SEVENTH FRAMEWORK PROGRAMME

FP7-ICT-2013-10

DEEP-ER

DEEP Extended Reach

Grant Agreement Number: 610476

D4.4

I/O software packages

Approved

Version: 2.0

Author(s): K. Thust (JUELICH)

Contributor(s): G. Congiu (Seagate), F. Kautz (FHG-ITWM)

Date: 04.05.2017

Project and Deliverable Information Sheet

<table>
<tr><td rowspan="7">DEEP-ER Project</td><td colspan="2">Project Ref. №: 610476</td></tr>
<tr><td colspan="2">Project Title: DEEP Extended Reach</td></tr>
<tr><td colspan="2">Project Web Site: http://www.deep-er.eu</td></tr>
<tr><td colspan="2">Deliverable ID: D4.4</td></tr>
<tr><td colspan="2">Deliverable Nature: Report</td></tr>
<tr><td rowspan="2">Deliverable Level:
PU *</td><td>Contractual Date of Delivery:
30 / June / 2016</td></tr>
<tr><td>Actual Date of Delivery:
30 / June / 2016</td></tr>
<tr><td colspan="2">EC Project Officer: Panagiotis Tsarchopoulos</td></tr>
</table>

* - 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

<table>
<tr><td rowspan="6">Document</td><td colspan="2">Title: I/O software packages</td></tr>
<tr><td colspan="2">ID: D4.4</td></tr>
<tr><td>Version: 2.0</td><td>Status: Approved</td></tr>
<tr><td colspan="2">Available at: http://www.deep-er.eu</td></tr>
<tr><td colspan="2">Software Tool: Microsoft Word</td></tr>
<tr><td colspan="2">File(s): DEEP-ER_D4.4_IO_software_packages_v2.0-ECapproved</td></tr>
<tr><td rowspan="4">Authorship</td><td>Written by:</td><td>K. Thust (JUELICH)</td></tr>
<tr><td>Contributors:</td><td>G. Congiu (Seagate), F. Kautz (FHG-ITWM)</td></tr>
<tr><td>Reviewed by:</td><td>G. Lapenta (KULeuven), E. Suarez (JUELICH)</td></tr>
<tr><td>Approved by:</td><td>BoP/PMT</td></tr>
</table>

Document Status Sheet

Version	Date	Status	Comments
1.0	30/June/2016	Final	EC submission
2.0	04/May/2017	Approved	EC approved

Document Keywords

Keywords:	DEEP-ER, HPC, Exascale, I/O Architecture, Benchmarking

Table of Contents

List of Figures

Executive Summary

This deliverable describes the three I/O pages that are used in the DEEP-ER project. These are the bridging components between the hardware (WP3) in one side and resiliency (WP5) and applications (WP6) in the other. A description of the BeeGFS filesystem is presented in this document, followed by a detailed presentation of the middleware packages SIONlib and E10. The focus of the latter two packages, that lay on top of BeeGFS, is the optimisation of I/O for task-local and collective I/O, respectively.

The different sections of this document give a short introduction about the basic features of the software packages and describe the developments that have been completed so far in the course of the DEEP-ER project.

1 Introduction

Scalable I/O is a challenging task in current HPC systems. For an increasing number of applications access to fast and reliable I/O systems becomes critical. The performance improvements in HPC for the last generations of supercomputers strongly favoured computation over I/O capabilities. This trend will cause I/O to become a major limiting factor for the scalability of Exascale systems.

In the DEEP-ER project this problem is addressed on different levels. As a fundamental layer the hardware architecture (WP3) provides storage that is local to the compute nodes (NVMe). With a constant ratio between the number of I/O units and computing power, this new system allows to scale to large systems. The BeeGFS parallel filesystem makes this scalable architecture available for the layers above it by providing the concept of cache domains.

In addition, WP4 has been working on the packages SIONlib and E10, which are developed from the application perspective to ease the use of this new architecture. Depending on the I/O strategy, applications can use one of the libraries or both, in order to maximise the benefits of the DEEP-ER concepts. This minimises the need for code intrusion in the applications while still using the advances of the underlying layers.

While deliverable D4.2 focused on the APIs that are accessible to the applications, the present deliverable describes the implementation of the APIs themselves.

2 BeeGFS

The connection between the storage devices and the applications is provided by the filesystem. In the DEEP-ER project BeeGFS constitutes the parallel filesystem for the I/O middleware (SIONlib or E10) and also gives direct access to the applications. See the following overview about the different parts of the BeeGFS development in the context to the DEEP-ER project.

2.1 Intermediate layer

The intermediate layer is implemented as designed in the deliverable 4.1 and shown in Figure 2.1-1. The global filesystem is a common BeeGFS installation. The servers use spinning hard-drives to store the file content and SSDs to store the metadata of the files. The cache filesystem is implemented by BeeOND and the files, including the metadata, are stored on the NVMe devices. BeeOND is a startup script which supports dynamic and flexible creation of a BeeGFS filesystem. It allows the creation of a BeeGFS instance during the placement of a job in the cluster by its queuing system. In the DEEP-ER project a cache domain consists of one node. The cache FS is created during the start up. This system has been deployed on the DEEP Cluster and on the DEEP-ER SDV.

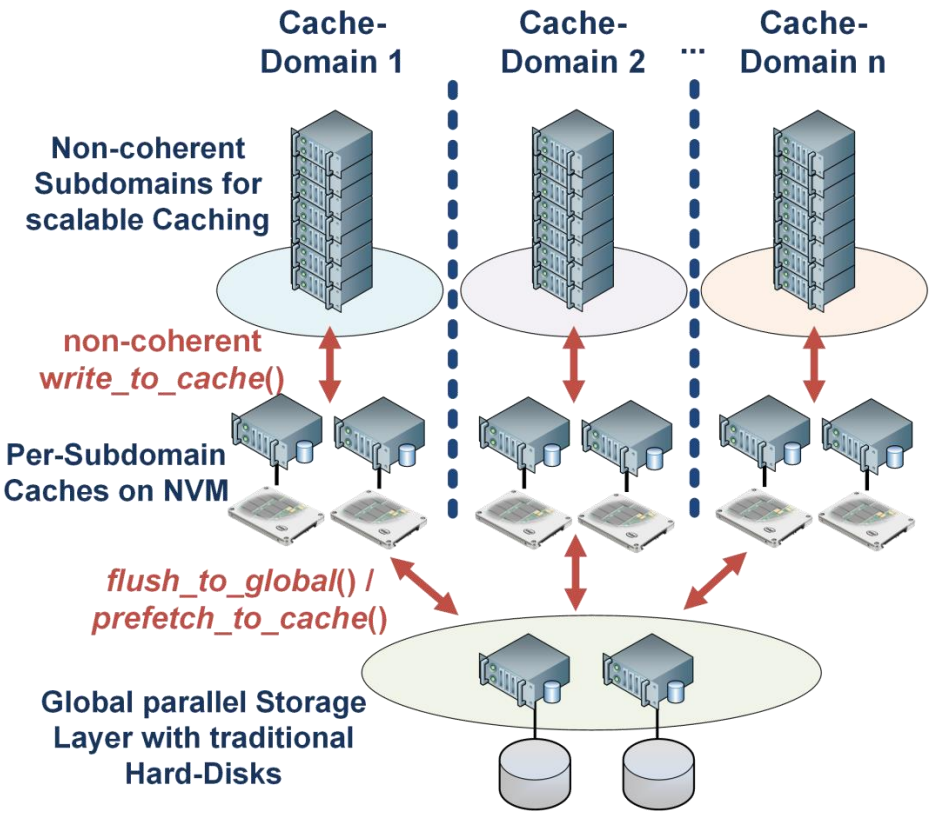

Figure 2.1-1: Cache architecture

2.2 I/O API extension

2.2.1 BeeGFS API

The BeeGFS Striping API allows creating files in the filesystem with a data stripe configuration that matches the I/O requirements of the application. The application can define how many targets, which stripe size and which stripe pattern (RAID0 or Buddy Mirroring) should be used. For example, the application can select more targets for big files and fewer targets for smaller files. This helps to optimize the I/O throughput of the applications. The API allows the application to query the stripe configuration and the used targets of a file from BeeGFS. It is also possible to check if a filesystem path is actually inside a BeeGFS filesystem or not.

2.2.2 Cache API

The cache API is implemented as specified in the deliverable 4.2. During the development of WP5 additional requirements were added to its specifications, leading to an extension of the API. All requirements were implemented and tested. Initially BeeGFS provided a synchronous API, which was later extended to work asynchronously. This helped the developers to integrate the cache API into their applications and test the functionality at an early stage in the project. The implementation of the synchronous version of the API could be finished earlier than the asynchronous version; this allowed a decoupling between the implementation of asynchronous I/O in BeeGFS as well as the integration into applications. The application developers can use the synchronous API with the same interfaces as the asynchronous API. To change from the synchronous API to the asynchronous version the application only needs to be re-linked with the new version of the library.

2.2.2.1 Cache API feature requests from the application developers

During the prototyping phase several telephone conferences and meetings with WP5 and WP6 took place to collect several additional requirements from the application developers. This lead to extra implementations and testing of the new features:

- The flush and the prefetch functions support a “follow symlink” feature in the new specification. Using a flag, the files and directories behind the symlinks are copied to the cache filesystem or the global filesystem. To avoid infinite recursion in the filesystem structure the number of symlinks to follow in a chain is limited to 20.
- CRC checksum calculation is supported during the flush and prefetch function. The cyclic redundancy check (CRC) code is a commonly used and highly efficient method to calculate checksums for streamed data. The calculation during the flush or prefetch avoids additional reads from the filesystem to calculate the CRC checksum of files by the user application.
- Two functions were implement and tested giving more control to the asynchronous flush and the prefetch operations. The API provides a function to check if the flush of a file or directory is ongoing, and a function to stop a running flush or prefetch.

Adaptions to the original API allowed fulfilling all application requirements.

2.2.2.2 Cache API implementation details

The synchronous version of the API implements the full function set of the specification and the requested API features “follow symlink” and “calculate CRC checksum”. The processing of the functions is done in the context of the thread that called the API function.

The asynchronous version of the API implements all functions of the synchronous API and in addition implements the requested “control functions” for the asynchronous flush or prefetch. All prefetch and flush operations are executed asynchronously in this version of the API. The asynchronous implementation allows improving the performance compared to the synchronous version in several ways. First the flush or prefetch functions are offloaded to a cache daemon, which is running on the same node in the asynchronous version. The offloading affects all prefetch or flush functions - the “follow symlink”, the “calculate CRC checksum” and the “byte range” version.

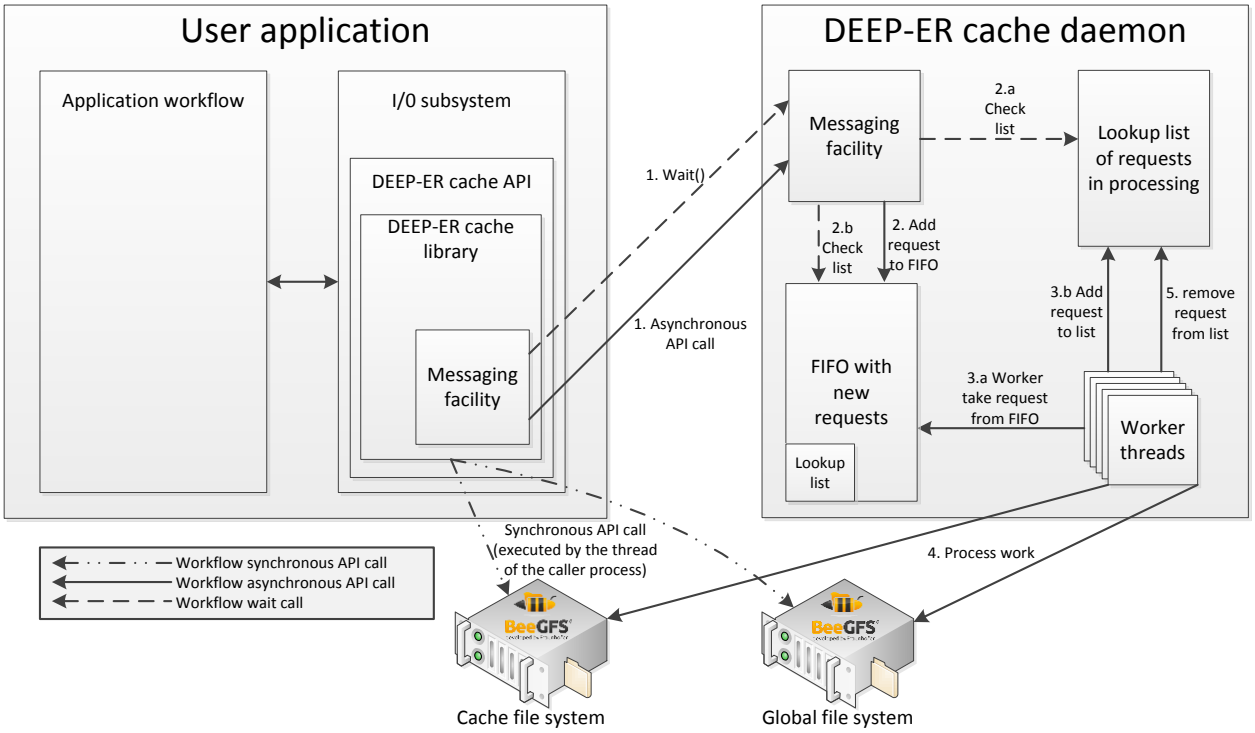

Figure 2.2-1: Asynchronous API architecture

A second performance improvement is implemented by splitting big files into byte range flushes or prefetches to use multiple threads. The number of worker threads of the cache daemon is configurable, because it has to be adjusted to the cluster environment. A thread count which is too high could have a negative impact on the cluster jobs on the node and if the thread count is to low the cluster jobs of the node have longer I/O waits. The minimal size of a file before it should be split is configurable as well. This size is also used as byte range size for the range flushes or prefetches, so big files are split multiple times for range flushes or prefetches.

All asynchronous requests are processed by the cache daemon in a first in - first out queue. Lookup tables are used to get an efficient access to the requests in the different stages. The failed requests, the CRC checksums of finished requests and requests that currently processed are organised in different lookup tables. Errors during the asynchronous flushes

and prefetches are reported to the API by the wait function. The architecture and the workflow of the synchronous/asynchronous operations are shown in Figure 2.2-1.

A code example for the usage of the cache API is given in Figure 2.2-2.

```
// create the directory on cache FS
funcError = deeper_cache_mkdir(dir.c_str(),S_IRWXU | S_IRGRP |
S_IROTH);
if(funcError == DEEPER_RETVAL_ERROR)
{
   // error handling
}

// prefetch the file from global FS to the cache FS
funcError = deeper_cache_prefetch(file.c_str(),
DEEPER_PREFETCH_NONE);
if(funcError == DEEPER_RETVAL_ERROR)
{
   // error handling
}

… // do something during the prefetch

// wait until the prefetch is finished
funcError = deeper_cache_prefetch_wait(file.c_str(),
DEEPER_PREFETCH_NONE);
if(funcError == DEEPER_RETVAL_ERROR)
{
   // error handling
}

// open file in the cache FS
int cacheFD = deeper_cache_open(file.c_str(), O_RDWR, S_IRWXU |
S_IRGRP | S_IROTH, DEEPER_OPEN_NONE);
if(cacheFD == -1)
{
   // error handling
}

… // modify the file in the cache FS by POSIX operations

// close file in the cache FS
funcError = deeper_cache_close(cacheFD);
if(funcError == DEEPER_RETVAL_ERROR)
{
   // error handling
}

// flush the file from the cache FS to the global FS and wait until
// the flush is finished
```

```
funcError = deeper_cache_flush(file.c_str(), DEEPER_FLUSH_WAIT);
if(funcError == DEEPER_RETVAL_ERROR)
{
   // error handling
}
```

Figure 2.2-2: Code example of the asynchronous cache API.

2.3 BeeGFS on the DEEP Booster nodes

In the DEEP-ER project the Intel Knights Corner (KNC)-based Booster constructed within DEEP has been used as software development and test platform. Porting BeeGFS to KNC involved changes in the beegfs-client code. The Intel Knights Landing (KNL) servers, that became available just recently and have been integrated in the DEEP-ER SDV, is different to its predecessor in that every BeeGFS package can be installed without any additional adaptation. The first preliminary tests of all BeeGFS components including the BeeOND scripts have been successfully done and everything works as expected.

2.4 Resiliency and reliability

A data mirroring system has been implemented, featuring a built-in high availability functionality in order to achieve better resiliency and reliability. Using this system, the data is still accessible to the cluster even if a node that contains a mirrored file should fail. Mirroring can be enabled on a per-directory basis, so specially selected data in the filesystem can be mirrored. Data mirroring on storage targets is based on so-called buddy groups, in general consisting of two targets, a primary target and a secondary target. They should be located on different physical servers and should be of the same size. It is not required to fulfil both requirements, but this configuration is recommended to get an optimal performance. A target that belongs to a buddy group is still available to store unmirrored data as well, making it possible to have a filesystem that only mirrors a certain subset of the data. Figure 2.4-1 shows how 2 mirrored files (yellow, red) and one unmirrored file (green) is striped across the targets when mirroring is enabled. Modifying operations will always be sent to the primary target first, which takes care of the mirroring process. File contents are mirrored synchronously, i.e. the client operation completes after both copies of the data were transferred to the servers. If, for some reason, the primary storage target of a buddy group is unreachable for the system, a failover to the secondary target will be issued. In this case, the secondary target will become primary target from that point on.

To avoid split-brain situations (e.g. a disconnection of about 50% of the whole system could trigger the two – now separated – systems to run independently from each other) failover information has to be propagated to every component of the system, before the actual failover is performed. The failover does not happen immediately and certain timeouts need to be maintained.

Originally, we studied the possibility to implement asynchronous data triplication as an additional step. However, several unforeseen issues, e.g. problems with the network protocol, and changes in the API requirements, forced us to redefine our priorities. Since for the applications the implemented replication feature is already a major enhancement for resiliency and reliability and they do not expect needing further levels or replication within the DEEP-ER project, we decided to set a low priority for triplication.

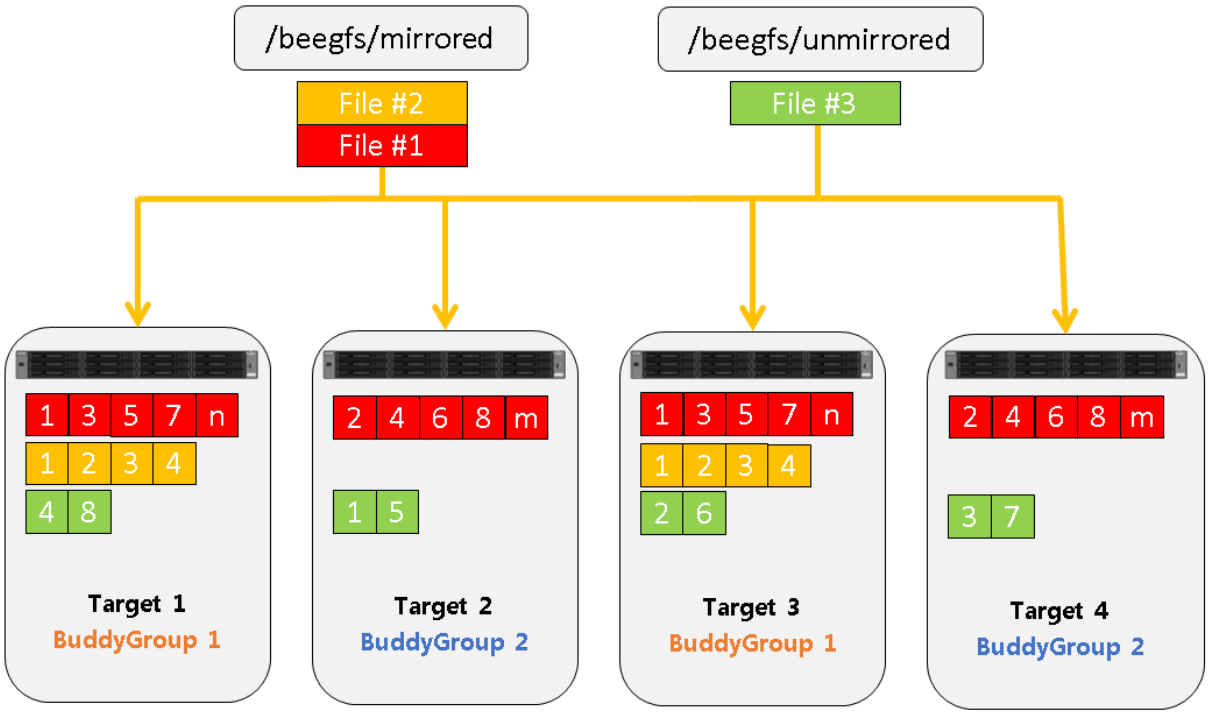

Figure 2.4-1: File Striping with BuddyMirrored and unmirrored files

2.5 Compression

Modern local file systems, such as Zettabyte File System (ZFS), support transparent compression and can be used as backend for BeeGFS. We successfully tested this feature and evaluated its interaction with BeeGFS. As a result, we can fully recommend using the underlying filesystem's compression if it is required.

2.6 Data placement

BeeGFS allows defining a set of preferred storage targets and metadata servers to store files. These policies can be set for every BeeGFS mount as part of the client module's configuration. The data placement policy can be defined for every BeeGFS mount point. It is possible to mount the same BeeGFS multiple times on the same node. Every mount could be configured with a different policy.

2.7 Monitoring and statistics of BeeGFS

During the project BeeGFS was extended with additional statistics to monitor the filesystem. The gathered data can be broken down to individual clients or users. This can be helpful to identify cluster jobs which are heavily using the filesystem or have bad I/O patterns. In addition, information gathered from the monitoring system can be used to identify performance-critical parts in the I/O subsystem of the applications.

2.7.1 Command line tool beegfs-ctl

The `beegfs-ctl` is a command line tool which provides statistics and monitoring information of BeeGFS as parsable output. The integration of the newly-implemented statistics in beegfs-ctl is covered below.

The storage statistics on a per-user basis can be queried with the `beegfs-ctl` command as follows:

```
beegfs-ctl --userstats --nodetype=storage --interval=1
```

```
====== 10 s ======
Sum:            11795 [sum]  7976 [getFSize]  798 [close]  1543 [ops-rd]  6.027 [MiB-rd/s]  1478 [ops-wr]  6.027 [MiB-wr/s]
nobody          8774 [sum]  7976 [getFSize]  798 [close]
root            3021 [sum]  1543 [ops-rd]  6.027 [MiB-rd/s]  1478 [ops-wr]  6.027 [MiB-wr/s]

====== 11 s ======
Sum:            4434 [sum]  2999 [getFSize]  301 [close]  578 [ops-rd]  2.258 [MiB-rd/s]  556 [ops-wr]  2.258 [MiB-wr/s]
nobody          3300 [sum]  2999 [getFSize]  301 [close]
root            1134 [sum]  578 [ops-rd]  2.258 [MiB-rd/s]  556 [ops-wr]  2.258 [MiB-wr/s]
```

Figure 2.7-1: Output of beegfs-ctl which contains storage statistics of users

The metadata statistics on a per-user basis can be queried with the `beegfs-ctl` command as follows:

```
beegfs-ctl --userstats --nodetype=metadata --interval=1
```

```
====== 11 s ======
Sum:            12 [sum]  1 [sAttr]  5 [stat]  6 [revalLI]
denis           8 [sum]  1 [sAttr]  3 [stat]  4 [revalLI]
root            4 [sum]  2 [stat]  2 [revalLI]

====== 12 s ======
Sum:            442 [sum]  48 [close]  48 [open]  100 [stat]  102 [revalLI]  144 [flckRg]
nobody          442 [sum]  48 [close]  48 [open]  100 [stat]  102 [revalLI]  144 [flckRg]
```

Figure 2.7-2: Output of beegfs-ctl which contains metadata statistics of users

The storage statistics on a per-client basis can be queried with the `beegfs-ctl` command as follows:

```
beegfs-ctl --clientstats --nodetype=storage --interval=1
```

```
====== 13 s ======
Sum:            13197 [sum]  8922 [getFSize]  893 [close]  1727 [ops-rd]  6.746 [MiB-rd/s]  1655 [ops-wr]  6.750 [MiB-wr/s]
storage10.ib.cluster  3608 [sum]  3280 [getFSize]  328 [close]
192.168.72.252  3382 [sum]  1727 [ops-rd]  6.746 [MiB-rd/s]  1655 [ops-wr]  6.750 [MiB-wr/s]
storage09.ib.cluster  2816 [sum]  2560 [getFSize]  256 [close]
storage02.ib.cluster  1948 [sum]  1770 [getFSize]  178 [close]
storage01.ib.cluster  1443 [sum]  1312 [getFSize]  131 [close]

====== 14 s ======
Sum:            13168 [sum]  8911 [getFSize]  890 [close]  1720 [ops-rd]  6.719 [MiB-rd/s]  1647 [ops-wr]  6.719 [MiB-wr/s]
192.168.72.252  3367 [sum]  1720 [ops-rd]  6.719 [MiB-rd/s]  1647 [ops-wr]  6.719 [MiB-wr/s]
storage01.ib.cluster  2902 [sum]  2638 [getFSize]  264 [close]
storage02.ib.cluster  2706 [sum]  2460 [getFSize]  246 [close]
storage09.ib.cluster  2255 [sum]  2050 [getFSize]  205 [close]
storage10.ib.cluster  1938 [sum]  1763 [getFSize]  175 [close]
```

Figure 2.7-3: Output of beegfs-ctl which contains storage statistics of beegfs-clients

The metadata statistics on a per-client basis can be queried with the `beegfs-ctl` command as follows:

```
beegfs-ctl --clientstats --nodetype=metadata --interval=1
```

```
====== 3 s ======
Sum:            8549 [sum]  944 [close]  944 [open]  1911 [stat]  1916 [revalLI]  2834 [flckRg]
192.168.72.252  8549 [sum]  944 [close]  944 [open]  1911 [stat]  1916 [revalLI]  2834 [flckRg]

====== 4 s ======
Sum:            8058 [sum]  889 [close]  889 [open]  1803 [stat]  1812 [revalLI]  2665 [flckRg]
192.168.72.252  8054 [sum]  889 [close]  889 [open]  1801 [stat]  1810 [revalLI]  2665 [flckRg]
node19.ib.cluster  4 [sum]  2 [stat]  2 [revalLI]
```

Figure 2.7-4: Output of beegfs-ctl which contains metadata statistics of beegfs-clients

In the figures Figure 2.7-1 to Figure 2.7-4 the output of the `beegfs-ctl` shows some statistics about the I/O operations. The first row (*Sum:*) of every interval shows a sum of all

operations which was done by all the user/client during the interval. The first column contains the UID/user name or IP-address/hostname of the I/O producer. The second column (*[sum]*) shows the sum of all operations which was done by a user/client. All following columns contain the number of operations of a special I/O operation. The monitoring system counts around 60 different metadata and storage operations. The following I/O operations happened in the examples:

- close: close file operation
- flockRg: byte range file lock (flock)
- getFSize: get local file size
- MiB-rd/s: read throughput in mebibytes per second
- MiB-wr/s: write throughput in mebibytes per second
- open: open file operation
- ops-rd: read operations
- ops-wr: write operations
- revalLI: revalidate intent operation, not directly triggered by the user
- sAttr: set file attribute
- stat: stat operation on a file

2.7.2 Webserver beegfs-admon

The beegfs-admon is a webserver that provides statistics and monitoring information. This data, and the information provided in the previous section, can be requested by using HTTP GET and POST requests. The beegfs-admon packages also contain a platform independent Java GUI to visualize the information. An example of the Java GUI with a metadata client statistics view and storage client statistics view are shown in Figure 2.7-5.

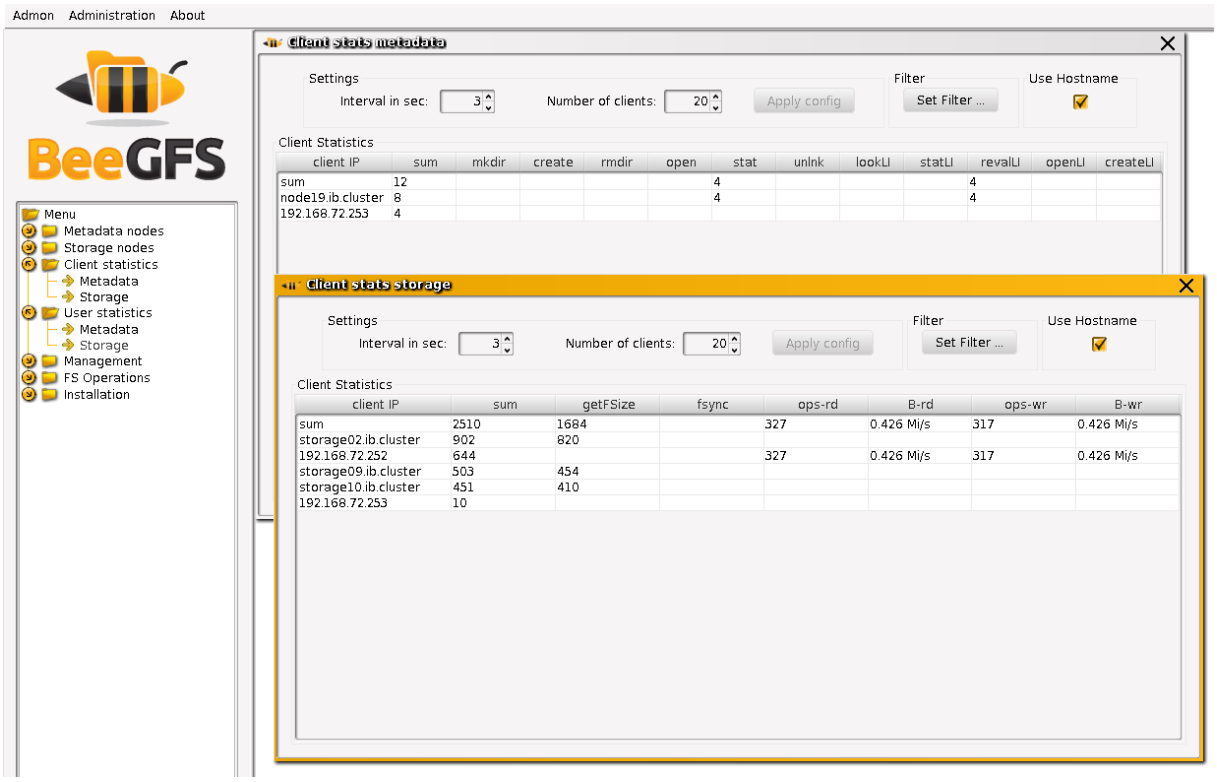

Figure 2.7-5: Client metadata and storage statistics in the admon GUI

2.8 Open Source

BeeGFS is released under an OpenSource license since the 23rd of February 2016. The source code is available on our website http://www.beegfs.com/content/source-code/.

2.9 Extoll Support

In the design of the I/O solution, the TCP over Extoll implementation constitutes the communication layer of BeeGFS. During the test on the DEEP-ER SDV we recognized a good I/O performance for 16 and fewer TCP streams. However, the throughput drops to less than 50 MByte/s in tests with more TCP streams. After several tests and benchmarks, this issue could be tracked down to limitations of TCP over Extoll. Although, the TCP over Extoll performance will be investigated and the implementation will be enhanced by Extoll, FHG-ITWM will implement support of the native Extoll protocol in BeeGFS to resolve the issues with TCP over Extoll and to achieve the maximum performance of Extoll. Originally, we expected that TCP over Extoll could provide nearly the full performance of the hardware, so this additional development work was not planned initially in the project and hence effects the distribution of the available resources. However, the high impact on the performance of the system justifies the change in priorities to maximize the projects benefits, since it is critical for achieving the project's main goals.

2.10 BeeGFS Summary and Conclusions

The BeeGFS approach in DEEP-ER addresses the problem of concurrent I/O on the storage system in common cluster environments. The cache filesystems help to partition the I/O and relieve the global storage system of the cluster. The applications can use the new filesystem architecture through the developed APIs. Exascale clusters will need a bigger storage system, what increases the probability that a part of the storage system fails. This topic is addressed by the implemented mirroring feature of BeeGFS. The new I/O statistics of the monitoring system help the application developers and the cluster administrators to identify performance issues.

3 SIONlib

3.1 Overview

One of the major goals of SIONlib is to provide support for parallel task-local I/O at large scale. In this section we will first cover the general approaches to optimise task-local I/O on large scale and describe the applied approaches of SIONlib to avoid different bottlenecks that occur when running traditional task-local I/O at large scale. The following part of this section will then cover the new developments in SIONlib done within the DEEP-ER project.

SIONlib is designed to speedup task-local I/O and to achieve this goal one of the key elements is the reduction of the number of files. In task-local I/O every task writes its data to its own file. For large task numbers this concept results in a correspondingly large file count. This in turn causes significant load on meta-data servers and introduces delays of several minutes in the program execution and hence limits scalability.

To avoid this bottleneck, SIONlib maps the task-local I/O of the application tasks to chunks of a shared file container, which reduces the number of files significantly. However, using fewer files than tasks results in shared files accessed concurrently by the application tasks. This can cause contention when file access is not coordinated. One point where serialisation in shared files access occurs is the concurrent access to single file system blocks from multiple tasks. A file system block is manipulated in “read, modify, write” cycles and can only be handled by a single task at a time. In order to prevent this serialisation SIONlib aligns access in the shared file to the file system block size, therefore ensuring exclusive access to a file system block by only one task.

In a similar way, as many files in a single directory cause meta-data contention, further scaling of task-local I/O to shared files leads to contention in the file meta-data management. Indeed, changes to file meta-data (e.g. i-node modifications) are typically performed by a single component of the file system. SIONlib addresses this bottleneck by using multiple files while keeping the number of files small compared to the number of tasks.

In the following we describe the developments of SIONlib that were done in the DEEP-ER project. The structure follows the DoW.

- *Integration of an additional parallel API driver layer for OmpSs. This will extend the list of currently supported parallel paradigms (MPI, OMP, hybrid).*
    - An analysis of this subtask by the involved groups from WP4 and WP5 showed that the current functionality of SIONlib already covers the described use case. One possible approach is the use of SIONlib’s key-value mode as described in section 3.2. This feature has been developed recently and was not available at the beginning of the project. In the desired use case, SIONlib operations will be performed by only one OmpSs task at a time, whereas the SIONlib file handle will be shared by all tasks, which perform I/O on the SIONlib file container. Accesses to the file will then be harmonized by coordinating the access to the shared file handle inside OmpSs.
- *Adaption of the existing implementation for MPI, OMP and hybrid to the MPI-implementations of DEEP-ER parallel environment.*
    - Large amounts of the code have been refactored in the course of the DEEP-ER project to support development and integration of new features (see section 3.3). In this way, the new buddy checkpointing feature (see section

3.4) could be implemented on top of existing SIONlib software layers. Beyond this additional functionality no further technical changes of the interfaces and implementation were needed to adapt SIONlib to the DEEP-ER parallel environment.

- *Enhancement of the SIONlib meta-data structure and the functionality of the underlying parallel software layer to support time-varying task-mappings.*
    - The aforementioned key-value mode of SIONlib (see section 3.2) is designed to handle time-varying task-mappings since it allows for a separation between the tasks directly involved in the I/O operations and those that are created only for computation. This feature can be used without losing the relation between those different tasks.
- *Enhancement of SIONlib to support buffering of task-local data in memory caches on compute- or I/O-elements. These would allow a deferred or asynchronous parallel I/O.*
    - The additional features of SIONlib developed by WP5 to support the SCR multi-level checkpointing library enable using asynchronous I/O. This is also the preferred way to exploit the asynchronous I/O feature of BeeGFS from SIONlib.
- *Adapt the internal testing and benchmarking features to cover the new features.*
    - The development of the new features for the project is test-driven. The SIONlib distribution package contains a set of parallel and serial tests for each of the new features, which allows verifying the correct execution of SIONlib during the installation process. This means testing is at the core of the development. The new features will be continuously benchmarked during and until the end of the project.

3.2 Key-value mode

The key-value mode in SIONlib allows correlating keys with payload data. Therefore SIONlib provides special write and read operations, which need a key as additional parameter. The data of write operations is then internally assigned to the according key. In this way data chunks written by one task can be separated into different sub-chunks, each with an individual key. This scheme can be used to separate data written by different threads of one process using the thread index as key.

Files that use the key-value mode can be exclusively accessed using this feature. The reason is that meta-data for the key-value is stored in the data sections of the file, which typically only contain user data for non-key-value opened SIONlib files. The benefit of this design decision is that SIONlib can handle key-value meta-data on a task-local level and does not have to manage this meta-data globally, with resulting global collective communication operations. Furthermore, this approach minimises the changes needed in meta-data handling and avoids changes in the file layout of the SIONlib file container.

In general, the implementation is not designed to replace a real key-value store or a database; instead it is specifically designed to handle the problem of time-varying task-mappings. Since the implementation is transparent for the user it can always be optimised toward different use cases without the need for significant changes in the user code. The preferred strategy when writing in key-value mode can be chosen in the open mode string, e.g.

```
sion_paropen_mpi("file.sion", "bw,keyval=inline", …);
```

Without explicit choice of the key-value mode the default is selected, which currently is the `inline` mode.

There are new API calls for writing and reading from/to key-value mode files, since the key is needed as additional argument. The two basic commands to make use of keys are:

```
size_t sion_fwrite_key(  const void *data,
                         uint64_t key,
                         size_t size,
                         size_t nitems,
                         int sid);
```

and

```
size_t sion_fread_key( void *data,
                       uint64_t key,
                       size_t size,
                       size_t nitems,
                       int sid);
```

These write to and read from a file with key-value mode, respectively.

The data layout for the current key-value implementation (called “inline”) adds a header to each write request containing the key and the size. So a single write call writes this meta-data and the actual data. Since the user is in part detached from the underlying file layout, the behaviour of write and read calls are changed: the size of every I/O call in non-key-value mode has to adhere the maximum of `chunksize` bytes. This limitation is dropped when key-value mode is used, so the size can be chosen freely. Although this might sacrifice write performance when used unintentionally, removing this limitation proved to be very convenient and may be considered for regular SIONlib write calls as well in future versions.

For more in-depth information on the key-value API we refer to the documentation on the SIONlib website [SIONlib website].

3.3 Redesign and restructuring

In order to simplify the development of new features for the project, SIONlib’s internal structure was redesigned. This significantly reduced the amount of work needed to develop and test the buddy checkpointing feature.

The original and revised application layouts are shown in Figure 3.3-1 and Figure 3.3-2, respectively.

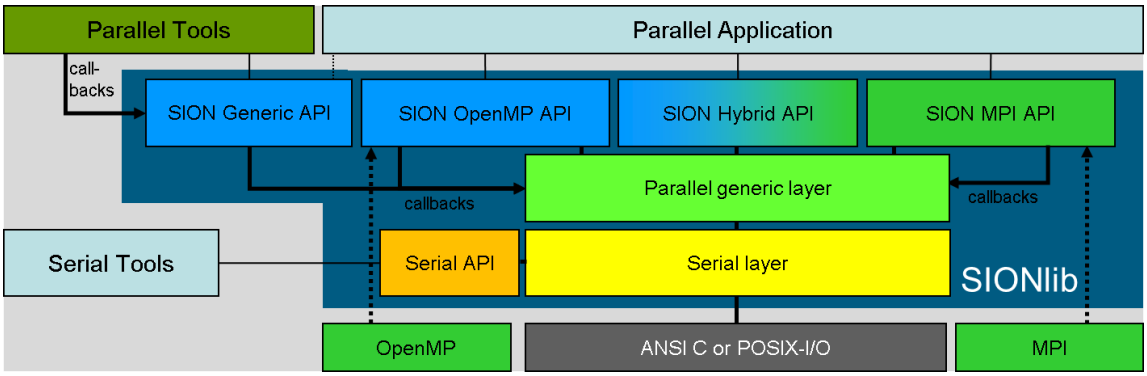

Figure 3.3-1: Original software layout of SIONlib

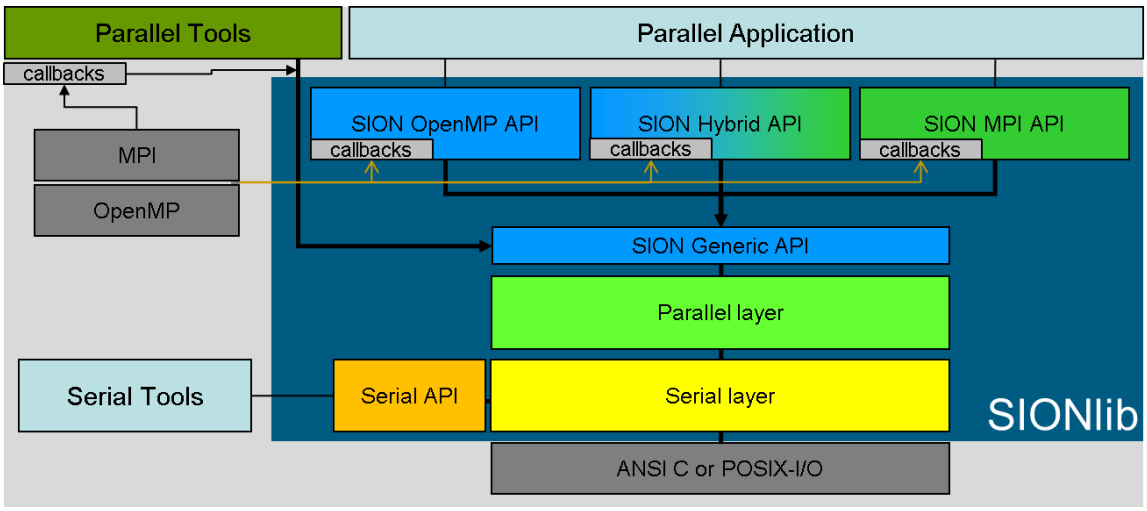

Figure 3.3-2: Revised software layout of SIONlib

In Figure 3.3-2 the ANSI C or POSIX-I/O layer provides the interface between SIONlib and the file system. The serial layer uses this interface to provide all the functionalities and algorithms required by the node to efficiently utilise the underlying file system API. For pre- or post-processing purposes this serial layer can be access directly from serial tools using the serial API. The parallel layer coordinates file access implementing only the parallel components of the algorithms and then calling the underlying serial layer.

Up to this layer the structure remained unchanged. The first API on top of the parallel layer was originally the MPI API. Over time the OpenMP and the hybrid API were added as a response to application demands and, in the case of tools (like Score-P or Scalasca), a generic API was added for more flexibility. This requirement comes from the fact that during start up such tools usually do not have sufficient knowledge about the communication layer that is used by the application and they need generic calls. This was the status before the project.

With the DEEP-ER developments all parallel APIs (MPI, OpenMP and hybrid) now use the generic API as opposed to the former layout. Since SIONlib was originally primarily designed for MPI there was much redundant functionality implemented for the other APIs as they were added. With the new structure the different parallel APIs only implement a small set of functions specific for this interface. These functions fulfil generic patterns, e.g. gathering data from all tasks and executing a command like writing the data to disk. The generic API uses these generic functions as building blocks to implement the generic parallel I/O algorithms.

This separation of communication and functionality has two major advantages. First of all it provides a single module to implement new functionalities, like buddy checkpointing which is described in the following section. A similar advantage also holds for the opposite direction: new communication layers only need to implement the limited set of communication primitives. There is no need for additional algorithms to be implemented for a new communication layer since the algorithms are based on the communication primitives.

3.4 Buddy checkpointing

3.4.1 Basic concepts

The concept of buddy checkpointing is driven by resiliency. While enabling highly scalable I/O, local storage also causes data loss in case a node fails. Since the aggregated performance of local storage is assumed to be significantly better than that of the global file system, parts of the work can be traded to add redundancy to the local layer while preserving high efficiency.

One obvious way to implement redundancy is to copy the local data to the local storage of a remote node. In this context the associated node is referred to as ‘buddy node’.

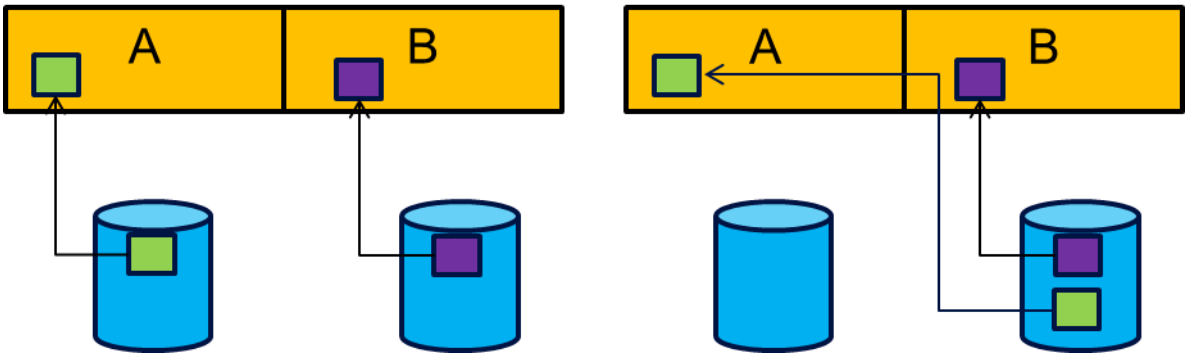

Figure 3.4-1: Normal restart from local data (left) and restart from local data using buddy checkpointing after node failure (right).

Figure 3.4-1 shows the data flow for different restart conditions. For the case where local checkpointing data is only used to speed up the restart process each cache domain provides the data for the local processes. Since the local storage is connected to the according nodes a node failure also causes the local storage to be inaccessible. To solve this problem we request the copied data from the buddy node. This buddy node will then provide the data after it reads its own local data. In order to transport the data SIONlib uses the communication layer of the application, e.g. MPI.

3.4.2 Design

With the new internal structure, buddy checkpointing could be integrated in the parallel layer without the need for different implementations for different APIs. Figure 3.4-2 shows the different components where the buddy checkpointing feature changes former behaviour.

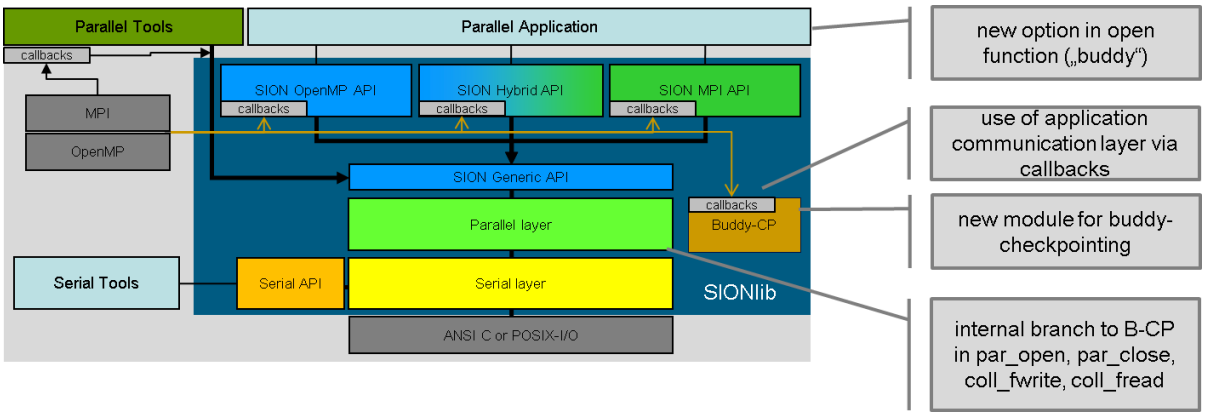

Figure 3.4-2: Buddy checkpointing integration

Internally the code now branches and runs different methods depending on whether buddy checkpointing is enabled or not. The relevant code parts are implemented in a new module, which makes use of the communication layer of the application. It is a general design decision of SIONlib to use the communication layer of the application in order to avoid additional dependencies.

In order to enable buddy checkpointing applications need to pass a new option “buddy=N” to the open option string. This option defines the level of buddy checkpointing and in case it is omitted it is set to the default value of 1 (meaning that one additional copy of the data is kept). The level equals the number of redundant copies of the data written, so a level of 2 means the data exists in the form it usually would for writing with SIONlib and two copies are present on disjoint local storages. Since there is no benefit in writing buddy data to the same node twice the number of cache domains represents a natural limit. Although there is no limitation in SIONlib, the performance penalty will usually determine a reasonable limit for the buddy level, which is significantly smaller than the number of cache domains. The failure of any cache domain during the application execution can considered significantly higher than the simultaneous failure of two cache domains. Hence, for most use cases the default level of 1 will be an good choice.

3.4.3 Communication scheme

The implementation of buddy checkpointing with the generic primitives eases the support for different communication layers and the maintenance of the code and tests. As a result the algorithms are also not formulated in the scope of e.g. MPI but with the generic primitives, which can be implemented for MPI as well as for OpenMP or hybrid applications. This leads to communication schemes, which incorporate the features of the potentially different underlying parallel APIs.

In general SIONlib supports different I/O strategies that were added over time, as there was demand from applications. The first basic scenario, which SIONlib was optimised for, is checkpointing a large amount of data of many tasks. For this scenario SIONlib only uses collective calls for open and close routines while all read and write routines are individual without any communication. Avoiding unnecessary communication is one of the optimisation strategies of SIONlib. The relevant bottlenecks change for a scenario where a large number of tasks are involved in I/O but only a small amount of data is written. In this case file system block alignment results in potentially large amounts of allocated but unused disk space. Therefore, SIONlib also supports a collective mode which needs to communicate during read

or write operations but can result in significantly denser files and better bandwidths, since less empty data needs to be read or written.

Since buddy checkpointing naturally involves communication one important feature is that it can only be used in the collective mode. In order to collect the data correctly different groups are formed to define different roles during buddy checkpointing.

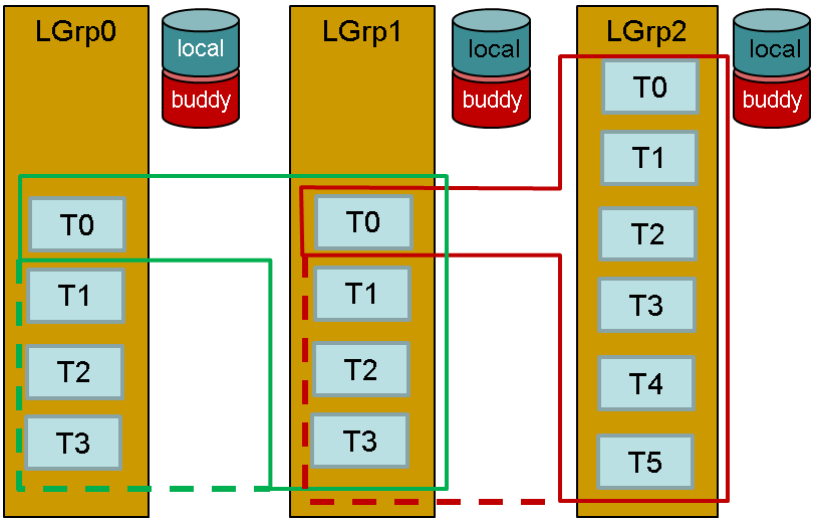

Figure 3.4-3: Communication groups

Figure 3.4-3 shows one example of possible communication groups for three cache domains with a different number of tasks. Inside these groups the tasks fulfil different roles. In the middle cache domain (LGrp1), task 0 (T0) acts as collector in the red group while taking the role of a sender in the green group. It is always the first task in a local group that collects the data of the other group members, both for local writes and for remote writes from the buddy group. For implementation reasons the remaining tasks in a local group are not excluded from the communication group, but are assigned an according role. This prevents them from sending their data not only to the first task of the buddy node but also to the first task in their own node. This is indicated with the dashed lines for both groups in the figure.

With these groups well defined SIONlib transfers the data between them. The algorithm for data transfer takes into account that a task may have different roles. In this way the data transfer becomes a multi-step process depending on different factors, like whether the number of cache domains is even or odd, and which level of buddy checkpointing is executed.

The implementation does not use the buddy nodes explicitly but a mapping that describes the buddy relation. This allows different levels of buddy checkpointing to reuse the algorithm that handles different roles.

3.4.4 Comparison with native buddy checkpointing in SCR

As part of the Co-design between WP5 and WP4, the buddy checkpointing functionality in SIONlib was identified as a good way to make use of the strengths of both, the scalable checkpoint / restart system (SCR) and SIONlib. Where SCR shines with its resiliency features, its task-local I/O strategy is the ideal candidate to be optimised with SIONlib.

The most obvious differences in I/O behaviour between pure SCR and SIONlib are how the data is read and the number of operations involved in the checkpointing/restart process. Since SCR has no control over the data during the write and read calls, it has to first copy all missing data to local storage in the buddy nodes. In Figure 3.4-4 this behaviour is depicted for the read and write case in the left column. The operations are labelled “W” and “R” for write and read respectively, and the same numbers mark processes that can run in parallel. The operations coloured in red mark additional operations that SCR does and that SIONlib does not have to do.

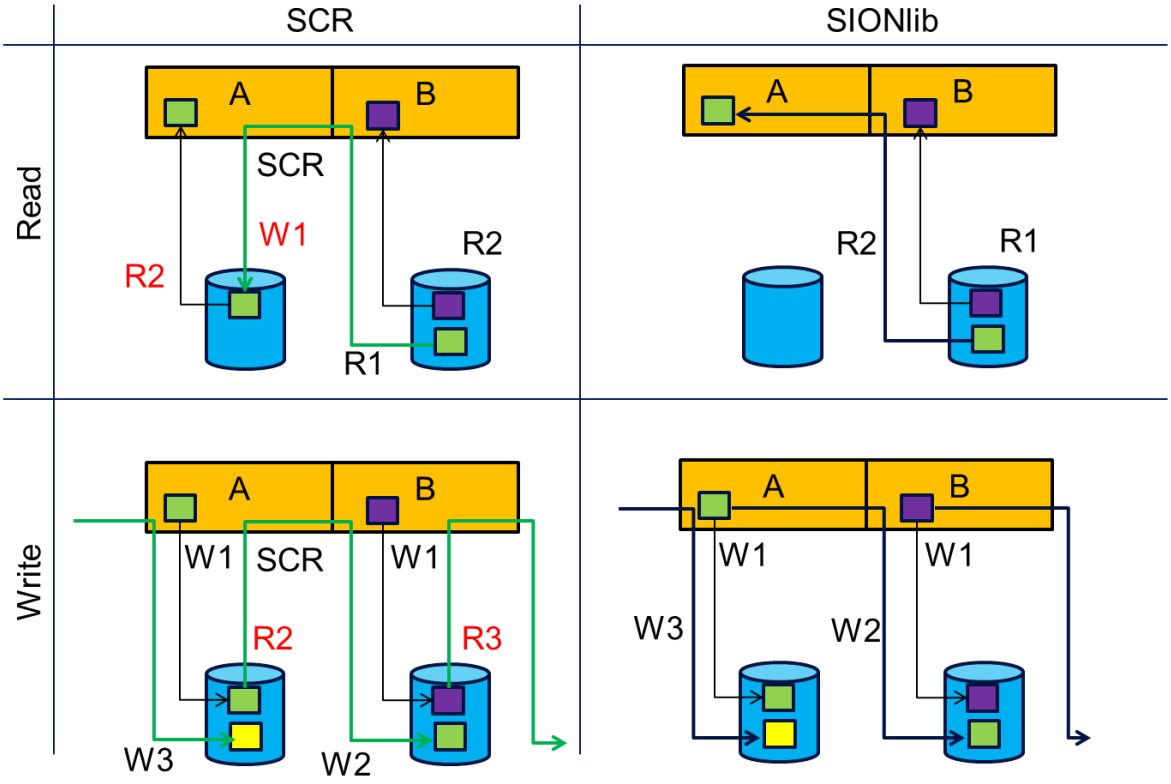

Figure 3.4-4: Comparison of read and write behaviour of SCR (left) and SIONlib (right)

First of all we describe the read behaviour. Since node A does not restart with its local data it needs to get the data from its buddy node B. In SCR this means copying the data to A from B and then reading it from the local storage. This operation needs to be done first; the green arrow labelled “SCR” shows the copying process from B to A and the operations are labelled “1”. With all local data recovered, both nodes can restart from their local data.

In the case of SIONlib restarting with no local data being available on node A means that the data is read on demand. First node B reads from the local storage and then sends the requested buddy data to node A. Since the local read operations are serialised and all data needs to be read from node B, the number of operations is the same for this node. Regarding node A, the difference between the two approaches is that SIONlib directly transfers the data from node B into node A’s memory without the need for additional reads or writes to the local storage of this node.

As indicated by the numbers, the two additional operations could be theoretically executed in parallel with the two read operations of node B.

An analysis of writing buddy checkpoints shows similar result. Since SCR does not handle the write calls, it needs to re-read the data written locally to be able to copy it to another

node. This adds a read operation to the scheme, which can be avoided by SIONlib. Regarding the parallel operations the two schemes also show a similar behaviour as in the read case. The reason why the operations “W2” and “W3” have different numbers is the communication pattern, e.g. odd and even nodes changing roles to avoid deadlocks.

Besides the reduction of I/O operations SIONlib can also reduce the number of local files per cache domain and hence also for the checkpoints that are migrated to the global storage. One single SIONlib-file per cache domain is expected to be a good choice for the envisioned hardware configuration.

3.4.5 Example

In Figure 3.4-5 we show a basic example how to write files with SIONlib using buddy checkpointing.

```
#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        lgroup    = -1;
  int        rank      = -1;

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

  if (rank < 2) {
    lgroup = 0;
  }
  else {
    lgroup = 1;
  }

  MPI_Comm_split(MPI_COMM_WORLD, lgroup, rank, &lComm);
  buffer = (char*)malloc(chunksize);

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

  sid = sion_paropen_mpi("simple.sion",
                         "w,buddy",
```

```
                              &numFiles,
                              MPI_COMM_WORLD,
                              &lComm,
                              &chunksize,
                              &fsblksize,
                              &rank,
                              NULL,
                              &newfname);

  if (sid >= 0) {
    sion_coll_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;
}
```

Figure 3.4-5: Basic usage of buddy checkpointing in SIONlib

In this example we create two local groups. The first two ranks belong to the first group and the remaining ranks to the second. If BeeGFS is used to manage the local storage, these groups would be chosen according to the BeeGFS cache domains.

The example given in D4.2 shows the regular writing of SIONlib files. Beyond the creation of local groups, the important changes to add buddy checkpointing are:

- `numFiles` is set to `-1` which tells SIONlib to use the communicator `lComm` to create I/O groups. These groups should be chosen according to the local storage.
- `sion_coll_fwrite` is used to write the data since buddy checkpointing involves communication with the buddy node in the write process, as described above.

3.5 Open Source

SIONlib is release under an Open Source (BSD 3-Clause) license. The source code and documentation are available on the SIONlib website [SIONlib website].

3.6 SIONlib Summary and Conclusions

In DEEP-ER, SIONlib provides buddy checkpointing to ease the use of local storage for resiliency. As checkpointing is a major use case for SIONlib, the extension to buddy checkpointing lets applications take advantage of local storage easily while still using a mature and long tested I/O library. This provides a benefit for applications already using

SIONlib, since there are no major changes needed to enable buddy checkpoining. On the other side, applications which integrate SIONlib mostly for the provided buddy checkpointing functionality benefit from the I/O optimisations which are already integrated into SIONlib.

4 E10

The Exascale10 description in D4.4 will address the following aspects:

1. Exascale10 contributions to the DEEP-ER project
2. Description of the proposed Exascale10 hints extensions for MPI-IO
3. Description of the Exascale10 architecture and implementation
4. Integration of Exascale10 within DEEP-ER applications
5. Exascale10 source code licensing and release
6. Summary and conclusions

4.1 Exascale10 Contributions

Exascale10 contributes to the DEEP-ER project by providing improvements to existing collective I/O implementations. The ROMIO middleware (a popular implementation of the MPI-IO specifications from Argonne National Laboratory) is used as substrate into which the new DEEP-ER hardware enabled functionalities are included, maintaining a familiar, widely adopted, I/O interface, minimising the integration effort of the new features into existing and future applications.

An important advantage is the use of the NVMe devices integrated in the nodes of the DEEP-ER Prototype. This fast, persistent, cache layer amplifies collective I/O performance, and more generally, any I/O operation. The new memory tier in the DEEP-ER Prototype is made available to applications through the MPI-IO interface by means of additional hints, described in detail in the rest of this document. The new hints rely on the underlying Exascale10 code inside ROMIO to efficiently move data to and from the cache layer.

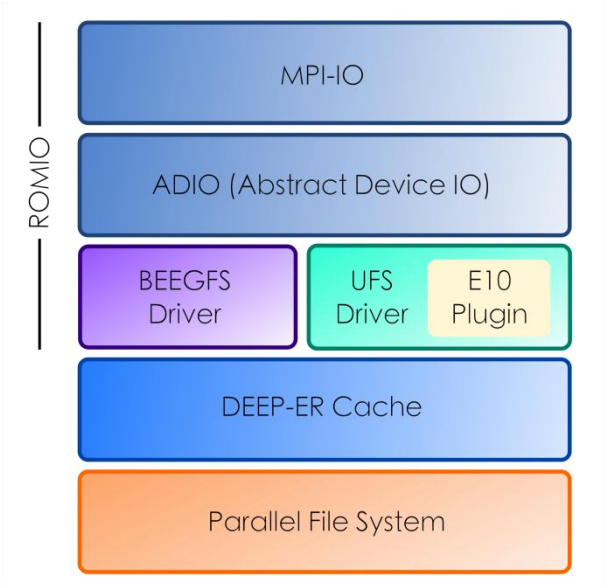

Figure 4.1-1: Exascale10 software stack

The Exascale10 effort in DEEP-ER is twofold. Firstly, an extension of the Universal File System driver in ROMIO (UFS) was developed, providing cache access to any parallel file system (e.g. Lustre, GPFS, etc). Secondly, a new BeeGFS driver was developed, taking advantage of the native BeeGFS cache APIs to provide the same functionalities of the

universal driver. A high level architecture of the software stack just described is shown in Figure 4.1-1.

4.2 Exascale10 Hints Extensions for MPI-IO

The Exascale10 hints extension for MPI-IO represents the only way users can access the DEEP-ER cache layer through MPI-IO. There is currently no additional API, although for the future it is planned to move the existing functionalities into a separate Exascale10 middleware, not relying on any other implementation.

Follows a list of hints and corresponding description:

- **`e10_cache`**: used to **`enable`** (or **`disable`**) access to the cache. If set to **`enable`** every collective I/O write operation will be directed to the cache. Additionally, the hint can be also set to **`coherent`** to provide cache coherency to parallel applications accessing the same file. In the coherent mode of operation, every write will acquire a lock on the requested file extent. This lock will be released only when data in the cache has been made persistent in the global file system. Default value is **`disable`**.
- **`e10_cache_path`**: used only in the UFS driver to tell the implementation where in the local file system the cache file should reside. BeeGFS does not need this hint since the cache layer is completely transparent to the user.
- **`e10_cache_flush_flag`**: used to tell the implementation when the data in the cache should be moved to the global file system. If set to **`flush_immediate`** will force the implementation to flush the data in the cache immediately after it has been written. If set to **`flush_onclose`** will tell the implementation to start the flush of the data in the cache when the file is closed. If set to **`flush_none`** will tell the implementation to skip the flush of the data in the cache. Default value is **`flush_immediate`**.
- **`e10_cache_discard_flag`**: used to tell the implementation whether a file should be removed from the cache (**`enable`**) or not (**`disable`**) when it is closed. Default value is **`enable`**.
- **`e10_cache_thread`**: used to tell the implementation how many synchronisation threads should be used to flush the data from the cache to the global file system. Default value is **`1`**.

The described hints take advantage of the UFS and BeeGFS driver implementations described in the following sections.

4.3 Exascale10 Architecture

The cache synchronisation task in the Exascale10 implementation of the UFS driver is delegated to a separate thread pool, created when the file is opened and destroyed when the file is closed. Communication between the main thread of the program and the synchronisation threads in the pool is provided through dedicated set of APIs and queues.

Synchronisation threads and queues are modelled using an object-oriented approach[1] and described by the following data structures and APIs in the UFS driver:

- **ADIOI_Sync_req_t:** data structure describing a synchronisation request used by the synchronisation thread to read data back from the cache and copy it to the global file system. The corresponding APIs provided to the threads are:
    - **ADIOI_Sync_req_init:** used by the main thread to create a new synchronisation request;
    - **ADIOI_Sync_req_fini:** used by the main thread to destroy a synchronisation request that has completed;
    - **ADIOI_Sync_req_init_from:** used by the main thread to create a new synchronisation request starting from an existing one;
    - **ADIOI_Sync_req_get_type:** used by the synchronisation thread to get the type of request, either `ADIOI_THREAD_SYNC` or `ADIOI_THREAD_SHUTDOWN`;
    - **ADIOI_Sync_req_{get,set}_key:** used by the synchronisation thread and the main thread to get/set the value for a specific key in the request. Available keys are:
        - **ADIOI_SYNC_TYPE**: the request type (as explained before);
        - **ADIOI_SYNC_OFFSET**: the offset of the synchronisation request;
        - **ADIOI_SYNC_DATATYPE**: the datatype of the synchronisation request;
        - **ADIOI_SYNC_COUNT**: the count for datatypes in the synchronisation request;
        - **ADIOI_SYNC_REQ**: the `MPI_Request` handle for the request;
        - **ADIOI_SYNC_ERR_CODE**: the return `error_code` for the request;
        - **ADIOI_SYNC_FFLAGS**: the cache flush flags for BeeGFS;
        - **ADIOI_SYNC_ALL**: all the above.
- **ADIOI_Atomic_queue_t**: data structure describing the queues used by the synchronisation thread and the main thread to communicate. The corresponding APIs provided to interact with the queue are:
    - **ADIOI_Atomic_queue_init:** used by main thread to create a queue;
    - **ADIOI_Atomic_queue_fini:** used by main thread to destroy a queue;
    - **ADIOI_Atomic_queue_push:** used by main thread to push a **ADIOI_Sync_req_t** to the queue;
    - **ADIOI_Atomic_queue_pop:** used by synchronisation thread to pop a **ADIOI_Sync_req_t** from the queue;
    - **ADIOI_Atomic_queue_front:** used by synchronisation thread to get the front **ADIOI_Sync_req_t** from the queue;
    - **ADIOI_Atomic_queue_back:** used by the synchronisation thread to get the back **ADIOI_Sync_req_t** from the queue;
    - **ADIOI_Atomic_queue_size:** used to check the number of **ADIOI_Sync_req_t** inside the queue;

[1] The use of an object-oriented approach makes future extractions of the functionalities contained inside ROMIO easier and allows for a better recycling of the existing code.

  - **ADIOI_Atomic_queue_empty:** used by the synchronisation thread to check the status of the queue.
- **ADIOI_Sync_thread_t**: data structure describing the synchronisation thread inside the thread pool. It contains three queues: 1) a pending queue (`pen_`), 2) a submitted queue (`sub_`) and 3) a waiting queue (`wait_`). The APIs provided to interact with the thread are:
  - **ADIOI_Sync_thread_init**: used by the main thread to create a new synchronisation thread. The corresponding routine starts a new POSIX thread with a pointer to `ADIOI_Sync_thread_start`, which internally pops `ADIOI_Sync_req_t`(s) from the `sub_` queue and satisfies them;
  - **ADIOI_Sync_thread_fini**: used by the main thread to destroy a synchronisation thread. The corresponding routine will create a `ADIOI_Sync_req_t` of type `ADIOI_THREAD_SHUTDOWN` and push it to the `sub_` queue;
  - **ADIOI_Sync_thread_enqueue**: used by the main thread to send `ADIOI_Sync_req_t`(s) to the thread. The corresponding routine will place a synchronisation request inside the `pen_` queue for later processing;
  - **ADIOI_Sync_thread_flush**: used by the main thread to signal the thread that all the previously sent `ADIOI_Sync_req_t`(s) should be satisfied. The corresponding routine moves all the synchronisation requests from the `pen_` queue to the `sub_` queue. A copy of the `ADIOI_Sync_req_t`(s) is also pushed to a `wait_` queue for later completion check from the main thread;
  - **ADIOI_Sync_thread_wait**: used by the main thread to wait for all the `ADIOI_Sync_req_t`(s) sent to a particular thread to complete. The corresponding routine invokes `MPI_Wait()` on all the `MPI_Request`(s) contained in the `ADIOI_Sync_req_t`(s) waiting in the `wait_` queue.

For BeeGFS there is no need for a synchronisation thread pool since this is already provided by the BeeGFS deamon. Thus the synchronisation thread APIs are modified to exploit the BeeGFS cache APIs and renamed using the suffix `ADIOI_BEEGFS_Sync_thread_*` in the BeeGFS driver.

4.3.1 ADIOI_Sync_req_t

The `ADIOI_Sync_req_t` data structure contains all the information required by the synchronisation thread to move the data in the file between the cache and the global file system. The data structure and the APIs used to interact with it are following reported:

```
struct ADIOI_Sync_req {
    // type of sync thread: ADIOI_THREAD_{SYNC,SHUTDOWN}
    int type_;
    // file extent offset in the file
    ADIO_Offset off_;
    // datatype used to write data to the file
    MPI_Datatype datatype_;
    // number of datatypes written to the file
    int count_;
    // MPI_Request used with MPI_Wait to check completion status
```

```
    ADIO_Request *req_;
    // error code returned by synchronisation routine
    int error_code_;
    // flush flags (BeeGFS only)
    int fflags_;
};

typedef struct ADIOI_Sync_req *ADIOI_Sync_req_t;
int ADIOI_Sync_req_init(ADIOI_Sync_req_t *r, ...);
int ADIOI_Sync_req_init_from(ADIOI_Sync_req_t *r,
                             ADIOI_Sync_req_t s);
int ADIOI_Sync_req_get_type(ADIOI_Sync_req_t r);
int ADIOI_Sync_req_get_key(ADIOI_Sync_req_t r, ...);
int ADIOI_Sync_req_set_key(ADIOI_Sync_req_t r, ...);
int ADIOI_Sync_req_fini(ADIOI_Sync_req_t *r);
```

Figure 4.3-1: Synchronisation request object and related APIs

4.3.2 ADIOI_Sync_thread_t

The `ADIOI_Sync_thread_t` data structure describes the thread(s) inside the synchronisation pool. The data structure and the APIs used to interact with it are following reported:

```
struct ADIOI_Sync_thread {
    // MPI File handle
    ADIO_File fd_;
    // pthread id
    pthread_t tid_;
    // pending queue
    ADIOI_Atomic_queue_t pen_;
    // submitted queue
    ADIOI_Atomic_queue_t sub_;
    // waiting queue
    ADIOI_Atomic_queue_t wait_;
};

typedef struct ADIOI_Sync_thread *ADIOI_Sync_thread_t;
int ADIOI_Sync_thread_init(ADIOI_Sync_thread_t *t, ...);
int ADIOI_Sync_thread_fini(ADIOI_Sync_thread_t *t);
void ADIOI_Sync_thread_enqueue(ADIOI_Sync_thread_t t,
                               ADIOI_Sync_req_t r);
void ADIOI_Sync_thread_flush(ADIOI_Sync_thread_t t);
void ADIOI_Sync_thread_wait(ADIOI_Sync_thread_t t);
```

Figure 4.3-2: Synchronisation thread and related APIs

4.3.3 Exascale10 Integration in ROMIO

Inside the ROMIO UFS driver there is a set of functions that were modified to integrate the new cache functionalities:

- **ADIOI_GEN_OpenColl**: used by `MPI_File_open()` to open a file collectively;
- **ADIOI_GEN_WriteStridedColl**: used by `MPI_File_write_{all,at_all}()` to write to a shared file collectively;
- **ADIOI_GEN_WriteContig**: used by `ADIOI_GEN_WriteStridedColl()` and any other `MPI_File_write()` operation to actually write data to the file. In the case of collective I/O this is invoked for every round of two phase I/O to move the data shuffled among processes to the file;
- **ADIOI_GEN_Flush**: used by `MPI_File_sync()` to flush all the previous writes to the file;
- **ADIO_Close**: used by `MPI_File_close()` to close an open file.

4.3.4 ADIOI_GEN_OpenColl

Normally this function returns a file handle to every process in the communicator. In the Exascale10 implementation this was extended to open an additional file in the cache (using the provided user path in `e10_cache_path`) and create the thread pool to start later synchronisation. The cache file is handled as additional `MPI_File` handle pointer (called `cache_fd`) inside the global file handle.

4.3.5 ADIOI_GEN_WriteStridedColl

In order to make sure that writes to the cache will not cause any error due to lack of space, local file space is allocated upfront for every collective write operation using a new function called `ADIOI_Cache_alloc()`[2]. The function will return an error code if the allocation has failed for any reason. The error code from every aggregator is then broadcasted to make sure that all the processes have succeeded allocating space. The allocation is done in `ADIOI_GEN_WriteStridedColl()` since this allows to allocate space only once for every collective write operation. If allocation was done in `ADIOI_GEN_WriteContig()` it would have required to allocate space for every round of two phase I/O, increasing the number of system calls introduced by the implementation and the number of global synchronisation points.

4.3.6 ADIOI_GEN_WriteContig

`ADIOI_GEN_WriteContig()` writes data to the file in the cache, creates a corresponding synchronisation request and sends it to a synchronisation thread in the pool. Following a code example of how this is done:

```
void ADIOI_GEN_WriteContig(ADIO_File fd, const void *buf, int count,
                           MPI_Datatype datatype, int file_ptr_type,
                           ADIO_Offset offset, ADIO_Status *status,
                           int *error_code)
{
    int err;
    char *p = (char *)buf;
```

[2] `ADIOI_Cache_alloc()` uses the `fallocate()` system call to allocate space. This system call does not write any data to the file system but only modifies the file system metadata to reserve blocks. Nevertheless, this system call only works with EXT4 and XFS file systems.

```
    ADIO_File fh = fd;
    if (fd->cache_fd && fd->cache_fd->is_open) {
        fh = fd->cache_fd;
        if (fd->hints->e10_cache_coherent == ADIOI_HINT_ENABLE)
            ADIOI_WRITE_LOCK(fd, offset,SEEK_SET, len);
    }
    ...
    while (bytes_xfered < len) {
        wr_count = len - bytes_xfered;
        err = write(fh->fd_sys, p, wr_count);
        bytes_xfered += err;
        p += err;
    }

    if (fd->cache_fd && fd->cache_fd->is_open &&
          fd->hints->e10_cache_flush_flag != ADIOI_HINT_FLUSHNONE) {
        ADIOI_Sync_req_t sub;
        int threads, curr_thread, idx;
        ADIO_Request *r = (ADIO_Request *)
                           ADIOI_Malloc(sizeof(ADIO_Request));
        *r = MPI_REQUEST_NULL;
        threads = fd->hints->e10_cache_threads;
        curr_thread = fd->thread_curr;
        idx = curr_thread % threads;

        // init sync req
        ADIOI_Sync_req_init(&sub, ADIOI_THREAD_SYNC,
                            offset, datatype, count,
                            req, 0);

        // enqueue sync request to thread
        ADIOI_Sync_thread_enqueue(fd->thread_pool[idx], sub);

        if (fd->hints->e10_cache_flush_flag ==
              ADIOI_HINT_FLUSHIMMEDIATE)
            ADIOI_Sync_thread_flush(fd->thread_pool[idx]);

        // select next thread in the pool
        fd->thread_curr = (curr_thread + 1) % threads;
    }
}
```

Figure 4.3-3: For every write a new request is created and submitted to the synchronisation pool

4.3.7 ADIOI_GEN_Flush

`ADIOI_GEN_Flush()` forces all the data written, and potentially still in the page cache, to be flushed to the file system. This function was modified to copy all the additional data in the NVM cache to the global file system. Following a code example of how this is done:

```
void ADIOI_GEN_Flush(ADIO_File fd, int *error_code)
{
    int err, idx, threads, curr_thread;
    static char myname[] = "ADIOI_GEN_FLUSH";

    if (fd->cache_fd == NULL ||
         (fd->cache_fd && !fd->cache_fd->is_open) ||
         (fd->cache_fd && fd->cache_fd->is_open &&
          fd->hints->e10_cache_flush_flag == ADIOI_HINT_FLUSHNONE))
        goto fn_flush;

    threads = fd->hints->e10_cache_threads;

    // Flush all the requests in each thread
    for (idx = 0; idx < threads; idx++)
        ADIOI_Sync_thread_flush(fd->thread_pool[idx]);

    // Wait for submitted requests to complete
    for (idx = 0; idx < threads; idx++)
        ADIOI_Sync_thread_wait(fd->thread_pool[idx]);

fn_flush:
    err = fsync(fd->fd_sys);
    ...
}
```

Figure 4.3-4: When the file is flushed all the pending requests are forced to the global file system and checked for completion

If the synchronisation requests were already flushed by the write function, `ADIOI_Sync_thread_flush()` will do nothing and just return immediately since the `pen_` queue is already empty.

4.3.8 ADIO_Close

`ADIO_Close()` function closes the `MPI_File` handle by invoking the corresponding `ADIOI_xxx_Close()` routine (where `xxx` is replace by the name of the file system driver). In this case close function was modified to invoke `ADIO_GEN_Flush()`, triggering the flushing of the data in the cache to the global file system, and call `ADIOI_GEN_Close()` on the `MPI_File` handle of the file in the cache. Following an example of how this is done:

```
void ADIO_Close(ADIO_File fd, int *error_code)
{
    if (fd->cache_fd) {
        if (fd->cache_fd->is_open) {
            (*(fd->fns->ADIOI_xxx_Flush))(fd, error_code);
            (*(fd->fns->ADIOI_xxx_Close))(fd->cache_fd, error_code);
             ADIOI_Sync_thread_pool_fini(fd);
        }
    }
```

```
    ...
}
```

Figure 4.3-5: When the file is closed the file is synchronised with the cache and the cache file is closed

As already said BeeGFS does not need to create any additional thread pool to synchronise the data in the cache to the global file system. Therefore, out of the previously described functions the following have been re-implemented in the BeeGFS driver:

- **ADIOI_BEEGFS_OpenColl**
- **ADIOI_BEEGFS_WriteContig**
- **ADIOI_BEEGFS_Flush**

All these functions will use the appropriate BeeGFS APIs for cache handling.

4.4 Exascale10 Integration

In the Exascale10 implementation collective write operations can write data to the cache instead of the global file system, taking advantage of fast NVMe devices installed in the compute nodes of the DEEP-ER Prototype and minimising the impact of the global file system performance on the total runtime. Potentially, since the number of NVMe(s) can grow with the number of compute nodes, our implementation can scale the write bandwidth linearly with the number of available memory devices. Write operations to the global file system can be overlapped with computation if the `e10_cache_flush_flag` is set to `flush_immediate`.

The limitation with this approach is that the file cannot be closed until the cache synchronisation is completed[3]. For this reason some changes might be required at the application level in order to take advantage of the new MPI-IO hints.

[3] If the `e10_cache` hint is set to `coherent` the file will not be over writable by other processes during synchronisation.

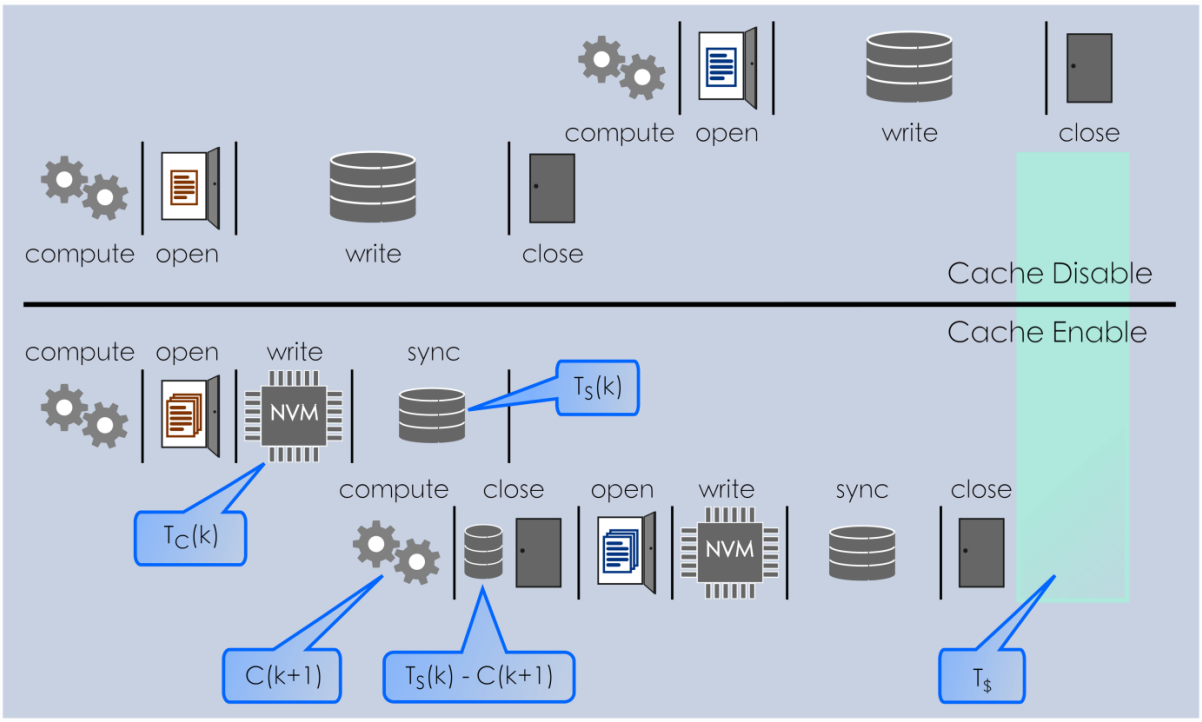

Figure 4.4-1: Example of standard HPC application workflow (above) and modified E10 workflow (below).

Figure 4.4-1 graphically shows a typical HPC application workflow. HPC codes perform some computation, generate data, and thus write this data to a shared file for later processing. This workflow is shown in the upper part of the figure (cache disable). After the compute part is completed a shared file is opened, data is written to it and then it is closed, ending the first phase of compute and I/O.

The lower part of the figure (cache enable) displays the workflow modification previously mentioned and required to take advantage of the DEEP-ER cache. Now, after the compute part is completed, the shared file (and a certain number of local cache files) is opened, data is written to the cache and then cache synchronisation (`ADIOI_Sync_thread_start`) is started together with the next compute phase, without closing the file. Instead, the file is closed at the end of the new compute phase, allowing the Exascale10 implementation to flush the data meanwhile. Figure 4.4-2 shows a C code example for the modified workflow just discussed.

```
MPI_Comm comm = MPI_COMM_WORLD;
MPI_File fh_1, fh_2, fh_3, ...;
MPI_Status status;
char *buf;
int count;

// compute #1
compute(&buf, &count);

// open shared file #1
MPI_File_open(comm, "file_1", MPI_MODE_CREATE, &fh_1);

// write data from compute #1 to shared file #1
MPI_File_write_all(fh_1, buf, count, MPI_CHAR, &status);
```

```
// instead of closing file #1 start compute #2
compute(&buf, &count);

// now close shared file #1
MPI_File_close(&fh_1);

// then open shared file #2
MPI_File_open(comm, "file_2", MPI_MODE_CREATE, &fh_2);

// write data from compute #2 to shared file #2
MPI_File_write_all(fh_2, buf, count, MPI_CHAR, &status);

// instead of closing file #2 start compute #3, and so on ...
...
```

Figure 4.4-2: Example of C code explicitly using E10 functionalities.

The average application perceived bandwidth for the cache enable case can be computed using the following formulas:

$$bw(k) = \frac{S(k)}{T_c(k) + \max(0, T_s(k) - C(k+1))} \quad (1)$$

$$\overline{BW} = \frac{\sum_{k=0}^{N-1} S(k)}{\sum_{k=0}^{N-1} T_c(k) + \max(0, T_s(k) - C(k+1))} \quad (2)$$

From Formula 1 it is clear that the bandwidth achievable when writing S(k) MB to the file depends on how fast data can be written to the cache ($T_c(k)$) and how much cache synchronisation ($T_s(k)$) can be overlapped with computation from the next phase (C(k+1)). The average bandwidth can be computed using Formula 2. The corresponding saved time is highlighted by the green band in Figure 4.4-1 ($T_\$$).

One important thing to notice is that synchronisation can be hidden as long as there is a following compute phase. Synchronisation costs for the last phase cannot be hidden since there is no further compute and will therefore impact the average bandwidth.

4.4.1 MPIWRAP

Although the described workflow modification is simple, it might not be possible for every application to change the source code to reflect this change. Therefore, a wrapper library for MPI-IO that performs the required changes for the application behind the scenes was developed.

The MPIWRAP library can be used to transparently feed the new hints to the application. Hints are described in a configuration file[4] parsed when the MPI library is initialised. Afterwards, hints for a certain file are set when the file is opened. An example of configuration file is shown Figure 4.4-3:

[4] The configuration file is defined using the Json file format.

```
{
    "Guardnames": "test_file",
    "File": [{
        "Path": "/work/deep47/test_file",
        "Type": "MPI",
        "cb_buffer_size": "4194304",
        "ind_wr_buffer_size": "524288",
        "cb_nodes": "64",
        "romio_cb_read": "enable",
        "romio_cb_write": "enable",
        "romio_no_indep_rw": "true",
        "striping_unit": "4194304",
        "striping_factor": "4",
        "e10_cache": "enable",
        "e10_cache_path": "/scratch",
        "e10_cache_flush_flag": "flush_immediate",
        "e10_cache_discard_flag": "enable",
        "e10_cache_threads": "1"
    }
}
```

Figure 4.4-3: Example of MPIWRAP configuration file

In Figure 4.4-3 the `Guardnames` field must be set to the basename used for the shared file. If the application is writing checkpoint data to a series of files with same basename and progressive numbering (e.g. test_file_1, test_file_2, etc) the `Guardanames` field will tell the MPIWRAP library that `MPI_File_Close()` of test_file_k will be ignored and return success. When the next test_file_k+1 in the sequence is opened, just after the compute phase k+1 has completed, the test_file_k has to be closed first.

The `Path` field contains the full name of the file (or its suffix) to which the hints will be applied. Hints can be applied to a specific file, to all the files inside a directory or to every file having the same suffix (e.g. /work/deep47/test_file will match /work/deep47/test_file*).

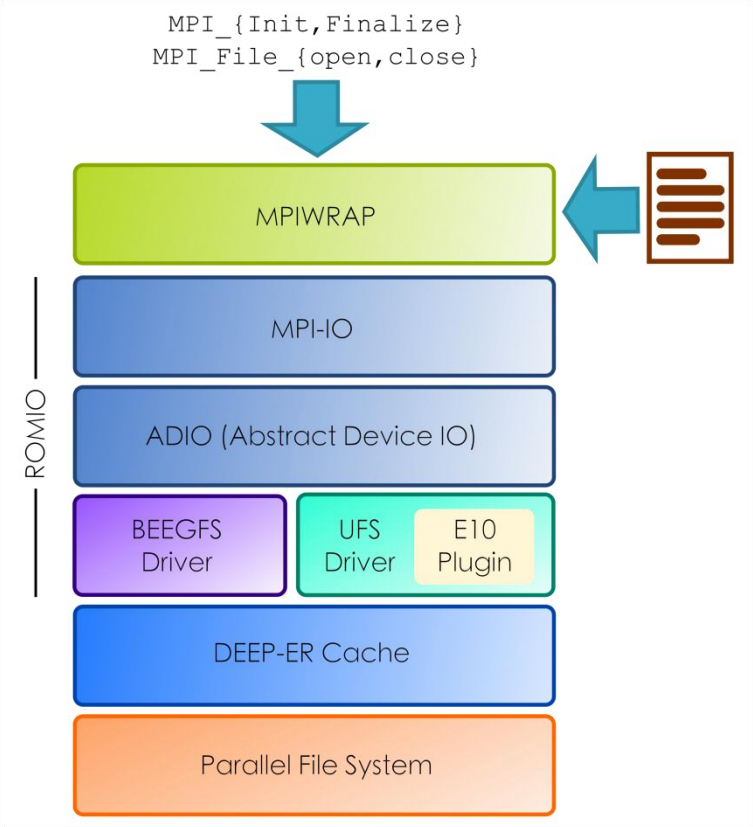

Figure 4.4-4: Exascale10 software stack including MPIWRAP on top of MPI-IO

The MPIWRAP library is layered on top of MPI-IO as show in Figure 4.4-4.

The following figure shows how the Exascale10 extensions can be easily and transparently used by applications simply setting the `MPI_HINTS_CONFIG` environmental variable:

```
#!/bin/bash -l
#PBS -N enable_64_4194304_524288.pbs
#PBS -e enable_64_4194304_524288.pbs.err
#PBS -o enable_64_4194304_524288.pbs.log
#PBS -l walltime=00:30:00
#PBS -l nodes=64:ppn=8
#PBS -m abe

module load parastation
mpiexec --env=MPI_HINTS_CONFIG \
     /homec/deep/deep47/coll_perf/enable_64_4194304_524288.hints \
     -np 512 /homec/deep/deep47/benchmarks/coll_perf/coll_perf \
     -f /work/deep47/test_file -i 4 -b 256
```

Figure 4.4-5: Example of bash script using MPIWRAP

The script in Figure 4.4-5 uses the configuration file shown in Figure 4.4-3 to set the hints in the MPI application. Afterwards the script can be submitted to the job scheduler, as it would be normally.

4.5 Source code licensing and release

The Exascale10 source code will be released under the same ROMIO license, since this has been used as fundamental building block during the DEEP-ER project, and will be made publicly available by the end of the project.

4.6 E10 Summary and Conclusions

The Exascale10 approach in DEEP-ER addresses the problem of multi-tier memory system integration in new HPC systems. Local NVM devices in compute nodes such as SATA or PCI Express SSDs are not currently integrated into existing MPI-IO implementations. Exascale10 makes these fast memory devices available to users through a set of MPI-IO hints extensions. Whenever the application cannot be changed an additional software layer is provided, which can make the use of the new hints almost transparent by describing them inside an apposite configuration file and by passing this to the application using an environment variable.

5 Summary and next steps

This deliverable has described the design and development of the I/O software packages. The APIs available for application developers cover different usage scenarios enable straight-forward integration into as many applications as possible.

In addition to the initial planning reflected on the DoW, an important force behind the developments presented in this document is the co-design between different work packages. Beyond the general support for the different software packages both inside and outside the work package we want to explicitly mention some of the co-design efforts with other work packages

- Collaboration with WP3 (architecture design)
  - Evaluation of NVMe devices
  - Integration of continuous benchmarking into the monitoring
- Collaboration with WP5 (resiliency)
  - Request to add CRC checksum calculation during flush/prefetch to the BeeGFS cache API
  - Addition of functions to check status of asynchronous operations
  - Design and development of SCR extensions and buddy checkpointing in SIONlib
- Collaboration with WP6 (applications)
  - Support for implementation of I/O layers in different applications
  - Documentation of the different I/O layers for the application developers
- Collaboration with WP6 (applications) and WP5 (resiliency)
  - Modification of the JUBE-integrated application GERShWIN to support SCR

With the implementations described in this deliverable the development of all I/O layers and their integration with each other and the resiliency features from WP5 are completed.

In the remainder of the project we will focus into helping the application developers to integrate the different I/O strategies into their applications and collect benchmarking measurements. In addition, the feedback from the users will be taken as new input to further streamline the usage of the DEEP-ER I/O software packages.

References

[BeeGFS documentation] http://www.beegfs.com/content/documentation/

[BeeGFS source code] http://www.beegfs.com/content/source-code/

[BeeGFS webseite] http://www.beegfs.com

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

[Exascale10 source code and documentation] http://github.com/gcongiu/E10