H2020-FETHPC-01-2016

DEEP-EST

DEEP Extreme Scale Technologies

Grant Agreement Number: 754304

D6.3

Complete Programming Environment Implementation

Final

Version:	1.0
Author(s):	V. Beltran (BSC), P. Martinez (BSC)
Contributor(s):	C. Clauß (ParTec), K. Keller (BSC), L. Bautista (BSC), J. Roob (ITWM), P. Reh (ITWM), L. Montigny (Intel), B. Steinbusch (JUELICH)
Date:	31.12.2019

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:	D6.3
	Deliverable Nature:	Report
	Deliverable Level: PU*	**Contractual Date of Delivery:** 31.12.2019
		Actual Date of Delivery: 31.12.2019
	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:** Complete Programming Environment Implementation	
	ID: D6.3	
	Version: 1.0	**Status:** Final
	Available at: http://www.deep-projects.eu/	
	Software Tool: LaTeX	
	File(s): DEEP-EST_D6.3_Complete_Programming_Environment_Impl.pdf	
Authorship	**Written by:**	V. Beltran (BSC), P. Martinez (BSC)
	Contributors:	C. Clauß (ParTec), K. Keller (BSC), L. Bautista (BSC), J. Roob (ITWM), P. Reh (ITWM), L. Montigny (Intel), B. Steinbusch (JUELICH)
	Reviewed by:	J. De Amicis (JUELICH) N. Eicker (JUELICH)
	Approved by:	BoP/PMT

Document Status Sheet

Version	Date	Status	Comments
1.0	31.12.2019	Final	

Document Keywords

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

Table of Contents

List of Figures

List of Tables

Executive Summary

This deliverable presents the programming environment developed to support the Modular Supercomputing Architecture (MSA) proposed in the DEEP-EST project. This document is based on the D6.2 deliverable, which has been updated with the latest developments done in each task. The original work stated in deliverable D6.1 has been refined and adapted based on application requirements, as well as on the new ESB design based on GPUs. The goal of the programming environment is twofold: Firstly, to facilitate the development of new applications and 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), the Extreme Scale Booster (ESB) and the Data Analytics Module (DAM). Although all the main developments planned for this Work Package have already been completed, the content of this deliverable will be updated to reflect any further development, tuning or optimisation done until the end of the project.

1 Introduction

Deliverable 6.1 presented the programming environment to be developed and adapted according to the requirements of the Modular Supercomputing Architecture (MSA) as described in D3.1 *System Architecture*, as well as to the initial requirements of the applications. In Deliverable 6.2 we described the overall programming environment including the different components and features defined in the initial document. We also highlighted any deviation from the original plan as required to add better support for the new ESB design. 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, while in others, modifications of the applications will be needed. In this Deliverable (D6.3) we are updating the previous one to reflect the new developments done and the status of the different components.

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

2 ParaStation MPI

2.1 Modularity-awareness

In order to accommodate the modular nature of the DEEP-EST system also on MPI level, ParaStation MPI has been extended by a new feature set called *topology awareness* that can be enabled at configuration time of the MPI library:

```
--with-topology-awareness
```

When enabled, each process rank within an MPI session gets provided by the additional information about its node and module affinity within the MSA. Following a generic approach, the required information can be queried through the MPI interface, for example, from the resource manager or any other subsequent entity that possesses a global view on the system. In doing so, the node and module affinity is internally handled in terms of IDs (hence *node IDs* and *module IDs*, respectively) so that processes with identical IDs belong logically to the same affiliation. For developing, benchmarking and debugging purpose, this topology awareness can also be controlled explicitly by the user in terms of the following environment variables:

```
PSP_MSA_AWARENESS=0|1       # Disable/enable the topology awareness on module level
PSP_MSA_AWARE_COLLOPS=0|1   # Disable/enable the usage of hierarchical collectives on module level
PSP_MSA_MODULE_ID=${my_id}  # For explicitly setting the (logical) MSA module affinity
```

2.1.1 Support on application level

For making MPI applications that run across different modules aware of the underlying topology, a mechanism for passing the above-mentioned affiliation information also to the application level is required. Obviously, this interface should on the one hand be aligned with the MPI standard, while on the other hand its usage should be as simple as possible. For that reason, we have leveraged the module identifiers from above in two ways for the realisation of such an interface. The first way is to let the application query for the module ID explicitly via an MPI info object, and the second way is to adapt the applications data flow and communication patterns implicitly by using modularity-aware MPI communicators that reflect the underlying topology.

Querying the local module ID via MPI_INFO_ENV

According to the MPI standard, an application can access a certain set of environmental information by querying the predefined MPI info object `MPI_INFO_ENV`. Just like other info objects, this predefined one contains key / value pairs in terms of character strings that can be used to represent arbitrary information. However, the MPI standard also defines a likewise predefined set of *keys* for this object that represent the arguments used for starting the MPI session. By extending this predefined set by a new info key named `msa_module_id` in ParaStation MPI, an MSA-aware application can easily query for this and may then adapt its program flow accordingly, as it is shown in Figure 1. (Please note that this key was previously named `deep_module_id`, see Deliverable 6.2. However, since the feature set of topology awareness has by now been merged back into the master branch of

ParaStation MPI, a renaming to `msa_module_id` as a more general and MSA-generic key took place.) As one can see, leveraging the generic mechanism of a string-coded key / value pair for passing the information about the module affinity for each process strictly preserves API compatibility with the MPI standard. That means that an MSA-aware application that utilises this new feature for querying the module affiliation may still be compiled and run on non-modular systems by just checking for the absence of the `msa_module_id` key in `MPI_INFO_ENV` (see Line 6 and 10 in Figure 1).

```
1  int module_id;
2  char value[MPI_MAX_INFO_VAL];
3
4  MPI_Info_get(MPI_INFO_ENV, "msa_module_id", MPI_MAX_INFO_VAL, value, &flag);
5
6  if (flag) { /* This MPI environment is modularity-aware! */
7
8    my_module_id = atoi(value); /* Determine the module affinity of this process. */
9
10 } else { /* This MPI environment is NOT modularity-aware! */
11
12   my_module_id = 0; /* Assume a flat topology for all processes. */
13 }
```

Figure 1: Example code for querying the MSA module ID via the `MPI_INFO_ENV` object.

Modularity reflecting MPI communicators

For adapting an MPI application to modular topologies, not only a steering of the program flow is required but also an adjustment of the communication patterns becomes necessary. Although such an adjustment can be conducted by means of explicitly distributing the information about the module affinities among all process ranks within the session, using dedicated MPI *communicators* that reflect the underlying topology would obviously be preferable as it would in particular increase readability and comprehensibility of the MSA-aware application code. Fortunately, the creation of new communicators by means of some kind of affiliation identifiers is quite straight-forward in MPI and `MPI_Comm_split` is here the right function of choice. This is because this function partitions the group of processes of an existing communicator into disjoint subgroups by means of so-called *colours*, which are actually integer values to be passed by each process during the collective function call. In fact, since each subgroup contains afterwards all processes of the same colour, using the module ID as the colour would create one new communicator for each module involved. According to this, each process within such a cross-module MPI session would then have the knowledge about its own position within the MSA in terms of its module ID and its module-local rank, as it is shown here in Figure 2.

In addition, for making things even easier for the application programmer, we have also implemented a new *split type* for the related `MPI_Comm_split_type` function that then makes internally use of the module identifier: `MPIX_COMM_TYPE_MODULE`. However, as the compiler/linker symbol of this new split type is not standardized, using it at application level makes the MPI program less portable as it is then no longer standard compliant.

```
1  MPI_Comm_split(MPI_COMM_WORLD, my_module_id, 0, &module_local_comm);
2
3  /*    After the split call, module_local_comm contains from the view of each
4   *    process all the other processes that belong to the same local MSA module.
5   */
6
7  MPI_Comm_rank(module_local_comm, &my_module_local_rank);
8
9  printf("My module ID is %d and my module-local rank is %d\n", my_module_id, my_module_local_rank);
```

Figure 2: Creation of a sub-communicator by using the moduel ID as *colour* in `MPI_Comm_split`.

2.1.2 Modularity-aware collective MPI operations

Modularity-awareness for collective MPI operations has been implemented for ParaStation MPI in accordance with the approach being described in Section 2.2.1 of Deliverable D6.1. Thus, the existing infrastructure of MPICH (i. e, the upper layer of ParaStation MPI) for the exploitation of SMP-awareness can be leveraged for also realising an appropriate modularity-awareness. For doing so, ParaStation MPI has to be built with its topology awareness enabled, as it has been introduced at the beginning of Section 2.1. The related code within MPICH can then use the topology information from ParaStation for the creation of appropriate shadow communicators for each of the common MPI communicators that are used at application level. These shadow communicators then represent the communication domains of the different MSA modules — actually quite similar to the modularity reflecting communicators as introduced in Section 2.1.1, but now transparent for the application.

The collective operations are then conducted in a hierarchical manner where the intra- and inter-module phases are strictly separated:

1. First do all module-internal gathering and/or reduction operations — if required.
2. Then perform the inter-module operation with only one process per module being involved. (These processes are so-called *local leaders* having the local rank 0.)
3. Finally, distribute the data within each module in a strictly module-local manner.

This approach is here exemplarily shown in Figure 3 for a Broadcast operation with nine processes and three modules. As one can see, the first step is to send the message from the root rank to the module-local leader. The second step is then the inter-module broadcast between the local leaders. And in the third step, the local leaders distribute the message within their modules. Besides Broadcast, the following collective operations of MPICH and ParaStation MPI are currently provided with this topology awareness and hence perform their operations in an analogous manner:

- `MPI_Bcast` / `MPI_Ibcast`
- `MPI_Reduce` / `MPI_Ireduce`
- `MPI_Allreduce` / `MPI_Iallreduce`
- `MPI_Scan` / `MPI_Iscan`
- `MPI_Barrier`

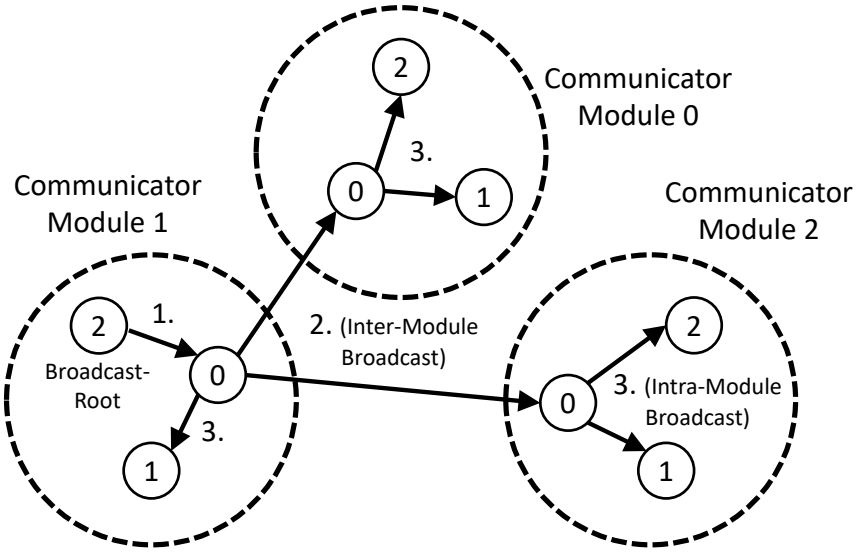


Figure 3: Example of a Broadcast pattern with modularity awareness enabled.

2.2 MPI Support for NAM-related RMA Operations

One distinct feature of the DEEP-EST prototype is the Network Attached Memory (NAM) and a new generation of the NAM library (called libNAM2) will soon be available. This library features a low-level API for accessing the NAM via Put / Get operations from every node within the EXTOLL network. However, for making the NAM programming more convenient and / or familiar for MPI application developers, Task 6.1 developed an interface for accessing the NAM also via MPI. This way, application programmers are able to use well known MPI functions (in particular those of the MPI RMA interface) for accessing NAM regions quite similar to other remote memory regions in a standardised (or at least harmonised) fashion under the single roof of an MPI world. In doing so, we followed an approach sticking to the current MPI standard as close as possible avoiding the introduction of new API functions wherever possible. Although the second generation of the NAM library is currently only available as an API skeleton, Task 6.1 has already prepared a user manual describing the semantics for accessing NAM memory via an extended MPI interface. Additionally, we developed a shared-memory-based implementation of this interface in ParaStation MPI that allows for emulating the NAM memory by using persistent shared-memory segments on the compute nodes instead.

2.2.1 Allocating and accessing NAM regions via MPI

The main issue when mapping the libNAM2 API onto the MPI RMA interface is the fact that MPI assumes all target and / or origin memory regions for RMA operations to be always associated with an MPI process being the owner of that memory. As a result, remote memory regions are always addressed by means of a process *rank* (plus handle, which is the respective window object, plus offset) in an MPI world. In contrast, the libNAM2 API merely requires an opaque handle for addressing

the respective NAM region (plus offset). Therefore, a mapping between remote MPI ranks and the remote NAM memory is required and this correlation is achieved by sticking to the notion of *ownership*, in the sense that definite regions of the NAM memory space are logically assigned to particular MPI ranks.

.

For requesting NAM memory via MPI, the well-known `MPI_Win_allocate` function has been extended. According to the MPI standard, this function allocates a local memory region at each process of the calling group and returns a pointer to this region as well as a window object that can then be used to perform RMA operations. For requesting NAM regions instead, we leverage the info argument for telling the MPI library to do the following. When setting the key / value pair `deep_mem_kind = deep_nam`, the MPI-internal memory management determines an available and probably contiguous NAM segment within the NAM allocation of the respective job (see Figure 4). This segment then backs the memory space of this new RMA window. Therefore, the segment will naturally be subdivided in terms of the *np* NAM regions (with *np* = number of processes in `comm`) that form the RMA window from the application's perspective.

```
1 MPI_Info_create(&info);
2 MPI_Info_set(info, "deep_mem_kind", "deep_nam");
3 MPI_Win_allocate(sizeof(int)*INTS_PER_PROC, sizeof(int), info, comm, NULL,
4                  &win);
```

Figure 4: The allocation of a NAM region by means of an MPI window object.

2.2.2 Releasing and Managing NAM Memory

NAM memory that has been allocated via `MPI_Win_allocate` will be freed — if not persistent, see below — by the collective call of `MPI_Win_free` and allocating and freeing such NAM regions repeatedly are valid operations in the scope of the granted NAM resources for a job. For that reason, ParaStation MPI has to implement an additional memory management layer on top of the NAM memory management as it is provided by the interplay between Slurm and the global NAM manager. That way, MPI applications are allowed to request for a certain amount of NAM space and that — as long as the granted limit of a job is not exceeded — subsequent requests and releases of non-persistent NAM regions can randomly be conducted via `MPI_Win_allocate` and `MPI_Win_free`. However, from the global resource manager's point of view, granted NAM resources are revoked either when the related job allocation ends or when a (persistent) NAM reservation is deleted.

2.2.3 Handling of persistent NAM regions in MPI

A central use-case for the NAM in DEEP-EST is the idea of facilitating workflows between different applications and / or application steps. For doing so, the data once put into NAM memory shall later be re-usable by other MPI applications. As a result, a mechanism is needed that supports denoting NAM regions — and hence also their related MPI windows — as *persistent*. In our implementation, this persistence is ensured if a window is allocated with an info object containing `deep_mem_kind`

= deep_nam_persistent as a key / value pair. If the creation of the persistent NAM window was successful, the related NAM regions become addressable as a joint entity by means of a logical *port name*. This port name can then be retrieved afterwards by querying the info object attached to that window via the info key deep_win_port_name.

Obviously, there needs to be a way for subsequent MPI sessions to attach to the persistent NAM regions that have been created previously by other MPI sessions. In ParaStation MPI, this can be achieved by means of an MPI_Comm_connect call being normally used for establishing communication between distinct MPI sessions. When passing a valid port name of a persistent NAM window plus an info argument with the key deep_win_connect and the value true (see Line 3 in Figure 5), this function returns an inter-communicator that then serves for accessing the remote NAM memory regions.

This approach retains the original segmentation of the NAM window, i. e., the window is still divided (and thus addressable) in terms of the MPI ranks of that process group that created the window before. Therefore, a call to MPI_Comm_remote_size on the returned inter-communicator reveals the former number of processes in that group. For actually creating the local representative for the window in terms of an MPI_Win datatype, we decided to alienate the MPI_Win_create_dynamic function with the inter-communicator as the input and the window handle as the output parameter (see Line 6 in Figure 5). (In fact, this approach "hijacks" the MPI_Win_create_dynamic function and its original purpose, but as its signature matches also to our purpose, we can go here without introducing new API functions in addition to MPI.)

```
1 MPI_Info_create(&win_info);
2 MPI_Info_set(win_info, "deep_win_connect", "true");
3 MPI_Comm_connect(port_name, info, 0, MPI_COMM_WORLD, &inter_comm);
4
5 MPI_Comm_remote_size(inter_comm, &group_size);
6 MPI_Win_create_dynamic(MPI_INFO_NULL, inter_comm, &win);
```

Figure 5: Connection to a NAM region of a different/previous MPI session and creation of the local representative for a memory window.

2.2.4 Querying information about remote window regions

After determining the size of the former process group via MPI_Comm_remote_size, there is still a demand for getting the information about the remote region sizes as well as the related unit sizes for the displacement. For doing so, ParaStation MPI provides the following new window attributes for querying this information: MPIX_WIN_SIZES and MPIX_WIN_DISP_UNITS. In contrast to their regular counterparts (these are MPI_WIN_SIZE and MPI_WIN_DISP_UNIT), these new attributes will return *arrays* filled with the related information to be indexed by the remote ranks, as shown in Figure 6.

```
1  MPI_Win_get_attr(win, MPIX_WIN_SIZES, &win_size_attr, &flag);
2  for(int i=0; i<group_size; i++) printf("[%d] segment size: %lld\n", i, win_size_attr[i]);
3
4  MPI_Win_get_attr(win, MPIX_WIN_DISP_UNITS, &win_disp_unit_attr, &flag);
5  for(int i=0; i<group_size; i++) printf("[%d] displacement unit %d\n", i, win_disp_unit_attr[i]);
```

Figure 6: Example code showing how to query information about remote NAM window regions.

2.3 CUDA-Awareness in MPI

On the one hand, the envisioned GPU-centric programming of the Extreme Scale Booster implies that the MPI programming interface is required to act as a central but lightweight entry point triggering and steering all GPU-related communication operations on the underlying hardware. On the other hand, CUDA-awareness refers to the idea that MPI applications may directly pass pointers to memory regions located on the GPU (the so-called *device* memory) to MPI functions such as `MPI_Send` or `MPI_Recv`. A non CUDA-aware MPI library would fail in such a case as the GPU's memory cannot be accessed directly. Instead, these memory regions have to be transferred explicitly by the application via special routines to the host memory beforehand — a step that is called *staging*. In contrast to this, a CUDA-aware MPI library recognises that a pointer is associated with a buffer within the device memory and can then perform the staging internally. Moreover, as such an MPI library is able to detect and to deal with device pointers, it may also strive for an exploitation of special GPU-related transport capabilities like GPUDirect, if provided by the underlying hardware and the runtime.

In fact, ParaStation MPI supports CUDA-awareness already at different levels. On the one hand, the usage of device pointers as arguments for send and receive buffers when calling MPI functions is supported. On the other hand, if an interconnect technology provides features like GPUDirect, ParaStation MPI is in principle able to bypass its own staging mechanism and to forward the required information for the exploitation of the offered hardware capabilities.

On the DEEP system, the CUDA awareness can be enabled by loading a module that links to a dedicated ParaStation MPI library providing CUDA support:

```
module load GCC
module load ParaStationMPI/5.4.0-1-CUDA
```

As CUDA-awareness might impact the MPI performance on systems parts where CUDA is not used, it might be useful (and the other way around necessary) to disable/enable the CUDA-awareness by setting the following environment variable: `PSP_CUDA=0|1`.[1]

Currently (effective December 2019), ParaStation MPI supports CUDA-awareness for Extoll just from the semantic-related point of view. Hence, the usage of device pointers as arguments for send and receive buffers when calling MPI functions is supported, but by an explicit staging when Extoll is used. The reason for this is that the Extoll runtime up to now does not support GPUDirect — but EXTOLL is currently working on this and as soon as GPUDirect will be supported by Extoll, this will also be integrated and enabled in ParaStation MPI. However, for InfiniBand communication, ParaStation MPI is already GPUDirect enabled and the following environment variables can be used to steer its usage:

[1] Please note that the additional code for CUDA awareness may slightly impact the performance even if `PSP_CUDA=0` is set and that hence using a non CUDA-aware module of ParaStation MPI instead might be preferable.

```
PSP_CUDA=0|1                  # Enable/Disable CUDA-awareness for psmpi/pscom (default = 0)
PSP_CUDA_AWARE_SHM=0|1        # Enable/Disable the CUDA-awareness of the shm plugin (default = 1)
PSP_CUDA_AWARE_OPENIB=0|1     # Enable/Disable the CUDA-awareness of the openib plugin (default = 1)
PSP_CUDA_AWARE_UCP=0|1        # Enable/Disable the CUDA-awareness of the ucp plugin (default = 1)
PSP_CUDA_ENFORCE_STAGING=0|1 # Enable/Disable the CUDA-awareness on the plugin level, i.e., enforce
                              # pscom-internal staging (default = 0 no staging of CUDA-aware plugins)
```

3 The OmpSs-2 Programming Model

The OmpSs-2 programming model implements features for improved tasking programmability and performance, support for message-passing libraries (MPI), as well as, for improved GPU programming of the modular supercomputer architecture (MSA).

We have divided this chapter into several sections. The first section presents NBody, a use-case application for feature demonstration. Section 3.2 describes features of the tasking model to improve performance and programmability. Later, several features are described in sections 3.3 and 3.4 that can leverage data-flow programming to simplify MPI and GPU programming on the MSA. Lastly several pointers are provided to documentation and manuals on features and usage on the SDV.

For a detailed overview of OmpSs-2 programming model, please refer to Deliverable 6.1 or to documentation on-line [1].

3.1 NBody example

NBody is a numerical simulation code that computes the interaction between discrete bodies in space. Such codes are often used in physics or astronomy. The presented code represents a time-discretized approximation for a solution after k time steps.

Figure 7a shows a sequential version of the simulation loop where two methods, `calculate_forces` and `update_particles` are called in each time step. `calculate_forces` computes the gravitational force vector in three dimensions for each particle by applying Newton's formula of universal gravitation on each particle pair in the simulation space. Once the gravitation vector is computed, acceleration vector is computed relative to the time step interval and the particle mass. Then a displacement vector is derived from the acceleration and velocity vectors. Finally, update_particles updates the velocity vector (using force, mass and timestep) and then updates particles' position using the displacement vector computed from velocity and timestep.

Figure 7b shows an implementation with MPI. The MPI-parallel implementation partitions particles per ranks and then on each rank data is partitioned per blocks (blocking). Since particles are distributed over `rank_size` nodes, forces can be computed only between local particles residing on that rank. Once those forces are computed, particles are sent to the next neighboring rank (`(rank+1)%rank_size`, implementing a circular shift). The MPI send and receive operations are implemented in the `exchange_particles` method. This method is called `rank_size-1` times. To allow concurrent send and receive operations, two scratch memories `remote1` and `remote2` are used.

Figure 7c illustrates the implementation of each principal function as shown previously in Figure 7b. It can be seen that each method calls a subroutine for each block and that each such block computation is declared as a task. Tasking adds node-level parallelism to the application.

It is interesting to point out that `exchange_particles` defines an *inout* dependency over a variable `serialize`. This dependency serializes the execution of all `exchange_particles_block` tasks. This corresponds to traditional MPI+OpenMP hybrid programming where the programming

[1] `https://github.com/bsc-pm/ompss-2-releases`

```
1 void nbody_solve(nbody_t *nbody, ...){
2   particles_block_t *particles = nbody->particles;
3   forces_block_t *forces = nbody->forces;
4   for (int t = 0; t < timesteps; t++) {
5     calculate_forces(forces, particles, num_blocks);
6     update_particles(particles, forces, num_blocks, time_interval);
7   }
8 }
```

(a) Serial NBody code showing the simulation loop.

```
1 void nbody_solve(nbody_t *nbody, ...){
2   particles_block_t *local = nbody->local, *remote1 = nbody->remote1, *remote2 = nbody->remote2;
3   forces_block_t *forces = nbody->forces;
4   for (int t = 0; t < timesteps; t++) {
5     particles_block_t *sendbuf = local;
6     particles_block_t *recvbuf = remote1;
7     for (int r = 0; r < rank_size; r++) {
8       calculate_forces(forces, local, sendbuf, num_blocks);
9       if (r < rank_size - 1) {
10        exchange_particles(sendbuf, recvbuf, num_blocks, rank, rank_size);
11      }
12      particles_block_t *aux = recvbuf;
13      recvbuf = (r != 0) ? sendbuf : remote2;
14      sendbuf = aux;
15    }
16    update_particles(local, forces, num_blocks, time_interval);
17  }
18  MPI_Barrier(MPI_COMM_WORLD);
19 }
```

(b) MPI-parallel simulation loop showing particle exchange and the use of two scratch variables.

```
1 void calculate_forces(forces_block_t *forces, particles_block_t *block1, particles_block_t *block2){
2   for (int i = 0; i < num_blocks; i++) {
3     for (int j = 0; j < num_blocks; j++) {
4       #pragma oss task in(*(block1+i), *(block2+j)) inout(*(forces+i)) label(calculate_fB)
5       calculate_forces_block(forces+i, block1+i, block2+j);
6 }}}
7
8 void exchange_particles(particles_block_t *sendbuf, particles_block_t *recvbuf, ...){
9   for (int i = 0; i < num_blocks; i++) {
10    #pragma oss task in(*(sendbuf+i)) out(*(recvbuf+i)) inout(*serialize) label(exchange_pB)
11    exchange_particles_block(sendbuf+i, recvbuf+i, i, rank, rank_size);
12 }}
13
14 void update_particles(particles_block_t *particles, forces_block_t *forces, ...){
15   for (int i = 0; i < num_blocks; i++) {
16     #pragma oss task inout(*(particles+i), *(forces+i)) label(update_pB)
17     update_particles_block(particles+i, forces+i, time_interval);
18 }}
```

(c) Three different methods called within the simulation loop show the blocked data organization on process level.

Figure 7: NBody simulation code represents a suitable scenario for performance optimization and programming model feature development.

model requires ordering to guarantee dead-lock free execution. This issue is discussed further in section 3.3.

The full code of the NBody simulation can be accessed on-line[2].

3.2 Improved tasking

Node-level parallelism is important as it allows to maintain the application's degree of concurrency while reducing the number of processes and consequently the memory footprint and related overheads. Tasking is a popular approach to implement node-level parallelism. It turns out that a top-down approach of adding tasks to an application is algorithmically meaningful and easy to implement. A top-down approach however requires an efficient support for task nesting.

Other advantages exist. Task nesting allows parallel task creation by the OmpSs-2 runtime. As task-dependencies are computed within the enclosing task, no synchronization of concurrent accesses to administrative data structures of the parent task is required. This improves runtime tasking performance. Consequently, nesting is an important design pattern to achieve scalable runtime behavior for large task numbers.

While nesting is beneficial, it can result in reduced concurrency as inner tasks that were created in different task nests never run in parallel due to the synchronized execution of the outer tasks. OmpSs-2 offers a new type of dependencies that allows to expose dependencies across nesting levels. This type is called *weak dependency*.

3.2.1 Weak dependencies

By defining a weak dependency [6] over a memory location (variable, or region) using the weak dependency clause in a task construct, the programmer expresses a dependency between inner tasks accessing the variable and outer tasks accessing that variable such that by disregarding the dependencies of the parent task, correct ordering of computation will be preserved.

Figure 8 shows an example for this dependency type in the Nbody application. In this case, the three principle methods of the simulation have been *taskified* using the task pragma.

Since all three tasks of the simulation loop are required to maintain ordering, tasking at this granularity does not allow for concurrency. However, using the *weakin*, *weakout* and *weakinout* clauses instead of the regular *in*, *out* and *inout* clauses, allows to expose dependencies of the inner tasks across nesting levels. Further, the control flow of the outer tasks (lines 6, 10 and 20) returns immediately after all inner tasks are instantiated, freeing up the stack quickly.

Figure 9 shows a visual representation of nested dependencies in a more complex scenario where four outer tasks create two inner tasks each. It can be seen that by using weak dependencies, the maximum degree of concurrency is 3 (9b), where tasks *T2.1*, *T2.2* and *T3.2* run in parallel. This is opposed to the achievable degree of concurrency of 2 in the original code (9a).

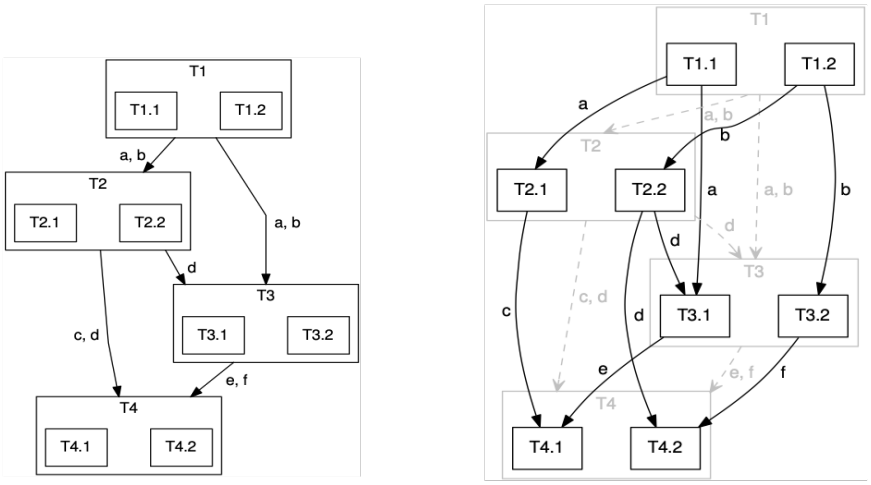
Support for weak dependencies is fully implemented and tested with example applications.

Other features for improved tasking are work-sharing tasks and support for array-type reductions.

[2] `https://pm.bsc.es/gitlab/ompss-2/examples/nbody`

```
1 void nbody_solve(nbody_t *nbody,...)
2   for (int t = 0; t < timesteps; t++) {
3     particles_block_t *sendbuf = local;
4     particles_block_t *recvbuf = remote1;
5     for (int r = 0; r < rank_size; r++) {
6     #pragma oss task weakin(local[0;num_blocks], sendbuf[0;num_blocks])\
7                      weakinout(forces[0;num_blocks]) label(calculate_forces)
8     calculate_forces(forces, local, sendbuf, num_blocks);
9     if (r < rank_size - 1) {
10      #pragma oss task weakin(sendbuf[0;num_blocks])\
11                       weakout(recvbuf[0;num_blocks]) label(exchange_particles)
12      exchange_particles(sendbuf, recvbuf, num_blocks, rank, rank_size);
13    }
14
15    particles_block_t *aux = recvbuf;
16    recvbuf = (r != 0) ? sendbuf : remote2;
17    sendbuf = aux;
18  }
19
20  #pragma oss task weakin(local[0;num_blocks]) weakinout(forces[0;num_blocks]) label(update_particles)
21  update_particles(local, forces, num_blocks, time_interval);
22 }
```

Figure 8: Nesting helps composability in applications and allows concurrent task instantiation, however, dependencies on coarser levels can limit concurrency.



(a) (b)

Figure 9: A weak dependency is a new dependency type that allows to extract concurrency and look-ahead across nesting levels. In this way, tasks *T2.1*, *T2.2* and *T.3.2* as shown in Figure 9b can be scheduled for concurrent execution.

3.2.2 Work-sharing tasks to exploit fine-grained structured-parallelism

Shared memory programming models such as OpenMP provides both worksharing and task constructs. The former relies on the efficient fork-join execution model to exploit structured parallelism; while the latter relies on fine-grained synchronization among tasks and a flexible data-flow execution model to exploit dynamic, irregular, and nested parallelism. On applications that show both structured and unstructured parallelism, both worksharing and task constructs can be combined. However, it is difficult to mix both execution models without penalizing the data-flow execution model. Hence, on many applications structured parallelism is also exploited using tasks to leverage the full benefits of a pure data-flow execution model. Indeed, task creation and management might introduce a non-negligible overhead that prevents the efficient exploitation of fine-grained structured parallelism, especially on many-core processors. To address this issue *worksharing tasks* are proposed. These are tasks that internally leverage worksharing techniques to exploit fine-grained structured loop-based parallelism.

Figure 10 shows the `update_particles_block` function of the NBody code as shown previously using the *for* construct to define a work-sharing task to exploit fine-grained structured parallelism.

```
1 void calculate_forces_block(forces_block_t *forces,
2         particles_block_t *block1, particles_block_t *block2){
3   ...
4   #pragma oss task for in(*block1, *block2)\
5   reduction(+:[blocksize]forces->x, [blocksize]forces->y, [blocksize]forces->z)
6   for (int i = 0; i < BLOCK_SIZE; i++) {
7     float fx, fy, fz, diffs[3];
8       for (int j = 0; j < BLOCK_SIZE; j++) {
9         force = f(block1, block2, i, j, diffs);
10        fx = force + diffs[0];
11        fy = force + diffs[1];
12        fz = force + diffs[2];
13      }
14      forces->x[i] += fx;
15      forces->y[i] += fy;
16      forces->z[i] += fz;
17 }}
```

Figure 10: The *task for* pragma allows the creation of a work-sharing tasks that will be executed concurrently by several CPUs

Conceptually, the work-sharing task defined in line 4 is similar to a regular task that inside contains a *paralell for* to exploit the parallel loop on line 6. The main difference between the approaches is that work-sharing tasks are optimised to avoid the implicit barriers of the OpenMP approach. Please refer to [7] for a detailed description and evaluation of work-sharing tasks.

The support for work-sharing task has been already included in the latest stable release of OmpSs-2.

3.2.3 Reduction support

Reductions support in OmpSs-2[3] allows the expression of parallel reductions over scalar- and array-types. This allows potentially lock-free, concurrent updates of a reduction variable. In OmpSs-2, the

[3] https://upcommons.upc.edu/handle/2117/129246

scope of a reduction starts at the first encounter of a task and finishes at either a task synchronisation point or at an unspecified point in time but before a data access of a task that is not participating in the reduction. The underlying runtime creates private copies of the variable or array that is used on the reduction. Then the runtime maximizes the reuse of these copies between tasks and nesting levels. Once the scope of a reduction is concluded, the runtime reduces private copies in parallel when possible.

Figure 10 shows the declaration of a reduction over three arrays representing the spacial dimensions of the force vector. The use of the *reduction* clause instead of an *inout* clause remove the serialization of all the tasks that contribute to calculate the final result of the `forces` array.

The support for scalar and array reduction has already been included in the latest stable release of OmpSs-2. The status of array reductions on CUDA kernels is discussed on Section 3.4.

3.3 Support of MPI and the MSA

Supporting hybrid MPI + X is beneficial as this approach allows to reuse existing MPI codes. MPI gives detailed control over data placement and synchronisation on process-level. However, strict ordering of work, manual synchronisation and a fork-join parallelism originating historically from threading control make achieving good performance with traditional MPI + X programming difficult.

The development of the Task-Aware MPI (TAMPI) library[4] was started in the context of the INTER-TWinE H2020 project. This library extends the functionality of standard MPI libraries by providing new mechanisms for improving the interoperability between parallel task-based programming models, such as OpenMP or OmpSs-2, and both blocking and non-blocking MPI operations. By following the MPI Standard, programmers must pay close attention to avoid deadlocks that may occur in hybrid applications (e.g., MPI + OpenMP) where MPI calls take place inside tasks. This is given by the out-of-order execution of tasks that consequently alter the execution order of the enclosed MPI calls. The TAMPI library ensures a deadlock-free execution of such hybrid applications by implementing a cooperation mechanism between the MPI library and the parallel task-based runtime system. Moreover, applications that rely on TAMPI do not require significant changes to allow the runtime to overlap the execution of computation and communication tasks. TAMPI provides two main mechanisms: the blocking mode and the non-blocking mode. The blocking mode targets the efficient and safe execution of blocking MPI operations (e.g., `MPI_Recv`) from inside tasks, while the non-blocking mode focuses on the efficient execution of non-blocking or immediate MPI operations (e.g., `MPI_Irecv`), also from inside tasks. On the one hand, TAMPI is currently compatible with two task-based programming model implementations: a derivative version of the LLVM OpenMP (yet to be released) and OmpSs-2. However, the derivative OpenMP does not support the full set of features provided by TAMPI. OpenMP programs can only make use of the non-blocking mode of TAMPI, whereas OmpSs-2 programs can leverage both blocking and non-blocking modes. On the other hand, TAMPI is compatible with mainstream MPI implementations that support the `MPI_THREAD_MULTIPLE` threading level, which is the minimum requirement to provide its task-aware features. The following sections describe in detail the blocking (OmpSs-2) and non-blocking (OpenMP & OmpSs-2) modes of TAMPI.

[4] Task-Aware MPI[8] `https://github.com/bsc-pm/tampi`

Blocking Mode (OmpSs-2)

The blocking mode of TAMPI targets the safe and efficient execution of blocking MPI operations (e.g., `MPI_Recv`) from inside tasks. This mode virtualizes the execution resources (e.g., hardware threads) of the underlying system when tasks call blocking MPI functions. When a task calls a blocking operation, and it cannot complete immediately, the underlying execution resource is prevented from being blocked inside MPI and it is reused to execute other ready tasks. In this way, the user application can make progress although multiple communication tasks are executing blocking MPI operations. This is done transparently to the user, that is, a task calls a blocking MPI function (e.g., `MPI_Recv`), and the call returns once the operation has completed as states the MPI Standard. This virtualization prevents applications from blocking all execution resources inside MPI (waiting for the completion of some operations), which could result in a deadlock due to the lack of progress. Thus, programmers are allowed to instantiate multiple communication tasks (that call blocking MPI functions) without the need of serializing them with dependencies, which would be necessary if this TAMPI mode was not enabled. In this way, communication tasks can run in parallel and their execution can be re-ordered by the task scheduler. This mode provides support for the following set of blocking MPI operations:

1. Blocking primitives: `MPI_Recv`, `MPI_Send`, `MPI_Bsend`, `MPI_Rsend` and `MPI_Ssend`.
2. Blocking collectives: `MPI_Gather`, `MPI_Scatter`, `MPI_Barrier`, `MPI_Bcast`, `MPI_Scatterv`, etc.
3. Waiters of a complete set of requests: `MPI_Wait` and `MPI_Waitall`.

Non-Blocking Mode (OpenMP and OmpSs-2)

The non-blocking mode of TAMPI focuses on the execution of non-blocking or immediate MPI operations from inside tasks. As the blocking TAMPI mode, the objective of this one is to allow the safe and efficient execution of multiple communication tasks in parallel, but avoiding the blocking of these tasks. The idea is to allow tasks to bind their completion to the finalization of one or more MPI requests. Thus, the completion of a task is delayed until (1) it finishes the execution of its body code and (2) all MPI requests that it bound during its execution are complete. Notice that the completion of a task usually implies the release of its dependencies, the freeing of its data structures, etc. For that reason, TAMPI defines two asynchronous and non-blocking functions named `TAMPI_Iwait` and `TAMPI_Iwaitall`, which have the same parameters as their standard synchronous counterparts `MPI_Wait` and `MPI_Waitall`, respectively. They bind the completion of the calling task to the finalization of the MPI requests passed as parameters, and they return "immediately" without blocking the caller. The completion of the calling task will take place once it finishes its execution and all bound MPI requests complete. Since they are non-blocking and asynchronous, a task after calling `TAMPI_Iwait` or `TAMPI_Iwaitall` passing some requests cannot assume that the corresponding operations have already finished. For this reason, the communication buffers related to those requests should not be consumed or reused inside that task. The proper way is to correctly annotate the communication tasks (the ones calling `TAMPI_Iwait` or `TAMPI_Iwaitall`) with the dependencies on the corresponding communication buffers, and then, annotating also the tasks that will reuse or consume the data buffers. In this way, these latter will become ready once the data buffers are safe to be accessed (i.e., once the communications have been completed). Defining the correct dependencies of tasks is essential to guarantee a correct execution order.

OmpSs-2 support for the TAMPI library that allows the inclusion of blocking and non-blocking MPI calls in tasks has been implemented and tested with different MPI libraries including ParaStation MPI on the DEEP-EST platform.

Examples

Figure 11b shows how non-blocking MPI calls are supported by using the `TAMPI_Iwaitall` API call. Here, the method `exchange_particles_block` sends and receives blocks of particles from neighboring ranks. By using `TAMPI_Iwaitall` once this task finalizes the execution of his body of code it will not release its dependencies until the associated requests (`requests[2]`) are also completed. In this way, the handling of explicit communications between nodes is naturally integrated with the dataflow execution model of OmpSs-2.

```
1 #pragma oss task in(*sendbuf) out(*recvbuf) label(exchange_particles_block)
2 void exchange_particles_block(const particles_block_t *sendbuf, particles_block_t *recvbuf, ...){
3   int src = MOD(rank - 1, rank_size);
4   int dst = MOD(rank + 1, rank_size);
5   int size = sizeof(particles_block_t);
6   if (rank % 2) {
7     MPI_Send(sendbuf, size, MPI_BYTE, dst, block_id+10, MPI_COMM_WORLD);
8     MPI_Recv(recvbuf, size, MPI_BYTE, src, block_id+10, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
9   } else {
10    MPI_Recv(recvbuf, size, MPI_BYTE, src, block_id+10, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
11    MPI_Send(sendbuf, size, MPI_BYTE, dst, block_id+10, MPI_COMM_WORLD);
12 }}
```

(a) OmpSs-2 support for blocking MPI calls.

```
1 #pragma oss task in(*sendbuf) out(*recvbuf) label(exchange_particles_block)
2 void exchange_particles_block(const particles_block_t *sendbuf, particles_block_t *recvbuf, ...){
3   ...
4   MPI_Request requests[2];
5   if (rank % 2) {
6     MPI_Isend(sendbuf, size, MPI_BYTE, dst, block_id+10, MPI_COMM_WORLD, &requests[0]);
7     MPI_Irecv(recvbuf, size, MPI_BYTE, src, block_id+10, MPI_COMM_WORLD, &requests[1]);
8   } else {
9     MPI_Irecv(recvbuf, size, MPI_BYTE, src, block_id+10, MPI_COMM_WORLD, &requests[1]);
10    MPI_Isend(sendbuf, size, MPI_BYTE, dst, block_id+10, MPI_COMM_WORLD, &requests[0]);
11  }
12  TAMPI_Iwaitall(2, requests, MPI_STATUSES_IGNORE);
13 }
```

(b) OmpSs-2 support for non-blocking calls.

Figure 11: The NBody `exchange_particles_block` uses the OmpSs-2 support for MPI to allow the inclusion of MPI calls in tasks while preserving correct ordering and absence of dead-locks.

3.3.1 Support of heterogeneous modules of the MSA

The OmpSs-2 programming model prototype foresees support for MSA by algorithmic design. In this way, it is up to the programmer to write code for the module type and to implement conditionals to differentiate between them. Figure 12 shows an example of the proposal using the NBody simulation

code from Figure 8. In this example the NODE_TYPE variable determines the control flow that is executed on the given node type. It can be seen that the algorithm differentiates between node types being IS_ESB and IS_CN. They denote the Extreme Scale Booster with GPUs and the Cluster Module node types. The control path that corresponds to the node type being the ESB includes calculate_forces as this computation will be offloaded to the GPU. Communication between nodes of different types is implemented with MPI in the send_forces and receive_forces methods. It is worthwhile mentioning that both intra-module and inter-module communications are done using TAMPI, so all the computations and communications inside and across different modules are orchestrated in a data-flow way. We explain support for accelerators in the next section.

```
1 for (int t = 0; t < timesteps; t++) {
2   if(NODE_TYPE == IS_ESB){ //compute on ESB node
3     particles_block_t *sendbuf = local;
4     particles_block_t *recvbuf = remote;
5
6     for (int r = 0; r < rank_size; r++) {
7       #pragma oss task...
8       calculate_forces(...);
9       if (r < rank_size - 1) {
10        #pragma oss task ...
11        exchange_particles(...);
12      }
13      ...
14    }
15    #pragma oss weakin(forces[0;num_blocks])
16    send_forces(forces, IS_CN)
17  }else if (NODE_TYPE == IS_CN){ //compute on COMPUTE NODE
18    #pragma oss weakout(forces[0;num_blocks])
19    receive_forces(forces, IS_ESB)
20    #pragma oss task ...
21    update_particles(...);
22  }
23 }
```

Figure 12: The support for heterogeneous nodes of the MSA can be accomplished algorithmically by checking application parameters, the environment or the MPI communicator type.

3.3.2 Future Work

The support of blocking and non-blocking MPI functions has already been tested and released. The next step will be to evaluate it at a large scale to really measure the potential benefits that could be expected from this approach on Exascale machines. Moreover, GPU Direct support over Extoll will be also evaluated once it becomes available. Finally, we will investigate how to extend TAMPI to support MPI one-sided primitives, which will be used to access the Network Attached Memory (NAM).

3.4 Support for accelerators

The OmpSs-2 prototype offers accelerator support through kernel offloading. In this approach the programmer annotate CUDA C kernels like regular tasks, which then can be invoked like regular functions. The OmpSs-2 runtime takes care of data movements and correct synchronization of host and device tasks and kernels following a data-flow execution model.

Figure 13 shows the `calculate_force_block_CUDA` kernel from the *NBody* application as shown in Figure 10. It is important to point out that the CUDA kernel code is located in a separate file that is compiled by the CUDA C compiler separately (`"kernel.cu"` as shown in in Figure 13a). In this example, the task annotation is added to the method declaration in the program header file (`nbody.h` as shown in Figure 13b). For completeness, the definition of the `forces_block_t` have been added to highlight that it is a *struct* of static arrays, thus suitable for host-device data movement. Data movement makes use of the CUDA unified memory. It is worth mentioning that the runtime has been extended to explicitly manage data transfers, so unified memory is no longer a hard requirement. The support of CUDA kernels without using unified memory will be available on the next stable release of OmpSs-2.

```
1 #include "nbody.h"
2 __device__
3 void calculate_forces_block_CUDA(forces_block_t *forces, particles_block_t *block1,
4 particles_block_t *block2){
5   /*CUDA code here*/
6 }
```

(a) CUDA kernel to compute the force vector in the NBody application.

```
1 typedef struct {
2   float x[BLOCK_SIZE]; /* x   */
3   float y[BLOCK_SIZE]; /* y   */
4   float z[BLOCK_SIZE]; /* z   */
5 } forces_block_t;
6 ...
7 #pragma oss task in(*block1, *block2) inout(*forces) device(cuda) ndrange(1, size, 128)
8 void calculate_forces_block_CUDA(forces_block_t *forces, particles_block_t *block1,
9      particles_block_t *block2);
```

(b) Header file (`nbody.h`) defining the `forces_block_t` type as well as a task for a GPU.

```
1 #include "nbody.h"
2 ...
3 void calculate_forces(forces_block_t *forces, ...){
4   for (int i = 0; i < num_blocks; i++) {
5     for (int j = 0; j < num_blocks; j++) {
6       calculate_forces_block_CUDA(forces+i, block1+i, block2+j);
7 }}}
```

(c) `calculate_forces`, the principal method from the simulation loop, creates a GPU task for each block.

Figure 13: Accelerator support in OmpSs-2 is implemented via kernel offloading where the OmpSs-2 runtime takes care of proper synchronization of kernels with host code as well as data transfers.

Currently we are working to provide support of array reductions on CUDA devices. In this way the example in Figure 13 could be rewritten to replace the *inout* clause on line 7 with a *reduction* clause. This change will allow the concurrent execution, on the same or on different GPUs, of several CUDA kernels that contribute to calculate the *forces* array.

3.5 Using OmpSs-2 on the SDV

To facilitate the use of the programming model including its tool chain for application developers, there are manuals and tutorials [5]. Examples showcase the use of all aforementioned programming features and include different examples including the presented NBody code. Furthermore, on the DEEP-EST TRAC [6] there is information on how to load modules, compile and trace applications on the SDV. This information is periodically updated to reflect any change on the software stack or the SDV system.

[5] https://pm.bsc.es/ompss-2
[6] https://deeptrac.zam.kfa-juelich.de:8443/trac/wiki/Public/User_Guide/OmpSs-2

4 Data Analytics programming model

The previous Deliverable D6.2 focused on collecting the frameworks and libraries used by the WP1 applications. This chapter describes the software installation workflow with EasyBuild in the first section followed by the list of programming models and frameworks installed on the DEEP-EST prototype. This section includes details about the Intel FPGA frameworks and the persistent memory toolkit. The list of tasks for the next period concludes this chapter.

4.1 Software Installation Methods

The software installation was carried out utilizing EasyBuild [21], which is a software build and installation framework focusing on HPC. All the EasyBuild build scripts were created with the full architecture specific optimizations available for each module of the DEEP-EST system. This includes compiling the DEEP-EST SW stack with the optimization flags for each module architecture.

All the EasyBuild scripts are freely distributed online in the EasyConfig GitHub repository [22]. Frequent updates and patches with the latest framework version are pushed to the repository. The frameworks are available as SW modules on each part of the DEEP-EST system using the parallel file systems. The module file created by EasyBuild contains all the other module dependencies of the framework. With this setup, any user can simply load the required framework without taking care of the underlying dependencies.

As an example, loading the module *TensorFlow/1.13.1-GPU-Python-3.6.8* on the DAM will automatically load the following modules beforehand:

- Python/3.6.8
- SciPy-Stack/2019a-Python-3.6.8
- h5py/2.9.0-serial-Python-3.6.8
- cuDNN/7.5.1.10-CUDA-10.1.105
- CUDA/10.1.105
- NCCL/2.4.6-1-CUDA-10.1.105

In this example, all the corresponding and necessary frameworks optimized for GPUs will be loaded under the hood on the DAM and ESB. As Easybuild generates the appropriate hierarchical module scheme, only modules which are compatible with each other are available for loading.

The same SW modules can be used across the parts of the DEEP-EST system. In this case, the Tensorflow module optimized for GPU can be used for the DAM and the ESB module as each module contains the same Nvidia V100 Tesla GPUs. On the CM, a CPU-optimized version of Tensorflow would be loaded instead.

Future frameworks will be installed using the same workflow offered by EasyBuild using custom EasyConfig scripts or the scripts hosted at GitHub.

4.2 Installed Frameworks

The list of required framework defined in Deliverable D6.2 was used for the software installation. In this section, we describe the machine learning frameworks, the OpenCL offloading programming model, Intel's new oneAPI programming model and the persistent memory toolkit.

4.2.1 Machine Learning Frameworks

The required machine learning frameworks from D6.2 are summarized in Table 1. Most of the frameworks have been installed on the prototype with EasyBuild. Only Apache Spark, BigDL and Dist-Keras are not installed yet; all the other software is installed and available on the CM and DAM. In addition, the latest version of Tensorflow 2.0 will be installed as requested by WP1.

Partner	Application	Requirements
KU Leuven	DLMOS	PyTorch, Horovod, scikit-learn, mpi4py
UoI	Deep-Learning	Tensorflow, Keras
CERN	DL	Apache Spark, TensorFlow, Keras, BigDL, Dist-Keras

Table 1: List of required frameworks

The list of installed machine learning frameworks required by WP1 with the corresponding version is described below:

- Python/{2.7.16,3.6.8}
- PyTorch/1.1.0-GPU-Python-3.6.8
- Horovod/0.16.2-GPU-Python-3.6.8
- scikit/2019a-Python-{2.7.16,3.6.8}
- mpi4py/3.0.1-Python-{2.7.16,3.6.8}
- TensorFlow/1.13.1-GPU-Python-3.6.8
- Keras/2.2.4-GPU-Python-3.6.8

The complete software stack available on the DEEP-EST prototype can be evaluated with the `module spider` command. With the current installation workflow, adding a new framework can easily be carried out.

Moreover, the highly optimized, extensively threaded math routines for scientific application in HPC (Intel MKL and Nvidia cuBLAS) along with the latest Deep Neural Network libraries (Nvidia cuDNN) were installed on the system for the corresponding hardware. The exact library version is described below:

- Intel MKL/2019.3.199
- cuBLAS/10.1.105
- cuDNN/7.5.1.10-CUDA-10.1.105

4.2.2 OpenCL Heterogeneous Programming Framework

All the WP1 applications currently using or planning to use their application on FPGA reported OpenCL as the main framework of interest as shown in Table 2. OpenCL is an open standard for parallel programming of heterogeneous systems [25].

One Intel FPGA Stratix 10 is installed on each of the 16 DAM nodes of the DEEP-EST prototype. To enable the exploitation of the Intel FPGAs available on the machine, Intel provided a FPGA Workshop at JSC focusing on OpenCL development. All the training materials have been shared with the application developers. In addition, emulation and hardware compilation modes have been tested and validated with the vector addition kernel using OpenCL and the Python wrapper for OpenCL (pyOpenCL). OpenCL will offer code portability to compare the application performance between CPU, GPU and FPGA.

Partner	Application	Software
ASTRON	Correlator/Imager	OpenCL
UoI	piVSM	OpenCL
CERN	CMSSW	OpenCL

Table 2: FPGA usage per application

4.2.3 oneAPI Programming Model

Intel publicly released the new oneAPI programming model at the SC19 conference in Denver. oneAPI is a unified standards-based programming model across multiple architectures [23]. The main goal of oneAPI is to simplify programming across a heterogeneous environment using Data Parallel C++. DPC++ is extending SYCL, a higher-level programming model for OpenCL. It allows code reuse across hardware targets, and enables high productivity and performance across CPU, GPU, and FPGA architectures as shown in Figure 14. oneAPI will be freely available.

Intel and JSC collaborated to install and support the beta version of Intel oneAPI on CM and DAM of the DEEP-EST prototype. In addition, the Intel oneAPI toolkit with the HPC examples was shared among the WP1 application developers. Intel oneAPI will support Intel FPGA Stratix 10, available in the DAM partition, during the first quarter of 2020. The goal is to install the Intel oneAPI as a module. In addition to this, training material will be provided by Intel.

4.2.4 Persistent Memory Toolkit

The Data Analytics Module is composed of 16 nodes with 384 GB RAM plus 2 or 3 TB of Intel Optane DC Persistent Memory (14 nodes with 2TB, 2 nodes with 3TB), as shown in Figure 15. Compared to DRAM, Intel Optane DC Persistent Memory has slightly higher latency, but offers much higher affordable capacities and data persistence. It can be configured in two principal modes: Memory Mode and App Direct Mode.

In Memory Mode, no changes to the application are required: the installed DRAM acts as a memory cache and the Intel Optane DC Persistent Memory offers large memory capacity to the OS and to applications. In App Direct Mode, DRAM and non-volatile memory are mapped onto separate

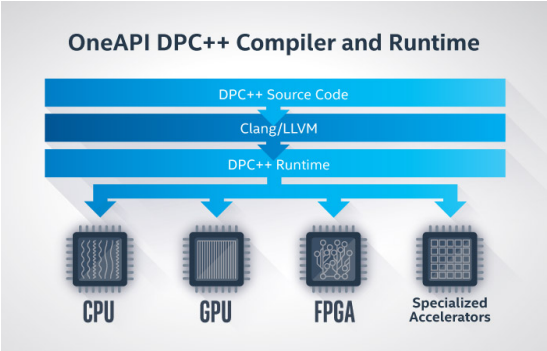


Figure 14: Intel oneAPI DPC++ Compiler Stack (Source: *https://software.intel.com/*)

memory spaces, and applications have to be modified in order to exploit the different characteristics of the two memory technologies. Access to the persistent memory occurs through regular load and store operations, yet the management of multiple address spaces requires code adaptations. Intel provided an online workshop on DCPMM and Intel VTune to evaluate the most frequently objects accessed by the application and to program with the Persistent Memory Development Kit (PMDK, see [24]) in App Direct mode. PMDK is available as Open Source, and also supports other persistent storage systems like SSDs.

A special use case of App Direct mode is to map a filesystem onto a non-volatile memory partition; for this, the fs-dax layer provided by PMDK enables file system access avoiding the need to go through a block device chain. For I/O-heavy applications, this usage mode can provide significant speed-ups.

In the DAM partition, half of the nodes will be configured in Memory Mode (8 nodes) whereas the other half will be configured in App Direct Mode (8 nodes).

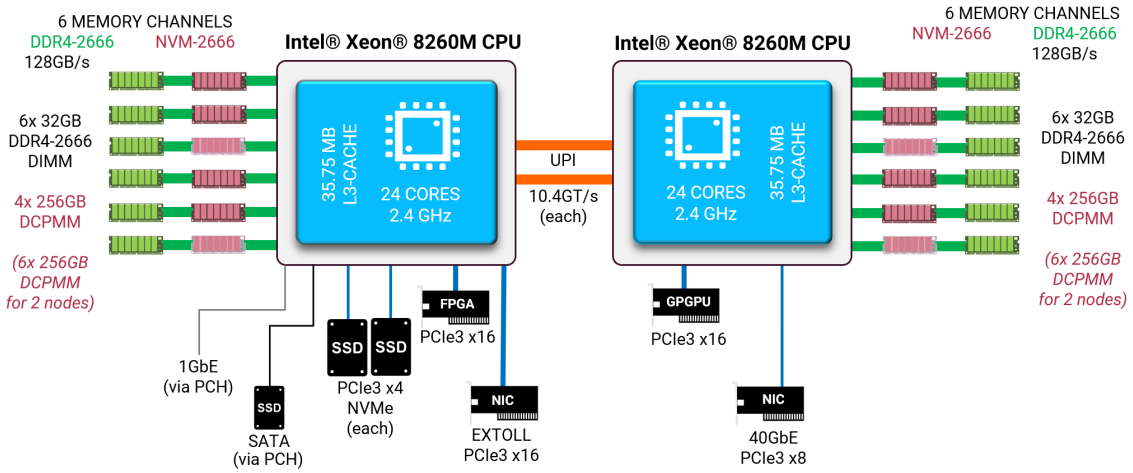


Figure 15: Scheme of the DAM node hardware

4.3 Future Work

In the next months, we will finish the installation of the last missing frameworks in the DEEP-EST prototype. In addition, we will support the frameworks on the ESB when this part of the prototype becomes available in early 2020. oneAPI will be tested on the DAM FPGA devices after the next oneAPI release supporting Intel FPGA Stratix 10 becomes available in early 2020. In particular the Intel oneAPI DL Framework Developer Toolkit will be installed to enable machine learning application on FPGA. We will continue to work in close collaboration with WP1 to have the most recent frameworks compiled and optimized on the DEEP-EST prototype.

To carry on with support to the WP1 application, EPCC will collaborate with the developers to improve their machine learning code.

5 I/O, file system, and storage

5.1 BeeGFS Storage Plugin Infrastructure

Storage plugins are a new feature for the BeeGFS storage server as motivated in D6.1. The BeeGFS storage plugin API is an abstraction layer for interchangeable plugins which wrap all accesses to chunk files on a BeeGFS storage server. This makes them an ideal feature for testing and comparison of different storage technologies and strategies.

The aim is to cover different storage technologies, with a focus on existing and future object stores. These backends could be POSIX filesystems, memkind, erasure coding, NVDIMMs or Amazon S3.

After optimisation and evaluation of the prototype, the storage plugin interface will see continued development and become part of the main BeeGFS product. Consequently, it will be available together with the published BeeGFS source code. This implies that – besides the plugins provided by the BeeGFS development team – interested third-party developers may write and publish custom plugins optimized for their architecture or technology.

5.1.1 Architecture Outline

With BeeGFS, the file system hierarchy and file attributes are stored on meta servers. Each file is striped across a number of chunks which reside on different storage servers.

To abstract away the direct POSIX file system calls done by the storage server, we implement a layer between the actual file system and the storage server core code. This layer maps the underlying file system to behave similar to an object store, which requires the handling of operations such as read, write, getting and setting of attributes for chunks identified by specific keys.

5.1.2 Interface Specification

For the DEEP-EST prototype, an interface specification is provided. This specification consists of:

- A C++11 Header file formalizing the API itself.
- Documentation on how to write and deploy plugins.
- An example "heap" plugin which keeps all information in RAM.
- A branch of the BeeGFS source tree able to compile and link storage plugins.

Together, these allow any interested participant to start the design and implementation of storage plugins.

5.1.3 Plugins

In the context of DEEP-EST, two plugins are developed:

- `heap` stores all data in C++ datastructures in normal RAM. While primarily intended for API demonstration, it is also a lightweight alternative to tmpfs for volatile storage.
- `pmem` is a prototype for BeeGFS chunk data storage on non-volatile memory NVDIMMs. It uses Intel's Persistent Memory Development Kit (PMDK) and the included libpmemobj to transactionally store data in hardware mapped directly into the storage server's virtual address space.

5.1.4 Prototype

The storage plugin abstraction layer has been developed into a prototype that demonstrates its features. Building, linking and loading of storage plugins is supported by the BeeGFS build system. Loading and configuration of plugins by administrators has been implemented and tested.

To demonstrate the viability of the storage plugin API, two example plugins, "heap" and "pmem", have been developed. While "heap" only stores data in RAM, "pmem" uses Intel's PMDK and libpmemobj for persistent memory NVDIMMs. These plugins will be delivered together with the remaining BeeGFS prototype system.

The packages built for deployment in DEEP-EST support installation together with the current stable release of BeeGFS to allow parallel general use and experimental deployment.

5.2 BeeOND Integration

BeeOND (BeeGFS on demand) is a framework that allows creating a temporary parallel file system on an arbitrary number of hosts. Setup and teardown can happen in a matter of seconds and it is easily to integrate into any resource manager / job scheduler. This makes it ideal as a per job scratch file system or cache layer. The BeeOND framework is ready to be integrated. Since BeeGFS version 7.1, BeeOND also supports the new BeeGFS storage pools. Pools allow to place data on a specific type of storage device within BeeGFS, which enables a maximum of flexibility regarding performance and storage space. They are assigned to storage target paths by providing a simple configuration file for all involved nodes. BeeOND automatically creates all the pools in the file. On each node, it checks each given path and – if it exists – adds it as a storage target to the system. Then, all included targets are assigned to their corresponding pools. This provides a simple and flexible way to include multiple nodes with different kinds of storage hardware into the BeeOND setup. As a last step, BeeOND then creates and assigns a BeeGFS root level directory for each configured pool. This default setup can be modified or overwritten by using the beegfs-ctl setpattern command on any directory in BeeGFS.

For example, lets assume there are a couple of nodes available. Each of the nodes has NVMe and spinning disk storage, so two storage pools shall be created. The NVMe pool is meant to be used as a scratch file system, so it is called *scratch*. The spinning disk pool is named *storage*. BeeOND sets up everything according to the configuration provided by the resource manager. After setup is complete, a BeeGFS mount is available on each node, for instance under `/mnt/beeond`. The two pools can then be found under `/mnt/beeond/scratch` and `/mnt/beeond/storage`. Data put in one of those directories will automatically go into the corresponding pool and therefore be only stored on the assigned type of storage.

5.3 BeeGFS Monitoring

As introduced in D6.1, we developed a new BeeGFS monitoring solution, called beegfs-mon. It collects runtime statistics in real time, such as data throughput and disk usage (a complete list can be found in the BeeGFS documentation). Unlike the previous solution, it uses a time series database backend to store the collected data. This allows us to use a powerful external analysis framework and visualization tools and therefore an improved, scalable monitoring of big clusters. It will also work well together with the rest of the system monitoring, e.g. by using the same tools for monitoring.

By default, beegfs-mon uses InfluxDB as database backend. This decision was made after evaluating several available databases such as Graphite or Prometheus. We have chosen InfluxDB because it is the best maintained, the best documented and most user-friendly time-series database available. It also provides a simple HTTP interface. Clustering is only supported as enterprise feature, but regarding BeeGFS, this is negligible because the load on the database side is not very high even for big clusters. Its overall performance proved to be pretty good. InfluxDB also works seamlessly together with the preferred visualization software, Grafana.

In D5.2, the decision has been made to use Apache Cassandra as store for the general DEEP-EST system monitoring. To make BeeGFS work with it too, we decided to add additional Cassandra support to beegfs-mon. It was designed to support multiple backends, so this has been a straightforward task. There are, however, two drawbacks in using Cassandra over InfluxDB: Because there is no native C++ or HTTP interface available for Cassandra, we require a third party library (https://github.com/datastax/cpp-driver, licensed under the Apache 2.0 license). It is loaded dynamically and is therefore required to be available on the system in a very specific version. Also, Grafana does not support Cassandra as a data source. Possibilities to get around this are either using a separate service that pulls the data from Cassandra and provides it via REST API to Grafana, or providing a full data source plugin. At the time of writing this deliverable, a decision regarding this has yet to be made.

BeeGFS-mon needs to be installed only at one node in the cluster. It regularly pulls the data from the BeeGFS servers and sends it to the database. Server data (meta, storage) are available per node and can, of course, be aggregated by the tools the chosen database query language provides. Client data (e.g. I/O operations) are collected by each server node on a per host or per system user base.

The software has been included in BeeGFS since the release of version 7.0, and is therefore publicly available.

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 take part 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. The mechanism has been extended by a new MSA-aware algorithm for the selection of collector processes. The algorithm is portable and relies on platform specific plugins to identify processes which run on parts of the system that are well suited for the role of I/O collector.

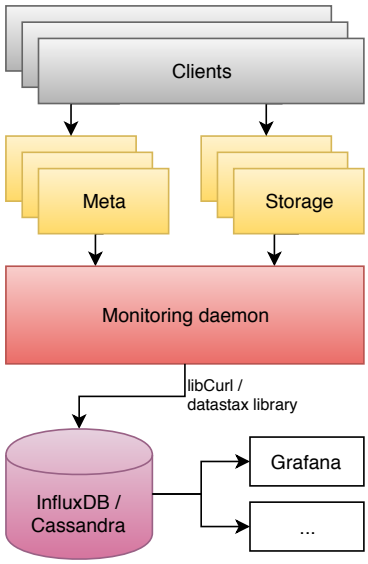


Figure 16: BeeGFS-mon data flow: The daemon pulls data from the server nodes in regular intervals and puts them into the database, where they are available for monitoring/evaluation.

So far, three plugins have been implemented:

- a new generic plugin – `hostname-regex` – which selects collectors tasks by matching the host name of the node they are running on against regular expressions that can be defined via environment variables,
- an older plugin – `deep-est-sdv` – that selects collector tasks using a similar mechanism, but with host names hard-coded to match cluster nodes of an earlier version of the DEEP-EST prototype,
- a mock-up plugin for testing purposes.

In the future, further plugins for new systems of the MSA type can be added.

In order to use the MSA aware collective I/O operations, a platform specific plugin has to be selected when installing SIONlib during the configure step.

```
./configure --msa=hostname-regex # ... more configure arguments
```

When opening a SIONlib file for access from several MPI processes in parallel, the user has to enable the MSA aware collective I/O mode. This is done using the `file_mode` argument of the open function. `file_mode` contains a string that consists of a comma separated list of key-value pairs. The word `collmsa` must appear in that list to select MSA aware collective I/O.

```
sion_paropen_mpi("filename", "...,collmsa,...", ...);
```

Also, when the new `hostname-regex` collector selection plugin is enabled, an environment variable `SION_MSA_COLLECTOR_HOSTNAME_REGEX` has to be defined to contain a POSIX basic regular expression to match against hostnames of nodes running candidates for the collector role (`SION_MSA_COLLECTOR_HOSTNAME_EREGEX` can be used alternatively, but should contain a POSIX extended regular expression).

Additionally, as is the case when using regular collective I/O, the size of collector groups has to be specified, either through the `file_mode` argument of the `sion_paropen_mpi` function or via the environment variable `SION_COLLSIZE`. Documentation on how to use SIONlib on the DEEP-EST SDV has been made available in the WiKi of the DEEP-EST TRAC[1].

5.5 SIONlib CUDA aware interface

In order to match more closely the programming interface offered by other libraries, such as ParaStation MPI (section 2.3), functions of SIONlib have been made CUDA-aware. This means that applications are allowed to pass device pointers pointing to on-device memory to the various read and write functions of SIONlib without the need to manually copy their contents to the host memory.

Like the MSA aware collective I/O operations, the CUDA aware interface has to be enabled when installing SIONlib. This is done by invoking the `configure` script with the argument `--enable-cuda` which optionally allows to specify the path to a CUDA installation.

```
./configure --enable-cuda=/path/to/cuda/installation # ... more configure arguments
```

When SIONlib has been installed with the CUDA aware interface enabled, the user may pass device pointers as the `data` argument to SIONlib's

- task-local
- key/value
- collective

read and write functions.

```
size_t sion_fwrite(const void *data, size_t size, size_t nitems, int sid);
size_t sion_fread(void *data, size_t size, size_t nitems, int sid);
size_t sion_fwrite_key(const void *data, uint64_t key, size_t size, size_t nitems, int sid);
size_t sion_fread_key(void *data, uint64_t key, size_t size, size_t nitems, int sid);
size_t sion_coll_fwrite(const void *data, size_t size, size_t nitems, int sid);
size_t sion_coll_fread(void *data, size_t size, size_t nitems, int sid);
```

SIONlib inspects the pointer and if it points to an on-device buffer performs a block-wise copy of the data into host memory before writing to disk or into device memory after reading from disk.

5.6 SIONlib I/O forwarding

MSA aware collective I/O has the potential of making more efficient use of the storage system by using a subset of tasks that are well suited for performing I/O operations as collectors. The collective I/O approach however imposes additional constraints that make it inapplicable in certain scenarios:

- by design, collective I/O operations force application tasks to coordinate in order to all perform the same sequence of operations. This is at odds with SIONlib's world view of individual files per task that can be accessed independently.

[1] `https://deeptrac.zam.kfa-juelich.de:8443/trac/wiki/Public/User_Guide/SIONlib`

- Collector tasks in general have to be application tasks, i.e. they have to run the user's application. This can generate conflicts on MSA systems, if the nodes that are capable of performing I/O operations efficiently are not part of the resources utilized by the user application.

I/O forwarding can help in both scenarios. It works by relaying calls to low-level I/O functions (e.g. `open`, `write`, `stat`, etc.) via a remote procedure call (RPC) mechanism from a client task (running the user's application) to a server task (running a dedicated server program) that then executes the functions on behalf of the client. Because the server tasks are dedicated to performing I/O, they can dynamically respond to individual requests from client tasks rather than imposing coordination constraints. Also, on MSA systems, the server tasks can run on modules different from the ones used by the user application.

I/O forwarding has been implemented in SIONlib through an additional software package, SIONfwd[2]. It uses a custom made, minimal RPC mechanism based only on MPI's message passing, ports, and pack/unpack mechanisms. In the future we intend to evaluate more general and more optimised third party RPC solutions as they become available.

To use I/O forwarding in SIONlib, the SIONfwd package first has to be installed (it uses a standard CMake based build system) and SIONlib has to be configured to make use of it:

```
./configure --enable-sionfwd=/path/to/sionfwd # ... more configure arguments
```

In the user application, just like MSA aware collectives, I/O forwarding has to be selected when opening a file (I/O forwarding is treated like an additional low-level API like POSIX and C standard I/O). This is done by adding the word `sionfwd` to the `file_mode` argument of SIONlib's `open` functions:

```
sion_paropen_mpi("filename", "...,sionfwd,...", ...);
```

Although in principle MPI contains a mechanism for dynamically spawning additional processes, it is not used to spawn the forwarding server processes for two reasons. First, the feature is loosely specified with many of the details left for implementations to decide. This makes it hard to precisely control process placement which is especially important on MSA systems. Second, the resources necessary to run the server tasks (additional compute nodes) in many cases have to be requested at job submission time anyway. Thus, the server tasks have to be launched from the user's job script before the application tasks are launched. A typical job script could look like this:

```
#!/bin/bash
# Slurm's heterogeneous jobs can be used to partition resources
# for the user's application and the forwarding server, even
# when not running on an MSA system.
#SBATCH --nodes=32 --partition=dp-cn
#SBATCH packjob
#SBATCH --nodes=4 --cpus-per-task=1 --partition=dp-dam

# Defines a shell function sionfwd-spawn that is used to
# facilitate communication of MPI ports connection details
# between the server and the client.
eval $(sionfwd-server bash-defs)

# Spawns the server, captures the connection details and
# exports them to the environment to be picked up by the
# client library used from the user's application.
sionfwd-spawn srun --pack-group 1 sionfwd-server
```

[2] https://gitlab.version.fz-juelich.de/SIONlib/SIONfwd

```
# Spawn the user application.
srun --pack-group 0 user_application

# Shut down the server.
srun --pack-group 0 sionfwd-server shutdown

# Wait for all tasks to end.
wait
```

More detailed instructions have been published on the DEEP-EST TRAC[3].

[3] https://deeptrac.zam.kfa-juelich.de:8443/trac/wiki/Public/User_Guide/SIONlib

6 Resiliency

In deliverable D6.2, we presented the novel FTI features differential checkpointing (dCP), incremental checkpointing (iCP) and the creation of HDF5 checkpoint files. Furthermore, we presented the novel checkpoint pragmas integrated into the mercurium compiler (openCHK).

These features conclude the work of the resiliency task described in the DEEP-EST proposal. However, as we have mentioned in deliverable D6.2, the implementation of dCP at that time depended on the level of fragmentation of the protected buffers into small blocks of dirty and clean data. During the 6 month between D6.2 and D6.3, we have worked on an implementation that removes this dependency.

The rest of this chapter is organized as follows: firstly we present a comparisson between the novel dCP implementation and the one presented in D6.2, afterwards we include the work presented in D6.2 in order to provide a self contained description of the accomplishments inside the resiliency task in this deliverable.

6.1 dCP implementations

Differential checkpointing (dCP) is a method to incorporate only differences of application states into the checkpoint data. Only the first checkpoint requires the full write of the protected data. Every time the application requests consecutive checkpoints, the data is merely updated by the differences. The application of this technique may lead to a significant reduction of checkpoint overhead. For instance in applications that operate on a multi-dimensional grid where some regions of the grid remain unchanged. However, the final degree of reduction depends on the application type and can vary as well depending on the checkpoint frequency.

We have implemented two distinct versions of dCP into FTI. Both implementations are based on hash algorithms. In order to determine the data differences, the protected buffers are decomposed into blocks of a user defined size. Each block is represented by the hash of its content. The hashes are calculated either by the MD5 or CRC32 algorithm. Both algorithm are characterized by high collision resistance and a small avalanche effect. Thus, we can detect practically every change of content in the blocks by variations in the hash digests.

In this section, we will compare the two available FTI dCP implementations. We refer to them as dCP-FTIFF and dCP-POSIX, which corresponds to the interfaces they are implemented upon. dCP-FTIFF is implemented on top of the FTI-FF IO interface and dCP-POSIX on top of the POSIX IO interface.

6.1.1 dCP-FTIFF

This implementation is based on immutable locations of the protected buffers in the checkpoint files. In order to allow for dynamic sizes of the protected buffers, we created a new file format that provides this functionality. Details of the file format can be found in our publication about this implementation [26]. The article was accepted and presented at CCGrid 2019.

Strong points of this implementation are: i) Minimized storage size, ii) No ASCII metadata files due to in-file metadata. A weak point of this implementation is that it leads to a significant loss in performance if the buffers are highly fragmented into very small blocks of changed (dirty) data [26]. Figure 17 shows the mechanism schematically.

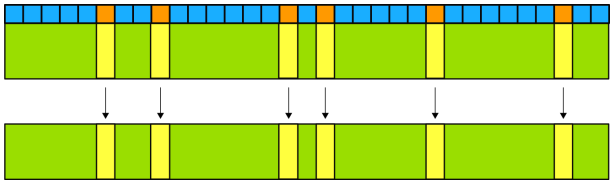


Figure 17: Above: buffer in memory + hash array. The red squares indicate dirty memory regions. Below: The buffer representation in stable storage. The dirty regions are overwritten by the updated data blocks.

6.1.2 dCP-POSIX

In order to improve the performance for highly fragmented buffers, we developed a second implementation of dCP based on the FTI POSIX interface. We take advantage of the buffering in the C standard IO library. As long as the data is written contiguously, the performance is independent of the chunk sizes that are written. In order to write the data contiguously, we append the dirty blocks to the existing checkpoint file.

We need some kind of metadata that directs FTI how to reconcile the data from the last checkpoint upon recovery. The metadata handling in our approach is twofold: 1) The buffers are decomposed into blocks of size b and each block is preceded with a small block of metadata. 2) The end of the current checkpoint layer is appended by a metadata block that keeps the global properties of the layer.

The preceded metadata keep the variable id and the offset of the datablock. The metadata that is appended to the file contains the blocksize b, the checkpoint id and the checkpoint layersize. The blocksize b for the data blocks corresponds to the blocksize that was used for creating the hasharrays. Figure 18 visualizes the file format.

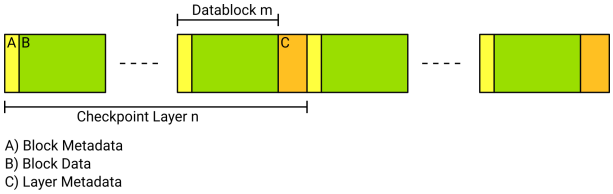


Figure 18: Schematic view of the metadata and data in the dCP-POSIX checkpoint file.

Strong points of this implementation are: i) Reduction of checkpoint overhead is directly proportional to the dirty-to-total data ratio[1], ii) The checkpoints contain the data from the former checkpoints. The weak point of this implementation is the increasing size of the checkpoint file. In order to limit the size,

[1] in contrast to dCP-FTIFF, where the reduction of overhead depends additionally on the level of fragmentation into dirty data blocks

the user may specify a maximum amount of checkpoint layers N. Every Nth differential checkpoint is then written as a full checkpoint and the former multilayer differential checkpoint is deleted.

The advantage of point ii) is not yet fully exploited. The obvious strength is that the checkpoint contains all the data from the last i checkpoints. This data can be reconstructed. In other words, it represents a compressed form of the last i checkpoints. In the current version, we take advantage of this, providing a fallback to layer $i-1$ if parts of layer i are corrupted. This allows to fallback up to the oldest layer if all the others contain corrupted data. However, to restart from a certain layer, the previous layers need to be uncorrupted until the oldest layer, to guarantee data integrity. The data corruption is detected by changes in the integrity sum, which is an MD5 hash that represents the contents of each layer.

6.2 Incremental Checkpointing (iCP)

Incremental checkpointing (iCP) enables to integrate subsets of the protected data individually into the checkpoint file. This has several uses in HPC applications. The most discussed usage is, that iCP can reduce the stress on the network by thinning out concurrent I/O data streams. That is, writing not all protected buffers at once to stable storage, but rather write the buffers at suitable locations in the execution flow in order to avoid reaching bandwidth saturation on the nodes.

However, there are several other situations in which iCP can be beneficial. For instance, buffers can be added to the checkpoint in a modular way, with full control of when the buffer is eventually stored. All buffers could potentially be written by different threads. Thus, we can achieve an overlap of checkpoint I/O and computation for decoupled datasets. The same applies for GPU applications with decoupled datasets and costly computations on the GPU side.

The implementation of iCP in FTI encloses a checkpoint region inside which the protected datasets can be added individually to the checkpoint file. The datasets can be added in any order. The checkpoint region is opened by the respective call to an initializing function and the region is closed by the matching finalize call.

We would like to explain the overlapping scenario from above in a short example. Iterations in n-body simulations consist essentially of two steps. Firstly, we need to compute the forces between the particles and secondly, we need to compute the displacement of the particles due to the forces. We can open the iCP region in iteration i, write the forces, F_i, computed in that iteration while we compute the positions, P_i. In the next iteration, we write the positions, P_i, from iteration i, while computing the forces, F_{i+1}, from iteration $i+1$ (see figure 19).

This scenario will additionally benefit from dCP support while using iCP, which we have enabled in FTI.

6.3 Checkpointing with pragmas (OpenCHK)

In deliverable D6.1 we proposed a pragma based interface that can be operated with several application based checkpoint/restart (CR) libraries. The advantages of checkpointing with pragmas is the portability of the code and the reduction in complexity. The developer does not need to know

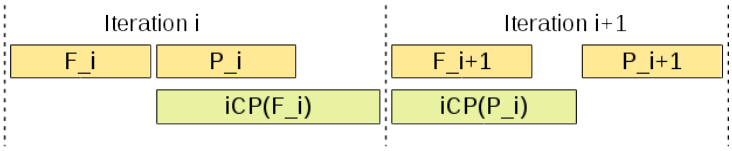


Figure 19: Overlapping checkpoint I/O with computation. In yellow the computations of the forces and particle displacements and in green the incremental checkpoints of the respective quantities.

about the specific API of any of the underlying CR libraries.

The mechanism is represented essentially by four directives (see figure 20). The directives trigger library initialization, finalization, checkpoint and recovery. In order to support distinct features of individual libraries (such as dCP in FTI), we added a kind clause that can take a descriptive parameter that points to a certain feature of the underlying library.

```
1 #pragma chk init comm()
2 #pragma chk store() id() level() [ [if()] [kind()] ]
3 #pragma chk load()
4 #pragma chk shutdown
```

Figure 20: The four directives (init, store, load and shutdown) and the respective clauses of the OpenCHK checkpoint pragma specification. The clauses comm, id and level are mandatory. The specification is available at `https://github.com/bsc-pm/OpenCHK-model`

The calls to the CR libraries are handled by an intermediate library, *transparent checkpointing library* (TCL), which can easily be modified to incorporate further libraries. Currently TCL supports FTI, SCR and VeloC. Users are free to choose their favorite back-end library depending on what is available in the target machine.

6.4 HDF5

The HDF5 file format is used in applications to organize scientific data in a file. Besides this, HDF5 offers support for parallel I/O and it is highly optimized for any kind of I/O operations. The library has bindings to C++, C, Fortran 90, Java and Phython and it is supported by Matlab and R. That is to say, that HDF5 files can be processed in various ways and offer a high portability of scientific data.

It is thus very interesting to have a checkpointing interface that uses HDF5, in order to achieve highly optimized I/O and at the same time merge interests of fault tolerance and data analysis.

As we proposed in deliverable D6.1, we integrated a simple HDF5 interface into FTI. Beside its application for resiliency, it hides the complexity that is exposed by the HDF5 library for simple use cases.

The interface enables to create named groups, named datasets and named types. It is possible to create a hierarchy of groups and to assign datasets and types to any of the created groups. It is also possible to create complex datatypes, for instance for C-structures. The support for HDF5 datasets is necessary to indicate the dimensional character of the stored variables.

The interface can be used in two different modes. The first mode creates one checkpoint file per process (N-N) and the second mode, one file shared among all processes (N-1). The aim of the second mode is to enable the elastic restart with a different amount of processes.

In order to operate the second mode, FTI allows to create shared datasets. Every application process can define subsets that belong to the shared datasets. The subsets need to be defined with offset and element count in order to select the corresponding region inside the shared dataset (see figure 21).

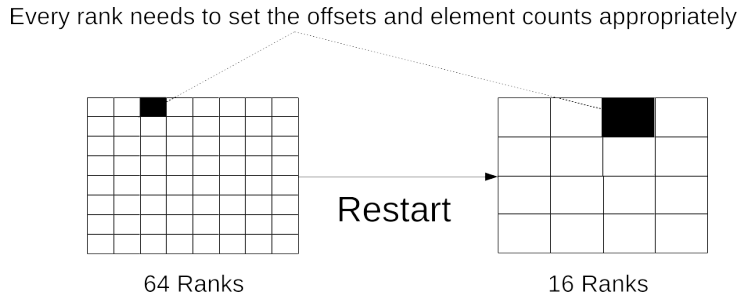


Figure 21: Elastic recovery with 16 ranks from a former execution with 64 ranks.

6.5 Co-Design

FTI on the cluster FTI is available on the SDV nodes for being tested by the application developers. Currently the module is accessible on the compute nodes; not on the login nodes. Thus, in order to link to the library, the user needs to request an allocation. The installed version is 1.2, which offers HDF5 checkpointing and dCP (more information about the release: `https://github.com/leobago/fti/releases`).

WP1 applications Table 2 in deliverable D6.1 section 6.2 lists the applications of WP1 and a short analysis regarding their need to incorporate checkpointing. We have focussed on xPic and NEST for the time being. There have been earlier attempts to implement checkpointing in NEST. However, due to the complex graph like structure of the runtime data it is a very challenging task. The application would benefit from checkpointing, as we stated in D6.1, not only for resiliency. It is very interesting to restart from checkpoints and explore different paths by changing some of the parameters. We have started a collaboration with the developers of NEST and we have implemented a checkpointing prototype that uses the BOOST serialization library to handle joints and loops inside the NEST data-structures (neurons and neuron-connections).

We have also implemented checkpointing inside xPic. The xPic developers are interested in performing elastic restarts from checkpoint files. In collaboration with the xPic team, we have accomplished an implementation of FTI in xPic, that is enables an elastic restart to a different number of processes and that allows the operation of dCP and iCP.

7 Summary

This document describes the programming environment developed to efficiently exploit the MSA architecture. The programming environment covers the most relevant parts of the software stack required to run on a supercomputer and it includes this 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 paragraphs we summarize the main developments already completed for each of these software components.

We have enhanced ParaStation MPI with modularity-aware MPI collective operations by extending the existing infrastructure of MPICH for node locality awareness. In addition, MPI application developers can optimize communication patterns and algorithmic program flows with respect to modularity also explicitly by querying for topology information via a new interface. Moreover, the MPI RMA API has been extended to support one-sided communications on the Network Attached Memory (NAM). An API manual has been written for this and a prototype implementation of this feature is available within ParaStation MPI that can even be used without an actual NAM by emulating its memory on the compute nodes. Finally, to support better the new ESB design, ParaStation MPI is being extended with CUDA-awareness features to improve both productivity and performance.

The OmpSs-2 programming model has been enhanced to improve programmability and exploit specific hardware features of each module. We have extended OmpSs-2's tasking model with an improved task nesting and dependency system support that relies on fine-grained synchronizations to unveil additional parallelism. Moreover, we have added support for scalar and array reductions. We have also implemented and evaluated *work-sharing* tasks to exploit fine-grained structured parallelism on multi- and many-core processors. Although the above-mentioned features were designed with the original ESB concept based on a many-core processor in mind, we are also extending or adapting some of them to the new GPU-based ESB. We have released a new version of our Task-Aware MPI (TAMPI library) that simplifies hybrid programming and supports both blocking and non-blocking MPI operations inside tasks. TAMPI will be enhanced to support one-sided MPI primitives to access the NAM. To address the new ESB design we have extended OmpSs-2 with support for CUDA C kernels, which can be annotated and invoked like regular tasks. Initially this feature was based on CUDA Unified Memory, but now we have a new implementation that also works with explicit memory transfers managed by the runtime that can improve performance for some workloads. This feature greatly simplifies the orchestration of complex applications that perform computations on both CPUs and GPUs, as well as (MPI) communications. Moreover, we have optimized three of the main components of the runtime system: the memory allocator, the scheduler and the dependency subsystem. Finally, we are working on a locality-aware scheduling in order to leverage the memory

hierarchies present on the different modules.

We have identified the data analytics and machine learning frameworks and libraries that are relevant for our applications. These frameworks and libraries have been installed and tuned on the DEEP-EST prototype.

We have extended the BeeGFS filesystem with a plugin architecture that facilitates the integration of new storage technologies with the filesystem. The development of an integrated monitoring facility to store time series has also been completed. The work on integrating the BeeOND on-demand filesystem with the resource manager and job scheduler developed in WP5 has been completed and it is ready to be integrated.

Finally, we have extended the Fault Tolerance Interface (FTI) multilevel checkpoint/restart library to increase the resiliency of applications running on the MSA architecture. Several features such as differential checkpointing (dCP), incremental checkpointing (iCP) and support for HDF5 are already available. We have also implemented the new OpenCHK interface based on pragmas that currently supports FTI, SCR and VeloC as back-ends. The next step is to enhance FTI to support applications running on GPUs.

List of Acronyms and Abbreviations

A

API	Application Programming Interface

B

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
BN	Booster Node (functional entity)
BoP	Board of Partners for the DEEP EST project
BSC	Barcelona Supercomputing Centre, Spain

C

Cassandra	The Apache Cassandra key-value store
CERN	European Organisation for Nuclear Research / Organisation Européenne pour la Recherche Nucléaire, International organisation
CM	Cluster Module: with its Cluster Nodes (CN) containing high-end general-purpose processors and a relatively large amount of memory per core
CN	Cluster Node (functional entity)
CPU	Central Processing Unit

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
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
DN	Nodes of the DAM
DNN	Deep neural network

E

EC	European Commission
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
EXTOLL	High speed interconnect technology for HPC developed by UHEI

F

FHG-ITWM	Fraunhofer Gesellschaft zur Foerderung der Angewandten Forschungs e.V., Germany
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
GPU	Graphics Processing Unit

H

H2020	Horizon 2020
HPC	High Performance Computing
HPDBSCAN	A clustering code used by UoI in the field of Earth Science
HW	Hardware

I

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
ISO	International Organisation for Standardisation

J

JLESC Joint Laboratory for Extreme Scale Computing
JUBE Jülich Benchmarking Environment
JUELICH Forschungszentrum Jülich GmbH, Jülich, Germany

K

KULeuven Katholieke Universiteit Leuven, Belgium

L

M

MPI Message Passing Interface, API specification typically used in parallel programs that allows processes to communicate with one another by sending and receiving messages
MPICH MPI implementation maintained by Argonne National Laboratory
MSA Modular Supercomputer Architecture

N

NAM Network Attached Memory
NEST Widely-used, publically available simulation software for spiking neural network models developed by NMBU
NMBU Norwegian University of Life Sciences, Norway

O

OmpSs BSC's Superscalar (Ss) for OpenMP
OpenCL Open Computing Language, framework for writing programs that execute across heterogeneous platforms
OpenMP Open Multi-Processing, Application programming interface that support multi-platform shared memory multiprocessing

P

ParaStation	Software for cluster management and control developed by JUELICH and its linked third party ParTec
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)
piSVM	Parallel classification algorithm
PMT	Project Management Team of the DEEP-EST project

Q

R

RDMA	Remote Direct Memory Access / Remote DMA-based Memory Access
RMA	Remote Memory Access

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-EST prototype is not yet available
SIONlib	Parallel I/O library developed by Forschungszentrum Jülich

T

TensorFlow	Open-source software library for dataflow programming

U

UHEI	Ruprecht-Karls-Universitaet Heidelberg, Germany
UoI	Háskóli Íslands University of Iceland, Iceland

V

W

X

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] J. M. Perez, V. Beltran, J. Labarta and E. Ayguadé, "Improving the Integration of Task Nesting and Dependencies in OpenMP," 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Orlando, FL, 2017, pp. 809-818.

[7] M. Maroñas, K. Sala, S. Mateo, E. Ayguadé and V. Beltran. "Worksharing Tasks, an Efficient Way to Exploit Irregular and Fine-Grained Loop Parallelism." To appear in IEEE 26th International Conference on High Performance Computing (HiPC 2019), Hyderabad, India.

[8] Kevin Sala, Jorge Bellon, Pau Farré, Xavier Teruel, Josep M. Perez, Antonio J. Pena, Daniel Holmes, VicenŞeltran, and Jesus Labarta. 2018. Improving the Interoperability between MPI and Task-Based Programming Models. In Proceedings of the 25th European MPI Users' Group Meeting (EuroMPI'18). ACM, New York, NY, USA, Article 6, 11 pages.

[9] Kevin Sala, Xavier Teruel, Josep M. Pérez, Antonio J. Peña, Vicenç Beltran, and Jesús Labarta: Integrating Blocking and Non-Blocking MPI Primitives with Task-Based Programming Models, CoRR, 2019.

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

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

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

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

[14] Best Practice Guide for Writing MPI+OmpSs Interoperable Programs

https://www.intertwine-project.eu/api-combinations

[15] 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/

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

[17] Paszke, Adam et al. "Automatic differentiation in PyTorch." (2017).

[18] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, Édouard Duchesnay, Scikit-learn: Machine Learning in Python, The Journal of Machine Learning Research, 12, p.2825-2830, 2/1/2011

[19] Sergeev, Alexander and Mike Del Balso. "Horovod: fast and easy distributed deep learning in TensorFlow." CoRR abs/1802.05799 (2018): n. pag.

[20] Joeri R. Hermans. Distributed Keras: Distributed Deep Learning with Apache Spark and Keras, CERN IT-DB https://github.com/cerndb/dist-keras

[21] Alvarez, Damian and O'Cais, Alan and Geimer, Markus and Hoste, Kenneth, Proceedings of the Third International Workshop on HPC User Support Tools (HUST-16), Scientific software management in real life: deployment of easybuild on a large scale system, 2016

[22] A collection of easyconfig files that describe which software to build using which build options with EasyBuild. http://easybuilders.github.io/easybuild/

[23] Intel oneAPI Toolkits(Beta): A Unified, Standards-Based Programming Model across multiple architectures. https://software.intel.com/en-us/oneapi

[24] PMDK: The Persistent Memory Development Kit (PMDK) https://pmem.io/pmdk/

[25] OpenCL: The open standard for parallel programming of heterogeneous systems https://www.khronos.org/opencl/

[26] K. Keller and Leonardo Bautista Gomez *Application-Level Differential Checkpointing for HPC Applications with Dynamic Datasets* 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID) https://doi.org/10.1109/CCGRID.2019.00015