H2020-FETHPC-01-2016

DEEP-EST

DEEP Extreme Scale Technologies

Grant Agreement Number: 754304

D6.1

Design and specification of programming environment

Approved

Version:	2.0
Author(s):	V. Beltran (BSC), J. Ciesko (BSC)
Contributor(s):	C. Clauss (ParTec), K. Keller (BSC), L. Bautista (BSC), P. Reh (ITWM), L. Montigny (Intel), B. Steinbusch (JUELICH)
Date:	26.09.2018

Project and Deliverable Information Sheet

DEEP-EST Project	**Project ref. No.:**	754304
	Project Title:	DEEP Extreme Scale Technologies
	Project Web Site:	`http://www.deep-projects.eu/`
	Deliverable ID:	D6.1
	Deliverable Nature:	Report
	Deliverable Level: PU*	**Contractual Date of Delivery:** 31/March/2018
		Actual Date of Delivery: 31/March/2018
	EC Project Officer:	Juan Pelegrin

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

Document Control Sheet

Document	**Title:** Design and specification of programming environment	
	ID: D6.1	
	Version: 2.0	**Status:** Approved
	Available at: `http://www.deep-projects.eu/`	
	Software Tool: LaTeX	
	File(s): DEEP-EST_D6.1_Design_and_specification_of_prog_env.pdf	
Authorship	**Written by:**	V. Beltran (BSC), J. Ciesko (BSC)
	Contributors:	C. Clauss (ParTec), K. Keller (BSC), L. Bautista (BSC), P. Reh (ITWM), L. Montigny (Intel), B. Steinbusch (JUELICH)
	Reviewed by:	E. Suarez (JUELICH) M. Girone (CERN)
	Approved by:	BoP/PMT

Document Status Sheet

Version	Date	Status	Comments
1.0	26/03/2018	Final	Final version
2.0	26/09/2018	Approved	EC approved

Document Keywords

Keywords:	DEEP-EST, HPC, Modular Supercomputing Architecture (MSA), Exascale, Programming environment

Table of Contents

List of Figures

Executive Summary

This deliverable presents the programming environment for the Modular Supercomputing Architecture (MSA) proposed in the DEEP-EST project. Planned work as stated in the Description of Work has been refined to take into account the application requirements described in D1.1 *Application co-design input*, the system software requirements presented in D5.1 *Collection of software requirements* and the MSA architecture introduced in D3.1 *System Architecture*.

The goal of the programming environment is twofold: Firstly, to facilitate the development of new applications as well as the adaptation of existing ones to fully exploit the MSA architecture. Secondly, this work contributes with optimizations and extensions of key software components such as message-passing libraries, task-based programming models, file systems, checkpoint/ restart libraries, and data analytics and machine learning frameworks to leverage specific hardware features that will be available on the Cluster Module (CM), Extreme Scale Booster (ESB) and Data Analytics Module (DAM).

The work presented in this deliverable will be refined further once the final decision on the specific hardware technologies to be includon the specific hardware technologies to be included in the DEEP-EST prototype is made. This corresponds to D3.2. *High-level system design*. Moreover, over the course of the project, further modifications will be done if new application requirements arise.

1 Introduction

This document describes the programming environment that will be developed and adapted according to the requirements of the Modular Supercomputing Architecture (MSA) as described in D3.1 *System Architecture*, as well as, the requirements of the applications. The main goal of the programming environment is to facilitate the development of applications so that they can effectively exploit the MSA architecture including specific hardware features of each module. In some cases this can be achieved transparently, in others, modifications of the applications will be needed.

The programming environment covers the most relevant parts of the software stack required to run on a supercomputer and it includes the following components:

- ParaStation MPI communication library (Section 2) to leverage distributed memory systems
- OmpSs-2 programming model (Section 3) to exploit many-core processors, deep memory hierarchies and accelerators
- Some of the most popular frameworks and libraries used for data analytics and machine learning (Section 4)
- BeeGFS filesystem (Section 5) to exploit the storage subsystem
- FTI/SCR multi-level checkpoint/restart libraries (Section 6) to enhance the application resiliency to system faults

In the following paragraph we summarize the main developments that will be conducted for each of these software components.

ParaStation MPI will be enhanced to support the heterogeneous nature of the MSA architecture, providing optimized modularity-aware MPI collective operations, integrating the Global Communication Engine (GCE) to speed up collective operations, supporting one-sided communications on the Network Attached Memory (NAM) and supporting MPI workloads with specific features.

The OmpSs-2 programming model will be enhanced to improve programmability and exploit specific hardware features of each module. For the ESB, OmpSs-2 will be extended with an improved task nesting and dependency system integration that can expose more parallelism in order to fully utilize large many-core processors. The scheduler of the runtime system will be enhanced to support locality-aware scheduling in order to leverage the cache hierarchy and Storage Class Memories (SCMs). Further, the tasking model will be extended to improve support of blocking I/O and networking calls. Finally, extensions will be developed to enable the support of GPUs with CUDA.

Data analytics and machine learning frameworks and libraries will be analysed and optimized for the DAM module. Support to the applications will be provided to ensure high performance of the Deep Learning frameworks on the hardware of the MSA.

The BeeGFS filesystem will be extended with a plugin architecture that facilitates the integration of new SCM memories such as the NAM with a filesystem. This includes the development of an integrated monitoring facility to store time series. The BeeOND on-demand filesystem will be integrated with the resource manager and job scheduler developed in WP5.

Finally, the Fault Tolerant Interface (FTI) multilevel checkpoint/restart library will be extended to increase the resiliency of applications running on the MSA architecture. Several key features such as reusing the application output as a checkpointing image will be developed.

All these developments will require a close collaboration with WP5, especially to integrate Para-Station MPI, the FTI checkpoint/restart library, the data analytics frameworks and BeeOND with the resource manager and job scheduler developed in WP5.

2 ParaStation MPI

This chapter details the design specifications for the envisioned DEEP-EST programming environment in particular with respect to the Message-Passing Interface (MPI) [1].

The distinct nature of the Modular Supercomputing Architecture (MSA) demands for a likewise modularity-aware MPI library. While ParaStation MPI has been extended in DEEP for being a Global MPI accommodating the Cluster-Booster concept, and has then further been enhanced by means for fault-tolerance in DEEP-ER, ParaStation MPI will now be the tailored implementation for the MSA and the DEEP-EST prototype. For this purpose, ParaStation MPI needs to be adapted and extended to meet the quite specific requirements, which can be described as follows:

- First of all, *transparent optimizations* for accommodating the modular nature of the MSA are required in order to prevent the inter-module communication from becoming a bottleneck. For example, support for modularity-aware collective operations, which feature appropriate communication patterns especially for the inter-module case, have to be implemented and also new hardware features like the Global Collective Engine (GCE) have to be exploited for improved scalability and efficiency.
- In addition, there will most probably be a demand for MPI-related features that are needed by other frameworks and whose implementation may even go beyond the current MPI standard. The most prominent example for such *explicit extensions* of the programming interface is the integration of Network-Attached Memory (NAM) accessibility into the current one-sided communication functionalities of MPI. This is because, although the MPI standard features *Put* and *Get* functions for accessing the content of remote memory regions directly, the current MPI semantics does not have the notion of globally accessible memory regions that are not associated with a particular MPI rank.
- Finally, a modularity-aware version of ParaStation MPI has to provide means for facilitating *workflows* and the required data exchange between different applications running in different modules of the DEEP-EST system. On one hand, task offloading in terms of MPI sub-applications to be spawned as workers onto appropriate system modules during the runtime of a respective master application has still to be supported comprehensively. On the other hand, especially with respect to more generic workflows, data processing chains that are built out of a whole succession of applications will have to be supported also by the MPI environment.

2.1 Application-transparent MPI Adaptations

As already stated in the introduction of this chapter, the extensions to be implemented for ParaStation MPI can be roughly divided into those that are transparent to the application and those, that widen the programming interface and demand for an explicit utilization by the application. This section deals with performance optimizations for collective communication operations, which belong to the former category.

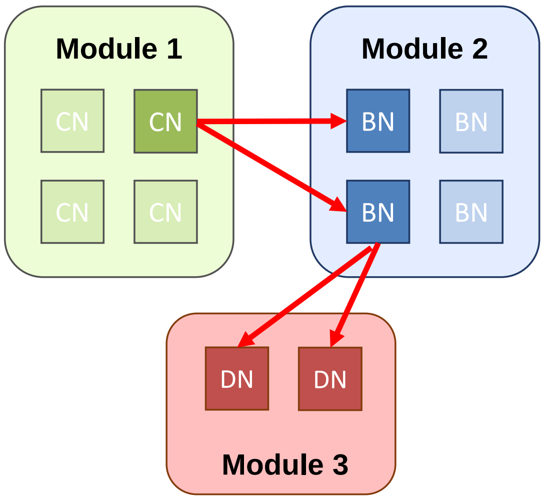
2.1.1 Modularity-awareness for Collective MPI Operations

Modularity-awareness is closely related to hierarchy-awareness. As the latter has also been a relevant aspect for Grid-, Wide-Area- and Meta-Computing, taking a closer look at the solutions for collective communication in those domains has shown to be worthwhile. So, for instance, Kielman et al. postulated in [2] the following set of rules that has to be fulfilled for gaining an improved performance of collective communication operations in clustered wide-area systems:

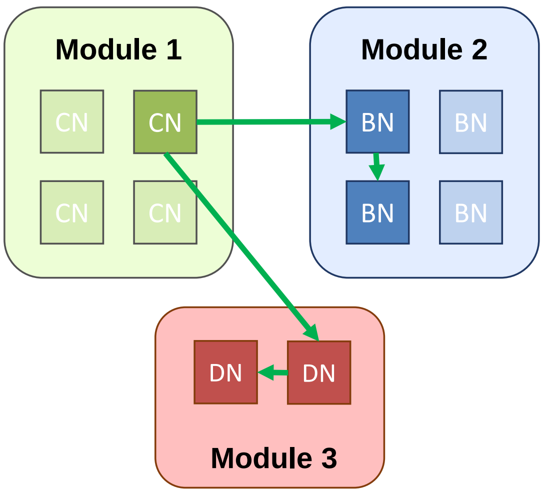
1. Every sender-receiver path used by an algorithm contains at most one wide-area link.
2. No data item travels multiple times to the same cluster.

The first rule limits the impact of the wide-area latency to a single hop, whereas the second rule saves precious wide-area bandwidth that might otherwise be wasted for the transmission of the same data in parallel to the same cluster (but to different processes within, of course).

Obviously, the application of these rules can easily be transferred to the MSA case, assuming that the inter-module communication — most probably across some gateway nodes acting as bridges between the modules — is prone to become a bottleneck. Figure 1 shows an example for the realization of a simple broadcast pattern on an MSA: While in (a) a simple binary tree is transparently applied and hence the rules above are broken, in (b) a topology-aware pattern is applied, which is in compliance with the two rules, and where the collective communication is accordingly optimized.



(a)



(b)

Figure 1: A broadcast pattern applied to a set of processes running within different modules:
(a) by means of a simple binary tree, resulting in a non-optimized mapping;
(b) by applying a pattern that takes the hierarchy information into account.

For realizing a corresponding modularity-awareness in ParaStation MPI for the DEEP-EST prototype, we have already identified a promising starting point in terms of the so-called SMP-awareness[1] feature of the MPICH library. This topology-awareness on SMP/node granularity, though only implemented for a subset of MPICH's collectives, is designed in a straight-forward

[1]SMP: *Symmetric Multi-Processor* — a compute node with multiple CPUs/cores working on a shared memory.

manner by applying a collective communication pattern always in two stages: once for the inter-node communication only and then once again for the intra-node communication [3].

As MPICH, which is a quite prevalent vanilla MPI implementation, serves also as the upper layer ofypararStation MPI, the extension and exploitation of this feature as a starting point for implementing modularity-awareness with respect to collective operations is not too far to seek. Therefore, the existing mechanism within MPICH has to be extended by a further tier in the hierarchy so that collective communication patterns are to be conducted in three (instead of two) stages: initially between the modules, then within the modules but between the nodes, and finally within the nodes.

However, as modules and nodes may even feature further hierarchies (so, for example, nodes may exhibit NUMA domains and modules may, in turn, be built up from sub-modules) we envisioned a more generic solution where the number of tiers is not fixed but can be adjusted.

2.1.2 Integration of the Global Collective Engine (GCE)

As Extoll's Global Collective Engine (GCE) will be a distinct addition to the Network Federation (NF) of the envisioned MSA and a unique feature of the DEEP-EST prototype, it is self-evident to integrate its support also in ParaStation MPI. However, as the GCE is bound to an Extoll fabric, only modules equipped with Extoll network interfaces will directly benefit from these accelerators for collective communication operations — which will definitely be the Extreme Scale Booster (ESB) as well as, quite likely, the Data Analytics Module (DAM).

Since MPI communication is possible across the whole MSA, the integration of the GCE into ParaStation MPI demands again for modularity-awareness in order to apply GCE-based accelerations only on those (sub-)groups of processes that are located within an Extoll fabric. According to this, the aforementioned approach of Section 2.1.1 will also be utilized here to differentiate between inter-module and module-internal communication paths of a collective pattern, so that the GCE can be utilized for self-contained sub-graphs within the Extoll-equipped modules. For instance, a global reduction operation is in a first step to be conducted internally for the module — and will hence be accelerated by the GCE at least for the Extoll part — but then has to be followed by a subsequent inter-module reduction step according to the hierarchy. That way, process groups whose members exclusively belong to an Extoll-equipped module can directly benefit from the GCE, whereas groups of processes with members in different modules may at least benefit *indirectly* as long as some sub-groups are within the Extoll fabric.

Regarding the actual integration of the GCE functionality into ParaStation MPI, the described modularity-awareness has to be coupled with a logic that is able to identify sub-groups of processes belonging to an Extoll fabric for setting-up and triggering related GCE mechanisms. Moreover, since also the GCE demands for a configuration on a ”per-group” granularity before the actual operation can be performed, it is quite natural to do so every time a new MPI communicator gets created. Therefore, if this mechanism can be paired with the above mentioned utilization of the shadow-communicators, the required integration into ParaStation MPI should not be too complicated.

2.2 MPI Support for NAM-related RMA Operations

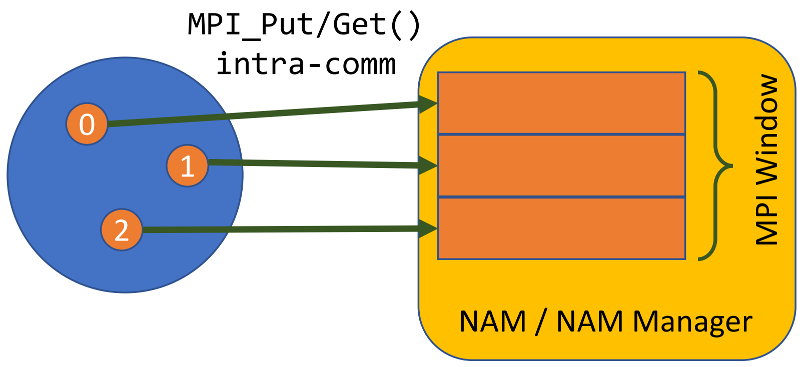
Currently, there are two not mutually exclusive scenarios under discussion for the way in which an application may access the NAMs in the DEEP-EST prototype. The first one is to access the NAMs as a fast storage abstracted by a file system; and the other is to provide access to the NAMs as remote but directly addressable memory regions in terms of MPI's Put and Get functions.

Most probably, the NAMs will be shipped by Extoll together with an extended version of their libNAM, which will allow for concurrent accesses of multiple processes to the same memory region. Obviously, there is a demand for coordinating such simultaneous accesses and the MPI RMA model seems to be a good candidate for doing so in a convenient manner. However, the current MPI standard unfortunately does not feature the idea of a globally accessible memory that is not associated with a certain process rank — and the question is how this could nevertheless be realized in an MPI environment without violating the standard severely. For doing so, we currently see two possible ways, which may even complement each other but will not be strictly standard compliant:

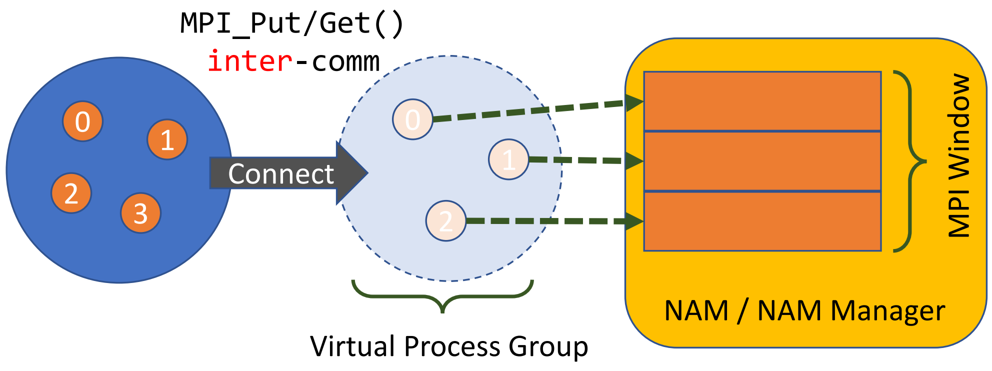
The first one is to stick to the notion of an ownership for the memory regions in a sense that each NAM region has to be allocated and then exposed by a particular MPI rank. According to this, the related MPI window regions (though globally accessible and located within the NAM) have then to be addressed by means of the rank of that process which allocated the NAM region before. The actual NAM memory allocation can then, for example, be hidden within an `MPI_Alloc_mem` or `MPI_Win_allocate` call where the MPI *info* argument contains a certain key/-value pair (e.g. `MEM_KIND=NAM`). The main difference between this approach and the usual MPI RMA memory model would be that strictly *no* load and store accesses, even to the *reputedly* local NAM memory regions, are allowed. That means that even for accesses to memory locations within the window that belong to the accessing process itself have to be always conducted via Put and Get calls.

Another way is the idea of having also *persistent* NAM regions that may live beyond the finalization of the respective MPI session. Such a semantics could likewise be supported by providing a related key/value pair (e.g. `MEM_KIND=NAM_PERSISTENT`); or by letting the application consciously omit the otherwise required `MPI_Free_mem` and/or `MPI_Win_free` calls. In particular the idea of having `MPI_Win_allocate` together with a certain entry within the passed info object would have the advantage that the allocation is collective over the used communicator. That in turn means that the NAM allocation could be represented by a *virtual* process group that still exists even if its actual MPI session has already been finished. According to this approach, the NAM and/or the resource manager would have to handle such virtual process groups together with their NAM allocations as a resource that can be requested by further MPI sessions. For this purpose, the original MPI session that allocated the persistent NAM regions could utilize the `MPI_Publish_name` function for announcing the external accessibility of this resource. A subsequent MPI session would then be able to query for a related service port via `MPI_Lookup_name`, which could in turn be used to attach to the persistent NAM regions by means of an `MPI_Comm_connect` call. As the remote ranks of the resulting *inter*-communicator would just be virtual, no actual MPI point-to-point or collective communication would be allowed on this communicator. Instead, the inter-communicator could rather be used to query via `MPI_Comm_remote_size` for the number of those processes the former NAM window was attached to — hence determining the size of the virtual process group. In addition, this inter-communicator would also serve as a handle for a subsequent call of `MPI_Win_create`. As only intra-communicators are commonly allowed here as arguments, the MPI library can check for this and then return a *special* window object that can be used to address the remote NAM regions, for which size and displacement can in addition be queried as common window attributes via `MPI_Win_get_attr`.

Figure 2 illustrates these two presented approaches and shows how the former (a) can be combined and extended by the latter one (b) for the case of persistent NAM regions. During the next project phase, the applicability and usability of both approaches will be discussed and evaluated together with WP1 and WP4 in order to have a prototype programming environment for this eventually implemented and tested right before Deliverable 6.2.



(a) Three processes that all allocated NAM regions and then have attached them to an MPI window.



(b) Four processes that are connected to a virtual process group representing persistent NAM regions.

Figure 2: Illustration of the two proposed ways for integrating NAM into an MPI world.

2.3 MPI Support for Workflows

This section discusses the required interfaces and their framework integrations that will be needed to support the envisioned workflows of applications — where two or more codes are run after each other across different modules of the DEEP-EST prototype while using one code's output data as another code's input — also from the MPI perspective.

As already detailed for Task 5.3 and Task 5.4 in Deliverable 5.1, there will most probably be different approaches for passing intermediate data between one (sub-)application and its related job-step to the subsequent one. Currently, we see in particular two new approaches for this inter-step communication. The first one is to use the NAM as a very fast temporary storage — be it used as a fast global storage via a filesystem or by means of the `MPI_Connect/Accept`-based approach as introduced in the prior section for accessing persistent NAM regions via Put and Get. The second one is to use MPI-based communication for passing the data directly between distinct but still running MPI sessions in a point-to-point and/or collective fashion.

For doing so, the latter approach requires that the related processing steps overlap in terms of their MPI sessions and that they can connect to each other for the data passing — which certainly can be realized by utilizing again `MPI_Connect/Accept`, but this time according to their original purpose. However, at this point a further interesting question arises: How can such a joining between distinct MPI sessions be coordinated?

Obviously, it is foremost the duty of the MPI Process Manager to handle such connecting operations under several aspects [4]: as a Process Manager commonly also acts as a connection broker, a distributed but globally accessible Naming Service needs to be provided in order to allow formerly disconnected processes of separate sessions to join each other. Furthermore, as the Process Manager controls the processes as well as their assigned resources, a reliable bookkeeping of all such connecting (and likewise disconnecting) operations between MPI sessions needs to be provided in order to ensure a proper resource release and process clean-up.

While the required orchestration between the Process Manager and the Job Scheduler for accomplishing these tasks belongs to Task 5.4 and Task 5.5, the design and implementation of the corresponding interface between the MPI application on one side and the MPI Process Manager on the other side naturally falls into Task 6.1 — and hence into this section of the deliverable at hand. For realizing such an interface, we envision a comprehensive implementation of the related means as they are foreseen by the MPI standard, which are, in particular, the feasibility to publish points of contact for MPI sessions via `MP_Publish_name` and likewise the possibility for an MPI session to query for such and other services via `MPI_Lookup_name`.

Although these features and the related functions are part of the current MPI standard, it has to be emphasized that an MPI library is not obliged to implement those interfaces in much more than the simplest way. Nevertheless, for the DEEP-EST prototype we actually target a comprehensive and fully customized realization that shall especially take the aspects of modularity as well as the distinct hardware features like the NAM appropriately into account.

3 The OmpSs-2 Programming Model

To support the modular supercomputer architecture (MSA) and to maintain a reasonable degree of compatibility to the applications of this project, we propose to maintain an hybrid programming model that combines message-passing libraries and task-based programming models to exploit distributed-memory and shared-memory respectively. However, the ambition to achieve high resource utilization and exascale performance requires new features and optimizations. In this task we will improve the OmpSs-2 (OMP Superscalar) [9] task-based programming model in four key areas: programmability, scalability, device support (offloading) and interoperability.

From the system architecture perspective, feature requirements can be grouped into support for many-core processors, deep memory hierarchies and I/O subsystem, hardware accelerators and distributed systems. This corresponds to the structure of this chapter. The next section gives an overview of the OmpSs-2 programming model.

3.1 The Programming Model in a Nutshell

OmpSs-2 is a declarative, data-flow, programming model developed at the Barcelona Supercomputing Center. It consists of a language specification, a source-to-source compiler for C, C++ and Fortran and a runtime system.

The language specification defines a set of directives that allow a declarative expression of tasks through the *task* pragma. Further, OmpSs-2 allows a programmer to annotate task parameters with *in*, *out* and *inout* clauses that correspond to the *input*, *output* or *inputoutput* access semantic of this parameter within that task. This information establishes a producer-consumer relationship between tasks, also called task dependency or data-flow. With this information the runtime is capable of task scheduling that maintains correctness of code while alleviating the programmer of implementing manual synchronization. Further, the *taskwait* construct allows task synchronization and instructs the calling thread to wait on all previously created tasks within that scope.

While this is similar to tasking in the recent specification of OpenMP, the OmpSs-2 runtime implements a different execution model. In OmpSs-2, every application starts with a predefined set of execution resources and an explicit parallel region does not exist. This view avoids the exposure of threading to the programmer as well as the requirement to handle an additional scope, the scope of a parallel region.

At compile time, the OmpSs-2 compiler processes pragma annotations and generates a temporary code file. This file includes both, user code as well as all required code for task generation, synchronization and error handling. In the final step of compilation, OmpSs-2 invokes the native compiler to create a binary file.

At runtime, the main thread (also called leader thread) progresses through code, creates tasks and stops at synchronization points (explicit or implicit barriers). Task creation comprises the creation of the task-object itself, which carries all descriptive information, and the handling of its dependencies. Once a task object has been created, the runtime inspects the dependency graph to determine the relationship to previously created tasks. If a dependency has been found, progression into deeper levels of the graph can be stopped and a representative node is added to the graph. In the opposite case, the task is placed into a ready-queue. Tasks in the ready-queue are picked up by worker threads, removed from the queue and executed.

3.2 Supporting Many-core Processors

The ESB nodes of the MSA prototype are characterized by a higher count of energy-efficient processor cores. Achieving high hardware utilization on this architecture is predominantly determined by three factors: algorithmic design that exposes enough concurrency, data-flow based task execution that replaces fork-join parallelism and reduces the need for coarse-grained synchronization and lastly runtime efficiency to create and execute tasks sufficiently fast.

To help developers write code that can efficiently run on the ESB module, we propose a set of concepts, annotations and improvements that we discuss below.

3.2.1 Improved Tasking Support

Single-threaded task creation is a costly process if a large number of tasks is created sequentially. Concurrent task creation is a solution that avoids the bottleneck. In this work we develop and evaluate nesting support to implement concurrent task creation where multiple outer tasks can create inner tasks concurrently.

Task-nesting splits the data-flow into nesting levels (domains) which reduces the effectiveness of a data-flow programming model. In this scenario coarser-grained outer tasks often need to wait until all sibling tasks finish execution which is typically implemented by a *taskwait* and often limits resource utilization. This can be avoided if the dependency information of the inner tasks is exposed to the outer levels. For this purpose we implement the concept of weak dependencies. This type of dependencies allows to maintain the data-flow based execution through nesting levels and increases the degree of concurrency(scalability).

Further, we aim for optimizing the runtime system, especially dependency analysis, work queuing and distribution to deal with high loads caused by concurrent task generation (runtime scalability).

3.2.2 Improved Programmability

To improve the support of common algorithms (programmability), we further add the support of *array-type reductions* and *task loops*. The language support for array-type reductions requires to support memory regions and regions with potential overlaps. Implementing runtime support for this type of reductions is not trivial as concurrent execution results in data races and a common technique of data privatization is limited in scalability for larger data types and SMP systems. We discuss this in further detail in section 3.3.

Task loops represent an efficient way to parallelize *for-loops* but unlike the *parallel for* work-sharing construct in OpenMP, it defines a different semantic. Task loops create tasks of an arbitrary size as defined by the user. Further, tasks created within this construct may be included into the application's data flow if the expression of dependencies for this construct is allowed.

It is important to point out that applications will likely require code modifications in order to reduce the number of fork-join loops and to adapt to the OmpSs-2 pragma annotations.

We anticipate that techniques offering efficient support of ESBs will also apply to compute nodes (CNs) and data analytics modules (DAMs) as their CPUs will include the same or smaller number of cores (possibly just different SKUs).

3.3 Supporting Deep Memory Hierarchies and I/O subsystem

The MSA prototype foresees the addition of a DIMM-based non-volatile memory (SCM) as another level of cache line addressable storage. This memory class is characterized by its ability to operate in memory mode or block mode. In this work we explore both use-cases.

SCMs configured in memory mode (attached to the memory bus), will likely implement a non-uniform memory access and will come with an API for memory operations (such as malloc). While we expect that the API will be used explicitly by the programmer, the runtime might benefit from it for internal operations as well.

In general, to support specialized memories, the programming model scheduler must be aware of memory locations (NUMA nodes), memory types (RAM, high-bandwidth-memory or SCM) and task data locations. This allows to maximize data reuse and to employ a given storage type for the right purpose. In this task we extend OmpSs-2 by a locality- and storage-type aware task scheduler. Further, we expect that specialized memories may be beneficial to hold scratch-pad data. Such data is generated by programming model runtimes to parallelize certain algorithms or to temporarily hold task data (software cache). Such a support would use the SCM API internally.

SCM devices configured for block access may benefit from the OmpSs-2 programming model by allowing an asynchronous execution of blocking I/O calls.

3.4 Support for Hardware Accelerators

Using hardware accelerators in applications requires the programmer to develop device kernels and to manage the execution of these kernels. This includes programming data transfers and synchronization with other kernels or threads.

In this task we extend the OmpSs-2 programming model to support the execution of accelerator kernels programmed in CUDA (device support). This step simplifies the use of accelerators as data movements between host and devices as well as synchronizations are taken care of by the runtime. Further, this work explores how to take advantage of modern features such as unified memory and on-demand paging on the proposed GPGPUs.

If FPGAs are programmed with OpenCL, we will consider adding support for this type of devices.

3.5 Support for Distributed Systems

To improve the performance of hybrid MPI+OmpSs-2 applications, support for MPI calls within tasks is required. The benefit is three-fold: Such code organization helps to avoid fork-join parallelism where traditionally concurrent execution phases are followed by sequential phases of MPI communication; secondly, the runtime system can pause the execution of blocking tasks thus freeing execution resources; and lastly, the execution of MPI calls can be streamlined following the application's data-flow.

To support the asynchronous execution of MPI calls with tasking, the OmpSs-2 runtime requires the ability to pause and resume tasks as well as a callback mechanism to associate events (resume events) to blocked tasks. This work reuses insights and functionality developed in the INTERTWinE [10] and DEEP-ER [11] projects.

4 Data Analytics programming model

This chapter presents the design specfication of the Data Analytics programming model in regards to the usage of the applications.

Of the six co-design use cases in the DEEP-EST project, four (from partners CERN, University of Iceland, University of Leuven, NMBU) do include significant data analysis steps which require specific SW frameworks. One of the applications in the NMBU use case does require specific Python packages, and is included in this section. The specific applications requirements are defined in D1.1.

Deep learning frameworks simplify deep learning model development and training by providing high-level primitives for complex and error-prone mathematical transformations, like gradient descent, back-propagation, and inference. They enable the development of different kinds of deep neural networks by separating the model construction and the model execution, and their use is state-of-the art in the Deep Learning field.

4.1 Objectives

We provide a list of the required frameworks currently used by the data analytics and machine learning applications in the DEEP-EST project. In addition, a recommended list of framework widely used in the DL community is added for potential test and improvements of the current applications. One example of the latter is BigDL, which looks promising for use in the CERN data analytics scenario.

The dependencies of each framework are defined as they will need to be installed on the different modules of the DEEP-EST prototype system (ESB, DAM, CM), and have to interoperate with the more traditional, HPC-style programming frameworks. The modules have been specified in D3.1 and during the co-design meeting in Jülich in February 2018, where we decided to use on the DAM, per node two Intel® Xeon® CPUs, one Nvidia Tesla V100, one Intel® Stratix® 10 FPGA, one 40Gbit/s Ethernet card and one TOURMALET card. Furthermore we also describe various optimizations of frameworks on each module.

On the modules of the MSA architecture, we propose to install all the frameworks with the last available optimizations. All of them will be kept up to date on the corresponding architecture. In this task, high-touch application support will be given to the end-users of WP1, and interoperability with the resource management system and the required traditional HPC frameworks will be provided.

We will support the developers to write efficient and optimized code for Machine Learning and Data Analytics. In order to take full advantage of multiprocessing architecture, we will help them to adapt their networks or scripts, switch to a different framework or in the extreme, hardware if a better performance is expected.

4.2 List of required frameworks

- **TensorFlow**

  Tensorflow is available as Open Source software from [5] and can be built/installed on a wide variety of systems. To ensure best performance, optimized versions of TensorFlow should be used, in particular on the DAM module. In this respect, highly optimized TensorFlow is available for the Intel® Xeon® SP [see D3.1] line of CPUs and for NVIDIA GPGPUs.

  The Intel optimized TensorFlow integrates optimizations for Intel Xeon SP processors and are available for the DEEP-EST project. Tensorflow is compiled with specifics flags and libraries (Intel MKL-DNN) on each architecture to accelerate performance, it takes advantage of the higher number of cores and the larger Advanced Vector Extensions SIMD instructions (AVX-512). Using the GPU mode, it will be particularly adapted to the DAM using the Nvidia Tesla V100. It should be noted that the FPGA available in the DAM will be used with the Intel Software API and OpenCL to run TensorFlow. Concerning the use by application, TensorFlow is currently adopted by KULeuven (DLMOS) and University of Island (DeepLearning).

  Dependencies: Python 2.7-3.6, Anaconda, Intel Parallel Studio XE 2018 (including the latest Intel C++ compilers), Intel MKL-DNN, bazel, gcc 7.2.0.

  Dependencies for GPU support: CUDA® Toolkit 9.0, NVIDIA drivers associated with CUDA Toolkit 9.0, cuDNN v7.0.

- **Keras**

  Keras [6] is a high-level neural networks API written in Python and capable of running on top of TensorFlow, CMTK, or Theano. With Keras, it is possible to quickly build simple or very complex neural networks. This framework simplifies the process of building deep learning applications. Instead of providing all the functionality itself, it uses either TensorFlow or Theano as backend and adds a standard, simplified programming interface on top. Keras will be available on the ESB, CM and DAM. It will profit of the backend's optimizations. At the time of writting, Keras is only using the TensorFlow backend by KULeuven (DLMOS) and University of Island (DeepLearning).

  Dependencies: Python 2.7-3.6, TensorFlow or Theano or CNTK.

- **Intel BigDL**

  Intel BigDL [7] is a distributed deep learning library for Spark that can run directly on top of existing Spark or Apache Hadoop clusters. To achieve high performance, BigDL uses Intel MKL and multi-threaded programming in each Spark task. On the DAM, it would especially take advantage of the Intel Persistent Memory DIMM (Apache Pass) to deal with high volume of data.

  It is currently used by the CERN (ML) application to perform data analytics at Big Data scale.

  Dependencies: Apache Spark, Hadoop, Scala.

- **Apache Spark**

Apache Spark [8] is an open-source engine developed specifically for handling large-scale data processing and analytics. Its real strength may lie in its ability to improve ETL (Extract, Transform, and Load) processes. ETL is a process used to integrate disparate data types and create a unified view of the data. It has a powerful distributed engine to scale out deep learning on massive datasets on distributed memory-based architecture. Several libraries to run on top of Spark for machine learning are now available, in particular, Spark MLlib and Intel BigDL are rapidly becoming a must-have for those working in AI. Apache Spark is used by the CERN application.

Dependencies: MLlib, BigDL.

- **Intel Distribution for Python**

  Python is a programming language used by researchers and engineers for machine learning and data science. Indeed most popular and widely used deep-learning frameworks are implemented in Python on the surface and C/C++ under the hood. It is used by all the applications of the DEEP-EST project associated to machine learning, it should be noted that NMBU is using specific libraries. The Intel Distribution for Python [12] use multiple cores and Single Instructions Multiple Data (SIMD) to accelerate deep learning applications. It offers performance speed-up on SciPy stack: NumPy, SciPy, and scikit-learn boosted by the Intel® Math Kernel Library. It will be installed on the ESB, CM and DAM.

  Dependencies: Intel DAAL, Anaconda, Docker, Numpy, Quantities, Scipy, six (Optional: pandas, nose).

The following table summarizes the required frameworks of the applications:

Application	**TensorFlow**	**Keras**	**bigDL**	**Spark**	**Python**
KULeuven (DLMOS)	yes	yes	no	no	yes
UoI (DeepLearning)	yes	yes	no	no	yes
NMBU	no	no	no	no	yes (specific packages)
CERN (ML)	considered	no	yes	yes	yes

Table 1: Fameworks currently used by the machine learning applications

4.3 List of recommended frameworks

- **Caffe & Caffe 2**

  Caffe would be installed on the ESB, DAM and CM with the Intel optimization. As for TensorFlow, an additional GPU version would be added on the DAM, it needs to be compiled with cuDNN.

  Dependencies: (GPU Support) with CUDA 7+ required for GPU mode with cuDNN, BLAS, Boost $\geq$ 1.55, Python 2.7.6.

- **Theano**

  Theano is a Python library for manipulating and evaluating expressions, especially matrix-valued ones, like with Numpy. It needs to be compiled. An Intel optimized version is available on Anaconda. However OpenCL support is still minimal indicating some difficulties for FPGA support on the DAM. CPU and GPU support is native.

Dependencies: with Python 2.7-3.6, NumPy $\geq$ 1.9.1 $\leq$ 1.12, SciPy $\geq$ 0.14 $<$ 0.17.1, BLAS (included in Intel MKL).

- **Intel Neon**

  Neon is Intel's reference deep learning framework committed to best performance on all hardware. Neon can either use CPU or GPU backend and would be particularly adapted to the DAM. It has been optimized for much better performance on CPUs by enabling Intel Math Kernel Library (MKL).

  Dependencies: Python 2.7 or Python 3.4+, CUDA SDK and drivers.

- **Intel DAAL**

  Like the Intel® Math Kernel Library (Intel® MKL), Intel DAAL is a highly optimized library of computationally intensive routines supporting Intel architecture including Intel® Xeon® processors However, most of Intel MKL was designed for when all the data to operate upon fits in memory at once. Intel DAAL can handle situations when data is too big to fit in memory all at once. Intel DAAL provides for data to be available in chunks rather than all at once. Intel DAAL is designed for use with popular data platforms including Hadoop, Spark, R, Matlab, etc. for highly efficient data access. Intel DAAL will be used by CERN for Big Data Analysis in case this application will need to do "out of core" processing of extremely large datasets.

5 I/O, file system, and storage

BeeGFS is a scalable cluster storage solution that works on top of standard POSIX filesystems. During the DEEP-ER project, BeeGFS has been extended by a caching layer that introduces cache groups which enable fast per group I/O without involving the global storage. In DEEP-EST, we will focus on the improvent of several parts of BeeGFS according to task 6.4. This includes the development of a storage plugin architecture, the improvement of the BeeOND framework and the provision of a new monitoring solution.

SIONlib is a library that enables scalable task-local I/O of application specific data with an internal format such as checkpoints. For the DEEP-EST project, the collective I/O mechanism of SIONlib will be made aware of the specific I/O characteristics of the MSA. Based on the collective I/O mechanism, we also intend to explore the possibility of I/O forwarding.

5.1 BeeGFS storage plugins

The new plugin architecture of the BeeGFS storage daemon will enable the use of various storage backends besides the currently used underlying POSIX file system. In particular, a plugin for the NAM will be developed so it can be used as a BeeGFS target.

5.1.1 Current situation

BeeGFS uses separate daemons for storing the metadata and the content of a file (meta and storage). The metadata contains the usual file metadata (ownership, access, extended attributes, . . .) as well as the internal information that is needed to locate the actual raw data that belongs to the file. The data is stored in so called chunk files, that are located on the storage targets that belong to the storage nodes. Data is usually striped and distributed to chunk files on multiple targets for performance reasons. When a client wants to access a file's content, it retrieves the corresponding storage targets and the path information that contains the internal location of the chunk file from the meta service. Afterwards it opens a connection to each storage server which then handles read/write operations to the chunk(s). The striping information tells the client from which target it has to request which part of the file.

5.1.2 Storage plugin architecture

With storage plugins, the handling of file content will be different. The data will still be split into chunks but those chunks will not necessarily be stored on an underlying POSIX file system. Instead, we will introduce an intermediate layer that abstracts the mapping of chunk IDs to chunk data. This way the storage server can handle data independently of the medium they are stored on. The server itself will only handle chunk IDs and the storage plugin takes care of mapping the chunk IDs to the data on the storage backend. By introducing this new architecture we enable the utilization of various storage technologies alongside the existing POSIX backend.

The interface specification will be openly available, so anyone can develop additional plugins tailored to their backend infrastructure.

The DEEP-EST prototype will contain a global NAM storage module that provides byte addressable memory directly to the network. As part of the project, we will implement a plugin that enables the use of the NAM as a BeeGFS storage backend. This new plugin will map chunk IDs to address ranges on the NAM. It will be developed using libNAM that was created during the DEEP-ER project. Utilizing the high transfer speed and low access time of the NAM will make it ideal for use in a scratch file system.

5.2 BeeOND integration

BeeOND (BeeGFS on demand) is a framework that allows creating an ephemeral parallel file system on an arbitrary number of hosts. Setup and teardown can happen in a matter of seconds and it is easily integratable into a cluster batch system. This makes it ideal as a per job scratch file system or cache layer. As part of the project, BeeOND will be utilized by the resource manager to set up such a storage environment on demand as a scheduled job requires it.

BeeOND is implemented as a collection of automation scripts, acting as an intermediate layer between the various BeeGFS services and the user. It handles tasks like starting and stopping the required daemons on remote hosts in the correct order, while communication is done using SSH. This simple approach ensures that integration of BeeOND into any other service is as easy as possible.

5.3 BeeGFS monitoring

As part of the project, BeeGFS introduces a new monitoring solution, called BeeGFS-mon. It takes advantage of a time series database (InfluxDB) to store the data collected by the daemons.

Each server node collects and buffers its own statistics. The beegfs-mon daemon pulls those from the meta and storage nodes in configurable, regular intervals and puts them into the database. Depending on the type of node (meta or storage), different measurements are available.

- **Meta:** Provides information about its workload, network traffic and availability.
- **Storage:** Provides information about its workload, network traffic, availability and combined disk usage of all targets. Additionally, storage nodes provide measurements for each storage target, containing disk usage, available inodes and consistency state.

Furthermore, each node collects the I/O calls made on beegfs from the clients. They are also collected by mon, added up and provides by client user ID or node address. Some of the data is collected in fixed one second intervals, other depends on the set pulling interval. The details can be found in the BeeGFS documentation. The data being collected and the way it is handled may change in the future.

Since the collected data points are stored in an InfluxDB database, they are easily accessible and queryable by using one of its supplied APIs. Also all tasks related to data management (retention policies, for example) are also handled by the database engine.

5.4 SIONlib MSA aware collective I/O

Recent versions of SIONlib contain mechanisms to perform I/O operations collectively, i.e. all processes of a parallel computation partake in these operations. This enables an exchange of I/O data between the processes, allowing a subset of all processes, the *collector* processes, to perform the actual transfer of data to the storage on behalf of other processes. Collector processes should typically be those processes that are placed on parts of the MSA with a high bandwidth connection to the parallel file system. Currently, this requires specialized knowledge and explicit configuration by a user of SIONlib. We intend to make the collective I/O mechanism of SIONlib aware of the specific properties of the MSA with regards to I/O bandwidth available to different parts of the system and provide an optimized default configuration for the choice of collector processes.

5.5 SIONlib I/O forwarding

In its current form, the collective I/O mechanism in SIONlib still requires the collector processes to be members of the set of processes initially comprised in a parallel computation. For computations which do not place any processes on system parts with a high bandwidth connection to the parallel file system, this limitation precludes an optimal choice of collector processes. We intend to explore an extension of the collective I/O mechanism into an I/O forwarding mechanism that spawns additional processes on system parts with a high bandwidth connection to the parallel file system to act solely as I/O collector processes.

6 Resiliency

The aim of this task is to enhance application resiliency with FTI, a multilevel checkpoint library maintained at the BSC. The basis of considerations for feature enhancements are the requirements of the applications within the project. The improvements shall leverage the DEEP-EST hardware and integrate with other middleware and runtime libraries developed within the DEEP-EST project, such as OmpSs and BeeGFS.

In alignment with the description of task 6.5 in the DEEP-EST proposal, we will provide a checkpoint pragma directive in OmpSs that enables the user to register the checkpoint data, the checkpointing frequency and the level of reliability of the checkpoints.

In addition to that we will enrich FTI with features that will increase performance of resilience measures and also broaden the spectrum of application that can leverage FTI.

We will focus on the implementation of an HDF5 interface, differential and incremental checkpointing and a flexible storage scheme (user defined file names and paths).

The accomplishments made on SIONlib leveraging a new buddy checkpointing scheme in combination with SCR during DEEP-ER will be integrated in the DEEP-EST project and an installation of SCR will be available on the prototype.

In the following, we give a short overview of the current FTI features followed by an analysis of the applications in order to clarify the advantages of the enhancements proposed and a description of these in the last section.

6.1 Core Features of FTI (Release V1.0)

The core FTI features are:

- Topology awareness (knowledge about on which node a process runs)
- Multilevel checkpointing (1: lowest security, 4: highest security)
  - Level 1: The checkpoint files are stored locally on the nodes
  - Level 2: In addition to Level 1 storing a copy on the partner node
  - Level 3: In addition to Level 1 encoding the checkpoint files of a group with `n` members in such a way, that a total loss of `n` files (checkpoint/encoded) can be afforded
  - Level 4: The checkpoint files are stored on the PFS
- Mapping the checkpoint data to one file on the PFS using MPI-I/O or SIONlib
- Delegating the post-processing to a dedicated head process (for levels 2-4)
- Ensuring data consistency on restart with checksums
- Support for partitioned communication patterns

6.2 Application Analyses

Name	Has Ckpt	Need Ckpt	Scale	Exec - Time	I/O - Size
NEST	no	yes	++++	+++	+++
GROMACS	yes	yes	++++	++++	+++
xPic	yes	yes	++++	+++	++++
CMS-SW	yes	yes	++++	++	++++
HPDBSCAN	no	yes	++++	++	++
piSVM	no	yes	++++	++	+
DNN	yes	yes	+	+++	+
Correlator	no	no	n.a.	++	++++
Imager	no	no	n.a.	++	n.a.

Table 2: Application Characteristics. Scale: from couple of processes to systemwide. Exec-Time: from minutes to months. I/O Size: from MBs to TBs

An overview of the DEEP-EST applications is given in Table 2. GROMACS and xPic both do already incorporate checkpointing (so does DNN through TensorFlow). However, the checkpointing is performed directly to the PFS. The staging of files to fast storage devices through FTI and flushing with a dedicated process to the PFS would be beneficial here.

Checkpointing in NEST is useful for several reasons. The scale of the application is high and the memory consumption is in the regime of hundreds of GB and practically the whole memory needs to be protected. Since production runs can reach 24 hours it is prone to failures. In addition to the resilience aspect it is useful to freeze and store execution states for a successive exploration of different paths the simulation could take. However, the modular structure of the runtime and the profile of the data sets (nested structures and wide usage of pointers) makes the implementation of checkpoint/restart difficult and it remains an open issue.

HPDBSCAN and piSVM are both relatively small applications measured in lines of source code (HPDBSCAN $\approx 1\,000$ and piSVM $\approx 10\,000$). Also the runtime is short compared to common production runs of large HPC simulations. But since they can scale to a high number of processes, checkpointing is reasonable to apply.

DNN is expected to have execution times up to 12 hours. Thus, a checkpointing mechanism is recommended. DNN is build on top of TensorFlow and it is written in Python. It may already use the in-build checkpoint mechanisms provided by TensorFlow.

6.3 Enhancements

The applications above represent a fairly good cross-section of applications in HPC. Based on these applications we will focus on enhancements for FTI that are aligned to the requirements of their HPC profile.

Although there is a large variety of features which would be interesting to implement in FTI, we will focus on the ones which have, to our understanding, the greatest impact. Nonetheless, we will explore some of the rather exotic features in a proof-of-concept manner.

Incremental checkpointing, the storage of checkpoint chunks chronologically as soon they are released by the applications to minimize the I/O throughput and avoid bottlenecks, can be beneficial for GROMACS and xPic since these applications deal with large datasets. xPic can serve as a use case here since the electric and magnetic fields are computed temporally separated.

In CMSSW, products generated from the event processing are flushed to disk in an incremental fashion when a certain threshold is reached. This matches our definition of incremental checkpointing. Notice that this is not checkpointing in the traditional sense since no restart will be performed on the data. Nevertheless, leveraging incremental checkpointing may be beneficial here, since the staging feature of FTI may speed-up the process.

Incremental checkpointing may also be useful for NEST. That is due to its modular character. When we say modular, we refer to the way how the implementation of the various models takes place. In spite of its complexity, it is an iteration based simulator. Every time-step, the state of the simulation evolves by iterating and updating the modules loaded for the simulation. Checkpointing could be implemented by incrementing the checkpoint data in each module separately.

Differential checkpointing, i.e. writing only the parts of the datasets that have changed since the last checkpoint, may be beneficial in xPic and GROMACS regarding the large amount of checkpoint data. GROMACS could serve as a use case here, since it contains datasets that are generated during the initialization but not changed further on. These are for instance atom types, md parameters, PME grid and others. Although the datasets are constant, they have to be stored in each checkpoint.

HDF5 output will be beneficial for many scientific applications and certainly for those that use intermediate simulation data for analyses. xPic performs all I/O using HDF5. That is for data analysis after the execution but also in intermediate steps and for checkpointing. Writing this data with FTI into HDF5 files may on one hand speed-up the execution by leveraging the staging feature of FTI, but also protect against data loss due to the various safety levels FTI offers.

We will provide API functions to define purpose and location of the output. In that way FTI will extend its usage to general I/O purposes. Typically, applications do not only perform I/O for checkpointing during runtime. Also intermediate results are often written frequently. To take advantage of the staging feature of FTI, it would be necessary to allow the user to specify data for non-resilience purpose. This data may be attached with a user defined path so that FTI may write both, the checkpoints and the analysis data.

7 Summary

In this Deliverable we have presented the software components that are the foundation of the programming environment. These software components include: the ParaStation MPI library, the OmpSs data-flow programming model, several well-known data analytics frameworks such as TensorFlow and Spark, the scalable BeeGFS filesystem, and the FTI and SCR multi-level checkpointing/restart libraries. All this software components will be used, adapted and enhanced to meet application requirements and leverage the specific hardware features of each module. The final objective of the whole programming environment presented in this Section is to ease the development of new applications and the adaptation of existing one that can easily make the most of the MSA concept endorsed by this project.

List of Acronyms and Abbreviations

A

API	Application Programming Interface
ASIC	Application Specific Integrated Circuit, Integrated circuit customised for a particular use
ASTRON	Netherlands Institute for Radio Astronomy, Netherlands

B

BADW-LRZ	Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften. Computing Centre, Garching, Germany
BDA	Big Data Analytics
BDEC	Big Data and Extreme-Scale Computing
BeeGFS	The Fraunhofer Parallel Cluster File System (previously acronym FhGFS). A high-performance parallel file system
BeeOND	BeeGFS-on-demand, parallel storage based on BeeGFS
BIC	Booster Interface Card (gateway nodes in DEEP)
BN	Booster Node (functional entity)
BoP	Board of Partners for the DEEP EST project
BSC	Barcelona Supercomputing Centre, Spain
BSCW	Repository used in the DEEP EST project to share all project documentation

C

CA	Consortium Agreement
Cassandra	The Apache Cassandra key-value store
CERN	European Organisation for Nuclear Research / Organisation Européenne pour la Recherche Nucléaire, International organisation
CLI	Command-Line Interface (a terminal/console-based user interface)
CM	Cluster Module: with its Cluster Nodes (CN) containing high-end general-purpose processors and a relatively large amount of memory per core
CME	Coronal Mass Ejections
CMS	Compact Muon Solenoid experiment at CERN's LHC
CN	Cluster Node (functional entity)
CNN	Convolutional Neural Networks
COTS	Commercial off-the-shelf
CPU	Central Processing Unit

CSIC	Spanish Council for Scientific Research

D

DAM	Data Analytics Module: with nodes (DN) based on general-purpose processors, a huge amount of (non-volatile) memory per core, and support for the specific requirements of data-intensive applications
DCDB	Data Centre Data Base (a tool developed in DEEP)
DDG	Design and Developer Group of the DEEP-EST project
DEEP	Dynamical Exascale Entry Platform (project FP7-ICT-287530)
DEEP-ER	DEEP - Extended Reach (project FP7-ICT-610476)
DEEP/-ER	Term used to refer jointly to the DEEP and DEEP-ER projects
DEEP-EST	DEEP - Extreme Scale Technologies
Dimemas	Performance analysis tool developed by BSC
DN	Nodes of the DAM
DNN	Deep neural network
DoW	Description of Work
DSL	Domain-specific Language
DRAM	Dynamic Random Access Memory. Typically describes any form of high capacity volatile memory attached to a CPU

E

EC	European Commission
EEHPC	Energy Efficient High Performance Computing
EEP	European Exascale Projects
EMP	EXTOLL Management Process
EPT4HPC	European Technology Platform for High Performance Computing
ESB	Extreme Scale Booster: with highly energy-efficient many-core processors as Booster Nodes (BN), but a reduced amount of memory per core at high bandwidth
EU	European Union
Exascale	Computer systems or Applications, which are able to run with a performance above 10^{18} Floating point operations per second
EXDCI	European Extreme Data & Computing Initiative
EXN	The EXTOLL Linux Ethernet emulation layer
EXTOLL	High speed interconnect technology for HPC developed by UHEI
Extrae	Performance analysis tool developed by BSC

F

fabri^3	Interconnect technology based on EXTOLL (pron. “Fabri-Cube”)
FFT	Fast Fourier Transform
FHG-ITWM	Fraunhofer Gesellschaft zur Foerderung der Angewandten Forschungs e.V., Germany
Flop/s	Floating point Operation per second
FP7	European Commission 7th Framework Programme
FPGA	Field-Programmable Gate Array, Integrated circuit to be configured by the customer or designer after manufacturing
FTI	Fault Tolerant Interface, a checkpoint/restart library

G

GCE	Global Collective Engine, a computing device for collective operations
GFlop/s	Gigaflop, 10^9 Floating point operations per second
GLA	General Learning Algorithms
GPU	Graphics Processing Unit
GROMACS	A toolbox for molecular dynamics calculations providing a rich set of calculation types, preparation and analysis tools

H

H2020	Horizon 2020
HBM	High Bandwidth Memory
HPC	High Performance Computing
HPDA	High Performance Data Analytics
HPDBSCAN	A clustering code used by UoI in the field of Earth Science
HW	Hardware
Hydra	The MPICH-native Process Manager

I

IC	Innovative Council
I^{2}C	Inter-Integrated Circuit computer bus
IB	see InfiniBand
IDC	International Data Corporation
InfiniBand	A networking communication standard for HPC clusters
InfluxDB	A time series database engine
Intel	Intel Germany GmbH, Feldkirchen, Germany
I/O	Input/Output. May describe the respective logical function of a computer system or a certain physical instantiation
IP	Intellectual Property

IPMI	Intelligent Platform Management Interface
iPic3D	Programming code developed by the KULeuven to simulate space weather
ISO	International Organisation for Standardisation

J

JLESC	Joint Laboratory for Extreme Scale Computing
JUBE	Jülich Benchmarking Environment
JUELICH	Forschungszentrum Jülich GmbH, Jülich, Germany
JURECA	Jülich Research on Exascale Cluster Architectures

K

KNL	Knights Landing, second generation of Intel® Xeon Phi (TM)
KNH	Knights Hill, next generation of Intel® Xeon Phi (TM)
KULeuven	Katholieke Universiteit Leuven, Belgium

L

LHC	Large Hadron Collider (LHC), the world's most powerful accelerator providing research facilities for High Energy Physics researchers across the globe
libNAM	Software layer for accessing and managing NAM (Network Attached Memory) modules
LLNL	Lawrence Livermore National Laboratory
LOFAR	Low-Frequency Array, an instrument for performing radio astronomy built by ASTRON

M

Megware	Megware Computer Vertrieb und Service GmbH, Chemnitz, Germany
MHD	Magneto-hydrodynamics
Mont-Blanc	European scalable and power efficient HPC platform based on low-power embedded technology
MoU	Memorandum of Understanding
MPI	Message Passing Interface, API specification typically used in parallel programs that allows processes to communicate with one another by sending and receiving messages
MPICH	MPI implementation maintained by Argonne National Laboratory

MSA	Modular Supercomputer Architecture
MUSIC	Multisimulation Coordinator (MPI-based library for coupled codes)
MQTT	Message Queuing Telemetry Transport (a publisher/subscriber-based messaging protocol)

N

NAM	Network Attached Memory
NCSA	National Centre for Supercomputing Applications, Bulgaria
NEST	Widely-used, publically available simulation software for spiking neural network models developed by NMBU
NF	Network Federation within the DEEP EST prototype
NMBU	Norwegian University of Life Sciences, Norway
NN	Neural Network
NUMA	Non-Uniform Memory Access
NV-DIMM	Non-Volatile Dual In-line Memory Module
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
NVRAM	Non-Volatile Random-Access Memory

O

OA	Open Access
ODC	Other direct costs
OGC	Open Geospatial Consortium
OmpSs	BSC's Superscalar (Ss) for OpenMP
Omni-Path Architecture	Communication architecture owned by Intel
OPA	see Omni-Path Architecture
OpenCL	Open Computing Language, framework for writing programs that execute across heterogeneous platforms
openHPC	A community effort that is initiated from a desire to aggregate a number of common ingredients required to deploy and manage HPC Linux clusters
OpenMP	Open Multi-Processing, Application programming interface that support multi-platform shared memory multiprocessing
Open MPI	MPI implementation maintained by the Open MPI Project
ORTE	Open MPI Runtime Environment (i.e. a Process Manager)

P

ParaStation	Software for cluster management and control developed by JUELICH and its linked third party ParTec
Paraver	Performance analysis tool developed by BSC
ParTec	ParTec Cluster Competence Center GmbH, Munich, Germany. Linked third Party of JUELICH in DEEP EST
PCIe	Peripheral Component Interconnect Express (a high-speed serial computer expansion bus standard)
PDU	Power Distribution Unit
PFlop/s	Petaflop, 10^{15} Floating point operations per second
Phi	see Xeon Phi
PI	Principal Investigator
piSVM	Parallel classification algorithm
PME	Particle mesh Ewald
PMI	Process Management Interface
PMT	Project Management Team of the DEEP-EST project
POSIX	Portable Operating System Interface
PRACE	Partnership for Advanced Computing in Europe (EU project, European HPC infrastructure)

Q

R

R&D	Research and Development
RAM	Random-Access Memory
RAS	Reliability, Availability, Serviceability
RDA	Research Data Alliance
RDMA	Remote Direct Memory Access / Remote DMA-based Memory Access
RDP	Reliable Datagram Protocol
REST	Representational State Transfer (an interface for web services)
RM	Resource Manager
RMA	Remote Memory Access
RMI	Remote Method Invocation
RML	Risk management list used in the DEEP-EST project

S

SCR	Scalable Checkpoint/Restart. A library from LLNL

SDV	Software Development Vehicle: HW systems to develop software in the time frame where the DEEP-EEST prototype is not yet available
SIMD	Single Instruction Multiple Data
SIONlib	Parallel I/O library developed by Forschungszentrum Jülich
SKA	Square Kilometer Array
SKU	Stock Keeping Unit
SLURM	Job scheduler that will be used and extended in the DEEP-EST prototype
Slurm	MHD code developed by KULeuven
SME	Small and Medium Enterprises
SMP	Symmetric Multi-Processing
SNMP	Simple Network Management Protocol
SRA	Strategic Research Agenda prepared by ETP4HPC
SSSM	Scalable Storage Service Module
STEM	Science, technology, engineering and mathematics
STS	Satellite time series
SW	Software

T

TCP/IP	Transmission Control Protocol and the Internet Protocol (a protocol family)
TensorFlow	Open-source software library for dataflow programming
TFlops	Teraflop, 10^{12} Floating point operations per second
ThinkParQ	Spin-off company of FHG ITWM
Tk	Task, Followed by a number, term to designate a Task inside a Work Package of the DEEP-EST project
ToW	Team of Work Package leaders of the DEEP-EST project
TRL	Technology Readiness Levels

U

UEDIN	University of Edinburgh, UK
UHEI	Ruprecht-Karls-Universitaet Heidelberg, Germany
UI	User Interface
UoI	Háskóli Íslands University of Iceland, Iceland
UPC	Universitat Politècnica de Catalunya. Barcelona, Spain

V

W

WLCG	Worldwide LHC Computing Grid
WP	Work package

X

x86	Family of instruction set architectures based on the Intel 8086 CPU
Xeon	Non-consumer brand of the Intel® x86 microprocessors (TM)
Xeon Phi	Brand name of the Intel® x86 manycore processors (TM)

Y

Z

Bibliography

[1] The Message Passing Interface Forum: *MPI: A Message-Passing Interface Standard – Version 3.1*, June 2015

[2] Kielmann, Thilo and Hofman, Rutger and Bal, Henri E. and Plaat, Aske and Bhoedjang, Raoul: *MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems*, in Proceedings of the seventh ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, vol. 34, num. 8, pp. 131–140, 1999

[3] Simon Pickartz and Carsten Clauss and Stefan Lankes and Antonello Monti: *Enabling Hierarchy-aware MPI Collectives in Dynamically Changing Topologies*, in Proceedings of EuroMPI/USA'17, Chicago, September 2017, pp. 25–28,
https://doi.org/10.1145/3127024.3127031

[4] George Bosilca, Thomas Herault, Ala Rezmerita, and Jack Dongarra: *On Scalability for MPI Runtime Systems*, in Proceedings of the 13th IEEE International Conference on Cluster Computing (CLUSTER), IEEE Computer Society, pages 187–195, September 2011,
http://ieeexplore.ieee.org/document/6061054/

[5] TensorFlow: open-source software library for dataflow programming
https://www.tensorflow.org/

[6] Keras: a high-level neural networks API, written in Python
https://keras.io/

[7] BigDL: Distributed Deep Learning on Apache Spark
https://github.com/intel-analytics/BigDL

[8] Apache Spark: a fast and general engine for large-scale data processing
https://spark.apache.org/

[9] OmpSs-2: The OmpSs-2 specification
https://pm.bsc.es/ompss-2-docs/spec/

[10] Best Practice Guide for Writing MPI+OmpSs Interoperable Programs
https://www.intertwine-project.eu/api-combinations

[11] F. Sainz and J. Bellon and V. Beltran and J. Labarta: *Collective Offload for Heterogeneous Clusters*, in Proceedings of the 22nd International Conference on High Performance Computing (HiPC), IEEE Computer Society, pages 376-385, December 2015
http://ieeexplore.ieee.org/document/7397653/

[12] Optimize performance of Python with integrated libraries and parallelism techniques
https://software.intel.com/en-us/distribution-for-python