SEVENTH FRAMEWORK PROGRAMME

FP7-ICT-2013-10

DEEP-ER

DEEP Extended Reach

Grant Agreement Number: 610476

D4.2

I/O interfaces description

Approved

Version: 2.0

Author(s): N.Eicker (JUELICH)

Contributor(s): K.Thust (JUELICH), S.Narasimhamurthy (Xyratex), S.Breuner (FHG-ITWM)

Date: 21.10.2014

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: D4.2	
	Deliverable Nature: Report	
	Deliverable Level: PU*	**Contractual Date of Delivery:** 30 / September / 2014
		Actual Date of Delivery: 30 / September / 2014
	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:** I/O interfaces description	
	ID: D4.2	
	Version: 2.0	**Status:** Approved
	Available at: http://www.deep-er.eu	
	Software Tool: Microsoft Word	
	File(s): DEEP-ER_D4.2_IO_interfaces_description_v2.0-ECapproved	
Authorship	**Written by:**	N.Eicker (JUELICH)
	Contributors:	K.Thust (JUELICH), S.Narasimhamurthy (Xyratex), S.Breuner (FHG-ITWM)
	Reviewed by:	J.Romein (ASTRON), E.Suarez (JUELICH)
	Approved by:	BoP/PMT

Document Status Sheet

Version	Date	Status	Comments
1.0	30/September/2014	Final	EC submission
2.0	21/October/2014	Approved	EC approved

Document Keywords

Keywords:	DEEP-ER, HPC, Exascale, I/O software, BeeGFS, SIONlib, Exascale10

Table of Contents

Executive Summary

This deliverable describes the different I/O interfaces within the DEEP-ER project, which include the BeeGFS file system, its extensions, and the SIONlib and Exascale10 I/O middleware layers. It briefly describes design concepts and gives an overview of the relevant parts of the API that can be used by DEEP-ER applications to perform I/O efficiently and reliably. This will be of primary interest to Applications (WP6) which can exploit these APIs for doing very efficient I/O at scale. This is also of interest to the Resiliency Software (WP5) which will work with some of the I/O middleware APIs defined by this deliverable.

1 Introduction

One of the key aspects of the DEEP-ER project is the development of strategies for handling I/O on future Exascale class machines. The three I/O software components within the DEEP-ER project offer different I/O strategies on how to approach I/O scalability and reliability for applications. Deliverable D4.1, submitted at month 6 described the background and motivation for the different I/O strategies and components employed in the DEEP-ER project [D4.1 Reference]. This deliverable extends D4.1 by actually describing the interfaces that will be used by the applications. The basic file system services are provided by the BeeGFS file system. BeeGFS extensions, SIONlib and Exascale10 work on top of BeeGFS and address scalability and reliability functionalities.

BeeGFS adds functionality and interfaces to address the NVRAM cache layer on top of the base storage platform. These interfaces could be used directly by the applications. Alternatively, this new functionality can be exploited by the middleware components, SIONlib and Exascale10, which provide higher level APIs for their I/O strategies and map them to BeeGFS calls.

We next describe the interfaces offered by the three components. We provide more detail on the Exascale10 component in this deliverable as it is a relatively new approach compared to BeeGFS and SIONlib.

2 BeeGFS interface description

2.1 General notes on API intentions

The DEEP-ER file system is designed to be compatible with the standard POSIX interface to access the global storage, but also provides extensions for explicit application control regarding how the DEEP-ER non-global cache layer is used.

2.2 POSIX interface

All standard POSIX I/O operations (e.g. `open()`, `close()`, `read()`, `write()`) will work on the DEEP-ER file system to allow running unmodified/legacy applications. However, these operations will only apply to data on the global storage layer and not take advantage of, nor “see”, any of the data on the cache layer, even if the data on the cache layer was previously written by the same node.

2.3 DEEP-ER interface extensions for the cache layer

The DEEP-ER cache layer consists of a configurable number of independent subdomains. It provides high scalability through avoidance of any direct interaction or communication between different subdomains, thus making file operations in different subdomains non-coherent. The cache layer is intended to hold temporary data on devices, which allow fast access, but have only limited capacity like NVM.

Long-term data storage is provided through a global storage layer with higher capacity, but lower performance and limited scalability on traditional rotational hard-drives.

The following sections describe how applications can access the cache layer and how data can be moved from or to the global storage layer. A more detailed description of the interface with parameter types and return values can be found in Appendix A.

2.3.1 *Create/remove/list directories on the cache layer*

`deeper_cache_{mkdir, rmdir, opendir, closedir};`

Create and use directories similar to the corresponding POSIX routines, but on the cache layer of the current cache domain.

Especially `mkdir` should transparently handle the case of missing parent directories on the cache layer for convenience, e.g. if user calls `deeper_cache_mkdir("/scratch/userA/projectX/checkpoints")` to create the "`checkpoints`" directory, it should succeed even though the parent directories ("`userA/projectX`") did not previously exist on the cache layer, but only on the global layer.

2.3.2 *Open a file on the cache layer for reading or writing*

`deeper_cache_open(path, flags, deeper_open_flags);`

`path`, `flags` and `return value` are similar to the POSIX `open`() call.

`deeper_open_flags` can be a combination of

- `DEEPER_OPEN_FLUSHONCLOSE` to automatically flush written data to global storage when the file is closed (asynchronously)
- `DEEPER_OPEN_FLUSHWAIT` to make `DEEPER_OPEN_FLUSHONCLOSE` a synchronous operation.
- `DEEPER_OPEN_DISCARD` to remove the file from the cache after it has been closed/flushed. Otherwise, the data will still remain on the cache layer.

Any following write()/read() on the returned file descriptor will write/read data to/from the cache layer. Data on the cache layer will be visible only to those nodes that are part of the same cache domain.

2.3.3 *Read-ahead a file/dir to cache*

`deeper_cache_prefetch(path, deeper_prefetch_flags);`

`path` is the path to a file or directory on the global layer that should be copied to the cache domain asynchronously.

`deeper_prefetch_flags` can be a combination of

- `DEEPER_PREFETCH_SUBDIRS` to recursively copy all subdirs, if given `path` leads to a directory.
- `DEEPER_PREFETCH_WAIT` to make this a synchronous operation.

This method can also be used by a cluster batch system to prefetch certain user data to the cache before a cluster job starts.

```
deeper_cache_prefetch_wait(path);
```

Wait for an ongoing prefetch operation from `deeper_cache_prefetch`() to complete.

2.3.4 *Flush to globally visible storage*

```
deeper_cache_flush(path, deeper_flush_flags);
```

Start an asynchronous flush to global storage of the given file.

`deeper_flush_flags` can be a combination of

- `DEEPER_FLUSH_{FLUSHWAIT, DISCARD}` with semantics similar to the corresponding `DEEPER_OPEN_...` flags.

```
deeper_cache_flush_wait(path);
```

Wait for an ongoing flush operation from `close`() or `deeper_cache_flush`() to complete.

2.3.5 *Byte-range operations*

The operations described above apply to whole files, so it is not possible to e.g. write the first half of a shared big file in cache domain A and the other half in cache domain B and afterwards merge both halves on the global storage layer. For this purpose, we have byte-range operations:

```
deeper_cache_flush_range(path, start_pos, num_bytes,
deeper_flush_flags);
```

```
deeper_cache_prefetch_range(path, start_pos, num_bytes,
deeper_prefetch_flags);
```

These functions operate similar to the corresponding prefetch/flush functions above, but only prefetch/flush a certain byte range from/to the global storage layer.

2.3.6 *Find out which compute nodes are part of the same cache domain*

```
deeper_cache_id(path, out_cache_id);
```

If `path` leads to a BeeGFS mount, `out_cache_id` returns a numeric ID that is the same for all nodes that are part of the same cache domain.

3 SIONlib interface description

3.1 General remarks

The basic functions of SIONlib's API are already stable and used by different applications today. One of the major design goals for the basic interface is to resemble the classical task-local I/O to make the transition as easy as possible.

3.2 Basic API calls

The following overview over the basic API calls uses the MPI interface to show the general structure of the calls. SIONlib also offers interfaces for serial, threaded (OpenMP), and hybrid (OpenMP and MPI) use.

3.2.1 *Open*

```
int sion_paropen_mpi(char*       fname,
                     const char* file_mode,
                     int*        numFiles,
                     MPI_Comm    gComm,
                     MPI_Comm*   lComm,
                     sion_int64* chunksize,
                     sion_int32* fsblksize,
                     int*        globalrank,
                     FILE**      fileptr,
                     char**      newfname);
```

- `fname`: name of the SIONlib file
- `file_mode`: "r" for read "w" for write and additional flags (see below)
- `numFiles`: number of multi files to use (-1 for automatic choosing from local communicator [see below])
- `gComm`: global MPI communicator
- `lComm`: local MPI communicator (= `gComm` if no adaption to I/O nodes is needed [see below])
- `chunksize`: maximum size to be written with single write call
- `fsblksize`: file system block size (-1 for automatic)
- `globalrank`: global rank of process
- `fileptr`: file pointer (NULL for not using an external file pointer)
- `newfname`: return value for actual file name if using multi files

Opens file `fname` in mode `file_mode` and returns the SIONlib file id `sid` which is the unique identifier used for the subsequent handling of the corresponding SIONlib file.

This call is collective and distributes the information about all chunk sizes to all tasks. With this information available for all tasks SIONlib is able to compute disjoint byte ranges for all tasks without further communication.

Opening a file in read mode uses `numFiles` and `chunksize` as "out" parameters and returns the contents from the opened file.

In addition to the “rb” and “wb” flags there are different flags which further control the behavior of SIONlib.

- `deeper (or beegfs)`: inform SIONlib about the underlying platform
- `localcache`: use BeeGFS's deeper API featuring access to the cache layer instead of standard POSIX or ANSI C
- `discard_local`: remove file from cache layer after successful sync to global file system
- `flush_sync`: flush file on close synchronous
- `flush_async`: flush file on close asynchronous
- `buddycheckpoint`: enable buddy checkpointing

In general the local communicator `lComm` can be choosen the same as `gComm`. In order to allow for additional optimizations `lComm` can be used to tell SIONlib about the mapping from ranks to I/O nodes. This causes SIONlib to choose one file per I/O node which leads to less metadata operations to non-local I/O nodes and increases the performance for large numbers of ranks. In this scenario the original file, which was requested for opening only, holds the mapping information from ranks to the multiple but few files which belong to the different I/O nodes. When using this approach the `newfname` variable holds the actual name of the file to be written to and which differs from the original one by an appended number. The splitting of one SIONlib file into multiple happens transparent for the user.

For historic reasons the file pointer `fileptr` can be given. This allows using standard `fread` and `fwrite` calls and hence reduces the amount of code to be changed for integrating SIONlib into an applications I/O routine. The downside to this strategy is that SIONlib is not able to perform certain optimizations since the write calls happen outside of the library. Furthermore additional convenience features like writing the `chunksize` byes more than once and checks preventing excess of the given chunksize are not available without further code changes. Hence passing a `NULL` pointer for `fileptr` and using the SIONlib file id `sid` and the according SIONlib functions for reading and writing functions is strongly encouraged.

3.2.2 *Read*

```
size_t sion_fread(void*  ptr,
                  size_t size,
                  size_t nitems,
                  int    sid);
```

- `ptr`: target pointer to write to
- `size`: size of single item
- `nitems`: number of items
- `sid`: SIONlib file id to read from

Read `size * nitems` bytes into buffer `ptr` from the file with SIONlib file id `sid`.

3.2.3 *Write*

```
size_t sion_fwrite(const void* data,
                   size_t      size,
                   size_t      nitems,
                   int         sid);
```

- `data`: source buffer of data
- `size`: size of single item
- `nitems`: number of items
- `sid`: SIONlib file id to write to

Write `size * nitems` bytes from buffer `data` to file with SIONlib file id `sid`.

3.2.4 *Close*

```
int sion_parclose_mpi(int sid);
```

- `sid`: SIONlib file id to close

Close file with SIONlib file id `sid`.

3.3 Example

```
#include <stdlib.h>
#include <string.h>

#include <mpi.h>
#include <sion.h>

int main(int argc, char* argv[])
{
  int        numFiles = 1;
  MPI_Comm   lComm;
  sion_int64 chunksize = 100;
  int        fsblksize = -1;
  char*      newfname  = NULL;
  char*      buffer    = NULL;
  int        sid       = -1;

  int        size = 0, rank = 0;
  int        idx  = 0;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  buffer = (char*)malloc(chunksize);

  memset(buffer, 'A' + rank, chunksize);
```

```
    sid = sion_paropen_mpi("simple.sion",
                           "w",
                           &numFiles,
                           MPI_COMM_WORLD,
                           &lComm,
                           &chunksize,
                           &fsblksize,
                           &rank,
                           NULL,
                           &newfname);

    if (sid >= 0) {
      sion_fwrite(buffer,
                  sizeof (char),
                  chunksize,
                  sid);

      sion_parclose_mpi(sid);
    }
    else {
      fprintf(stderr, "on rank %d: error sid = %d\n", rank, sid);
    }

    free(buffer);

    return 0;
  }
```

This example shows a minimal write operation.

3.4 Further API Features

On top of this basic API there are different calls which handle more complex tasks. These tasks include reading SIONlib files with a different number of tasks than used during write and collecting data before write to enable output of small data efficiently.

The full API reference with the description of all user functions and the utility tools which are included in SIONlib can be found on the [SIONlib website].

3.5 Extensions in DEEP-ER

For the DEEP-ER project SIONlib will implement a mechanism for buddy checkpointing. This feature will be triggered during the open call by adding the flag "`buddycheckpoint`".

Buddy checkpointing will enable faster restarts from checkpoints if the same cache layer is still available. In addition to its own checkpointing data each task will also save data from its buddy task which SIOINlib ensures to be chosen from a different physical storage by using BeeGFS's new API features. From the application point of view only the open calls change while the write calls are transparently changed by SIONlib.

4 Exascale10 interface description

The Exascale10 description in D4.2 will address the following aspects

1. Exascale10 approach for collective I/O in DEEP-ER
2. Description of collective I/O interfaces
3. Description of MPI I/O hints extensions
4. Interfacing with the rest of DEEP-ER I/O software components

Since Exascale10 is a disruptive new approach, we wish to describe the key interfaces in some detail in the following sections and provide more detailed interface specification/description of parameters, provided on top of the MPI-IO, in the appendix.

4.1 General remarks on collective I/O

In the DEEP-ER Prototype, collective I/O will be re-architected to take advantage of NVMe devices on Booster Nodes. This will allow addressing two of the big problems that current implementation of collective I/O will encounter when moving to ultra large scales.

Firstly, the node concurrency level in future Exascale machines will grow much bigger than the available memory per core [MTA] actually reducing the available memory per core to a few Megabytes. This will also increase the memory pressure in terms of available I/O memory bandwidth per core.

In this scenario the usage of large collective I/O buffers becomes infeasible, forcing any implementation to deal with reduced available memory, drastically increasing the number of phases of data shuffle and file I/O in every collective I/O operation.

Secondly, the global synchronization overhead required at the end of every collective write operation might involve many thousands of processes moving the bottleneck from the I/O phase to post write synchronization.

The Exascale10 solution for collective I/O will address all the problems previously mentioned by leveraging the Booster Node architecture as well as file system functionalities. This will be done by, firstly, integrating the NVMe cache support in the MPI I/O middleware through the definition of additional MPI hints and APIs and, secondly, providing Exascale enhancements for the MPI collective I/O implementation (currently based on the extended two phase algorithm, ext2ph) that will go under the name of ExCol.

4.2 Description of collective I/O interfaces

In D4.1 we identified the following requirements for the interfaces:

1. Seamless integration with existing applications and frameworks
2. Re-use of as much of the existing foundations on similar solutions as possible

In this section we describe the currently available collective I/O interfaces for read and write operations in MPI I/O, along with the semantic extensions introduced by Exascale10 for the DEEP-ER collective I/O implementation. Additionally, we define a new collective I/O interface for data prefetching that can be used by applications to populate the local caches (NVMe) before data is collectively read, thus making possible also for reads to overlap I/O from the global file system with computation.

4.2.1 *Write to a shared file collectively*

```
MPI_File_write_all(MPI_File fh, void *buf, int count, MPI_Datatype
datatype, MPI_Status *status)
```

This is the routine provided by MPI I/O to write collectively to a shared file. The routine is blocking and at the end, in order to make sure that data in the file is consistent (data might be still buffered at some level in the node), it requires the user to call `MPI_File_sync()` or `MPI_File_close()` to flush all the buffers to the global file.

We want to keep the interface unchanged and at the same time extend its semantic. In fact, in our case the write operation will happen in two I/O phases. In the first phase data will be moved from all the MPI processes to a smaller number of I/O aggregators that write it to local cache devices. Once the first phase is completed `MPI_File_write_all()` returns. Nevertheless, non-blocking synchronization of local caches' content with global file system can be started right away before the control is returned to the application (using an appropriate internal synchronization function to be defined) if the proper option is selected (see hints in the next section).

The progress of the non-blocking synchronization phase (second I/O phase) can be "controlled" by invoking `MPI_File_sync()` on the `MPI_File` handle, that for this purpose will be extended to keep track of the non-blocking operation (similar to `MPI_File_iwrite()-> MPI_Wait()`). When called `MPI_File_sync()` or `MPI_File_close()` will block until synchronization is completed. This allows applications to overlap I/O in the synchronization phase of collective I/O with computation (see code example in the last sub-sections).

If at some point the application wants to read the data back, for example using POSIX, first of all it must make sure that local caches have been synchronized with global storage (local caches are not coherent) by invoking either `MPI_File_sync` or `MPI_File_close`.

4.2.2 *Load data into the user's buffer*

```
MPI_File_read_all(MPI_File fh, void *buf, int count, MPI_Datatype
datatype, MPI_Status *status)
```

This routine is similar to the write version but in this case data is first loaded into the local cache device by I/O aggregators and afterwards sent to the correct processes. If `MPI_File_prefetch_all()` is issued earlier, the data in the cache will be already available.

4.2.3 *Prefetch data into the local NVMe cache collectively*

```
MPI_File_prefetch_all(MPI_File fh, int count, MPI_Datatype,
MPI_Status *status)
```

This routine has almost the same signature of the routine used for collective read operations. The difference in this case is that it only transfers data to local caches without filling the final buffers (which indeed do not appear) and it does that without blocking. When invoked, later on, `MPI_File_read_all()` completes the collective read operation by filling the final user buffers.

Additional notes:

1. The naming convention for non-blocking routines in MPI is to prefix the operation name with an "i" (e.g. `MPI_File_{iread,iwrite}`). Additionally non-blocking routines return a pointer of type `MPI_Request` that can be used later on to check operation completion passing it to, for example, `MPI_Wait()`. In this case, since `MPI_File_prefetch_all` is intended to be paired with the corresponding collective read operation, `MPI_File_read_all` will be responsible for checking status completion before filling the user buffers. All the synchronization required will be handled internally by the middleware.

2. The routine will use MPI file views to build the aggregate file region. To avoid `MPI_File_read_all()` reworking out the aggregate file region again (which involves expensive global communication) the `MPI_File` handle will be extended to keep the collective I/O state that can be used afterwards by `MPI_File_read_all()` to recover the collective read operation from where `MPI_File_prefetch_all()` left it.

An independent I/O version of the routine will also be provided as part of the MPI I/O extensions for ROMIO (BeeGFS ADIO driver), named `MPI_File_prefetch`. Once again the signature is almost identical to `MPI_File_read` except for the missing user buffer.

More information on these MPI I/O templates as seen by MPI applications is available in [MPI Report].

Detailed specifications and parameters of these interfaces are available in the appendix.

4.3 Description of MPI I/O hints extensions

Collective I/O functionalities in MPI I/O can be controlled by means of hints. Hints can be set using the `MPI_Info_set()` routine and afterwards passed to the implementation using the `MPI_File_open()` routine. Here we use the hints mechanism provided by MPI I/O to extend the available functionalities with the DEEP-ER collective I/O approach.

- **`e10_cache`**: controls whether or not I/O operations will go through the cache layer. "`enable`" is used to specify that writes will be directed to the local cache. In the case of reads if the data is not cached (cache miss) data will be retrieved directly from global storage without additionally copying it to the local cache. To populate the cache with data the user has to explicitly call `MPI_File_prefetch_all` (this behaviour is compliant to `deeper_cache_open` from BeeGFS). "`disable`" is used to bypass the local cache and read/write directly from/to the global file system. If not specified, by default, I/O operations will always go to the global file system.

- **`e10_cache_path`**: string containing the path of the file in the local cache during I/O (this is not required for BeeGFS but is needed for portability in order to allow other file systems that do not support an additional cache layer to use fast local SSDs).

- **`e10_cache_flush_flag`**: controls when the flush (synchronization) of the cache should happen. "`flush_immediate`" is used to flush asynchronously right away before `MPI_File_write_all()` returns. "`flush_onclose`" is used to delay the synchronization until either `MPI_File_sync()` or `MPI_File_close()` is invoked. If not specified, by default, `flush_immediate` is selected.

- **e10_cache_discard_flag:** controls whether or not the file in the cache should be discarded after synchronization is over. "enable" is used to discard the local data after synchronization. "disable" is used to retain the file. If not specified, by default, the file will not be discarded.
- **e10_collective_mode:** controls the collective I/O implementation to use. "ext2ph" is used to specify that standard extended two phase I/O algorithm should be used. "excol" is used to specify that the new Exascale10 implementation should be used instead (the implementation is being defined at this stage). If not specified, by default, the standard ext2ph implementation is selected.

Other MPI I/O hints not listed above can be still used by the new implementation (e.g. hints to set striping information to be afterwards used for optimal file domain partitioning).

4.3.1 Collective write example

```
#include <mpi.h>
...
MPI_Info hints;
MPI_File fh;
MPI_Status stat;
char buff[1048576];
char filename[] = "/scratch/userA/projectX/checkpoints";
int  ret;

MPI_Info_create(&hints);
MPI_Info_set(hints,"e10_cache","enable");
MPI_Info_set(hints,"e10_collective_mode","excol");
MPI_Info_set(hints,"e10_cache_flush_flag","flush_immediate");
MPI_Info_set(hints,"striping_factor","65536");
MPI_Info_set(hints,"striping_unit","8");

MPI_File_open(MPI_COMM_WORLD, filename, MPI_MODE_WRONLY |
              MPI_MODE_CREATE, hints, &fh);

MPI_Info_free(&hints);MPI_File_set_view(fh, ...);

ret = MPI_File_write_all(fh, buff, 1048576, MPI_CHAR, &stat);

if (ret){
    /* start error handling here */
}

/* overlap synchronization and computation here ... */

ret = MPI_File_sync(fh); /* check the status of synchronization */

if (ret){
    /* start error handling here */
}

/* now data can be safely read back from global storage */
ret = MPI_File_read_all(fh, buff, 1048576, MPI_CHAR, NULL);

MPI_File_close(&fh);
```

To be noted in the previous example is that, since data in the cache is not discarded after synchronization, unless differently specified (through the corresponding hint) the collective read operation should also be able to access the data from the cache. ***Collective Read Example***

```
#include <mpi.h>
...
MPI_Info hints;
MPI_File fh;
MPI_Status stat;
char buff[1048576];
char filename[] = "/scratch/userA/projectX/checkpoints";
int  ret;

MPI_Info_create(&hints);
MPI_Info_set(hints,"e10_cache","enable");
MPI_Info_set(hints,"striping_factor","65536");
MPI_Info_set(hints,"striping_unit","8");

MPI_File_open(MPI_COMM_WORLD, filename, MPI_MODE_WRONLY |
              MPI_MODE_CREATE, hints, &fh);

MPI_Info_free(&hints);

MPI_File_set_view(fh, ...);

ret = MPI_File_prefetch_all(fh, 1048576, MPI_CHAR, &stat);

if (ret){
    /* start error handling here */
}

/* overlap prefetch and computation here (doesn't need data yet) */

/* read data from local NVM */
ret = MPI_File_read_all(fh, buff, 1048576, MPI_CHAR, NULL);

if (ret){
    /* start error handling here */
}

/* do computation here (needs prefetched data now) */

MPI_File_close(&fh);
```

4.4 Interfacing with the rest of DEEP-ER I/O software components

The Exascale10 collective I/O strategy will access the local NVMe caches through the interfaces exposed by BeeGFS. As an example, synchronization hints previously introduced can be passed down by the Exascale10 implementation to the BeeGFS cache management routines to synchronize locally cached data with the global file system (i.e. through `deeper_cache_flush_range`).

5 Summary and Conclusions

This deliverable described the I/O APIs that will be provided by the I/O middleware components (BeeGFS Extensions, SIONlib and Exascale10) and the BeeGFS file system for DEEP-ER Applications. Detailed semantics of the use of these APIs (with descriptions of the input and output parameters) were provided that can be exploited by WP6 (Applications) and WP5 (Resiliency Software).

BeeGFS provides a POSIX compatible file system interface, as well as extensions to efficiently utilize NVRAM in the DEEP-ER architecture as a local cache layer on top of the file system.

SIONlib provides an API to efficiently deal with the overheads of task local files created by applications, as well as resiliency functionality (Buddy check pointing), which will be the focus of its DEEP-ER implementation.

Exascale10 provide an interface to deal with very efficient collective I/O at scale first by exposing the NVRAM tiers and then providing advanced collective I/O aggregation (building on top of MPI-IO)

The ensuing deliverables provide the implementations of these functionalities and study its use and exploitation by the applications and the resiliency layers within DEEP-ER.

In this deliverable we have described the I/O interfaces in some detail within the DEEP-ER project. We next pursue the implementation of these functionalities.

6 References

[SIONlib website] http://www.fz-juelich.de/jsc/sionlib

[MPI Report] http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[ThakurGL96] Thakur, R., Gropp, W. and Lusk, E., An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In the Proceedings of the Proc. of the 6th Symposium on the Frontiers of Massively Parallel Computation.

[MTA] Vetter, Jeffrey S., et al. "HPC Interconnection Networks: The Key to Exascale Computing." *High Speed and Large Scale Scientific Computing*. Eds. W. Gentzsch, L. Grandinetti, and G. Joubert. IOS Press Log in to post comments Google Scholar BibTex Tagged XML Archives News Archive Colloquium Archive Conference Archive ORNL| Computing and Computational Sciences Directorate| Computer Science and Mathematics Division| ORNL Disclaimer, 2009.

7 Appendix A - BeeGFS interface specification (deeper_cache.h)

```
/**
 * Create a new directory on the cache layer of the current cache
domain.
 *
 * @param path path and name of new directory.
 * @param mode permission bits, similar to POSIX mkdir (S_IRWXU
etc.).
 * @return 0 on success, -1 and errno set in case of error.
 */
int deeper_cache_mkdir(const char *path, mode_t mode);

/**
 * Remove a directory from the cache layer.
 *
 * @param path path and name of directory, which should be removed.
 * @return 0 on success, -1 and errno set in case of error.
 */
int deeper_cache_rmdir(const char *path);

/**
 * Open a directory stream for a directory on the cache layer of the
current cache domain and
 * position the stream at the first directory entry.
 *
 * @param name path and name of directory on cache layer.
 * @return pointer to directory stream, NULL and errno set in case
of error.
 */
DIR *deeper_cache_opendir(const char *name);

/**
 * Close directory stream.
 *
 * @param dirp directory stream, which should be closed.
 * @return 0 on success, -1 and errno set in case of error.
 */
int deeper_cache_closedir(DIR *dirp);

/**
 * Open (and possibly create) a file on the cache layer.
 *
 * Any following write()/read() on the returned file descriptor will
write/read data to/from the
 * cache layer. Data on the cache layer will be visible only to
those nodes that are part of the
 * same cache domain.
 *
 * @param path path and name of file, which should be opened.
 * @param oflag access mode flags, similar to POSIX open (O_WRONLY,
O_CREAT, etc).
 * @param deeper_open_flags zero or a combination of the following
flags:
```

```
 *        DEEPER_OPEN_FLUSHONCLOSE to automatically flush written
data to global
 *           storage when the file is closed, asynchronously;
 *        DEEPER_OPEN_FLUSHWAIT to make DEEPER_OPEN_FLUSHONCLOSE a
synchronous operation, which
 *        means the close operation will only return after flushing
is complete;
 *        DEEPER_OPEN_DISCARD to remove the file from the cache
layer after it has been closed.
 * @return file descriptor as non-negative integer on success, -1
and errno set in case of error.
 */
int deeper_cache_open(const char* path, int oflag, int
deeper_open_flags);

/**
 * Close a file.
 *
 * @param fildes file descriptor of open file.
 * @return 0 on success, -1 and errno set in case of error.
 */
int deeper_cache_close(int fildes);

/**
 * Prefetch a file or directory (including contained files) from
global storage to the current
 * cache domain of the cache layer, asynchronously.
 * Contents of existing files with the same name on the cache layer
will be overwritten.
 *
 * @param path path to file or directory, which should be
prefetched.
 * @param deeper_prefetch_flags zero or a combination of the
following flags:
 *        DEEPER_PREFETCH_SUBDIRS to recursively copy all subdirs,
if given path leads to a
 *           directory;
 *        DEEPER_PREFETCH_WAIT to make this a synchronous operation.
 * @return 0 on success, -1 and errno set in case of error.
 */
int deeper_cache_prefetch(const char* path, int
deeper_prefetch_flags);

/**
 * Prefetch a file similar to deeper_cache_prefetch(), but prefetch
only a certain range, not the
 * whole file.
 *
 * @param path path to file, which should be prefetched.
 * @param pos start position (offset) of the byte range that should
be flushed.
 * @param num_bytes number of bytes from pos that should be flushed.
 * @param deeper_prefetch_flags zero or a combination of the
following flags:
 *        DEEPER_PREFETCH_SUBDIRS to recursively copy all subdirs,
if given path leads to a
```

```
 *           directory;
 *        DEEPER_PREFETCH_WAIT to make this a synchronous operation.
 * @return 0 on success, -1 and errno set in case of error.
 */
int deeper_cache_prefetch_range(const char* path, off_t start_pos,
size_t num_bytes, int deeper_prefetch_flags);

/**
 * Wait for an ongoing prefetch operation from
deeper_cache_prefetch[_range]() to complete.
 *
 * @param path path to file, which has been submitted for prefetch.
 * @param deeper_prefetch_flags zero or a combination of the
following flags:
 *        DEEPER_PREFETCH_SUBDIRS to recursively wait contents of
all subdirs, if given path leads
 *           to a directory.
 * @return 0 on success, -1 and errno set in case of error.
 */
int deeper_cache_prefetch_wait(const char* path, int
deeper_prefetch_flags);

/**
 * Flush a file from the current cache domain to global storage,
asynchronously.
 * Contents of an existing file with the same name on global storage
will be overwritten.
 *
 * @param path path to file, which should be flushed.
 * @param deeper_flush_flags zero or a combination of the following
flags:
 *        DEEPER_FLUSH_WAIT to make this a synchronous operation,
which means return only after
 *           flushing is complete;
 *        DEEPER_FLUSH_DISCARD to remove the file from the cache
layer after it has been flushed.
 * @return 0 on success, -1 and errno set in case of error.
 */
int deeper_cache_flush(const char* path, int deeper_flush_flags);

/**
 * Flush a file similar to deeper_cache_flush(), but flush only a
certain range, not the whole
 * file.
 *
 * @param path path to file, which should be flushed.
 * @param pos start position (offset) of the byte range that should
be flushed.
 * @param num_bytes number of bytes from pos that should be flushed.
 * @param deeper_flush_flags zero or a combination of the following
flags:
 *        DEEPER_FLUSH_WAIT to make this a synchronous operation.
 * @return 0 on success, -1 and errno set in case of error.
 */
int deeper_cache_flush_range(const char* path, off_t start_pos,
size_t num_bytes, int deeper_flush_flags);
```

```
/**
 * Wait for an ongoing flush operation from
deeper_cache_flush[_range]() to complete.
 *
 * @param path path to file, which has been submitted for flush.
 * @return 0 on success, -1 and errno set in case of error.
 */
int deeper_cache_flush_wait(const char* path);

/**
 * Return a unique identifier for the current cache domain.
 *
 * @param out_cache_id pointer to a buffer in which the ID of the
current cache domain will be
 *        stored on success.
 * @return 0 on success, -1 and errno set in case of error.
 */
int deeper_cache_id(const char* path, uint64_t* out_cache_id);
```

8 Appendix B - SIONlib interface specification

```
/* Serial and utility functions */

/* I/O */

/*!
 * @brief Write data to sion file.
 *
 * This is the basic function for writing data to the underlying
 * storage.
 *
 * @param[in]  data   pointer to data to be written
 * @param[in]  size   size of a single item
 * @param[in]  nitems number of items to be written
 * @param[in]  sid    sion file id to be written to
 *
 * @return     number of elements written
 */
size_t sion_fwrite(const void* data,
                   size_t      size,
                   size_t      nitems,
                   int         sid);

/*!
 * @brief Read data from sion file.
 *
 * This is the basic function for reading data from the underlying
 * storage.
 *
 * @param[out] data   pointer to data to be written
 * @param[in]  size   size of a single item
 * @param[in]  nitems number of items to be written
 * @param[in]  sid    sion file id to be written to
 *
 * @return     number of elements read
 */
size_t sion_fread(void*  ptr,
                  size_t size,
                  size_t nmemb,
                  int    sid);

/* helper functions */

/*!
 * @brief Function that indicates whether the end of file is reached
for this task.
 *
 * This means that the file pointer points behind the last byte of
last chunk of this task.
 * If all bytes of the current chunk are already read and there are
more chunks available
 * for this task, sion_feof will advance the file pointer to the
start position of the
 * next chunk in the sion file.
```

```
 * The function is a task local function, which can be called
independently from other MPI tasks.
 *
 * @param[in]   sid     sion file handle (in)
 *
 * @return      SION_SUCCESS (1) if end of last chunk reached
 *              SION_NOT_SUCCESS (0) otherwise
 */
int sion_feof(int sid);

/*!
 * @brief Flushed sion file.
 *
 * @param[in]   sid             sion file handle
 *
 * @return     SION_SUCCESS if ok
 */
int sion_flush(int sid);

/* get information (with sion datatypes) */
/*!
 * \ingroup sion_common
 * @brief Returns edianness of data in file sid
 *
 * @param[in] sid sion file handle
 *
 * @return    1-> big endian, 0-> little endian
 */
int sion_get_file_endianness(int sid);

/*!\brief SION get endianness
 *
 * @return 1-> big endian
 *         0 ->little endian
 */
int sion_get_endianness(void);

/*!@brief Function that returns the number of bytes available in the
current chunk.
 *
 * This function returns the number of bytes remaining in the
current chunk (bytes not read).
 * It is a local function, which can be called independently from
other MPI tasks.
 *
 * @param[in]   sid             sion file handle
 *
 * @return      rc>0  number of bytes
 *              rc<=0 file position is after end of block
 */
sion_int64 sion_bytes_avail_in_chunk(int sid);

/* Seeking */

/*!
 * @brief Seek to a new position (multi-file version).
```

```
 *
 * This function seeks the file pointer to a new position according
to to the specified parameters.
 * It can only be used if the sion file was opened for reading in
serial mode.
 * In write mode currently only the rank can be modified!
 * SION_CURRENT_BLK and SION_CURRENT_POS are required
 *
 * @param[in]   sid             sion file handle
 * @param[in]   rank            rank number of the process
(SION_CURRENT_RANK to select the current rank)
 * @param[in]   currentblocknr  block number (SION_CURRENT_BLK to
select the current block)
 * @param[in]   posinblk        position in the block
(SION_CURRENT_POS to select the current position)
 * @param[out]  **fileptr       file pointer to corresponding file
of a multi-file set
 *
 * @retval      SION_SUCCESS if file pointer can be moved to new
position
 */
int sion_seek_fp(int        sid,
                 int        rank,
                 int        currentblocknr,
                 sion_int64 posinblk,
                 FILE**     fileptr);

/* serial interface routines */

/*!
 * @brief Open a sion file in serial mode.
 *
 * This function opens all tasks of a sion file in serial mode. The
 * meta-data of all tasks will be loaded into memory, which allows
 * accessing different parts of the file by using sion_seek_fp(),
for
 * example to switch between ranks. Default position of the
 * file pointer is in front of the first byte of the first task.
 *
 * In the case of multi-file sion-files sion_open() open internally
 * each physical file with sion_open recursively. sion_open manage
for
 * each physical file an own sion file descriptor and data
structure.
 *
 * @param[in]           fname           name of file, should be
equal on all tasks
 * @param[in,out]       file_mode       like the type parameter of
fopen (currently recognized options: "rb", "wb")
 * @param[in,out]       ntasks          number of tasks used to
write this file
 * @param[in,out]       nfiles          number of physical files
 * @param[in,out]       chunksizes      chunksize for each task
 * @param[in,out]       fsblksize       blocksize of filesystem,
must be equal on all processors
```

```
 * @param[in]           globalranks     rank numbers for which the
file should be open;
 *                                      will be stored in sion file,
useful if comm is not MPI_COMM_WORLD
 *                                      typical: globalrank = rank
in MPI_COMM_WORLD
 * @param[out]          fileptr         file pointer for this task
 *
 * @retval              sid             sion file handle or -1 if
error occured
 */
int sion_open(char*        fname,
              const char*  file_mode,
              int*         ntasks,
              int*         nfiles,
              sion_int64** chunksizes,
              sion_int32*  fsblksize,
              int**        globalranks,
              FILE**       fileptr);

/*!
 * @brief Open a sion file for a specific rank
 *
 * This function opens a sion file for a specific rank.  It can be
 * used to open the sion file independently from each task.
 * (e.g. if no MPI is available or only a subset of tasks chunks are
needed)
 *
 * Using this function the meta data at the beginning of the sion
file is read
 * by each task instead of read once and distribute (sion_open).
 * sion_open_rank reads only the tasks specific meta data  from the
meta data
 * block at the end of the sion file.
 *
 * All metadata will be read and stored in internal data structures.
 *
 * Warning: Only read operations are currently supported.
 *
 * @param[in]           fname           name of file, should be
equal on all tasks
 * @param[in,out]       file_mode       like the type parameter of
fopen (currently recognized options: "rb", "wb")
 * @param[in,out]       chunksize       chunksize for this task
 * @param[in,out]       fsblksize       blocksize of filesystem,
must be equal on all processors
 * @param[in]           rank            rank number for which the
file should be open;
 *                                      will be stored in sion file,
usefull if comm is not MPI_COMM_WORLD
 *                                      typical: globalrank= rank in
MPI_COMM_WORLD
 * @param[out]          fileptr         file pointer for this task
 *
 * @retval              sid             sion file handle or -1 if
error occured
```

```
 */
int sion_open_rank(char*       fname,
                   const char* file_mode,
                   sion_int64* chunksize,
                   sion_int32* fsblksize,
                   int*        rank,
                   FILE**      fileptr);

/*!
 * @brief Close a sion file.
 *
 * This function closes a sion file which was opened in serial mode
with sion_open() or sion_open_rank().
 * In write mode this function will also save all meta data to the
meta data block of the sion file
 * at the beginning and the end of the file.
 * The function is a task local function, which can be called
independently from other MPI tasks.
 *
 *  @param[in]  sid     sion file handle
 *
 *  @return     SION_SUCCESS if close is ok
 */
int sion_close(int sid);

/* Parallel Interface: MPI  */

/*!
 * @brief Open a sion file using MPI.
 *
 * This function opens a sion file using MPI. The chunksizes are
 * communicated globally which enables each task to calculate the
 * correct offset to prevent different ranks from trying to access
the
 * same parts of the file.
 *
 * @param[in]      fname        name of file, should be equal on all
tasks
 * @param[in,out]  file_mode    like the type parameter of fopen
(currently recognized options: "rb", "wb")
 * @param[in,out]  numFiles     number of multi files to use (-1 for
automatic choosing from local communicator)
 * @param[in]      gComm        global MPI communicator
 * @param[in]      lComm        local MPI communicator (= gComm if
no adaption to I/O nodes is needed)
 * @param[in,out]  chunksize    maximum size to be written with
single write call
 * @param[in,out]  fsblksize    file system block size (-1 for
automatic)
 * @param[in]      globalrank   global rank of process
 * @param[in]      fileptr      file pointer (NULL for not using an
external file pointer)
 * @param[out]     newfname     return value for actual file name if
using multi files
 *
```

```
 * @retval         sid          sion file handle or -1 if error
occured
 */
int sion_paropen_mpi(char*       fname,
                     const char* file_mode,
                     int*        numFiles,
                     MPI_Comm    gComm,
                     MPI_Comm*   lComm,
                     sion_int64* chunksize,
                     sion_int32* fsblksize,
                     int*        globalrank,
                     FILE**      fileptr,
                     char**      newfname);

/*!
 * @brief Close a sion file using MPI.
 *
 * Closing a sion file using MPI is a collective operation. In this
 * call the amount of data actual written is sent to the master rank
 * which writes this information to the second metadata block.
 *
 * @param[in]  sid  sion id
 *
 * @retval     sid  SION_SUCCESS if successful
 *                  SION_NOT_SUCCESS otherwise
 */

int sion_parclose_mpi(int sid);
```

9 Appendix C - E10 interface specification

```
int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info
info, MPI_File *mpi_fh);
/**
 * Opens a file (collective).
 * @param comm is the object describing the MPI communicator.
 * @param filename is the string identifying the file name.
 * @param amode is the integer identifying the file access mode.
 * @param info is the object containing the MPI hints.
 * @param mpi_fh is the MPI file handle.
 * @return MPI_SUCCESS or the appropriate error code.
 *
 * E10 hints that can be used as values in the 'info' argument:
 * - e10_cache: enable/disable local cache support for I/O.
 * - e10_cache_path: string containing the path of the file in the
local cache (not needed for BeeGFS).
 * - e10_cache_flush_flag: when cached data should be synched, either
'flush_immediate' or 'flush_onclose'.
 * - e10_cache_discard_flag: enable/disable closed file discard from
local cache.
 * - e10_collective_mode: select collective I/O mode either 'ext2ph' or
'excol'.
 */

int MPI_File_close(MPI_File *mpi_fh);
/**
 * Closes a file.
 * @param mpi_fh is the MPI file handle.
 * @return MPI_SUCCESS or the appropriate error code.
 *
 * E10 notes:
 * when 'e10_cache' hint is 'enable' and 'e10_cache_flush_flag' is
'flush_onclose' this routine also starts cache synchronization with
global storage.
 */

int MPI_File_sync(MPI_File mpi_fh);
/**
 * Causes all previous writes to be transfered to the storage device.
 * @param mpi_fh is the MPI file handle
 * @return MPI_SUCCESS or the appropriate error code.
 *
 * E10 notes:
 * when 'e10_cache' hint is 'enable' the user can call this routine to
check up on completion of MPI-IO write operations ('flush_immediate')
before to read data back or to start cache synchronization with global
storage ('flush_onclose').
 */

int MPI_File_read(MPI_File mpi_fh, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status);
/**
 * Independent read using individual file pointers.
```

```
 * @param mpi_fh is the MPI file handle.
 * @param buf is the user buffer to be filled.
 * @param count is the read size expressed in datatype units.
 * @param datatype is the MPI datatype object describing the data
layout in the file.
 * @param status is the MPI status object handle.
 * @return MPI_SUCCESS or the appropriate error code.
 *
 * E10 notes:
 * when 'e10_cache' is 'enable' data will be retrieved from the cache.
If the cache is empty (cache miss) data will be retrieved from global
storage.
 */
```

```
int MPI_File_read_at(MPI_File mpi_fh, MPI_Offset offset, void *buf,
int count, MPI_Datatype datatype, MPI_Status *status);
/**
 * Independent read using explicit offset.
 * @param mpi_fh is the MPI file handle.
 * @param offset is the file offset.
 * @param buf is the user buffer to be filled.
 * @param count is the read size expressed in datatype units.
 * @param datatype is the MPI datatype object describing the data
layout in the file.
 * @param status is the MPI status object handle.
 * @return MPI_SUCCESS or the appropriate error code.
 *
 * E10 notes:
 * when 'e10_cache' is 'enable' data will be retrieved from the cache.
If the cache is empty (cache miss) data will be retrieved from global
storage.
 */
```

```
int MPI_File_read_all(MPI_File mpi_fh, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status);
/**
 * Collective read using individual file pointers.
 * @param mpi_fh is the MPI file handle.
 * @param buf is the user buffer to be filled.
 * @param count is the read size expressed in datatype units.
 * @param datatype is the MPI datatype object describing the data
layout in the file.
 * @param status is the MPI status object handle.
 * @return MPI_SUCCESS or the appropriate error code.
 *
 * E10 notes:
 * when 'e10_cache' is 'enable' data will be retrieved from the cache.
If the cache is empty (cache miss) data will be retrieved from global
storage.
 */
```

```
int MPI_File_read_at_all(MPI_File mpi_fh, MPI_Offset offset, void
*buf, int count, MPI_Datatype datatype, MPI_Status *status);
/**
 * Collective read using explicit offset.
```

```
 * @param mpi_fh is the MPI file handle.
 * @param offset is the file offset.
 * @param buf is the user buffer to be filled.
 * @param count is the read size expressed in datatype units.
 * @param datatype is the MPI datatype object describing the data
layout in the file.
 * @param status is the MPI status object handle.
 * @return MPI_SUCCESS or the appropriate error code.
 *
 * E10 notes:
 * when 'e10_cache' is 'enable' data will be retrieved from the cache.
If the cache is empty (cache miss) data will be retrieved from global
storage.
 */
```

int MPI_File_write(MPI_File mpi_fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);

```
/**
 * Independent write using individual file pointers.
 * @param mpi_fh is the MPI file handle.
 * @param buf is the user buffer to be written.
 * @param count is the read size expressed in datatype units.
 * @param datatype is the MPI datatype object describing the data
layout in the file.
 * @param status is the MPI status object handle.
 * @return MPI_SUCCESS or the appropriate error code.
 *
 * E10 notes:
 * when 'e10_cache' is 'enable' data will be written to the cache.
Synchronization can be started either immediately ('flush_immediate')
or at close time ('flush_onclose'). Cache synchronization completion
status can be checked using either MPI_File_sync or MPI_File_close.
 */
```

int MPI_File_write_at(MPI_File mpi_fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);

```
/**
 * Independent write using explicit offset.
 * @param mpi_fh is the MPI file handle.
 * @param offset is the file offset.
 * @param buf is the user buffer to be written.
 * @param count is the read size expressed in datatype units.
 * @param datatype is the MPI datatype object describing the data
layout in the file.
 * @param status is the MPI status object handle.
 * @return MPI_SUCCESS or the appropriate error code.
 *
 * E10 notes:
 * when 'e10_cache' is 'enable' data will be written to the cache.
Asynchronous synchronization can be started either immediatelly
('flush_immediate') or at close time ('flush_onclose'). Cache
synchronization completion status can be checked using either
MPI_File_sync or MPI_File_close.
 */
```

int MPI_File_write_all(MPI_File mpi_fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);

```
/**
 * Collective write using individual file pointers.
 * @param mpi_fh is the MPI file handle.
 * @param buf is the user buffer to be written to the file.
 * @param count is the write size expressed in datatype units.
 * @param datatype is the MPI datatype object describing the data
layout in the file.
 * @param status is the MPI status object handle.
 * @return MPI_SUCCESS or the appropriate error code.
 *
 * E10 notes:
 * when 'e10_cache' is 'enable' data will be written to the cache.
Asynchronous synchronization can be started either immediatelly
('flush_immediate') or at close time ('flush_onclose'). Cache
synchronization completion status can be checked using either
MPI_File_sync or MPI_File_close.
 */
```

int MPI_File_write_at_all(MPI_File mpi_fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);

```
/**
 * Collective write using explicit offset.
 * @param mpi_fh is the MPI file handle.
 * @param offset is the file offset.
 * @param buf is the user buffer to be written to the file.
 * @param count is the write size expressed in datatype units.
 * @param datatype is the MPI datatype object describing the data
layout in the file.
 * @param status is the MPI status object handle.
 * @return MPI_SUCCESS or the appropriate error code.
 *
 * E10 notes:
 * when 'e10_cache' is 'enable' data will be written to the cache.
Asynchronous synchronization can be started either immediatelly
('flush_immediate') or at close time ('flush_onclose'). Cache
synchronization completion status can be checked using either
MPI_File_sync or MPI_File_close.
 */
```

int MPI_File_prefetch(MPI_File mpi_fh, int count, MPI_Datatype datatype, MPI_Status *status);

```
/**
 * Independent prefetch using individual file pointers.
 * @param mpi_fh is the MPI file handle.
 * @param count is the prefetch size expressed in datatype units.
 * @param datatype is the MPI datatype object describing the data
layout in the file.
 * @param status is the MPI status object handle.
 * @return MPI_SUCCESS or the appropriate error code.
 *
 * E10 notes:
 * this routine requires 'e10_cache' is set to 'enable'.
 */
```

int MPI_File_prefetch_at(MPI_File mpi_fh, MPI_Offset offset, int count, MPI_Datatype datatype, MPI_Status *status);

```
/**
 * Independent prefetch using explicit offset.
 * @param mpi_fh is the MPI file handle.
 * @param offset is the file offset.
 * @param count is the prefetch size expressed in datatype units.
 * @param datatype is the MPI datatype object describing the data
layout in the file.
 * @param status is the MPI status object handle.
 * @return MPI_SUCCESS or the appropriate error code.
 *
 * E10 notes:
 * this routine requires 'e10_cache' is set to 'enable'.
 */
```

int MPI_File_prefetch_all(MPI_File mpi_fh, int count, MPI_Datatype datatype, MPI_Status *status);

```
/**
 * Collective prefetch.
 * @param mpi_fh is the MPI file handle.
 * @param count is the prefetch size expressed in datatype units.
 * @param datatype is the MPI datatype object describing the data
layout in the file.
 * @param status is the MPI status object handle.
 * @return MPI_SUCCESS or the appropriate error code.
 *
 * E10 notes:
 * this routine requires 'e10_cache' is set to 'enable'.
*
 */
```

int MPI_File_prefetch_at_all(MPI_File mpi_fh, MPI_Offset offset, int count, MPI_Datatype datatype, MPI_Status *status);

```
/**
 * Collective prefetch using explicit offset.
 * @param mpi_fh is the MPI file handle.
 * @param offset is the file offset.
 * @param count is the prefetch size expressed in datatype units.
 * @param datatype is the MPI datatype object describing the data
layout in the file.
 * @param status is the MPI status object handle.
 * @return MPI_SUCCESS or the appropriate error code.
 *
 * E10 notes:
 * this routine requires 'e10_cache' is set to 'enable'.
 */
```

List of Acronyms and Abbreviations

A

API: Application Programming Interface

B

BIOS: Basic I/O system. Boot and system initialisation code run before the OS starts

BLN: Brick local network. Used to locally connect the Brick modules

BMC: Board management controller. Used to physically monitor and manage a compute blade.

BN: Booster Node (functional entity)

BoP: Board of Partners for the DEEP-ER project

C

D

DAG: Directed acyclic graph.

DDG: Design and Developer Group of the DEEP-ER project

DEEP: Dynamical Exascale Entry Platform

DEEP-ER: DEEP Extended Reach: this project

DEEP-ER Prototype: Demonstrator system for the extended DEEP Architecture, based on second generation Intel® Xeon Phi™ CPUs, connecting BN and CN via a single, uniform network and introducing NVM and NAM resources for parallel I/O and multi-level checkpointing

DEEP Architecture: Functional architecture of DEEP (e.g. concept of an integrated Cluster Booster Architecture), to be extended in the DEEP-ER project

E

EC: European Commission

Exascale: Computer systems or Applications, which are able to run with a performance above 1018 Floating point operations per second

F

FhGFS: Fraunhofer Global File system, a high-performance parallel I/O system to be adapted to the extended DEEP Architecture and optimised for the DEEP-ER Prototype

FLOP: Floating point Operation

FP7: European Commission 7th Framework Programme.

G

H

HPC: High Performance Computing

I

ICT: Information and Communication Technologies

Intel: Intel Germany GmbH Feldkirchen,

I/O: Input/Output. May describe the respective logical function of a computer system or a certain physical instantiation

J

JUELICH: Forschungszentrum Jülich GmbH, Jülich, Germany

K

L

M

MPI: Message Passing Interface, API specification typically used in parallel programs that allows processes to communicate with one another by sending and receiving messages

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

NVM: Non-Volatile Memory. Used to describe a physical technology or the use of such technology in a non-block-oriented way in a computer system

NVMe: Short form of NVM-Express

NVM-Express: An interface standard to attach NVM to a computer system. Based on PCI Express it also standardises high level HW interfaces like queues.

O

OpenMP: Open Multi-Processing, Application programming interface that support multiplatform shared memory multiprocessing

P

PCI: Peripheral Component Interconnect, Computer bus for attaching hardware devices in a computer

PCIe: Short form of PCI Express

PCI Express: Peripheral Component Interconnect Express started as an option for a physical layer of PCI using high-performance serial communication. It is today's standard interface for communication with add-on cards and on-board

devices, and makes inroads into coupling of host systems. PCI Express has taken over specifications of higher layers from the PCI baseline specification.

PFlop/s: Petaflop, 1015 Floating point operations per second

PM: Person Month or Project Manager of the DEEP project (depending on the context)

PMT: Project Management Team of the DEEP-ER project

POSIX: Portable Operating System Interface. A family of standards specified by the IEEE for maintaining compatibility between operating systems.

Q

R

S

SSD: Solid State Disk

SW: Software

T

TFlop/s: Teraflop, 1012 Floating point operations per second

U

V

W

X

Y

Z