC and will be utilized by the applications of the DEEP-ER project. While task-local file creation and access becomes a problem on Exascale systems because of overload problems with meta-data servers of most modern

parallel file systems, SIONlib circumvents this restriction by providing task-local file access to single so-called SIONlib files. Here all processes are carrying out simultaneous I/O on a single file (or on a limited amount of files). Processes are writing and reading chunks of data aligned to file system blocks so that each block is assigned to exactly one process, avoiding access race-conditions. Furthermore, the reduced number of files significantly removes the load on the meta-data servers and thus scales better than traditional task-local I/O approaches. Figure 1 gives an overview about the SIONlib structure. Further information on SIONlib can be found in [2].

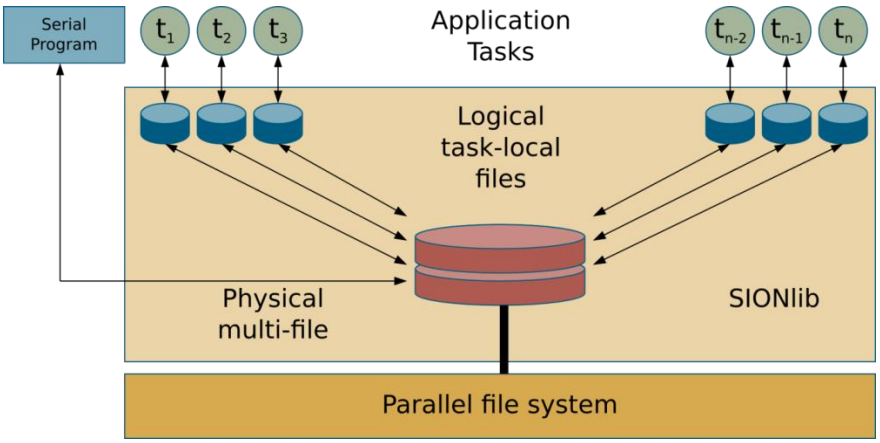

Figure 1: SIONlib structure overview

It is possible to configure SIONlib to create more than one file for the output of all processes. SCR needs information of all files created during a checkpoint, so SIONlib has to provide an interface from which this information is retrieved.

4 BeeGFS

BeeGFS is a parallel file system developed by Fraunhofer ITWM [3]. In the context of the DEEP-ER project it should provide a scalable I/O facility for application checkpoint data.

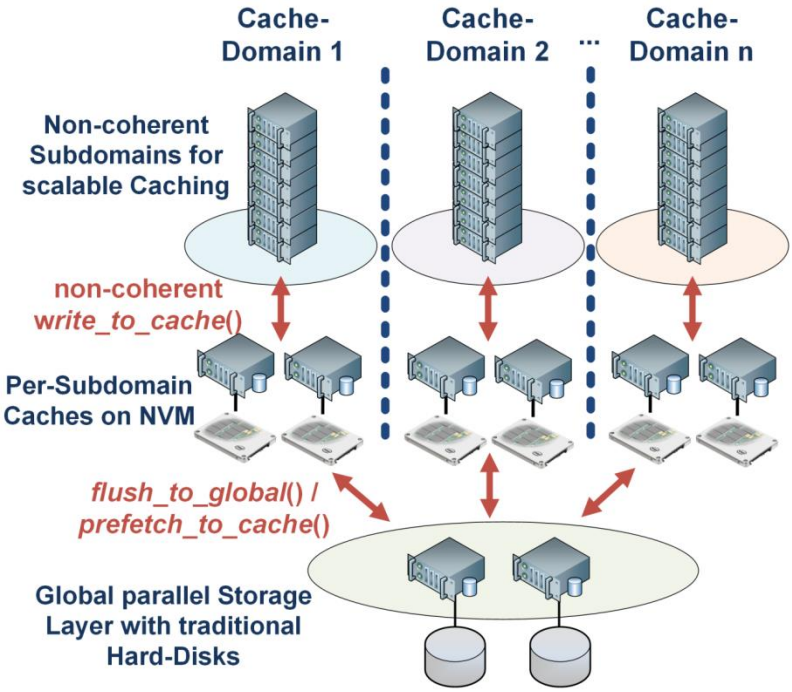

Figure 2: BeeGFS structure overview

A key aspect within the application-based checkpoint/restart software is its mechanism to cache files within so called “cache domains” which are constituted by the node-local storages of a few nodes within a supercomputer. Within the DEEP-ER project we can store checkpoints at the first level and BeeGFS will background-transfer them to the main storage of the system and vice versa (Figure 2). With DEEP-ER’s approach, fewer nodes have to synchronize their view on the cache reducing the coherency-overhead significantly. Further information on BeeGFS can be found on [3].

5 DEEP-ER extensions

The situation in DEEP-ER will, by definition, be different compared to other supercomputer architectures because its focus will be on resiliency features. While the standard SCR library expects only a node-local cache and a generic file system for background storage, the software to be used in DEEP-ER has to expose the underlying capabilities of BeeGFS and SIONlib to the application developer (Figure 3).

BeeGFS cache domains are constituted of node-local storage and mapped incoherently to underlying background storage. Files within a cache domain are created on local storage and can then be flushed by the file system itself to the background storage. In this scenario built-in flushing and storing features of SCR need to be replaced by appropriate calls to the BeeGFS API. This takes load from the processes because the work is done by BeeGFS. BeeGFS, like SCR, is able to fetch/store data in a synchronous or asynchronous way, which makes the mapping of SCR functions to BeeGFS calls straight forward. It is foreseen that

SCR uses the BeeGFS cache domains to store task-local files and to utilize it to flush them to the background storage. For this task SCR needs to be aware of the different cache domains in order to arrange its process communicators accordingly. Typically this results in the choice of having one communicator per cache domain.

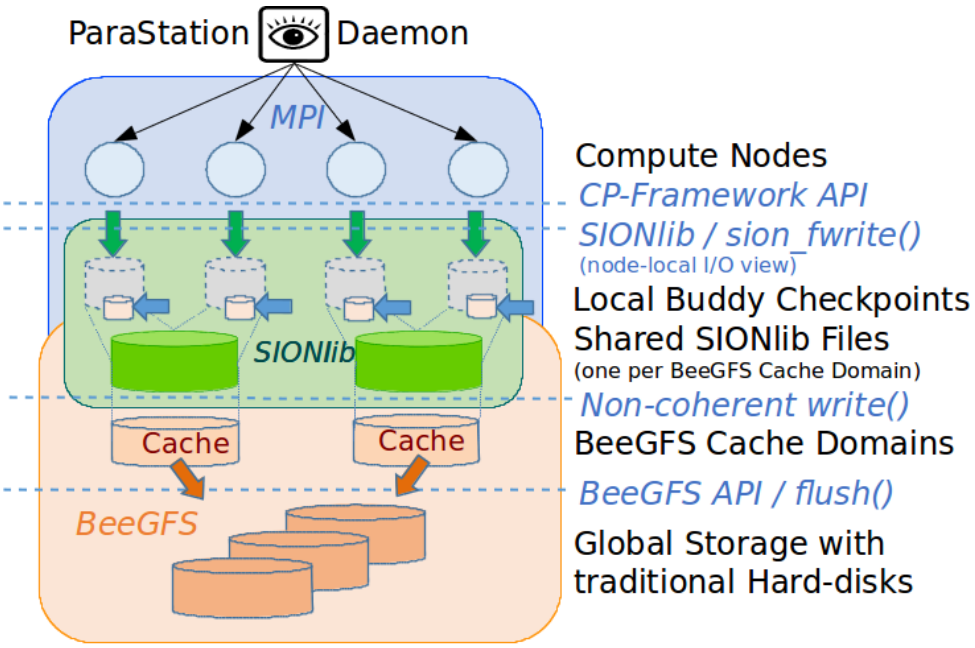

Figure 3: DEEP-ER Resiliency Software Layers

As stated before, handling task-local files on an Exascale system will become crucial since meta-data servers on most modern parallel file systems will not, or only to a certain degree, be able to scale to an Exascale level during file creations. Our approach to overcome these limitations is to aggregate streams of several processes into a single file. This means, that not every process is responsible for its own file, but a special process of a process group needs to take care of flushing and fetching a single file for the group. With respect to the features of BeeGFS stated above, the process is the master process of the cache domain communicator.

To apply this approach to SCR, all code locations have to be identified, where file handling takes place. It needs to be ensured that only one special process does the file operations, otherwise this would lead to multiple undesired deletes and copies.

It is foreseen, that buddy-checkpointing will also be handled by SIONlib. Process streams are, in this scenario not only aggregated into a single file, but also into the buddy file. This means, that SCR needs to be informed about the SIONlib files created, to not break SCRs meta-data handling. Appropriate calls to retrieve this information have to be provided by SIONlib. Letting SIONlib do the buddy-checkpointing eliminates the need for SCR to copy buddy files hence and forth between buddy nodes because the files are written directly.

6 Expected results and conclusion

The acceptance of resiliency software highly depends on the complexity of its interface. While carrying out resiliency tasks like creating a checkpoint, background data transfers and restoring becomes more complex on future Exascale systems the end-user does not need to know about the underlying mechanisms.

Enabling resiliency features in DEEP-ER will be as easy as adding only a handful of additional function calls to the code. In addition, the preferred method of I/O will be supported by the resiliency software.

Our approach addresses these issues by providing a simple interface to the user. Besides two functions for initialization and finalization of the library, the user in principle only has to add four additional lines of code at a place, where I/O would be usually done. Configuration is easily carried out by defining environment variables or can be more fine grained when using a configuration file.

However, integration of resiliency features into an application is straight-forward due to the simple user-interface and easy configurability.

Performance issues caused by task-local file handling of SCR will be circumvented by the usage of SIONlib. File streams of several processes will be aggregated into only a few files taking load from meta-data servers. In Addition, file transfers between node-local storage and background-storage will be carried out by the file system software itself which also leads to a reduced load on the user processes.

7 Bibliography

[1] A. Moody, “Overview of the Scalable Checkpoint/Restart Library,” 2009.

[2] W. Frings, F. Wolf and V. Petkov, “Scalable Massively Parallel I/O to Task-Local File,” *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis,* 2009.

[3] J. Heichler, “An introduction to BeeGFS,” http://www.beegfs.com/docs/Introduction_to_BeeGFS_by_ThinkParQ.pdf, 2014.

List of Acronyms and Abbreviations

B

BeeGFS: The Fraunhofer Parallel Cluster File System (previously acronym FhGFS). A high-performance parallel file system to be adapted to the extended DEEP Architecture and optimised for the DEEP-ER Prototype.

D

DEEP: Dynamical Exascale Entry Platform

DEEP-ER: DEEP Extended Reach: this project

E

Exascale: Computer systems or Applications, which are able to run with a performance above 10^{18} Floating point operations per second

I

ITWM: Institut für Techno- und Wirtschaftsmathematik. An Institute of the Fraunhofer Society

J

JUELICH: Forschungszentrum Jülich GmbH, Jülich, Germany

L

LLNL: Lawrance Livermore National Laboratory

M

MTBF: Mean Time Between Failures

N

NAM: Network Attached Memory, nodes connected by the DEEP-ER network to the DEEP-ER BN and CN providing shared memory buffers/caches, one of the extensions to the DEEP Architecture proposed by DEEP-ER

R

RAID: Redundant Array of Independent Discs

S

SCR: Scalable Checkpoint/Restart. A library from LLNR

SSD: Solid State Disk

SIONlib: Parallel I/O library developed by Forschungszentrum Jülich