SEVENTH FRAMEWORK PROGRAMME

Research Infrastructures

FP7-ICT-2013-10

DEEP-ER

DEEP Extended Reach

Grant Agreement Number: 610476

D4.1

Definition of Requirements for I/O

Approved

Version:	2.0
Author(s):	N. Eicker (JUELICH)
Contributor(s):	S. Breuner (FHG-ITWM), G. Congiu (Xyratex), S. Narasimhamurthy (Xyratex), K. Thust (JUELICH)
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: D4.1	
	Deliverable Nature: Report	
	Deliverable Level: PU*	**Contractual Date of Delivery:** 31 / March / 2014
		Actual Date of Delivery: 31 / March / 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:** Definition of Requirements for I/O	
	ID: D4.1	
	Version: 2.0	**Status:** Approved
	Available at: http://www.deep-er.eu	
	Software Tool: Microsoft Word	
	File(s):DEEP-ER_D4.1_Definition_of_Requirements_for_IO_v2.0-ECapproved	
Authorship	**Written by:**	N. Eicker (JUELICH)
	Contributors:	S. Breuner (FHG-ITWM), G. Congiu (Xyratex), S. Narasimhamurthy (Xyratex), K. Thust (JUELICH)
	Reviewed by:	M.Richter (Intel), E.Suarez (JUELICH)
	Approved by:	BoP/PMT

Document Status Sheet

Version	Date	Status	Comments
1.0	31/Mar/2014	Final version	EC submission
2.0	21/Oct/2014	Approved	EC approved

Document Keywords

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

Table of Contents

List of Figures

Executive Summary

This deliverable summarises the application I/O requirements within the DEEP-ER project and introduces the three I/O components, namely, FhGFS, SIONlib and Exascale10. They address different aspects of I/O within the DEEP-ER project. The application I/O requirements have been systematically addressed through a detailed questionnaire that was submitted to the application developers. This deliverable also explores the inputs needed for these I/O components from the various work packages.

1 Introduction

The DEEP-ER project intends to develop a next generation highly scalable and efficient I/O system based on the Fraunhofer file system (FhGFS). It will support I/O intensive applications, using optimised I/O middleware SIONlib and Exascale10 with respect to complementary aspects of their requirements. Figure 1 sketches the overall architecture of this I/O middleware. Besides the actual applications a main consumer of these capabilities will be a multi-level checkpoint scheme to be developed by WP5. Even though not explicitly mentioned in the figure, checkpointing here is assumed to be part of the application.

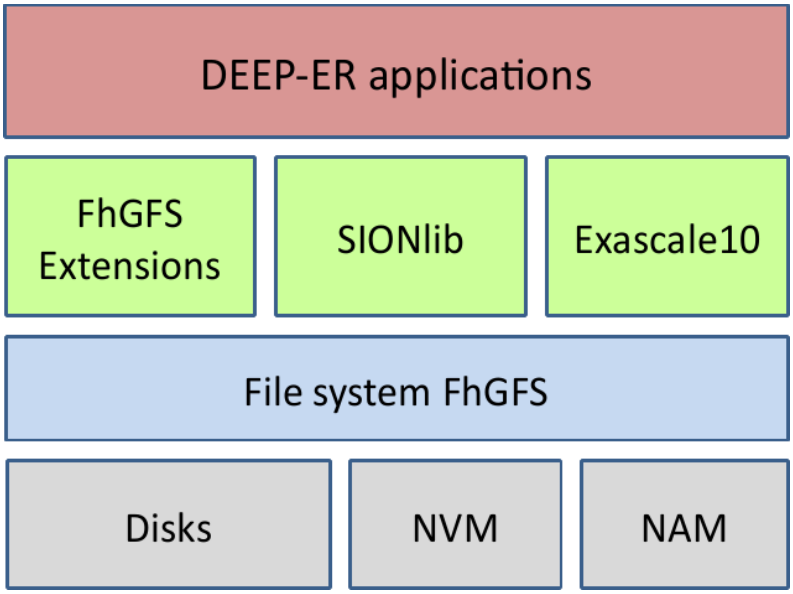

Figure 1: DEEP-ER I/O architecture

In order to enable WP4 to provide a detailed design and realisation of a corresponding I/O infrastructure to the DEEP-ER project, a thorough investigation of the I/O requirements of the DEEP-ER applications with respect to the three technologies in the focus of this work-package is necessary. For that, in the context of the Design and Development Group (DDG) of the project, together with representatives from WP3 and WP5, a questionnaire was composed and submitted to the application developers of WP6. The aim of this questionnaire was to characterise the applications not only with respect to I/O, but also regarding aspects interesting for WP3 (System architecture) or WP5 (Resiliency software). The present document focuses on the application's I/O aspects.

Based on the results of the questionnaire the overall architecture of the DEEP-ER I/O system is sketched. It will employ three different technologies that shall act in a complementary way within the project. As a global parallel file system FhGFS will be utilised to provide the common basis of I/O operations in the context of DEEP-ER. On top of this SIONlib will act as a compound-layer to allow the DEEP-ER applications to make use of global FhGFS most efficiently with minimal effort of adaptation. This reflects the fact that currently almost all applications within the projects conduct task-local I/O. Last but not least Exascale10 addresses the so-called small I/O problem. Here the ever-increasing level of parallelism results in a huge number of small I/O operations in order to achieve the global access of file data in Exascale parallel applications.

Furthermore this document specifies the requirements of the DEEP-ER I/O system and its components on both, the underlying hardware architecture and the low-level software components building the foundation of the overall system-software stack.

The overall structure of the document is as follows: We start with a brief compendium of the results with respect to I/O from the application questionnaire. After that, we discuss the DEEP-ER I/O system along its components within the next three chapters. Finally we conclude our results.

This document addresses two main audiences. On the one hand it allows application developers aiming to port their application to the DEEP-ER system with a high-level overview on the DEEP-ER I/O system specifying the main-design goals. On the other hand it outlines the requirements of the DEEP-ER I/O system and its components on the sub-systems to be provided by other Work Packages in the project, specifically WP3 and WP5. With this respect it builds the basis for further co-development with other WPs. Furthermore, this document is intended to present an early sketch of the DEEP-ER I/O architecture and, therefore, might be seen as part of the overall DEEP-ER design documentation.

2 I/O Requirements of DEEP-ER applications

The questionnaire submitted by WP3, WP4 and WP5 to WP6's application developers tried to shed some light on several aspects of I/O that may have impact on the design of both the hardware and software subsystems in the DEEP-ER project. Some of them, in particular, are very important for WP4 since they directly influence the design of the storage subsystem and its related components (i.e. file system, middleware and I/O libraries).

In this section we explore the most relevant aspects that were presented to WP6 in the aforementioned questionnaire along with the generated outcomes and a discussion on the impact of these on WP4's objectives.

2.1 Ratio of I/O and Processing

The ratio between I/O and processing expresses the impact that I/O has on the total runtime of the application and gives some indication on whether or not the application could be I/O bound. At the moment, for the most I/O intensive DEEP-ER applications, this value ranges from 50% for the *NEST*[1] application (JUELICH) in the case where computation is spread over all the nodes but with only one node doing all the I/O, to more than 100% for the *Oil*

[1] The application NEST is an in-kind contribution to the DEEP-ER project by partner JUELICH.

Exploration application (BSC) in the worst case scenario, where I/O time for one step is one order of magnitude larger than computing time and I/O happens every 4 to 9 steps

On the other hand, one of the less I/O intensive applications, the *MAXW-DGTD* application (INRIA), has only 10% ratio between I/O and processing, given that it is currently not implementing any checkpoint/restart strategy. Similarly, for all the other applications the impact of I/O on the total runtime is small and in general none of the DEEP-ER applications seemed to be I/O bound at the beginning of the project. Things are likely to change when checkpoint/restart will be taken into account. Additionally, developers are also exploring the parameter space(s) of their applications for scale-out and this is bound to generate more I/O. Therefore, even though most applications seem not to be I/O bound for the time being we expect this to change in the near future.

2.2 I/O Interfaces

The I/O interface used by an application directly reflects on its I/O behaviour. At the current state of development, as emerged from the answers to the questionnaire, most of the DEEP-ER applications are using the POSIX I/O interface to perform task local I/O - the only exceptions being the *Crystal Deformation & Earthquake Source Dynamics* application (BADW-LRZ) and the *Lattice QCD* application (UREG) which can use both POSIX I/O and MPI-IO. Thus, every process in the parallel application writes output data to a separate file either by directly invoking the POSIX write API or indirectly through high level I/O libraries, such as HDF5.

Per task I/O is simple to implement and in general performs quite well for existing scales, but as applications scale out to hundreds of thousands, or even millions of processes, this puts a huge burden on the file system metadata service. At that point, the file system will have to create a very large number of files at the same time and perform metadata operations on them (stat, lookup, etc).

The per task I/O pattern happens to be a very good use case for SIONlib, since application developers may want to keep the implementation of I/O as simple as possible without sacrificing performance and introducing additional development overhead. SIONlib can almost transparently convert per task I/O into parallel I/O to a few shared files behind the scenes, thus reducing the burden on the metadata service. Basically only the file-open operation has to be adapted leaving all other I/O activity untouched.

Exascale10 based ExCol on the other hand is most effective with parallel I/O on large shared files (which is done at smaller scales using MPI-IO today). The *Crystal Deformation & Earthquake Source Dynamics* and *Lattice QCD* applications are already using MPI-IO. Even though other applications are not using MPI-IO at the moment, from the questionnaire answers most of them plan to start using this interface either directly or indirectly through high level I/O libraries (i.e. parallel HDF5 and pnetCDF). This provides an opportunity for the use of ExCol for performance optimisation.

2.3 Size of written data chunks

The size of written data chunks and the possibility for this information to be known a-priori before the file is opened is important for SIONlib. Indeed, SIONlib must be aware of the maximum size of data written by a single write operation in order to function effectively. For all DEEP-ER applications this information can be easily worked out once the input

parameters and the problem partitioning are decided. Therefore, the corresponding parameters can be provided to the operations that actually open or write files to the file system.

2.4 Cache coherency requirements

Coherency of file system caches is another important aspect that impacts the overall I/O and storage subsystem behaviour and performance. Cache coherency requirements reflect on the necessity of the file system to make each and every single write operation immediately visible to all the file system clients. If the application uses files to exchange information between processes, there will be a huge burden on the file system which will have to guarantee cache coherency using very expensive locking schemes. On the other hand, if a file is being used only by the process that has created it, the file system can relax coherency requirements using, for example, an eventual coherent model. This means that writes to the file can be made persistent only when the file is closed, by flushing the cache content to the file, or when data in the cache is explicitly synchronised with the file.

From the questionnaire answers we know that several applications are using files just to write restart information (*Space Weather* from KULeuven and *Lattice QCD* from UREG are good examples) or to save some model parameter that will be eventually deleted, hence obviating the need for strong cache coherency.

This questionnaire outcome confirms the value of the intended DEEP-ER storage interface, which will allow applications to provide hints to relax coherency on files, thus avoiding expensive checks and at the same time enabling the usage of fast local solid state drives in the client nodes as cache.

2.5 New I/O routines

Applications developers were also asked, if they could envisage any new I/O routine provided by the file system that could highly improve their application performance. From an evaluation of the answers to the questionnaire it appears that none of the application can greatly benefit from such new I/O routine.

2.6 Summary

In summary, the questionnaire provided valuable inputs to WP4 for architecting the I/O subsystem.

In the following chapters we discuss in more detail the individual components of the I/O solution for DEEP-ER and their requirements on the underlying hard- and software components. At the same time this sketches a first draft of the I/O architecture WP4 will specify in more detail during the following months.

3 FhGFS

This chapter discusses the FhGFS part of the DEEP-ER I/O subsystem including its requirements to other Work Packages.

3.1 Component Overview

FhGFS [FhGFS website] is a distributed file system, which was specifically designed for parallel IO. It provides a POSIX I/O interface to the applications on the compute nodes and transparently distributes stored file contents and file system metadata across multiple servers to allow scaling of throughput and capacity.

Keeping the POSIX interface instead of defining a completely new API for I/O is one of the fundamental aspects of the DEEP-ER storage architecture. Other scale-out I/O approaches for high numbers of nodes like Lustre DAOS [Lombardi13] as part of the US DOE Fast Forward Exascale initiative [DOEFF] or Hadoop [ShvachkoKRC10] typically try to abandon the POSIX interface and instead use completely new APIs. This requires major modifications to the applications, makes the applications non-portable between different systems and prevents usage of other standard or legacy tools that are not available for such non-standard I/O interfaces.

Instead, to overcome the scalability limitations that are inherent from some of the very strict requirements of the POSIX interface semantics, the DEEP-ER model will add minor extensions to the POSIX interface. This approach will allow application developers to relax semantics like cache coherence through trivial modifications, while remaining fully backwards compatible with applications that do not make use of these extensions and managing the complexities of e.g. different types of storage devices completely transparent for the applications.

3.2 Requirements on FhGFS in DEEP-ER

To enable the storage subsystem to scale to the level that is intended by DEEP-ER, it is important to reduce the amount of communication with central system components (such as a global storage or metadata server that would be accessed by all compute nodes) as much as possible and instead switch to a design that foresees to scale linearly with the number of compute nodes in the system.

New storage technologies (such as non-volatile memory (NVM)) allow significantly faster access than traditional hard-drives, especially for non-sequential I/O. Unfortunately currently such devices are only affordable in limited capacity sizes. Nevertheless, the DEEP-ER storage system design will be able to incorporate such devices and exploit their advantages.

The following picture illustrates the storage system design for DEEP-ER that fulfils these requirements.

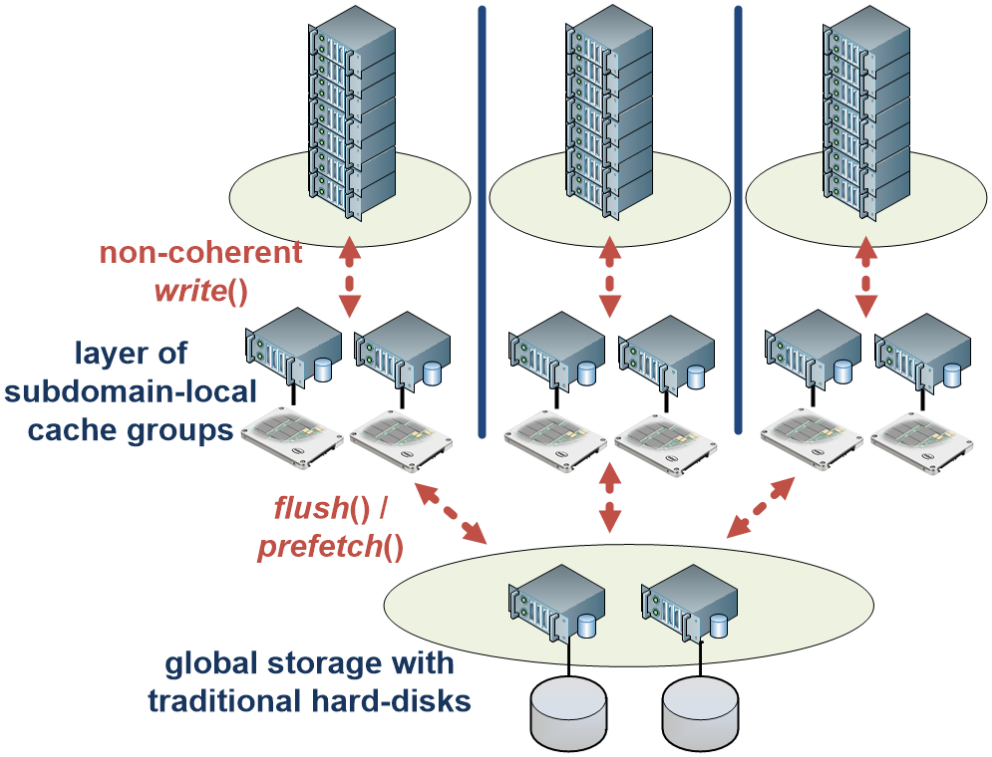

Figure 2: DEEP-ER storage system design

Hard-drives provide the high capacity for globally shared data (shown at the bottom layer in the picture). To avoid as much communication as possible with this centralised layer of globally shared storage servers, a cache layer of per-subdomain storage servers based on new storage technologies like NVM shall be added (as sketched in the middle layer of the picture). The cache layer allows applications to write data that is not shared with other subdomains (i.e. non-coherent) and thus provides the ability to scale I/O performance linearly with the number of compute nodes. Both of these layers will be managed through FhGFS.

Access to data on the global storage layer will be seamless to the applications, while access to the cache layer will require the applications to provide hints to the file system. Such hints are required since the file system is unable to foresee which files need to be coherent across the cluster – at least if we assume the absence of an information management system at a centralised instance, which would be counterproductive to our design-goal of scalability.

However, from the results of the questionnaire it became clear that most of the data that is generated by the applications does not need to be coherent and thus the cache layer promises to increase the efficiency of I/O significantly. This is especially true for temporary data, such as checkpoints, which might typically reside only on the cache layer and eventually be deleted without the need to ever flush them to the global storage layer.

3.3 Constraints of FhGFS on other Work Packages

3.3.1 General System Requirements of FhGFS

The standard version of FhGFS can run on any little or big endian platform that supports Linux. The FhGFS client can connect via a TCP/IP or ibverbs-compatible (e.g. InfiniBand)

network to the storage servers. The storage and metadata servers of FhGFS use disks formatted with a standard local Linux POSIX file system (e.g. ext4) to store file system and user data.

3.3.2 WP3

The DEEP-ER storage system design requires a small number of traditional, dedicated storage servers based on conventional hard-drives for the global storage layer.

The cache layer needs to scale with the number of compute nodes and thus requires a set of per-subdomain (e.g. per-rack) non-volatile memory devices. These might be located inside one or more of the compute nodes or in dedicated hosts, as long as there is a way to access them through a local Linux POSIX file system. In principal, this layer is also suitable to utilise devices like NAMs (instead of NVM), if they are made available to FhGFS through a POSIX file system interface.

3.3.3 WP4, WP5, WP6

To take advantage of the cache layer, the DEEP-ER applications, the DEEP-ER checkpointing framework and other DEEP-ER I/O middleware such as SIONlib will need to provide hints to FhGFS though I/O API extensions, if the generated data can be stored non-coherent in a cache subdomain. If the data needs to become globally accessible later (e.g. a checkpoint that needs to be read from a compute node in another subdomain), the application will be able to flush the data from the cache layer to the global storage layer.

If no such hints are provided, generated files will always be stored in the global layer, so that the file system is backwards-compatible with legacy applications and unmodified I/O sections of the DEEP-ER applications, which are not performance-critical.

4 SIONlib

This chapter describes the definition of the requirements for SIONlib in the context of the DEEP-ER project.

4.1 Component Overview

SIONlib [SIONlib website][FringsWP09][FrecheFS10] is motivated by task-local I/O and offers an API which – from the user's perspective – offers a single output file per task. The emphasis of SIONlib is on optimising the I/O performance with a minimum amount of effort for the user. This especially holds for code changes required in order to morph task-local I/O to the use of the SIONlib API, since the basic calls resemble standard ANSI C calls.

SIONlib pursues different strategies in order to achieve a performance increase for the application's I/O. Some of them are originating from the knowledge of the underlying file system, which includes preventing multiple tasks from writing to the same file system block or proper alignment of write operations to the file system block size. The other main design idea is the reduction of meta data operations on the file system by reducing the number files significantly. Ideally, only one file is written or the number is kept in the order of the number of I/O nodes.

These features only require communication during the open call but do not need further communication for subsequent I/O operations, except for the final closing of the file.

By dropping the demand of no communication at all during I/O, SIONlib also features collective I/O. This is specifically designed for I/O patterns, where a huge number of tasks output very little amounts of data (as compared to the file system block size).

4.2 Requirements on SIONlib in DEEP-ER

SIONlib's general API is already existing and mature. Possible extensions to the existing API will be made available to WP5 (see below) either inside of SIONlib or in a thin layer between SIONlib and the resiliency functionality of WP5. The concept of "non-coherent" storage provided by FhGFS complements the structure of SIONlib perfectly and promises to yield a close to ideal scaling. This feature could be directly used with the current version of the SIONlib API.

Possible extensions of SIONlib to support OmpSs natively are under investigation and will be defined later. These will lead to a seamless support of the DEEP programming model [EickerLMS13].

To support the two main strategies for resiliency in this project, i.e. common checkpointing and "buddy" checkpointing, some extensions to the current concept of SIONlib might be required. Common checkpointing will usually act globally, i.e. each task writes its data required for restarts to a globally visible file system. This approach is already possible with SIONlib and has been used in many applications. The performance and scalability is likely to benefit from using local file systems. This requires internal extensions to SIONlib, but these will neither affect the user interface itself nor have any requirements on other WPs.

4.3 Constraints of SIONlib on other Work Packages

4.3.1 General System Requirements of SIONlib

For SIONlib to work on any architecture it needs a file system providing a POSIX interface. In general, SIONlib can handle global files or shared files, which might be distributed over several local file systems (per node or per set of nodes). In the latter case SIONlib is internally using the so-called SIONlib multi-files. All changes required to store such files to local disks are internal and do not affect the SIONlib user interface.

Since SIONlib uses MPI, OpenMP or hybrid interfaces for internal communication, these need to be present on the system.

4.3.2 WP3

The benefits of new hardware like NAM are more likely to be exploited by either the underlying file system (FhGFS) or directly by higher level functionality like the resilience mechanism developed in WP5. Both, NAM devices and local storage, will be utilised by SIONlib via local file systems. No adaption of SIONlib is required to support the DEEP programming model that employs MPI_Comm_spawn() to run parts of the application on the Booster nodes of the system. Furthermore, SIONlib files will be accessed only within a single communicator (i.e. intra-communicator on CNs or BNs) and not via a global scope spanning several communicators, which would require a complicated inter-communicator functionality.

4.3.3 WP4

Since SIONlib does not directly access underlying storage hardware, the operating system has to provide a file system interface as an abstraction layer. In particular for storing centralised meta-data, which is required to enable local storage-devices via SIONlib a global file system is required to be accessible from all nodes of the system. Therefore, such type of file system – in the case of DEEP-ER this is FhGFS – must be available from the BN.

4.3.4 WP5

For buddy checkpointing there are mainly two different ways of implementation:

- A POSIX interface to some local file system that allows to access data from remote files
- Using a SIONlib's internal strategy similar to the SIONlib coalescing I/O strategy, where tasks could become collectors and receive and store data from other tasks.

The logic of when and which data to checkpoint and how to restart from checkpoints is out of the scope of SIONlib and shall be implemented by WP5 preferable on top of SIONlib.

Restarting also needs to keep the rank numbers of the processes unchanged for all but the processes which failed. This might create new requirements on the resource management as provided by the DEEP project.

4.3.5 WP6

SIONlib's task-local strategy requires the applications to follow the same approach. For instance, tasks should not access data which is written by another process. Furthermore, the amount of data to be written during output should be known before opening the output file.

According to the replies from the questionnaire, the DEEP-ER applications fulfil both requirements, making them good candidates for integrating SIONlib as an I/O library.

5 Exascale10

5.1 Component Overview

5.1.1 Value proposition

The Exascale10 workgroup intends to provide a set of middleware components to address many of the I/O issues of Exascale I/O intensive applications. Feedback from application experts worldwide obtained through multiple workshops has provided a reference architecture for the Exascale10 middleware stack and core APIs [Exascale10 white paper][E10 website]. This architecture addresses specific I/O problems as well as the need for analytics and simulation along with machine learning mechanisms to adapt the I/O subsystem to continuously changing I/O workloads, thus providing optimal performance. The middleware components are generalised to be interoperable with various backend file systems and object stores.

In DEEP-ER Exascale10 looks at building a middleware component to address a specific performance problem of Petascale I/O applications that will be of central importance at

Exascale: the so called *"small I/O problem"*. This component is proposed to work on top of FhGFS and will be made available through the Exascale10 software middleware to the community for further exploitation by Exascale I/O intensive applications going forward. We next go into more detail on this specific problem addressed by Exascale10. Given that Exascale10 is a fairly new initiative compared to the other I/O solutions described in this deliverable, it seems reasonable to provide more background on this specific topic here.

5.1.2 The small I/O problem

HPC applications typically process large amounts of data that have to be read/written from/to a large shared file residing in a global parallel file system. To make the data set manageable, this is usually partitioned into smaller subsets and assigned to a certain number of processes to be elaborated in parallel. Image processing applications operating on very large images, for example, have to partition the original image into a number of smaller sub-images that afterwards are elaborated independently in parallel.

Data sets (such as N-dimensional arrays) in parallel file systems are logically flattened into a linear array of bytes and then striped over several I/O servers (logical data layout and physical data layout do not match in parallel file systems), resulting in the loss of the original spatial data locality. Because of this characteristic, accesses to spatially contiguous regions in the data set may result in non-contiguous accesses to the file. Therefore, applications having very fine grained data partitioning may generate a huge number of small, non-contiguous I/O requests to the parallel file system.

The small I/O problem refers to a performance issue common in HPC applications issuing many small non-contiguous read/write operations to the parallel file system. Indeed, parallel file systems provide best I/O bandwidth performance for large contiguous requests while they typically provide only a fraction of the maximum available I/O bandwidth in the opposite case [ThakurEtAl99][AveryALLN06][HeSSYT11]. This is primarily due to the large number of Remote Procedure Calls (RPCs) generated by the file system clients that overwhelms I/O servers and the resulting high number of hard disk drives' head movements in every I/O server (seek overhead).

Having recognised the small I/O problem, Collective I/O was proposed by the MPI-IO community as solution [ThakurEtAl99]. Collective I/O exploits global I/O knowledge from all the processes of a HPC application reading/writing data to/from a shared file. This knowledge is used to build an aggregate view of the accessed file region and coalesce all the corresponding small non-contiguous requests into a fewer number of big contiguous accesses that can be later issued to the parallel file system. In collective I/O only a subset of the available processes actually performs I/O, these processes are called "aggregators", since they gather and aggregate all the requests on behalf of the other processes, whose only role is to send/receive data to/from aggregators.

The large I/Os performed by aggregators reduces the actual number of I/O requests that needs to be accounted for by the I/O stack and brings about large sequential accesses in the I/O subsystem. [ThakurEtAl99] quantifies the improvements that are obtained through such collective I/O mechanisms at scales that range from tens to few hundreds of processes. Figure 3 below pictorially exemplifies the basic collective I/O mechanism just described.

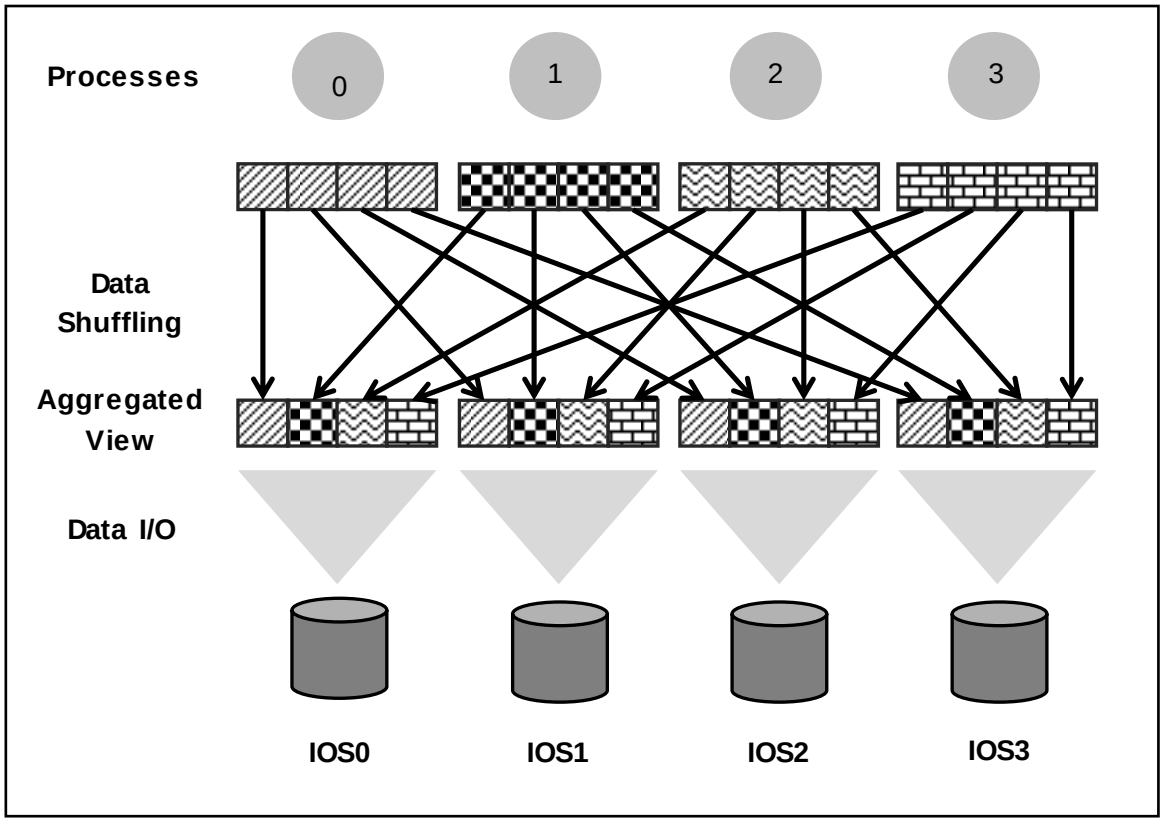

Figure 3: Collective I/O in MPI-IO

In Figure 3 there are four processes each of which plays the role of aggregator. Collective I/O proceeds in two phases: *Data Shuffling Phase* and *Data I/O Phase*. Data shuffling takes place between all the processes and aggregators and is aimed to build the logically contiguous file regions (or file domains) that will be later accessed during the *Data I/O Phase*.

Two-phase I/O has a number of problems that limit its scalability. The main issues are, (a) synchronisation overheads between aggregators exchanging data with every other process – where the aggregator I/O is bottlenecked by the slowest process [YuJV08], (b) aggregators' contention on file system stripes (primarily due to stripe boundary misalignment of file domains) [LiaoA08], (c) aggregators' contention on I/O servers (related to non-optimal file domain partitioning strategy), (d) pressure that large collective buffers can have on system memory (due to scarce amount of available memory per single core in future HPC clusters), and (e) high memory bandwidth requirements due to the large number of data exchanges between processes and aggregators [LuCTZ12]. There is also a mismatch between logical file layouts and the collective I/O mechanism which does not seem to take the physical data layouts into consideration [ChenSTRG11][ZhangJD09].

5.1.3 Solution framework

For applications in DEEP-ER to exploit two-phase collective I/O like mechanisms, what is needed is a rethink of the collective I/O framework and the existing collective I/O implementation [ROMIO Implementation] taking the above problems into consideration. The "ExCol" solution will leverage the existing ROMIO to bring about Exa-scalability for DEEP-ER

applications. Figure 4 provides a snapshot of the proposed architecture. ExCol will eventually be part of the open source Exascale10 software middleware.

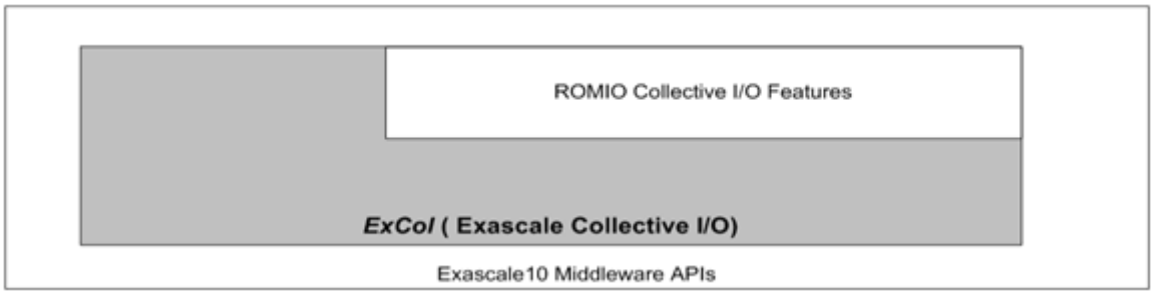

Figure 4: ExCol solution

ExCol is ideal for intensive I/O on large shared files, where smaller I/Os are aggregated intelligently over and above ROMIO collective I/O implementations – and hence is very complimentary to SIONlib. A more primitive version of ExCol can also be provided for task local file I/O as well.

5.2 Requirements on Exascale10 in DEEP-ER

The DEEP-ER prototype requires an I/O middleware to work on top of the FhGFS file system to address the needs of *highly scalable application I/O*. The key requirement from DEEP-ER on Exascale10 hence was to address specific crucial I/O related bottlenecks, which the applications could easily exploit without demanding big changes in the file system/storage backend architecture or the applications themselves. The small I/O problem addressed by ExCol is a very key Exascale10 based I/O performance optimisation, which Exascale10 brings into the project. Furthermore, from the perspective of the DEEP-ER applications, ExCol is very complimentary to the problems addressed by FhGFS and the SIONlib middleware, with the complementarity being very valuable to DEEP-ER from a cost benefits perspective(avoiding feature redundancies).

5.3 Constraints of Exascale10 on other Work Packages

5.3.1 WP3

The ExCol solution would work in the Booster Nodes utilising the KNL many-core processors so that they can improve upon the collective I/O operations that ensues at the Booster cores [D3.1]. The shared files could reside in NVM within the Bricks (in case they are specified as non-coherent) or they can reside behind an external I/O server in the global file system. The Brick architecture hence is expected to meet the requirements of ExCol.

5.3.2 WP4

FhGFS meets the requirements of ExCol in terms of providing the file system client on top of which ExCol would work. There is good synergy with SIONlib in that ExCol addresses a very complimentary problem to that of SIONlib (in terms of addressing I/O to large shared files). We have identified the possibility of ExCol providing further support to SIONlib in its collective I/O operations. The requirements of such a solution however will be investigated in the future.

5.3.3 WP5

ExCol could possibly provide scalable collective I/O functionalities to the SCR framework which is the foundation of WP5. SCR leverages MPI (and MPI-IO) and hence forms a potential candidate for collective I/O optimisation.

5.3.4 WP6

Answers to the applications questionnaire have revealed that five of the seven applications plan to use MPI-IO on shared files. Hence, ExCol is ideally positioned to address the bottlenecks that are expected to arise when the applications scale out and when they operate in I/O intensive modes.

6 Conclusion

This deliverable has introduced and described the various I/O components within DEEP-ER project, which forms a very critical extension to the DEEP project, for dealing with application I/O and checkpoint/restart I/O. Various key aspects of application I/O ranging from the ability to provide a scale-out file system (with capabilities of specifying data coherency within the different storage tiers), optimising I/O performance for small files and dealing with small I/O at scale can be addressed by the I/O components as described in the deliverable. Requirements towards the different work packages have also been summarised.

The next deliverable for WP4, D4.2, will focus on defining the I/O interfaces for these components.

References

[D3.1] "Hardware Architecture Specification", DEEP-ER Deliverable D3.1

[FhGFS website] The Fraunhofer Parallel File System: http://www.fhgfs.com

[Lombardi13] Lombardi, J. (2013); DAOS Changes to Lustre; Lustre User Group Conference, San Diego, California, April 2013

[DOEFF website] US Department of Energy: Fast Forward Exascale Initiative; https://sites.google.com/site/exascaleinitiative/fast-forward, retrieved 2014-03-20

[ShvachkoKRC10] Shvachko, K. et al (2010); The Hadoop Distributed File System; 26th IEEE Symposium on Massive Storage Systems and Technologies, Incline Village, Nevada, May 2010

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

[**FringsWP09**] Frings, W.; Wolf, F.; Petkov, V. (2009); Scalable Massively Parallel I/O to Task-Local Files; Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, Portland, Oregon, November 14 - 20, 2009, SC'09,

SESSION: Technical papers, Article No. 17, New York, ACM, 2009.ISBN 978-1-60558-744-8. - S. 1 – 11 DOI: 10.1145/1654059.1654077

[**FrecheFS10**] Freche, J.; Frings, W.; Sutmann, G. (2010); High Throughput Parallel-I/O using SIONlib for Mesoscopic Particle Dynamics Simulations on Massively Parallel Computers Parallel Computing: From Multicores and GPU's to Petascale ed.: B. Chapman, F. Desprez, G.R. Joubert, A. Lichnewsky, F. Peters and T. Priol, Amsterdam, IOS Press, 2010. Advances in Parallel Computing Volume 19. - 978-1-60750-529-7. - S. 371 – 378 DOI: 10.3233/978-1-60750-530-3-371

[**EickerLMS13**] Eicker, N.; Lippert, T.; Moschny, T.; Suarez, E.; The DEEP project: Pursuing cluster-computing in the many-core era; Proc. of the 42nd International Conference on Parallel Processing Workshops (ICPPW) 2013, Workshop on Heterogeneous and Unconventional Cluster Architectures and Applications (HUCAA), Lyon, France, October 2013, pp. 885-892 DOI: 10.1109/ICPP.2013.105

[**Exascale10 white paper**] http://www.xyratex.com/performance-very-large-scale

[**E10 website**] www.eiow.org

[**ROMIO Implementation**] http://www.mcs.anl.gov/research/projects/romio/

[**AveryALLN06**] Ching, Avery; Choudhary, A.; Wei-Keng Liao; Ward, L.; Pundit, N., "Evaluating I/O characteristics and methods for storing structured scientific data," *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International* , vol., no., pp.10 pp.,, 25-29 April 2006.

[**LiaoA08**] Wei-keng Liao and Alok Choudhary. 2008. Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols. In*Proceedings of the 2008 ACM/IEEE conference on Supercomputing* (SC '08). IEEE Press, Piscataway, NJ, USA, , Article 3 , 12 pages

[**ThakurEtAl99**] Thakur, R.; Gropp, W.; Lusk, E., "Data sieving and collective I/O in ROMIO," *Frontiers of Massively Parallel Computation, 1999. Frontiers '99. The Seventh Symposium on the* , vol., no., pp.182,189, 21-25 Feb 1999

[**LuCTZ12**] Yin Lu; Yong Chen; Thakur, R.; Yu Zhuang, "Abstract: Memory-Conscious Collective I/O for Extreme-Scale HPC Systems," *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:* , vol., no., pp.1360,1361, 10-16 Nov. 2012.

[**YuJV08**] Weikuan Yu and Jeffrey Vetter. 2008. ParColl: Partitioned Collective I/O on the Cray XT. In *Proceedings of the 2008 37th International Conference on Parallel Processing* (ICPP '08).

[**HeSSYT11**] Jun He, Huaiming Song, Xian-He Sun, Yanlong Yin, Rajeev Thakur, Pattern-aware file reorganisation in MPI-IO, Proceedings of the sixth workshop on Parallel Data Storage, November 13-13, 2011, Seattle, Washington, USA

[**ChenSTRG11**] Yong Chen; Xian-He Sun; Thakur, R.; Roth, P.C.; Gropp, W.D., "LACIO: A New Collective I/O Strategy for Parallel I/O Systems," *Parallel & Distributed Processing*

Symposium (IPDPS), 2011 IEEE International , vol., no., pp.794,804, 16-20 May 2011

[**ZhangJD09**] Xuechen Zhang; Jiang, S.; Davis, K., "Making resonance a common case: A high-performance implementation of collective I/O on parallel file systems," *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, vol., no., pp.1,12, 23-29 May

List of Acronyms and Abbreviations

A

API: Application Programming Interface

B

BADW-LRZ: Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften. Computing Centre, Garching, Germany

BN: Booster Node (functional entity)

BoP: Board of Partners for the DEEP-ER project

BSC: Barcelona Supercomputing Centre, Spain

BSCW: Basic Support for Cooperative Work, Software package developed by the Fraunhofer Society used to create a collaborative workspace for collaboration over the web

C

CN: Cluster Node (functional entity)

D

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

DEEP: Dynamical Exascale Entry Platform

DEEP-ER: DEEP Extended Reach: this project

DEEP-ER Network: high performance network connecting the DEEP-ER BN, CN and NAM; to be selected off the shelf at the start of DEEP-ER

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

DEEP System: The prototype machine based on the DEEP Architecture developed and installed by the DEEP project

E

EC: European Commission

EC-GA: EC-Grant Agreement

EU: European Union

Exaflop: 1018 Floating point operations per second

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

GPU: Graphics Processing Unit

H

HPC: High Performance Computing

I

ICT: Information and Communication Technologies

IEEE: Institute of Electrical and Electronics Engineers

Intel: Intel Germany GmbH Feldkirchen,

J

JUELICH: Forschungszentrum Jülich GmbH, Jülich, Germany

K

KNL: Knights Landing, second generation of Intel® Xeon Phi™

KULeuven: Katholieke Universiteit Leuven, Belgium

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

O

OmpSs: BSC's Superscalar (Ss) for OpenMP

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

P

PMT: Project Management Team of the DEEP-ER project

Q

R

S

SC: International Conference for High Performance Computing, Networking, Storage, and Analysis, organised in the USA by the Association for Computing Machinery (ACM) and the IEEE Computer Society

T

U

UREG: University of Regensburg, Germany

V

W

WP: Work Package

X

Y

Z