SEVENTH FRAMEWORK PROGRAMME

FP7-ICT-2013-10

DEEP-ER

DEEP Extended Reach

Grant Agreement Number: 610476

D4.3

Definition of test cases and patterns

Approved

Version: 2.0

Author(s): C. Manzano (JUELICH)

Contributor(s): J. Kreutz (JUELICH), O. Buechner (JUELICH), M. Cintra (Intel), A. Jakobs (JUELICH), K. Thust (JUELICH), D. Alvarez (JUELICH), R. Leger (Inria), S. Solbrig (UREG), A. Johnson (KULeuven), M. Hanzich (BSC), M. Petschow (ASTRON), J. Romein (ASTRON), F. Geier (ParTec), R. Krotz (ParTec)

Date: 09.12.2015

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: D4.3</td></tr>
<tr><td colspan="2">Deliverable Nature: Report</td></tr>
<tr><td>Deliverable Level:
PU *</td><td>Contractual Date of Delivery:
31 / December / 2014</td></tr>
<tr><td>Actual Date of Delivery:
17 / December / 2014</td></tr>
<tr><td colspan="3">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: Definition of test cases and patterns</td></tr>
<tr><td colspan="2">ID: D4.3</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_D4.3_Definition_of_test_cases_and_patterns_v2.0-ECapproved.docx</td></tr>
<tr><td rowspan="4">Authorship</td><td>Written by:</td><td>C. Manzano (JUELICH)</td></tr>
<tr><td>Contributors:</td><td>J. Kreutz (JUELICH), O. Buechner (JUELICH), M. Cintra (Intel), A. Jakobs (JUELICH), K. Thust (JUELICH), D. Alvarez (JUELICH), R. Leger (Inria), S. Solbrig (UREG), A. Johnson (KULeuven), M. Hanzich (BSC), M. Petschow (ASTRON), J. Romein (ASTRON), F. Geier (ParTec), R. Krotz (ParTec)</td></tr>
<tr><td>Reviewed by:</td><td>A. Emerson (CINECA), 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	17/December/2014	Final version	EC submission
2.0	09/December/2015	Approved	EC approved

Document Keywords

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

Table of Contents

List of Figures

List of Tables

Executive Summary

This deliverable describes a set of synthetic and application-based benchmarks, which will be used to evaluate the DEEP-ER I/O Architecture with respect to I/O performance, scalability and integration effort.

For the description of the application based benchmarks, the I/O requirements defined in Deliverable 4.1 [D4.1], together with detailed input from the application developers (WP6), have been taken into account. Furthermore, the checkpointing strategy has been considered (WP5).

Besides the specification of the benchmarks, test scenarios and the methodology are described. The actual results of the I/O performance measurements and comparison with production platforms will be documented at a later stage in Deliverable 4.5.

1 Introduction

As established in the Description of Work (DoW), Task 4.5 pursues the following objectives:

- Specification of a set of synthetic I/O benchmarks for the component-based evaluation, a set of DEEP-ER application benchmarks, benchmark parameter sets, and test scenarios;
- Integration of the benchmark codes into the JUBE Benchmarking Environment;
- Adaptation of the synthetic benchmarks to use the new I/O layer and APIs;
- Collaborate with Task 6.1 to support the application developers (with performance indicators);
- Perform the actual measurements, implementing automatic test procedures;
- Comparison with I/O performance results of JUELICH production systems;
- Analyse, compare and document the results of the measurements.

This document describes a set of I/O benchmarks and its integration in the JUBE platform, fulfilling the first objectives of Task 4.5 and establishing a series of guidelines, which will allow the remainder of the proposed goals to be completed successfully.

The document is structured in 7 main sections: Section 1 comprises the present introduction. Section 2 describes the platforms which will be used for the testing and analysis, giving a technical overview. Section 3 specifies the benchmarking strategy, which will be followed. Section 4 introduces the JUBE Benchmarking Environment and explains how the tool will help to achieve the Task 4.5 objectives. Section 5 explains the selected synthetic benchmarks and defines a series of specific test cases to be used in the environments of Section 2. Section 6 presents the set of applications chosen for benchmarking, gives a general outlook of each focusing on the I/O part and, like Section 5, defines the corresponding series of test cases. Finally, Section 7 provides a review of the work done and gives a description of the future steps.

The work presented in this document will enable WP4 to perform the actual benchmarking measurements, analyse them and extract conclusions. Application developers will also benefit from the description done here, since having a better understanding of the performance indicators will guide them to make use of the DEEP-ER I/O Architecture more efficiently. The same applies for the resiliency software work package, which will use this deliverable as a basis for executing checkpointing performance measurements within the Task 5.5.

2 Test environments

Within this section we proceed to explain the test scenarios and methodology that will be used for the analysis and evaluation of the DEEP-ER I/O Architecture.

2.1 DEEP Cluster

The DEEP Cluster was installed primarily as testing platform for the DEEP project. Equipped with Sandy Bridge multicore processors and QDR InfiniBand network, the DEEP Cluster is highly flexible and capable of hosting the programming and compiling infrastructure that is needed to make use of the novel Cluster-Booster Architecture. While the Booster part is being designed and assembled within the DEEP project and is not yet ready, the Cluster can already be used for tests and development of the software stack. The Cluster is ideally suited for testing the I/O performance of DEEP-ER applications using the Fraunhofer parallel file system (BeeGFS), available in the compute nodes under */work*.

During the DEEP-ER project the DEEP Concept will be extended by including new I/O elements like, for instance, NAM and NVM. A new Cluster-Booster hardware platform – the DEEP-ER Prototype – will be built to demonstrate the use of these new memory technologies. Further improvements with respect to the DEEP System will be the update to new generation processors on the Cluster and Booster side of the machine and the simplification of the network configuration. On the software side, new functionality will be added to the BeeGFS parallel file system, the SIONlib library will be adapted to the new I/O components and the middleware E10 will be integrated in the system. Once the DEEP-ER Prototype is available, it will be used for further testing within the Task 4.5.

2.1.1 System architecture of the DEEP Cluster

General description

The DEEP Cluster is a liquid-cooled, energy-efficient, scalable HPC system from the Aurora product line produced by Eurotech. Highlights of the Aurora system used for the DEEP Cluster compute nodes include the following:

- Nodes: 128x Intel® Xeon® E5 series CPUs (Sandy Bridge).
  - 2 sockets/node.
  - 340 GFLOPs/node.
  - 32GB RAM / node.
- Main memory: 4 TB (aggregate).
- Overall peak performance: 45 TFlops.
- Network Architecture: 60Gbps 3D torus & QDR InfiniBand.
- Compatibility: Full x86 compatibility – Intel Cluster Ready.
- Operating system: CentOS 6.3.

For the management and server nodes an external server rack has been assembled with two login servers (frontends), six file system servers, a storage system and a Gigabit Ethernet switch to set up the administration network. This rack also contains the higher-level high speed network switches (8 FDR InfiniBand switches).

Storage nodes description

The storage part of the JUELICH DEEP Cluster consists of 6 Dell PowerEdge storage servers, each one connected to one JBOD with 45 x 2 TB disks via a SAS switch. The BeeGFS parallel file system is installed in the storage nodes. The characteristics and configuration of the DEEP storage system are described in Table 1:

DEEP Storage servers:		
6x DELL PowerEdge R520 storage servers (deep-fs01 – deep-fs06), each with:		
	2x Intel Xeon Quad-Core CPUs (ES-2403)	
	32 GB memory	
	1x H310 RAID controller with:	2x 300 GB SAS disk (RAID1)
		Mellanox ConnectX-3 HCA
	1x SAS port	
SAS switch:		
1x LSI 6140 SAS switch connecting the storage servers with the JBOD:		
	6x SAS cables to connect each storage server to the SAS switch	
	2x SAS cables to connect the SAS switch to the JBOD	
JBOD:		
1x SGI JBOD 2245 with:		
	2x 6 Gb/s SAS 4x wide ports	
	45x 2TB disks:	4x disks for deep-fs01 and deep-fs02
		20x disks for deep-fs03 and deep-fs04,
		20x disks for deep-fs05 and deep-fs06
		1x disk as spare one
Disks configuration:		
RAID sets on each server:		
	deep-fs01	RAID1: 2x mirrored disks
	deep-fs02	RAID1: 2x mirrored disks
	deep-fs03	RAID6: 10x disks
	deep-fs04	RAID6: 10x disks
	deep-fs05	RAID6: 10x disks
	deep-fs06	RAID6: 10x disks

Table 1: Characteristics and configuration of DEEP storage system

Going more into detail about the characteristics of the system, there are 2 SAS cables from the JBOD to the switch, 1 cable providing connection to the 24 front side disks and 1 cable providing connection to the 21 rear side disks. Each server in a group[1] sees also the disk space of the other server in the group and can make a fail over in case of the absence of the partnering server. The separation of the disks is done by zoning on the SAS switch. On each

[1] There are 3 groups in the system, the one formed by deep-fs01 and deep-fs02, the one formed by deep-fs03 and deep-fs04 and the one formed by deep-fs05 and deep-fs06.

RAID set a standard ext4 file system was created. Because the write performance of the RAID6 sets was poor, the stripe_cache_size on the md devices of the servers deep-fs03 to deep-fs06 has been increased from 256 to 16384.

Regarding BeeGFS, the current version is 2014.01.r8, installed in July 2014. The BeeGFS services[2] are running in the DEEP Cluster according to the distribution shown in Table 2.

DEEP Cluster node	DEEP Cluster node description	BeeGFS roles	BeeGFS services
deep-fs01	Storage node	Management, Metadata, Administration, Monitoring	fhgfs-mgmtd, fhgfs-meta, fhgfs-admon
deep-fs02	Storage node	Metadata	fhgfs-meta
deep-fs0[3-6]	Storage nodes	Storage	fhgfs-storage
deep[1-128]	Compute nodes	Client, Helper	fhgfs-client, fhgfs-helperd
deepm	Administration (master) node	Client, Helper	fhgfs-client, fhgfs-helperd
deepl	Login node	Client, Helper	fhgfs-client, fhgfs-helperd

Table 2: BeeGFS services in the DEEP Cluster

The BeeGFS parallel file system is mounted in the DEEP Cluster compute, administration and login nodes under */work*. Figure 1 illustrates the servers and storage concept for the DEEP Cluster.

[2] See the BeeGFS online documentation [BeeGFS website] for details about the services.

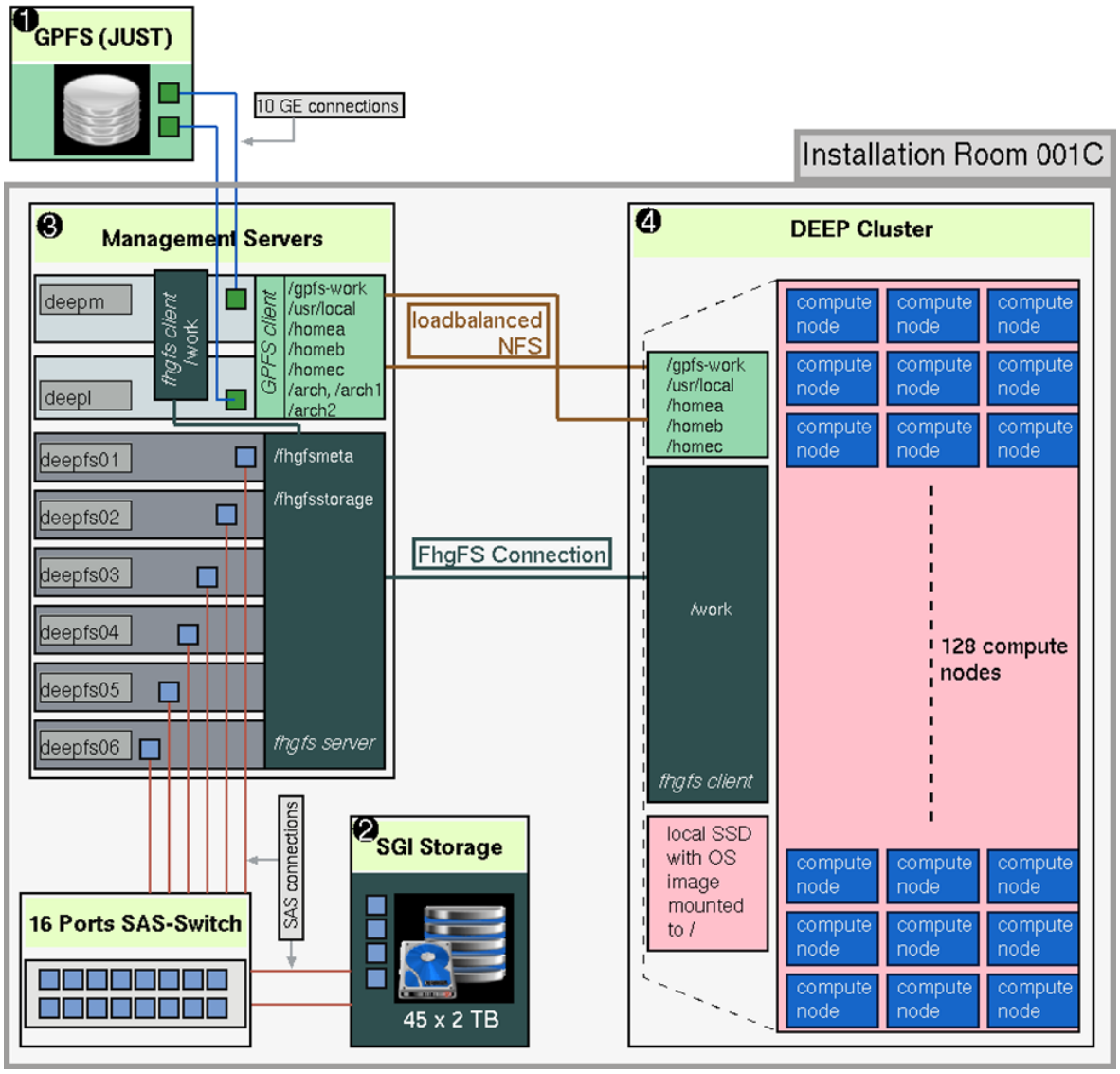


Figure 1: DEEP Cluster and storage concept (note that FhgFS is the old name of BeeGFS)

2.2 NVM evaluator

One key component of the DEEP-ER concept is the use of node-local storage to complement a traditional parallel file system attached to I/O nodes. As a first step in evaluating this architectural design, we have obtained 2 units of a state-of-the-art PCIe-based SSD, which uses the recent NVMe industry standard interface. As part of WP4 we have designed a test methodology for these early SSD samples.

The purpose of the NVM test setup is threefold:

1. to assess the performance of the NVMe SSD in order to provide performance parameters for the DEEP-ER application developers;
2. to assess the potential impact of the performance gains from NVMe over SATA alternatives for a subset of (unmodified) DEEP-ER applications; and
3. to assess how I/O system software (e.g. file system) can be tuned to exploit the performance characteristics of the NVMe SSD.

2.2.1 System architecture of the NVM evaluator

Our initial NVM testing will use P3700, the latest generation NVMe SSD from Intel. Two of these devices were installed in two PCIe slots of the “knc2” machine, which is one of the two Xeon-Phi-equipped test machines at JSC. The “knc2” machine is currently configured with 2 Xeon Phis and 2 P3700s. For future tests with larger applications, one of the P3700s can be moved to “knc1”. Both machines run CentOS 6.4. A diagram depicting this set-up is given in Figure 2.

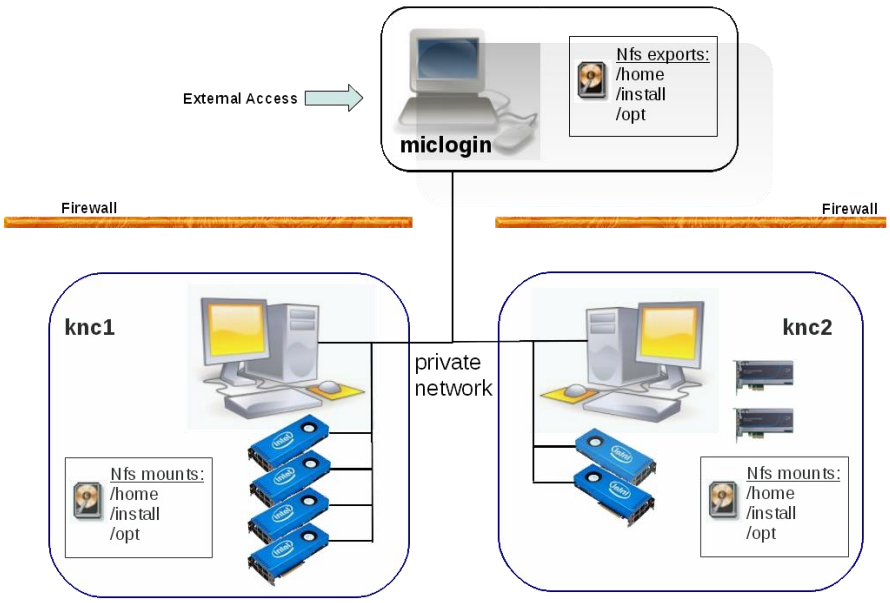


Figure 2: Test machines at JUELICH where the NVMes are mounted

3 Test methodology

Testing on the DEEP Cluster and NVM evaluator will be done at three levels:

- **With synthetic benchmarks.** For the DEEP Cluster, I/O benchmarks like IOR and partest, as well as a metadata benchmarks like mdtest, will be used on a regular basis, together with checks for testing the consistency of the results (LinkTest). For the NVM evaluator, standard storage I/O micro-benchmarks commonly used to measure disk and file system I/O performance will be used. Examples of such micro-benchmarks include IOzone [IOzone website] and FIO [FIO website]. The goal of this testing is to characterise the baseline performance of the DEEP I/O layers, as well as of the new NVMe SSD and to identify the I/O strategies required to achieve peak performance.
- **With mock-ups of some of the DEEP-ER applications.** For this testing, we will use synthetic applications that behave similarly to some key DEEP-ER applications in terms of I/O. More specifically, the exact computation is replaced by spin-loops and only the memory access and I/O access are modelled in detail in terms of the read/write characteristics. In addition to simplifying the experimentation process, this approach also allows the parameterization of the I/O behaviour to model a larger variety of inputs than it is possible with the real applications and their input sets. The goal of this testing is to characterize the expected performance of the mock-ups in the DEEP-ER I/O Architecture and how they benefit from the improved performance characteristics of the NVMe SSD.
- **With real DEEP-ER applications.** For this testing, we will use some of the actual DEEP-ER applications. To run on the NVM evaluator, scaled down versions will be used, more specifically, versions of the applications especially set up to run on one or two nodes. We expect again to characterize the performance of the DEEP-ER I/O layers and to get a better understanding of how the applications behave with the NVMe SSDs. The aim in the end is to obtain more accurate results about the impact that the different DEEP-ER I/O strategies and the special NVM devices have on the applications.

The corresponding testing tools will be integrated in the JUBE benchmarking environment (see Section 4).

4 JUBE

Benchmarking activities should be automated to guarantee fair and reproducible comparisons between different platforms or environments. Automated benchmarking also simplifies the managing of different combinations of parameters in a large parameter space and reduces the risk of introducing errors in the process.

The JUBE Benchmarking Environment enables a systematic benchmarking and allows adapting custom workflows to new architectures. In DEEP-ER, JUBE will be used to automatic test and analyse the DEEP-ER I/O Architecture, its usability, performance and overheads.

4.1 Description JUBE

4.1.1 File format

The input file-format for JUBE uses the well-spread markup language XML. JUBE offers schema-validation files, which help preventing syntactic errors already in editors that support schema validation before an input file is actually parsed by JUBE. The structure is designed to reduce the amount of text duplication while retaining enough verbosity to debug possible problems.

4.1.2 XML structure

In this section we give an overview of a simple JUBE XML file.

JUBE uses the `jube` tag as its root. Inside there is a `benchmark` element which includes the actual benchmark description. See for instance the following example of JUBE XML file:

```
<jube>
  <benchmark name="parameterspace" outpath="bench_run">
    <!-- Configuration -->
    <parameterset name="param_set">
     <!-- Create a parameterspace out of two template parameters -->
     <parameter name="compile_opt">-O1,-O3</parameter>
     <parameter name="nodes" type="int">1,2,4</parameter>
    </parameterset>

    <!-- Operation -->
    <step name="run_something">
     <use>param_set</use> <!-- use parameterset "param_set" -->
     <do>echo "$compile_opt $nodes"</do> <!-- shell command -->
    </step>
  </benchmark>
</jube>
```

One of the key functionalities of JUBE is the separation of data and commands in a way that allows commands with different input data to be executed in a similar manner. This concept is most obviously reflected in the `parameterset` elements inside the `benchmark` tag. By adding multiple possibilities to a `parameter` inside a `parameterset`, JUBE will automatically use each possible combination. A possible example is shown in the previous code listing. It runs an application with different compile options (e.g. "O1" and "O3") and

using different numbers of nodes (e.g. 1, 2 and 4). JUBE would then run this benchmark for all combinations of compile options and number of nodes which are, for this example, the six combinations ("O1", 1), ("O3", 1), ("O1", 2), ("O3", 2), ("O1", 4) and ("O3", 4).

The actual execution is described in the `step` element which is also part of a `benchmark`. It defines a single block of execution, e.g. a compilation with the relevant parameters. Furthermore, it describes the dependencies to other steps, e.g. an execution needs the compilation to be performed first of all.

As a final task the performed benchmarks should be analysed. This process is split into the `analyzer` part and the `result` part. An example is shown in the following code listing:

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
  <benchmark name="result_creation" outpath="bench_run">
    <!-- Configuration -->
    <parameterset name="param_set">
      <!-- Create a parameterspace with one template parameter -->
      <parameter name="number" type="int">1,2,4</parameter>
    </parameterset>

    <!-- Regex pattern -->
    <patternset name="pattern">
    <pattern name="number_pat" type="int">Number: $jube_pat_int</pattern>
    </patternset>

    <step name="write_number">
      <use>param_set</use>
      <do>echo "Number: $number"</do>
    </step>

    <!-- Analyse -->
    <analyzer name="analyse">
      <use>pattern</use> <!-- use existing patternset -->
      <analyse step="write_number">
        <file>stdout</file> <!-- file which should be scanned -->
      </analyse>
    </analyzer>

    <!-- Create result table -->
    <result>
      <use>analyse</use> <!-- use existing analyzer -->
      <table name="result" style="pretty" sort="number">
        <column>number</column>
        <column>number_pat</column>
      </table>
    </result>
  </benchmark>
</jube>
```

The `analyzer` element describes the data to be extracted and how to extract it, which in this case uses the `patternset` “pattern” on standard output. Using this data `result` defines how the extracted data should be shown to the user. For the above code listing this means

printing a human-readable ASCII formatted table with the columns containing the original number as the first column and the number which is extracted from the output of the according step as the second column. The `patternset` makes use of the pre-defined regular expression pattern `$jube_pat_int` and the result is just the same as the original number for this example.

Since this overview does not try to replace the JUBE tutorial only some basic features are described. These should give the reader a rough idea of the design principles and general usage of JUBE.

4.2 Further documentation JUBE

A complete and detailed documentation including tutorials for getting started, a description of the more advanced features and a reference describing all available functionality can be found in the online documentation [JUBE website].

4.3 Integration JUBE

JUBE is designed to be non-blocking for operations like sending jobs to a queuing system. This is often preferable since waiting for batch jobs to finish can be quite time-consuming and should not interrupt the usual work flow. However, this behaviour is undesirable in DEEP-ER, where JUBE is to be integrated in a time-based job scheduler and where I/O benchmarks should be prevented from being run in parallel. Since I/O uses shared resources, running different benchmarks in parallel would cause interference and hence render the results useless. Therefore, JUBE includes the jube-autorun shell script which eases the task of running JUBE synchronously. This script will be used to perform continuous benchmarks in special reservations. The frequencies will most likely be daily or weekly, depending on the duration of the benchmark and the need for changes in the codes.

Two kinds of benchmarking will be performed. The first with mostly constant benchmarks, which will be used to measure changes in the system performance, for example when driver changes are applied or hardware is substituted. The second with application-benchmarks to measure the change in application performance over time, for example due to integration of new I/O strategies or modifications in the I/O libraries used by the application.

5 Synthetic benchmarks

The following set of synthetic benchmarks will be integrated in the JUBE benchmarking environment to analyse the DEEP-ER I/O components: IOR, mdtest, partest and LinkTest.

- IOR: is a standard I/O benchmark and will be used to set a baseline *read* and *write* performance in the test environments. In some cases, like in the NMV evaluator, another similar standard I/O benchmark might be used such as IOzone or FIO, which might be more suitable for the platform (for instance by performing single client I/O throughput experiments). Due to the similarities between the aforementioned benchmarks and IOR, we have chosen to only include the description of this last one in this deliverable. A complete description of IOzone and FIO can be found online [IOzone website][FIO website].
- mdtest: will be used to get a baseline performance for metadata operations like open, stat and close on files and directories.
- partest: is part of the SIONlib installation and will be used to test different I/O strategies in a similar way to IOR.
- LinkTest: is not, strictly speaking, an I/O analysis tool but a parallel ping pong test between all possible MPI connections. The LinkTest will allow us to detect possible anomalies in the functioning of the DEEP-ER Interconnect, which might interfere with the I/O performance measured by the I/O benchmarks.

5.1 IOR

The IOR benchmark [IOR website] can be used for testing the performance of parallel file systems in HPC. The software uses MPI for process synchronization.

IOR provides the capability to test aggregate I/O rates via several typical middleware libraries including MPI collective I/O calls and HDF5 library calls, in addition to POSIX I/O calls. Input arguments allow, among others, to determine the variance of the overall I/O size, individual transfer size, file access mode (single shared file, one file per client), and whether the data are sequentially or randomly accessed. These and other inputs can be used to mimic the I/O patterns of real HPC applications.

5.1.1 IOR test cases for DEEP-ER

IOR will be used to measure the baseline I/O-performance of the BeeGFS global parallel file system, in which the infrastructure for I/O operations in the DEEP Cluster is based. Performing the measurement on a regular basis will help us to assess the consequences of changes in the cluster, as well as in the applications themselves. This task will be done with help of the JUBE benchmarking environment (see Section 4). In order to obtain this performance reference, different test cases with different parameter-sets have been identified.

Test runs have shown that the parameters[3] displayed in Table 3 are relevant for the tests, whereas the remaining IOR parameters do not need to be modified (i.e. are set to default).

Parameter	**Values**

[3] For a description of the whole set of parameters and its default values please check Annex A.

Parameter	Values
api	POSIX / MPIIO
reorderTasksConstant	1 (true)
taskPerNodeOffset	1
filePerProc	1 (one file per task) / 0 (shared file)
segmentCount	1
blockSize	4 GiB
transferSize	8 MiB, 128MiB

Table 3: IOR relevant parameters

For the tests on the DEEP Cluster, we assume that transferring more than 4 x 32 GB = 128 GB per session should avoid any write-caching effect as this is the total RAM size of the storage servers.[4] As for the test cases, we identified a small test case to run in 8 nodes, a medium one to run in 32 nodes, and a big one to run in 64 nodes:

	Small	Medium	Big
Number of tasks	128	128	256
Nodes	8	32	64
Tasks per node	16	4	4
API	MPIIO/POSIX	MPIIO	MPIIO
Files per process	0/1	0	0
Reorder tasks constant	1	1	1
Transfer size	8 MiB	128 MiB	128 MiB
Block size	4 GiB	4 GiB	4 GiB
Aggregate size	512 GiB	512 GiB	1024 GiB

Table 4: IOR test cases

5.2 mdtest

The mdtest benchmark [mdtest website] will be used to investigate the speed at which metadata operations are performed using the BeeGFS file system. mdtest uses MPI to coordinate the operations and collect the results.

The benchmark allows selecting the number of files per process which are to be created, the depth of the directory structure, the number of MPI threads to perform the test among others.

[4] We have 4 storage servers (deep-fs0[3-6]) with 32 GB RAM each. The 2 metadata servers (deep-fs0[1-2]) don't need to be taken into account for avoiding the write-caching effect. See Section 2.1.1 for more details about the system architecture.

5.2.1 mdtest test cases for DEEP-ER

As most of the DEEP-ER applications are using the POSIX I/O interface to perform task local I/O, it might be interesting to check how the file system performs when applications scale up and significantly increase the number of processes, putting a huge burden on the file system metadata service. This can be nicely simulated making use of this benchmark test.

For the tests, taking into account the I/O requirements described in Deliverable D4.1 [D4.1], we make the following assumptions about the intended use of the parallel file system:

- A large number of temporary files are created and later deleted, i.e. the performances of file create and delete operations are critical.
- The directory structure is kept flat, i.e. performance of directory create and delete operations is not relevant.
- The performance of any other operation changing the file system metadata like link creation or deletion, file status get and set operations etc. is not critical.

Table 5 shows, from the whole set of mdtest parameters[5], those relevant for the DEEP-ER tests.

Parameter	**Description**
-n <items_per_task_per_tree>	every task will create/stat/remove # files/dirs per tree
-F	perform test on files only (no directories).
-C	only create files/dirs
-z <depth>	depth of hierarchical directory structure

Table 5: mdtest relevant parameters

With these parameters, a series of test cases have been identified (see Table 6).

Total number of files created	65536	65536	65536	65536	65536	65536
Number of files per task	16384	4096	1024	4096	1024	256
MPI tasks	4	16	64	16	64	256
Nodes	2	8	32	2	8	32
Tasks per node	2	2	2	8	8	8
Directory depth	1|2	1|2	1|2	1|2	1|2	1|2
File creation only/File delete	-C|-	-C|-	-C|-	-C|-	-C|-	-C|-

Table 6: mdtest test cases

5.3 partest (SIONlib)

As part of the SIONlib installation, partest is a tool for benchmarking parallel I/O. It offers options to test specific parameters for different I/O strategies such as MPI-IO, task-local files and SIONlib.

Configuring benchmarks with partest is done via command line arguments. All parameters have default values, so only those which differ from the default settings need to be set. In

[5] See Annex B for the whole set of parameters.

order to obtain a complete list of all available commands and their default values the "--help" argument can be used. In the results output partest prints a list of all variables used for the benchmark, including the default ones. These are then followed by the time measurements for the read and write operations.

5.3.1 partest test cases for DEEP-ER

For the DEEP-ER Platform the 0 and 3[6] test types will be the most suitable and both will be used. The type 0 test is a SIONlib standard test. It uses collective calls for opening and closing files but all the write or read calls in between do not need any further communication. For small systems like the DEEP System and the later DEEP-ER Prototype, writing task-local files (test type 3) is likely to be the reference value for maximum performance since the metadata server is unlikely to be the bottleneck when performing I/O. The metadata performance should be benchmarked using another tool like mdtest (see Section 5.2).

The buffer size will be chosen similar to the file system block size, which is 512 kB on the DEEP Cluster.

The parameter "localsize" will be set so that a usage of roughly a third of the available RAM is used if summed up over all tasks per node. This reduces caching effects of the file system.

For more information the option "verbose" might be used, which shows statistics for all individual tasks. This is only useful for small setups (< 32k tasks).

In order to test the actual write performance and reduce the effect of caching by the system, tests with and without the "posix" option will be used. Without this option the standard ANSI "fwrite" would be applied, which buffers data internally, typically with a buffer size of one file system block.

For checkpointing, the benchmark setup should use the collective interface ("collread" and "collwrite"). In addition to this option the environment variable "SION_COLLSIZE" needs to be set. In contrast to the standard SIONlib behaviour, which only uses collective calls for opening and closing files, in collective mode also the read and write calls are collective. Therefore, nodes are grouped by the caching layer to which they have access, and all but one node in a group do send their data to the group collector, which then performs the actual write operation.

Another way of reducing caching effects is to use "taskoffset", which cyclically shifts the task-mapping between read and write phases. In this way each task, instead of reading its own data, reads the data belonging to another task. This is only useful for tests not using the caching layer, since it contradicts the storage concept, in which data is only available for some local group.

5.4 LinkTest

The LinkTest program is a parallel ping pong test between all possible MPI connections of a machine. The output of this program is a full communication matrix which shows the bandwidth and message latency between each processor-pair, together with a report including the minimum bandwidth. The program can also be used for communication stress

[6] See Annex C with the partest parameter description.

tests by running it repeatedly for a specific duration. The LinkTest software has been developed by the Juelich Supercomputing Centre and is freely available [LinkTest website].

The LinkTest runs for *n* processors in *n* steps, where in each step *n/2* pairs of processors perform the MPI ping pong test[7]. Assignment of MPI tasks is performed once at the beginning of the program, according to the underlying hardware and operating system (e.g. on Linux the hostname and rank/core are used for identification). The selection of the pairs is random but it is guaranteed that, after running all steps, all possible pairs have been covered. A top *N* analysis of the results is done and the poorest *N* connections are identified, where *N* can be specified by the user. SIONlib is used for writing an output file containing the results of the whole communication matrix. An analysing tool included with the software can be used to generate pattern files, a list of bad links, and a graphical output illustrating the communication matrix and providing a histogram about timings and bandwidths ranges.

Several options and parameters can be passed to the program to target different aspects of the communication to be measured. The most important ones are the size of the messages being sent, the number of iterations, and whether to run in serialized or parallel mode. See the homepage [LinkTest website] for a detailed description on how to use the MPI LinkTest program.

5.4.1 LinkTest test cases for DEEP-ER

The LinkTest program is being used to check the status of the interconnect within the DEEP-ER system. This includes the bandwidth and latency of single connections, as well as the overall capability of the network in terms of possible bottlenecks or congestion. To do so, the program has to be run with different setups:

1. Latency check: This test is executed in serial mode for all possible connections (to include inter-node measurements, two tasks per node are being used). For latency tests the message size should be very small and is set to 1 byte. To get meaningful values, 10 iterations are sufficient for the ping pong execution.
2. Bandwidth check: Just like the latency check, the bandwidth configuration is also started in serial mode with two tasks per node, but it uses large message sizes to measure the maximum bandwidth achieved for single connections. A messages size of 512 KB is applied. Test runs have shown that 50 to 100 iterations have to be performed to reduce the deviance of the measured values. Hence, the number of iterations is set to 75.
3. Check for bottlenecks: To test the network topology for possible bottlenecks a large amount of parallel communication must be set up. Hence this test is executed in parallel mode using one task per CPU core. The number of iterations and the message size from the bandwidth test can be reused.

To analyse the results of these benchmarks and their distribution, a graphical overview will be created using the analysis tool provided by the MPI LinkTest software. Problems of single connections, routing or hardware problems (e.g. faulty switches) –all of them leading to special patterns in the communication matrix and to limited network capabilities when doing

[7] The sender sends a message with a certain data size to the receiver and waits for a reply from the receiver. The receiver receives the message from the sender and sends back a reply with the same data size.

massive parallel communication, which then result in substantial drawback of bandwidth–, will be obvious from the generated output. Figure 3 shows the graphical output for an exemplary communication matrix for a serially executed LinkTest run using 4 nodes and 16 tasks per node on the DEEP Cluster.

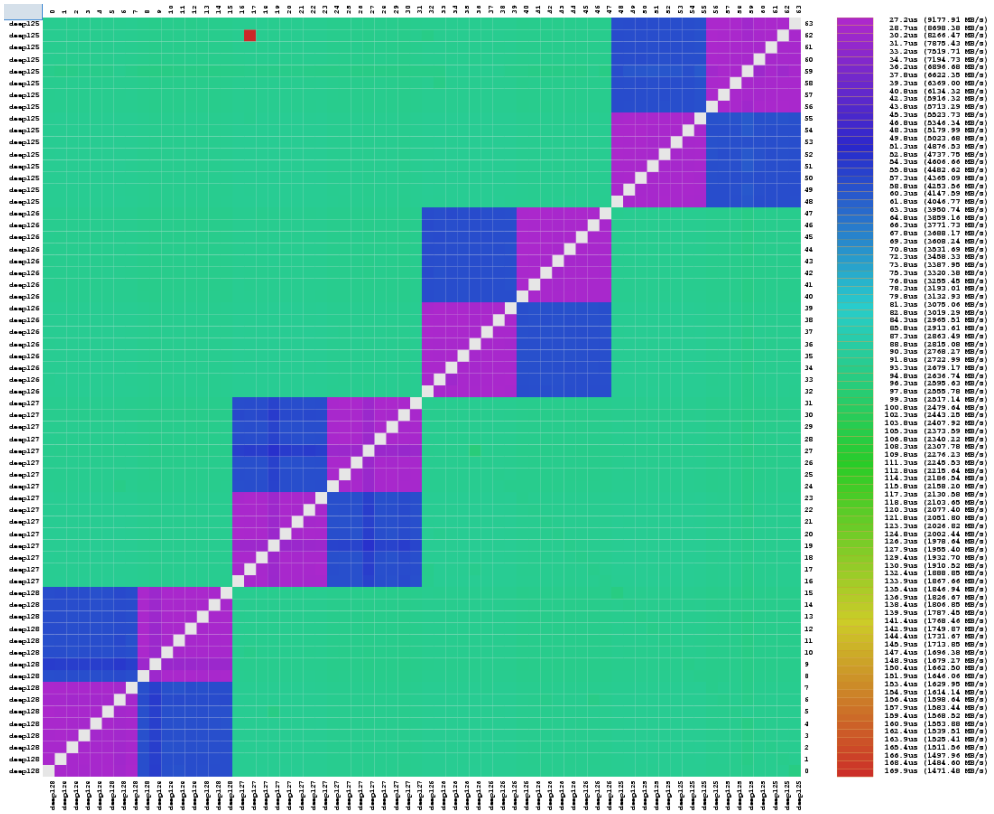
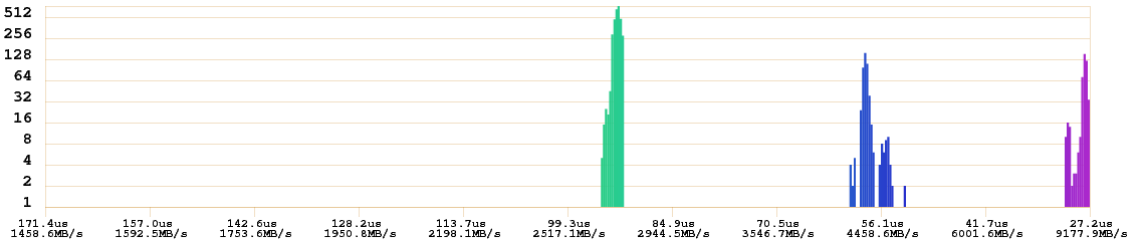


Figure 3: Graphical output of the LinkTest communication matrix on the DEEP Cluster

The legend on the right hand side of Figure 3 illustrates how the colours are mapped to the communication time (and bandwidth). The smallest size coloured squares within the communication matrix illustrate the performance of a single connection. Connections where both ends are identical (using the same MPI task) are located on the diagonal, which is coloured white since the communication time for these connections is assumed to be zero. As one can see, the colours reflect very well the positions of the tasks residing on the same host (4 squares bisected by the diagonal) and even the thread pinning on the CPUs (blue and purple colour within the 4 host squares), which turns out to be different depending on the host. The inter-node communication looks very consistent (green colour) implying that a fat tree network is available providing the same conditions (e.g. number of hops) for all connections between all the nodes. The little red square in the top left range of the matrix indicates that there is one connection that might have a problem and should be further investigated, since the bandwidth achieved is much lower compared to the remaining inter-node connections.

Figure 4 shows a histogram for the above communication matrix, also provided by the graphical output of the MPI LinkTest software. The three types of connection pairs (same CPU, same node and different CPU, different nodes) show up as three peaks within the histogram. Additional information on the test run, e.g. number of iterations and message size, is provided in table format.



length_of_message:	262144 bytes (256.00 KBytes)	number_of_tasks:	64
number_of_messages:	50	Execution order:	Serial
Alltoall:	0	Mixing PE rank:	No
Min Value:	27.2us (9177.91 MB/s)		
Max Value:	171.4us (1458.58 MB/s)		
Avg Value:	80.9us (3088.54 MB/s)		

Figure 4: Histogram of the LinkTest communication matrix on the DEEP Cluster

6 Application-based benchmarks

In order to characterize the DEEP-ER I/O Architecture, a series of application-based benchmark tests will be run, in addition to the synthetic ones described above, within Tasks 4.5 and 5.5. Making use of different test cases, we expect to detect weak and strong scaling, as well as to probe the suitability of the DEEP-ER I/O software.

The applications that will be used for the benchmark tests are iPiC3D, MAXW-DGTD, Radio Astronomy, Full Waveform Inversion and Chroma. The DEEP-ER applications[8] TurboRVB and SeisSol won't be used for considering their I/O behaviour already represented by the patterns of the selected applications.[9]

The iPiC3D and Chroma applications can make use of MPI-IO, the former one using as well the HDF5 high level I/O library. MAXW-DGTD has a task-local I/O schema and was described at first as non-I/O intensive. However, a checkpointing strategy has been added to the code which is likely to change this. The application from the BSC partner (Full Waveform Inversion) is I/O bound, where the I/O time for one step is one order of magnitude longer than it takes to compute it, so it makes a good candidate for exploring the scalability of the cluster. Finally, the Radio Astronomy application receives UDP station data and buffers it. After scattering blocks of data across the Booster Nodes and the processing part, the output data are buffered and written to disk. Within this deliverable we are interested in this last imaging part.

In some cases, like for instance in the case of the Full Waveform Inversion application, we will be using mock-ups of the applications, which perform like the original ones regarding I/O, memory and compute load. This approach allows modelling a larger set of inputs than it is possible with the real applications and therefore is best suited for regular benchmarking.

Just as with the synthetic benchmarks, we expect that regular testing and recording of the results will allow us to monitor the behaviour of the DEEP Cluster and NVM evaluator. To do this, the applications and mock-ups will be integrated in the JUBE benchmarking environment.

The following subsections describe the main characteristic of the DEEP-ER applications from which application-based benchmarking will be done. In each application, its most important parameters have been identified. Different parameter-values will be selected to create various test cases, depending on the platform size and the specific benchmarking goals. The application's I/O patters are also described below, giving already an idea of the kind of I/O measurements that can be performed with them.

6.1 iPiC3D (KULeuven)

iPic3D is a particle-in-cell (PIC) code that simulates plasma using a semi-implicit method. Like most PIC codes, it consists of two parts, a particle solver that simulates the motion of charged particles in response to the electromagnetic field, and a field solver that simulates the electromagnetic field evolution in response to "moments" (e.g. net current and charge density) of the particles. The domain is discretized with a regular 3D mesh, whose submeshes are distributed among the processes performing the simulation. The application

[8] See Deliverable 6.1 [D6.1] for a complete description of the DEEP-ER applications.
[9] See Deliverable 4.1 [D4.1] for more information about the I/O patterns of the applications.

is currently implemented with MPI, with ongoing efforts to implement an efficient OpenMP parallelisation.

The structure of the application is sketched in Figure 5. The initialisation phase reads the input file, consisting of a simple text file that contains the basic parameters, such as the physical problem (initial and boundary conditions), the dimensions of the grid, the number of particles per mesh cell, the number of cycles to execute, and the frequencies with which field-data and particle-data should be written.

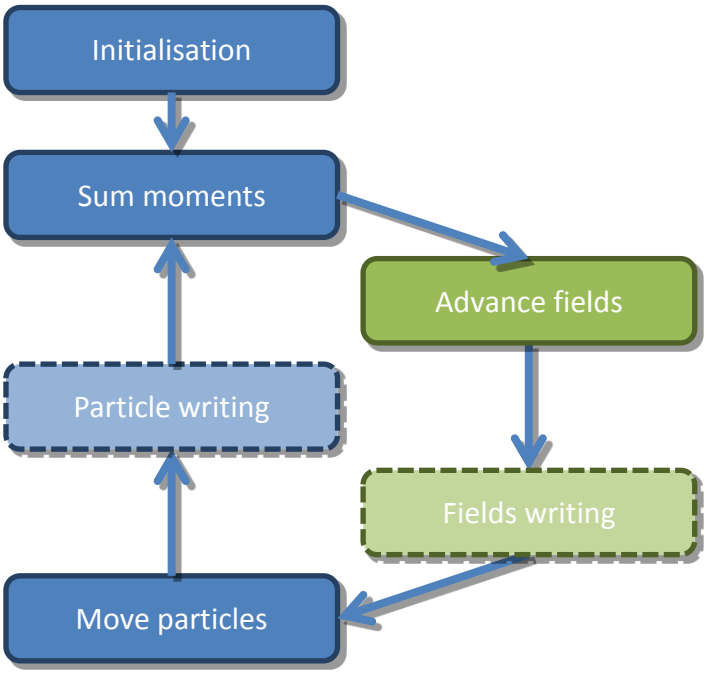


Figure 5: Phases of the iPiC3D algorithm

The initialisation phase can be very lightweight from the I/O perspective if the problem is mathematically determined, although it is also possible to initialize iPic3D by reading field and particle data. After every F cycles, fields are written to disk. Similarly, particles are written at every P iteration, with P larger than F, since the size of particle data is typically two orders of magnitude larger than the field data. Until now, I/O has never been a performance bottleneck for iPic3D. Nevertheless, the I/O scheme has recently changed to use parallel I/O, and the impact on the whole application runtime has yet to be analysed carefully.

6.1.1 iPiC3D parameters

The three main parameters that affect the current I/O behaviour are the number of mesh cells, the number of particles per mesh cell, and the number of MPI processes.

- Number of mesh cells: determines the size of the field data.
- Number of particles per mesh cell: together with the previous parameter, determine the size of the particle data.
- Number of MPI processes: determines the number of processes that have to be synchronised when writing field or particle data. Additionally, there are two parameters to specify the frequency of fields and particles writing.

# of processes	Field output	Particle output

# of processes	Field output	Particle output
1	~24KB	4 MB
4x4	~384KB	64 MB
16x16	~6MB	1 GB
64x64	~96MB	16 GB
256x256	~1.5GB	256 GB
1024x1024	~24GB	4096 GB

Table 7: iPiC3D test cases for 512 mesh cells per process and 128 particles per mesh cell (data per iteration)

6.1.2 iPIC3D I/O patterns

Under the current I/O scheme, one HDF5 file is generated every F iterations for field data and one HDF5 file is generated every P iterations for particle data, regardless of the number of processes. This increases the I/O overhead, since it requires synchronisation, but eliminates the need to merge data later as a post-processing step. The writing relies on H5hut, a library that uses pHDF5 and thus MPI-IO.

6.2 MAXW-DGTD (Inria)

The Inria application MAXW-DGTD is based on a Discontinuous Galerkin - Time Domain (DGTD) solver of the 3D Maxwell-Debye equation system. The solver is used to simulate the propagation of electromagnetic waves through human tissues of which the realistic geometry is accurately described by an unstructured tetrahedral mesh.

At the beginning of the project, MAXW-DGTD was simply MPI-parallel. The cell-local finite element formulation of the DGTD method leads to favourable data-locality properties in the most processing-intensive loops. Thus, an OpenMP implementation has been developed and optimised.

MAXW-DGTD consists of 3 phases: a pre-processing phase, the time-stepping loop, and a post-processing phase, which can be seen in Figure 6. The different steps behind this workflow happen in parallel, i.e. all the items are performed by all MPI processes. A newer version has replaced the blocking MPI_Sendrecv calls by non-blocking calls.

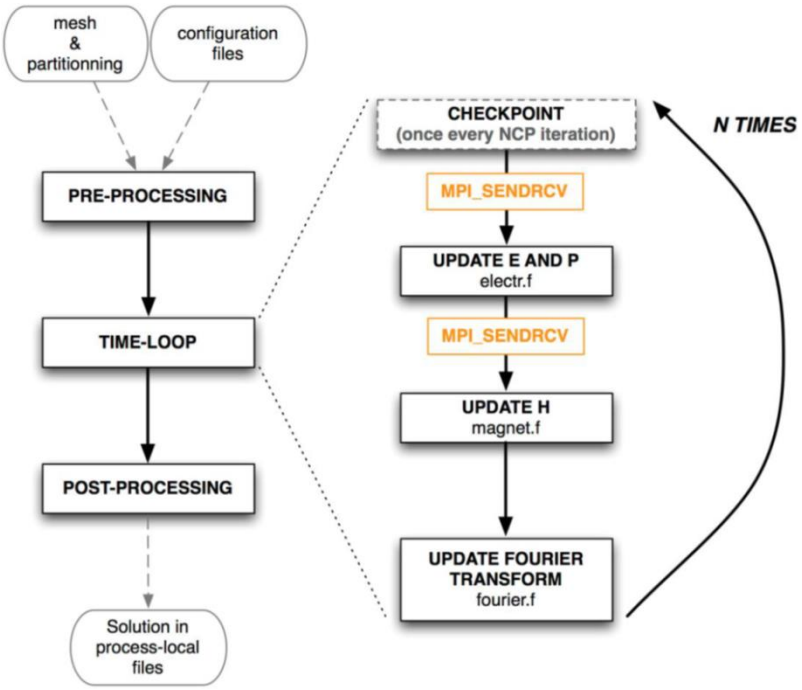


Figure 6: Phases of the MAXW-DGTD algorithm

Typically about 90% of the execution time is spent in the time stepping loop, the other 10% are shared between the pre- and post-processing phases when no checkpointing strategy is used.

6.2.1 MAXW-DGTD parameters

There are two main parameters that influence the size and complexity of the calculations and therefore the size of the output:

- The number of cells in the mesh (nt): it is provided by the mesh-file itself and cannot be changed by the user.
- The order of the cell-local Lagrange polynomials (“P1” to “P5”): it can go up to 5 (“P1”-“P5”) and is freely modifiable by the user. However, the problems that have been selected for DEEP-ER won't really benefit (in terms of precision) from an order higher than 3 for numerical reasons. Consequently, the cell-local polynomial order is typically set between 1 and 3, leading to a number of degrees of freedom per cell (“np”) ranging from 4 to 20.

Other parameters also affecting I/O are:

- The number of submeshes: it translates to the number of MPI processes (”nproc”) can also be set by the user. It typically ranges from 1 to 1024 in steps of the power of two. This number doesn't influence the amount of data subject to I/O; however, it changes the number of output files (as CP and final output is process-local).
- The number of spatial points observed and the frequency of checkpoints writing also affect the I/O of the application.

For a complete set of parameters see Table 8 which depicts several test cases. In this table, “Final time” refers to the dimensionless physical time of the simulation run.

	“HEAD P1”	“WOMAN P1”	“HEAD P3”	“WOMAN P3”
#cells: nt	1853832	5536852	1853832	5536852
#dofs per cell: np	4	4	20	20
#proc: nproc	1<nproc<1024	1<nproc<1024	1<nproc<1024	1<nproc<1024
Final time	2.5	5	2.5	5
#time iterations	11 250	646 680	34 380	1 969 787
Final output size	680MB	1.98GB	3.31GB	9.9GB
CP size	1 GB	2.9GB	4.3GB	12.8GB

Table 8: MAXW-DGTD test cases

6.2.2 MAXW-DGTD I/O patterns

I/O operations are performed in three phases of the application:

1. In the pre-processing step the two input files are read; one contains the unstructured tetrahedral mesh, the other one holds the information needed to define the submeshes and the communication lists between those submeshes. When the application is run in parallel mode, every MPI thread reads both files and extracts the relevant data. The size of the two files combined is a few hundred Mbytes.
2. In the time-stepping loop I/O is generally done at each timestep, with the loop consisting of a few tens of thousands of timesteps. In these I/O operations seven double precision real numbers per selected spatial point are written, with up to ten spatial points. Each point is written to a separate file. Which processes write the data depends on where the spatial points are located.
3. In the last phase of the application, the post-processing, the calculated solution is written to several files in different forms. The physical fields are stored in the Fourier space for a given frequency. Each process writes its own file for the fields that are associated with the tetrahedra of its submesh. These files are of a total size of 96 x np x nt bytes respectively; np is the number of degrees of freedom in a tetrahedron when the interpolation order is p and nt represents the effective number of tetrahedra.

The mesh sizes range from 1853832 to 5536852 cells. With the different orders of precision (“P1”-“P3”) this leads to the minimum and maximum sizes of (680 / nproc) MB and (9.9 / nproc) GB for the Fourier solution for each MPI process.

In addition to these I/O operations, a simple checkpointing mechanism was integrated in the application. Each process writes to a corresponding file the fields that are required for a recovery of the computation. This is done every few iterations; the period can be chosen by the user. The necessary data are the E, H and P fields (e_field(np,3,nt), h_field(np,3,nt), and pol(np,3,nt)), the current time, the number of the iteration and the state of the Fourier transform (ufourl(np,3,nt) and ufour(6,nt)). The total size of the data is (120 np + 96)xnt + 12 bytes per MPI process, which leads to a range of (1/nproc) GB to (12.8/nproc) GB of data per checkpoint, depending on the test case (see Table 8).

6.3 Radio Astronomy (ASTRON)

To form an image of the sky, the signals from the radio telescopes' antenna stations are combined in a *central signal processing* (CSP) unit. First, the incoming signals are filtered, correlated, and integrated by the so called correlator pipeline. In general, as the amount of incoming data are too large to be stored, the correlation pipeline needs to be performed in real-time. While in the future the image reconstruction is likely done in real-time as well, currently, the output of the correlator (called *visibilities*: corresponding to samples in the image's Fourier domain) is stored on disk for later offline processing (see Figure 7). Second, in the so called *science data processing* (SDP) part, the visibilities are read from disk and, in different frequency bands, sky images are reconstructed. To remove instrumental and environmental effects, the image reconstruction includes instrument calibration and the creation of an accurate sky model to correct the visibilities. For different science cases, the image reconstruction includes further integration in time and frequency.

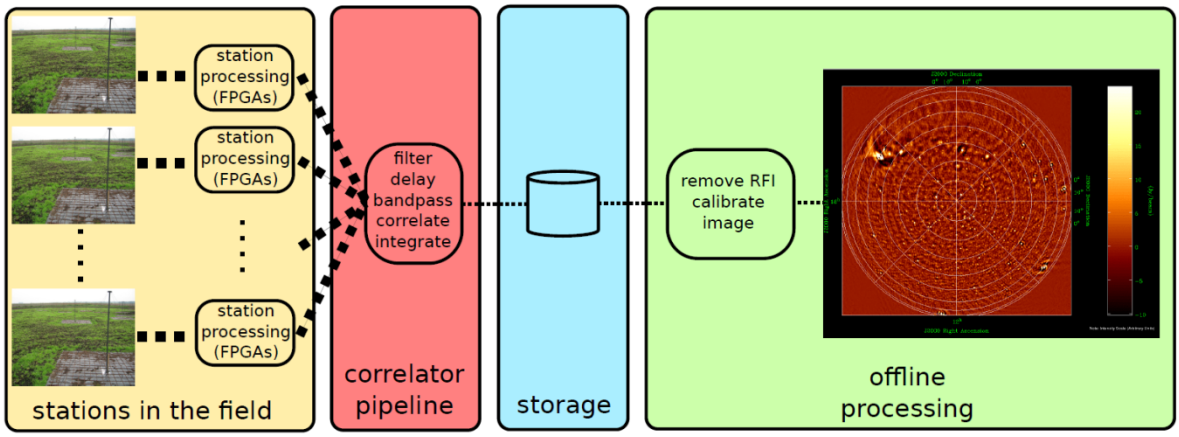


Figure 7: Image processing pipeline used in Radio Astronomy

6.3.1 Radio Astronomy parameters

We introduce the symbols used in the following in Table 9.

Symbol	Meaning
r	number of receivers or antenna stations
b	number of baselines; $b = \frac{1}{2}r(r+1)$
p	number of polarizations per receiver (usually, $p = 2$)
n_{sb}	number of frequency bands (subbands) per antenna station
s_{sb}	sample rate per subband
s_{r}	sample rate per receiver, $s_{\mathrm{r}} = n_{\mathrm{sb}} s_{\mathrm{sb}} p$
c	number of channels each frequency band is split into in the CSP
S_{int}	integrated samples, i.e., given integration time T_{int}, $S_{\mathrm{int}} = T_{\mathrm{int}} s_{\mathrm{sb}}$
$\lvert V_{\mathrm{sb}} \rvert$	size of visibilities per time step for one subband, $V_{\mathrm{sb}} = \mathrm{sizeof}(\texttt{complex<float>}) \cdot bcp^2$
$\lvert W_{\mathrm{sb}} \rvert$	size of weights for visibilities; $W_{\mathrm{sb}} = \mathrm{sizeof}(\texttt{int16\_t}) \cdot bc$

Symbol	Meaning
T_{step}	time steps; i.e., with observation time T_{obs}, $T_{\text{step}} = (T_{\text{obs}} s_{\text{sb}})/(c S_{\text{int}})$
n_{pixel}	image dimension; the total number of pixels is n_{pixel}^2

Table 9: Radio Astronomy symbols for various parameters

Assuming `complex<int8_t>` samples, the input streams deliver $\text{sizeof}(\texttt{complex<int8\_t>}) \cdot s_{\text{r}} r$ [bytes/s] of data and $(|V_{\text{sb}}| + |W_{\text{sb}}|) s_{\text{sb}} n_{\text{sb}}/(c S_{\text{int}})$ [bytes/s] have to be written to permanent storage. A given attainable input and output bandwidth therefore limits the number of antenna stations that can be used for an observation.

Besides this external I/O, internally, the correlation requires a data transpose over the network. Each input stream contains data from all n_{sb} frequency bands in p polarizations of one station, while correlation requires the data of all stations for one frequency band in p polarization (all subbands are processed independently). Assuming r *input processes* (one for each input stream) and n_{sb} *compute processes*, each input process needs to scatter all incoming data to n_{sb} compute processes, which in turn gather the data from all r input processes.

After the visibilities are computed and stored, sky images are created. As image reconstruction, including instrument calibration, is an area of active research, the exact requirements for future telescopes are not known. However, based on current technology, we provide *estimates* of the I/O requirements.[10] In particular, we assume the use of the AW-projection algorithm [CornwellGB08] and the creation of one image per sub-band. In this case, the requirements on the CSP and SDP are summarized in Table 10.

Processing	Input (rate)	Output (rate)
CSP (Correlation)	$\text{sizeof}(\texttt{complex<int8\_t>}) \cdot s_{\text{r}} r$	$\frac{(\lvert V_{\text{sb}}\rvert + \lvert W_{\text{sb}}\rvert) s_{\text{sb}} n_{\text{sb}}}{c S_{\text{int}}}$
SDP (Imaging)	$n_{\text{sb}} T_{\text{step}} (\lvert V_{\text{sb}}\rvert + \lvert W_{\text{sb}}\rvert)$	$\text{sizeof}(\texttt{float}) \cdot n_{\text{sb}} n_{\text{pixel}}^2$

Table 10: Radio Astronomy I/O and compute requirements of the CSP (Correlation) and SDP (Imaging)

For each subband and each time step, we read the visibilities and their weighting factors. The output consists of the final images. For an entire observation, $n_{\text{sb}} T_{\text{step}} (|V_{\text{sb}}| + |W_{\text{sb}}|)$ [bytes] are read from disk and $\text{sizeof}(\texttt{float}) \cdot n_{\text{sb}} n_{\text{pixel}}^2$ [bytes] are stored. The I/O requirements of the imaging strongly depend on the number of time steps of an observation, which is the number of visibility snapshots taken during the observation. Depending on the science case, an observation varies from a few minutes to many hours. For simplicity, we consider a fixed observation time of 12 hours.

In Table 11 we define two cases: *Present*, and *Future*, which are artificial, but correspond to realistic requirements of today's and near future telescopes. Given these parameters, the *Present* use case requires respectively 8 GB/s and 5 GB/s input and output data rate for correlation. For one observation, approximately 230 TB of data are read and 4 GB are finally stored. For correlation, the *Future* use case requires a data rate of 310 GB/s and 1.3 TB/s for input and output, respectively. Interestingly, the output data from the correlation pipeline is

[10] For a more detailed analysis, also based on current technology, see [JongeriusWNC14].

larger than the input, which is one reason why alternative algorithms and data compression techniques are currently under investigation. Furthermore, most likely, the imaging will also be done in real-time. If however the processing is based on current technology, during an observation, the correlator outputs 56 PB and the final images have a size of 4 TB.

As mentioned previously, the exact requirements strongly depend on the specific parameters of an observation, the exact telescope configuration, and the algorithms used for the data processing. As the next generation of telescopes is currently being designed, the above model and its implied requirements should be taken as rough estimates on the I/O requirements for two specific sets of parameters given in Table 11.

	Present	**Future**
$\mathbf{r}$	160	1000
$\mathbf{n_{sb}}$	60	256
$\mathbf{s_{sb}}$ $[\boldsymbol{kHz}]$	200	300
$\mathbf{c}$	256	1000
$\mathbf{S_{int}}$	1000	1000
$\mathbf{T_{obs}}$ [h]	12	12
$\boldsymbol{n_{\text{pixel}}}$	4096	65536
Input data rate	8 GB/s	310 GB/s
Output data rate	5 GB/s	1.3 TB/s
Data read per observation	230 TB	56 PB
Data stored per observation	4 GB	4 TB

Table 11: Radio Astronomy parameters for two use cases

6.3.2 Radio Astronomy I/O patterns

I/O requirements are dominated by the following stages:

1. For each antenna station an input streams of UDP packets and placed into input buffers.
2. Each input stream scatters to compute processes, which in turn gather data from all input streams.
3. The correlated data (i.e. the visibilities) are written to disk.
4. The visibilities are read from disk to calibrate the instrument and reconstruct the images.
5. The final images are stored into permanent storage.

6.4 Full Waveform Inversion (BSC)

Full Waveform Inversion is a cutting edge technique that aims to acquire the physical properties of the subsoil from a set of seismic measurements. Starting from a guess (initial model) of the variables being inverted (e.g., sound transmission velocity), the stimulus introduced and the recorded signals, Full Waveform Inversion (FWI) performs several

phases of iterative computations to reach the real value of the set of variables being inverted with an acceptable error threshold.

As the original code is proprietary from a third party, we have agreed in the Consortium to generate a mock-up code that performs like the original one with respect to the I/O to global and local file systems, traffic to main memory, and computational load.

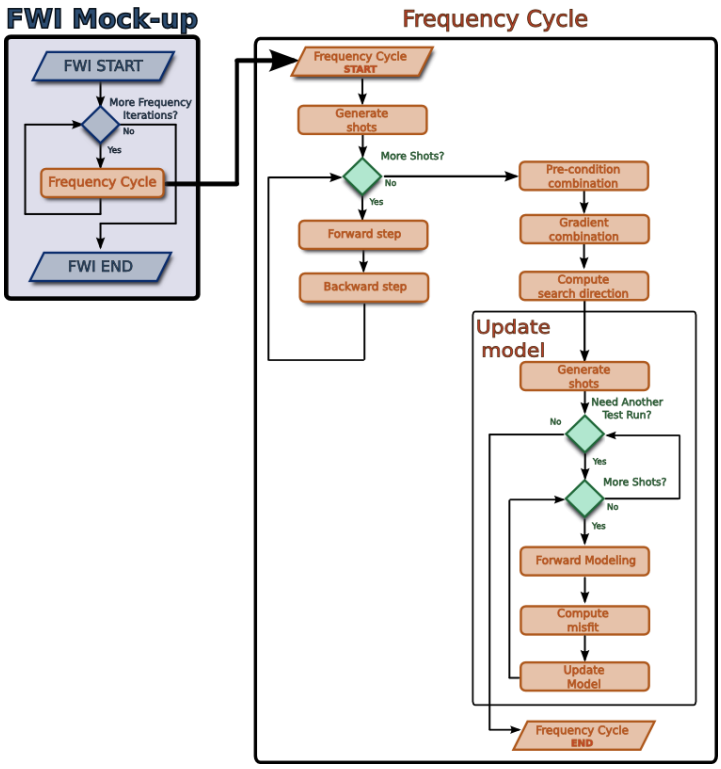


Figure 8: Main workflow for FWI application

Figure 8 shows the main structure of the FWI application, replicated by the mock-up, in which there is a main loop for different frequencies, within which these processing steps are performed:

1. Generate the set of overlapping “shots” (partitions of propagation wavefields[11], with their sources and receivers). In this phase we also need to re-interpolate the input data to suit the frequency requirements.
2. For each shot
    a. Forward step: compute propagation of source wavelet[12], together with the absorbing boundary conditions and extract the traces (pressure value at every receiver position). At every stack[13] step the forward propagation wavefield must be stored to disk (snapshot) to contribute to the gradient[14] and also must contribute to the pre-condition. At the end, the pre-condition must be saved to disk.

[11] Wavefield: computational array across which the waves are propagated.
[12] Wavelet: Discretised wave used as a stimulus for the wavefield.
[13] Stack: the number of steps between consecutive writes of simulation state (forward propagation) or reads of an old simulation state (backward propagation).
[14] Gradient: array of the same size of the wavefield, in which holds the maximum variation direction for that point in the wavefield.

   b. Backward step: compute the difference between the actual and the simulated seismograms (called adjoint receiver), and a measure of the misfit (currently an L2 norm) of these adjoint receivers. Then the propagation of these receivers must be computed. Starting from a given iteration, at every stack step the snapshots previously computed must be read (one at a time) and combined to update the gradient.
3. When all shots have been computed, all pre-conditions must be combined, and also the gradients must be combined.
4. Combine pre-conditioner and gradient and compute the search direction with an appropriate optimization method (e.g. steepest descent, conjugate gradient, etc.)
5. In order to update the model with the information from the search direction, a factor alpha must be computed. This alpha factor is used to modify the global model in order to minimize the cost function (misfit).
   a. Generate the set of overlapping shots.
   b. For each test (usually three), compute the forward propagation.
   c. Compute the misfit.
6. Finally, the velocity model is updated with the information given by the search direction and the optimum alpha found in point e.
7. If the misfit value is below the stop condition, there is no need to continue updating the velocity model for the current frequency, so it must continue with the next one.

It should be emphasised that phases 2 and 5. (the ones dealing with the shots) are fully parallel. This means that once the input data for a shot has been generated, this shot is processed independently of the others, thus making this application embarrassingly parallel.

6.4.1 FWI parameters

The main parameters for the FWI mock-up includes the frequency of the wavelet introduced as the source ($Freq$), the dimensions of the model (velocity field) in physical units (l_i) and the minimum propagation velocity (V_{min}). Note that the frequency parameter is not a scalar parameter but a vector, as the inversion process is mainly built around a loop of increasing frequency (see Figure 8).

As the computational model has to be discretized, the input parameters l_i and V_{min} will determine the size of the wavefield in computational points. The relation between these input parameters and the wavefield size for a single dimension is shown in Equation 1. Total wavefield size is $n_z \cdot n_x \cdot n_y$. We note that as each dimension grows in number of points linearly with the frequency, in a 3D field it will grow in size (i.e. memory) with the power of three of the frequency. Moreover, the time discretization (calculated internally in the mock-up) will depend linearly on the spatial discretization calculated in Equation 1. Hence, the number of time steps will grow and the total computational time will increase with the power of four of the frequency, due to the increase in memory size and number of total time steps.

$$n_i = \frac{l_i \cdot Freq}{V_{min}} \qquad i = z, x, y$$

Equation 1: Wavefield size for a single dimension

The remaining parameters are the number of sources and the number of receivers per source. The number of sources will determine the number of shots (see Figure 8) being

processed, one source for each shot. Finally the receivers per shot will define the set of seismic traces that will be the output of the internal shot processing.

Table 12 shows small, medium and large configurations that reproduce real-life scenarios for the FWI system. In addition, we propose an Exascale case, i.e. a configuration which is too expensive for current Petascale systems.

	Small	**Medium**	**Large**	**Exascale**
Set of frequencies (Hz)	2, 4, 6, 8	2, 4, 6, 8	2, 4, 6, 8	2, 4, 6, 8, 10, 12, 14, 16
Minimum velocity (mts/sec)	1000	1000	1000	1000
Dimensions (depth km * X km * Y km)	3 * 8 * 8	3 * 16 * 16	6 * 16 * 16	6 * 30 * 30
Number of shots	750	3000	15000	30000
Number of receivers (per shot)	750	3000	3000	12000

Table 12: Small, medium, large and Exascale test case for FWI system

6.4.2 FWI I/O patterns

The I/O pattern of the FWI application makes use of two different file systems:

- a large (and generally slow) global file system which is used to keep the main input data and the final results of the whole application, and to generate intermediate files required as inputs for the processing of each single shot;
- and a small (usually fast) local, per-node file system which is used for the temporary files generated during the processing of a single shot (forward and backward steps in Figure 8).

As the global file system is used mainly for general input and final results, the access to such I/O resources are, in general, restricted to the preprocessing (i.e. FWI start and Frequency cycle start) and postprocessing (i.e. FWI end and Frequency cycle end), and its usage does not hinder the application performance.

The local I/O is the main bottleneck of the application and it is issued by the forward and backward steps. During the forward processing step, the state of the program must be saved at defined intervals of time steps (snapshot). The appropriate number of time steps between writes is computed for each simulated frequency. Then, in the backward processing step, the saved snapshots are retrieved in the inverse order than they were written, used for correlating with the backward processing wavefield, and then discarded. In order to prevent the program from stalling when the local I/O is performed, we have chosen to design the I/O as an OmpSs task. By doing this we are able to keep one thread just waiting for the transfer to finish, whilst the rest of threads are overlapping this transfer with the computation of the forward or backward steps.

6.5 Chroma (UREG)

The Chroma application of UREG is a lattice QCD code. The purpose of Chroma is to do ab-initio simulations of the strong nuclear force. In the method used, space-time is discretised and the infinite dimensional path integral is converted to a large but finite dimensional integral. This integral can now be solved by Monte Carlo Methods. For a 4-dimensional cubic lattice, the Bosonic fields (the force between fermions) are represented by the links between two neighbouring lattice sites and are described by 3x3 complex matrices.

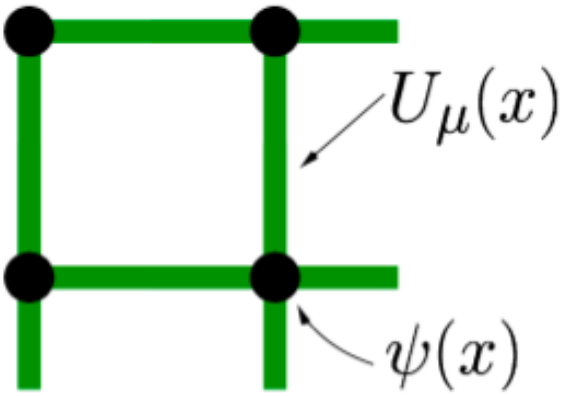


Figure 9: Symbolic lattice for QCD (Chroma application)

To perform the QCD simulations, the Hybrid Monte Carlo (HMC) method is used in this application. The different phases of a HMC are illustrated in the flowchart of Figure 10. The “fermionic” (particle) parts involve the solver, a part of the code that dominates the total runtime.

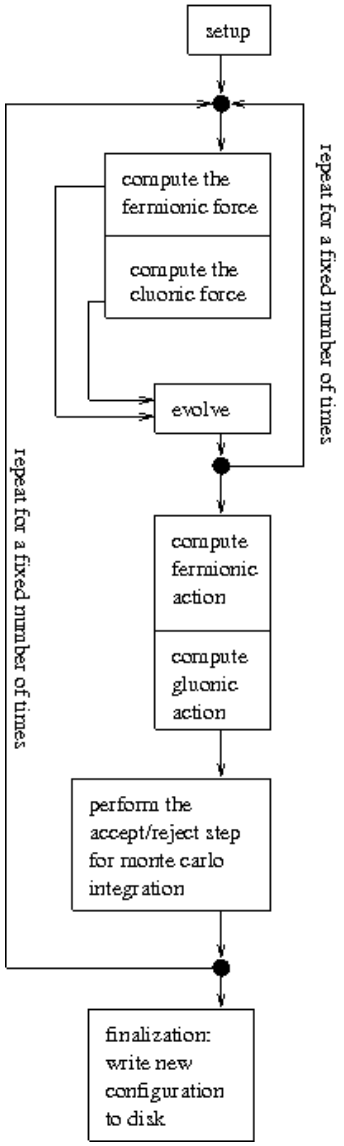


Figure 10: Phases of a HMC (Chroma application)

6.5.1 Chroma parameters

The only parameter that is changed for different test cases is the lattice volume. For weak scaling tests on one KNC, three different volumes were selected (see Table 13). The smallest one is bound by the algorithm of the application; the large one is limited by the memory of 8 GB on the current KNC. The physical state-of-the-art test case would be a lattice volume of 64x64x64x128, which would run on a system with 256 KNCs and therefore is not relevant in the current status of the project.

	Small	**Medium**	**Large**	**“State of the art”**
Volume	16x8x8x8	16x16x16x8	16x16x16x32	64x64x64x128
File size	4.5 MB	18 MB	72 MB	18 GB

Table 13: Chroma test cases

6.5.2 Chroma I/O patterns

The application mainly consists of three phases; the initialization, the main loop and the finalization. Altogether only two files are read or written during the execution of the program. The smaller file is an XML file which contains the simulation parameters. It is negligible here due to its small size. The significant file is the one that holds the gauge fields U (the links between fermions) and some metadata. The size of this file depends on the test case used (see Table 13). If checkpointing is activated, the files are written O(1000) times.

In the initialization phase, two different starting positions are possible. If a new Markov chain is started, the gauge fields U are set to unity, which is called a “cold start”. In this case, only the small XML file is read in. If an existing Markov chain is continued, in addition to the parameters, the file with the gauge fields is read in.

In the main loop the gauge fields can be stored for checkpointing reasons every few iteration steps. In addition, the simulation parameters are written in the XML file. These files can be used as the input for the initialization phase to restart the application. Typically, many Markov steps are discarded before an output file is written, but that can be adjusted by the user.

In the finalization phase, the latest configuration of the fields and the simulation parameters are written to disk one last time.

7 Summary and next steps

This deliverable described the test cases, environments and tools that will be used within Task 4.5 to perform an assessment of the DEEP-ER I/O Architecture. The goal of the assessment is to evaluate the I/O elements with regard to I/O performance, scalability and integration effort. For doing this, synthetic and application-based benchmarks will be used, as well as a benchmarking environment (JUBE) which will allow to monitor and keep track of the evolution of the system.

At the moment of writing this deliverable, the integration efforts already started: some synthetic and application benchmarks have been ported to JUBE and the configuration required for running time-based benchmarking jobs is nearly finished. Additionally, the benchmarks IOR, mdtest and LinkTest have been already run in the DEEP Cluster for starting to set the baseline performance of the system, and a series of tests with a mock-up of the FWI application have been performed in the NVM evaluator.

The work will continue by integrating the remaining benchmark codes into JUBE, establishing a series of routines for logging the tests results and running the actual tests on the platforms described within this deliverable, these are, the DEEP Cluster and the NVM evaluator. Once the DEEP-ER Prototype is available it will also be used for the benchmarking activity. We will analyse the measurements, compare them with I/O performance results of production systems and get performance indicators which will be used by the application developers. All this work will be documented at a later stage in Deliverable D4.5. Task 5.5 will make use too of the described benchmarking setup for resiliency performance measurements which will be reported in Deliverable D5.4.

References

[D4.1] “Definition of Requirements for I/O”, DEEP-ER Deliverable D4.1

[BeeGFS website] http://www.fhgfs.com/wiki/FhGFS

[IOzone website] www.iozone.org

[FIO website] www.freecode.com/projects/fio

[JUBE website] http://www.fz-juelich.de/jsc/jube

[IOR website] http://www.cs.sandia.gov/Scalable_IO/ior.html

[mdtest website] http://mdtest.sourceforge.net

[LinkTest website] http://www.fz-juelich.de/jsc/linktest

[D6.1] “Applications code analysis”, DEEP-ER Deliverable D6.1

[JongeriusWNC14] Rik Jongerius et al., *An End-to-End Computing Model for the Square Kilometre Array*. IEEE Computer, Vol. 47 (9), pp. 48-54, 2014

[CornwellGB08] T.J. Cornwell et al., *The Noncoplanar Baselines Effect in Radio Interferometry: The W-Projection Algorithm*. IEEE J. Selected Topics in Signal Processing, Vol. 2 (5), pp. 647-657, 2008

Annex A - IOR parameters

A.1 General IOR parameters

Option	Description (default in brackets[])
refNum	user supplied reference number, included in long summary [0]
api	must be set to one of POSIX, MPIIO, HDF5, or NCMPI depending on test [POSIX]
testFile	name of the output file [testFile] NOTE: with filePerProc set, the tasks can round robin across multiple file names '-o S@S@S'
hintsFileName	name of the hints file []
repetitions	number of times to run each test [1]
multiFile	creates multiple files for single-shared-file or file-per-process modes; i.e. each iteration creates a new file [0=FALSE]
reorderTasksConstant	reorders tasks by a constant node offset for writing/reading neighbor's data from different nodes [0=FALSE]
taskPerNodeOffset	for read tests. Use with -C & -Z options. [1] With reorderTasks, constant N. With reordertasksrandom, >= N
reorderTasksRandom	reorders tasks to random ordering for readback [0=FALSE]
reorderTasksRandomSeed	random seed for reordertasksrandom option. [0] >0, same seed for all iterations. <0, different seed for each iteration
quitOnError	upon error encountered on checkWrite or checkRead, display current error and then stop execution; if not set, count errors and continue [0=FALSE]
numTasks	number of tasks that should participate in the test [0] NOTE: 0 denotes all tasks
interTestDelay	this is the time in seconds to delay before beginning a write or read in a series of tests [0] NOTE: it does not delay before a check write or check read
outlierThreshold	gives warning if any task is more than this number of seconds from the mean of all participating tasks. If so, the task is identified, its time (start, elapsed create, elapsed transfer, elapsed close, or end) is reported, as is the mean and standard deviation for all tasks. The default for this is 0, which turns it off. If set to a positive value, for example 3, any task not within 3 seconds of the mean displays its times. [0]

Option	Description (default in brackets[])
intraTestBarriers	use barrier between open, write/read, and close [0=FALSE]
uniqueDir	create and use unique directory for each file-per-process [0=FALSE]
writeFile	writes file(s), first deleting any existing file [1=TRUE] NOTE: the defaults for writeFile and readFile are set such that if there is not at least one of the following -w, -r, -W, or -R, it is assumed that -w and -r are expected and are consequently used -- this is only true with the command line, and may be overridden in a script
readFile	reads existing file(s) (from current or previous run) [1=TRUE] NOTE: see writeFile notes
filePerProc	accesses a single file for each processor; default is a single file accessed by all processors [0=FALSE]
checkWrite	read data back and check for errors against known pattern; can be used independently of writeFile [0=FALSE] NOTES: * data checking is not timed and does not affect other performance timings * all errors tallied and returned as program exit code, unless quitOnError set
checkRead	reread data and check for errors between reads; can be used independently of readFile [0=FALSE] NOTE: see checkWrite notes
keepFile	stops removal of test file(s) on program exit [0=FALSE]
keepFileWithError	ensures that with any error found in data-checking, the error-filled file(s) will not be deleted [0=FALSE]
useExistingTestFile	do not remove test file before write access [0=FALSE]
segmentCount	number of segments in file [1] NOTES: * a segment is a contiguous chunk of data accessed by multiple clients each writing/reading their own contiguous data; comprised of blocks accessed by multiple clients * with HDF5 this repeats the pattern of an entire shared dataset
blockSize	size (in bytes) of a contiguous chunk of data accessed by a single client; it is comprised of one or more transfers [1048576]
transferSize	size (in bytes) of a single data buffer to be transferred in a single I/O call [262144]

Option	Description (default in brackets[])
verbose	output information [0] NOTE: this can be set to levels 0-5 on the command line; repeating the -v flag will increase verbosity level
setTimeStampSignature	set value for time stamp signature [0] NOTE: used to rerun tests with the exact data pattern by setting data signature to contain positive integer value as timestamp to be written in data file; if set to 0, is disabled
showHelp	display options and help [0=FALSE]
storeFileOffset	use file offset as stored signature when writing file [0=FALSE] NOTE: this will affect performance measurements
memoryPerNode	Allocate memory on each node to simulate real application memory usage. Accepts a percentage of node memory (e.g. "50%") on machines that support sysconf(_SC_PHYS_PAGES) or a size. Allocation will be split between tasks that share the node.
memoryPerTask	Allocate secified amount of memory per task to simulate real application memory usage.
maxTimeDuration	max time in minutes to run tests [0] NOTES: * setting this to zero (0) unsets this option * this option allows the current read/write to complete without interruption
deadlineForStonewalling	seconds before stopping write or read phase [0] NOTES: * used for measuring the amount of data moved in a fixed time. After the barrier, each task starts its own timer, begins moving data, and the stops moving data at a pre-arranged time. Instead of measuring the amount of time to move a fixed amount of data, this option measures the amount of data moved in a fixed amount of time. The objective is to prevent tasks slow to complete from skewing the performance. * setting this to zero (0) unsets this option * this option is incompatible w/data checking
randomOffset	access is to random, not sequential, offsets within a file [0=FALSE] NOTES: * this option is currently incompatible with:

Option	Description (default in brackets[])
	-checkRead -storeFileOffset -MPIIO collective or useFileView -HDF5 or NCMPI
summaryAlways	Always print the long summary for each test. Useful for long runs that may be interrupted, preventing the final long summary for ALL tests to be printed.

Table 14: General IOR parameters

A.2 POSIX-ONLY IOR parameters

Option	Description (default in brackets[])
useO_DIRECT	use O_DIRECT for POSIX, bypassing I/O buffers [0]
singleXferAttempt	will not continue to retry transfer entire buffer until it is transferred [0=FALSE] NOTE: when performing a write() or read() in POSIX, there is no guarantee that the entire requested size of the buffer will be transferred; this flag keeps the retrying a single transfer until it completes or returns an error
fsyncPerWrite	perform fsync after each POSIX write [0=FALSE]
fsync	perform fsync after POSIX write close [0=FALSE]

Table 15: POSIX-ONLY IOR parameters

A.3 MPIIO-ONLY IOR parameters

Option	Description (default in brackets[])
preallocate	preallocate the entire file before writing [0=FALSE]
useFileView	use an MPI datatype for setting the file view option to use individual file pointer [0=FALSE] NOTE: default IOR uses explicit file pointers
useSharedFilePointer	use a shared file pointer [0=FALSE] (not working) NOTE: default IOR uses explicit file pointers
useStridedDatatype	create a datatype (max=2GB) for strided access; akin to MULTIBLOCK_REGION_SIZE [0] (not working)

Table 16: MPIIO-ONLY IOR parameters

A.4 MPIIO-, HDF5-, AND NCMPI-ONLY IOR parameters

Option	Description (default in brackets[])
collective	uses collective operations for access [0=FALSE]
showHints	show hint/value pairs attached to open file [0=FALSE] NOTE: not available in NCMPI

Table 17: MPIIO-, HDF5-, AND NCMPI-ONLY IOR parameters

Annex B - mdtest parameters

B.1 General mdtest parameters

Option	Description
-b <branching_factor>	branching factor of hierarchical directory structure
-B	no barriers between phases (create/stat/remove)
-c	collective creates: task 0 does all creates and deletes
-C	only create files/dirs
-d <testdir>	the directory in which the tests will run
-D	perform test on directories only (no files)
-e <number_bytes_to_read>	number of bytes to read from each file
-E	only read files
-f <first_number_of_tasks>	first number of tasks on which the test will run
-F	perform test on files only (no directories)
-h	prints help message
-i <iterations>	number of iterations the test will run
-I <items_per_tree>	number of items per tree node
-l <last_number_of_tasks>	last number of tasks on which the test will run
-L	files/dirs created only at leaf level
-n <items_per_task_per_tree>	every task will create/stat/remove # files/dirs per tree
-N <stride_length>	stride # between neighbor tasks for file/dir stat (local=0)
-p <seconds>	pre-iteration delay (in seconds)
-r	only remove files/dirs
-R <seed>	randomly stat files/dirs (optional seed can be provided)
-s <stride>	stride between the number of tasks for each test
-S	shared file access (file only, no directories)
-t	time unique working directory overhead
-T	only stat files/dirs
-u	unique working directory for each task
-v	verbosity (each instance of option increments by one)
-V <verbosity>	verbosity value
-w <number_bytes_to_write>	number of bytes to write to each file
-y	sync file after write completion
-z <depth>	depth of hierarchical directory structure

Table 18: General mdtest parameters

NOTES:

- -N allows a "read-your-neighbour" approach by setting stride to tasks-per-node. Do not use it with -B, as it creates race conditions.
- -d allows multiple paths for the form '-d fullpath1@fullpath2@fullpath3'
- -B allows each task to time itself. The aggregate results reflect this change.
- -n and -I cannot be used together. -I specifies the number of files/dirs created per tree node, whereas -n specifies the total number of files/dirs created over an entire tree. When using -n, integer division is used to determine the number of files/dirs per tree node. (E.g. if -n is 10 and there are 4 tree nodes (z=1 and b=3), there will be 2 files/dirs per tree node.)
- -R and -T can be used separately. -R merely indicates that if files/dirs. are going to be stat'ed, then they will be stat'ed randomly.

Annex C - partest parameters

In the following only the options relevant for the project are described in more detail. The names correspond to the long option names. For the short options and the possible arguments please consult the "--config" argument.

C.1 File settings partest

- filename: File name of direct access file. In case of using multiple files this is the file name which is used as base file name while the additional file names get suffixed by numbers.
- numfiles: Number of files to use. If set to -1 the optimum number of files is computed by partest.
- chunksize: Chunk size used in open call. The chunk size is the amount of contiguous space for a task and hence the maximum size which can be written in one write call.
- fsblksize: Size of file system blocks. If set to -1 the size is automatically determined by SIONlib.Test configuration

C.2 Configuration partest

- testtype: Type of test. The available types are
  - 0: SIONlib, standard. This uses collective calls for opening and closing files but all the write or read calls in between do not need any further communication.
  - 1: SIONlib, independent. This type will not be used in the project.
  - 2: MPI IO: Write using MPI IO.
  - 3: Task-local file.
- bufsize: Size of blocks written in one single write call.
- totalsize: Global total size of data written.
- localsize: Local size of data written by each processor.
- factor: Factor for random size of chunk size. Should be avoided to ensure reproducibility.
- read: Switch read off/on.
- write: Switch write off, on, or double write.

C.3 Special options partest

- verbose: Verbose print info for each task. This is useful for small runs since it shows statistics for all tasks. Due to the high amount of additional output this should be avoided for > 32k tasks.
- nochecksum: Suppress checking correctness of data.
- debugtask: Debug task 0.
- Debugtask: Debug task n.

- posix: Use POSIX calls instead of ANSI calls. The important difference is the system internal buffering of the ANSI calls which can be prevented by choosing the unbuffered POSIX calls.
- collwrite: Use collective write if possible. This uses the collective calls of SIONlib, which is similar to the two stage MPI I/O. Nodes take roles of either collector or sender and during the collective write data is first collected and then only the collectors write the data. This mode also needs the environment variable SION_COLLSIZE to be set to the number of senders per collector or to -1 to let SIONlib compute a reasonable number. For further information see the documentation of SIONlib.
- collread: Use collective read if possible (see above).
- taskoffset: Shift tasks numbering for reading by offset compared to the mapping while writing to prevent data caching of file system. This is not compatible to the proposed checkpointing strategies.
- byteoffset: Start offset. Write <bytes> first before using blksize.
- serialized: Serialize I/O. Only I/O of #tasks are running in parallel (-1 -> all tasks in parallel, -2 -> use transactions, def: -1).
- unlinkfiles: Remove files after test.

C.4 partest parameters for Blue Gene/L, Blue Gene/P, Blue Gene/Q

- bigionode: Order tasks by BG I/O-node (0 none, 1 I/O-node, 2 I/O-bridge).
- bgtaskpernode: Number of tasks per BG I/O-node.
- bgtasksort: Sort task inside local communicator (0 distance to I/O-node, 1 global rank).

C.5 MPI-IO, GPFS partest options

- hintlargeblock: Hint MPI-IO, IBM, Large Block IO.
- hintiobufsize: Hint MPI-IO, IBM, IO bufsize in KB.
- hintsparseacess: Hint MPI-IO, IBM, sparse access.

C.6 Notes on size formats in partest

Suffixes like "MiB" can be added to sizes in order to scale them correctly. The available suffixes are "kilo" to "tera" for base 10 scaling and "kibi" to "tebi" for base 2. Two letter suffixes like "GB" trigger base 10 while one and three letter suffixes correspond to the binary ones.

List of Acronyms and Abbreviations

A

API: Application Programming Interface
ASTRON: Netherlands Institute for Radio Astronomy
Aurora: The name of Eurotech's cluster systems

B

BeeGFS: The Fraunhofer Parallel Cluster File System (previously acronym FhGFS). A high-performance parallel file system to be adapted to the extended DEEP Architecture and optimized for the DEEP-ER Prototype
Blue Gene: IBM project aimed at designing supercomputers that can reach operating speeds in the PFLOPS (petaFLOPS) range, with low power consumption.
Blue Gene/L: First generation Blue Gene
Blue Gene/P: Second generation Blue Gene
Blue Gene/Q: Third generation Blue Gene
BN: Booster Node (functional entity); refers to a self-booting KNL board (Node board architecture) including the NVM and NIC devices connected by PCI Express or a Brick (Brick architecture)
BoP: Board of Partners for the DEEP-ER project
Brick: Modular entity forming a Booster Node in the Brick Architecture, composed of Host modules, NVMe and NIC devices all connected by an PCI Express switch
BSC: Barcelona Supercomputing Centre, Spain

C

Chassis: Mechanical entity mounted in a rack. A chassis typically aggregates multiple mechanical sub-units (here: Bricks) through a chassis level infrastructure (e.g. backplane, power, cooling)
CINECA: Consorzio Interuniversitario, Bologna, Italy
CN: Cluster Node (functional entity)
CSP: Central Signal Processing. In astronomy, to form an image of the sky, the signals from the radio telescopes' antenna stations are combined in such a CSP unit.

D

DEEP: Dynamical Exascale Entry Platform
DEEP-ER: DEEP Extended Reach: this project
DEEP-ER Interconnect: High performance network connecting the Booster and Cluster nodes, the NAM and service nodes with each other to form the DEEP-ER Prototype.
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 PhiTM 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 Cluster: The prototype machine based on the DEEP Architecture developed and installed by the DEEP project

E

E10: Exascale 10. Parallel I/O software developed by a consortium of partners around the EOFS community. Partner Seagate is responsible for the development needed for the DEEP-ER project.

Eurotech: Eurotech S.p.A., Amaro, Italy

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 optimized for the DEEP-ER Prototype

FIO: Flexible I/O. IO workload generator used as an industry standard benchmark, stress testing tool, and for IO verification purposes.

FLOP: Floating point Operation

G

GFlop/s: Gigaflop, 10^9 Floating point operations per second

GPFS: General Parallel File System. High-performance clustered file system developed by IBM.

GRS: German Research School for Simulation Sciences GmbH, Aachen and Juelich, Germany

H

HDF5: Hierarchical Data Format: A set of file formats and libraries designed to store and organize large amounts of numerical data

H5hut: Library implementing several data models for particle-based simulations that encapsulates the complexity of parallel HDF5.

HPC: High Performance Computing

HW: Hardware

I

IB: InfiniBand

IBM: International Business Machines Corporation. American multinational technology and consulting corporation.
Intel: Intel Germany GmbH Feldkirchen, Germany
iPic3D: Programming code developed by the University of Leuven to simulate space weather
I/O: Input/Output. May describe the respective logical function of a computer system or a certain physical instantiation
IOR: IOR (Interleaved Or Random) benchmark program

J

JUBE: Jülich Benchmarking Environment
JUELICH: Forschungszentrum Jülich GmbH, Jülich, Germany

K

KNC: Knights Corner, Code name of a processor based on the MIC architecture. Its commercial name is Intel® Xeon PhiTM.
KNL: Knights Landing, second generation of Intel® Xeon PhiTM
KULeuven: Katholieke Universiteit Leuven, Belgium

L

LinkTest: Parallel ping-pong test between all possible MPI connections of a machine

M

MAXW-DGTD: Application based on a Discontinuous Galerkin - Time Domain (DGTD) solver of the 3D Maxwell-Debye equation system
MPI: Message Passing Interface, API specification typically used in parallel programs that allows processes to communicate with one another by sending and receiving messages
MPI-IO: MPI Input Output

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
NCMPI: Parallel NetCDF
NetCDF: Network Common Data Form. A set of software libraries and data formats that support the creation, access, and sharing of array-oriented scientific data
NIC: Network Interface Card, Hardware component that connects a computer to a computer network
NVM: Non-Volatile Memory. Used to describe a physical technology or the use of such technology in a non-block-oriented way in a computer system
NVMe: Short form of NVM-Express

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

O

OmpSs: BSC's Superscalar (Ss) for OpenMP

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

OS: Operating System

P

ParTec: ParTec Cluster Competence Center GmbH, Munich, Germany

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

PCIe: Short form of PCI Express

PCI Express: Peripheral Component Interconnect Express started as an option for a physical layer of PCI using high-performance serial communication. It is today's standard interface for communication with add-on cards and on-board devices, and makes inroads into coupling of host systems. PCI Express has taken over specifications of higher layers from the PCI baseline specification.

PFlop/s: Petaflop, 10^{15} Floating point operations per second

POSIX: Portable Operating System Interface

Q

R

Rack: Compartment to mechanically assemble multiple chassis to form the final computer

S

SDP: Science Data Processing. In astronomy a SDP is used to reconstruct sky images in different frequency bands based on the output data obtained from a correlator.

SIONlib: Scalable I/O library developed by JUELICH and GRS for the parallel access to task-local files

SSD: Solid State Disk

SW: Software

T

TFlop/s: Teraflop, 10^{12} Floating point operations per second

U

UREG: University of Regensburg, Germany

V

W

WAN: Wide Area Network
WP: Work Package

X

x86: Family of instruction set architectures based on the Intel 8086 CPU

Y

Z