H2020-FETHPC-01-2016

DEEP-EST

DEEP Extreme Scale Technologies

Grant Agreement Number: 754304

D6.2

Prototype Programming Environment Implementation

Final

Version:	1.0
Author(s):	V. Beltran (BSC), J. Ciesko (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:	31.03.2019

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:	D6.2
	Deliverable Nature:	Report
	Deliverable Level: PU*	**Contractual Date of Delivery:** 31.03.2019
		Actual Date of Delivery: 31.03.2019
	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:** Prototype Programming Environment Implementation	
	ID: D6.2	
	Version: 1.0	**Status:** Final
	Available at: http://www.deep-projects.eu/	
	Software Tool: LaTeX	
	File(s): DEEP-EST_D6.2_Prototype_Implementation.pdf	
Authorship	**Written by:**	V. Beltran (BSC), J. Ciesko (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:	S. Krempel (ParTec) J. Kreutz (JUELICH)
	Approved by:	BoP/PMT

Document Status Sheet

Version	Date	Status	Comments
1.0	29.03.2019	Final	Final version

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 status of the programming environment for the Modular Supercomputing Architecture (MSA) proposed in the DEEP-EST project. The planned work as stated in deliverable D6.1 has been refined and implemented in respect to new application requirements, as well as, 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), Extreme Scale Booster (ESB) and Data Analytics Module (DAM) modules. The work presented in this deliverable will be further refined based on the continuous co-design cycles established between hardware and software architects and application developers.

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, the initial requirements of the applications. In this document we give an update on the progress made implementing different components and features defined in the initial document. We also highlight any deviation from the original plan as required to add better support for 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, in others, modifications of the applications will be needed.

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

- ParaStation MPI communication library (Section 2) to leverage distributed memory systems
- OmpSs-2 programming model (Section 3) to exploit many-core processors, deep memory hierarchies and accelerators
- Some of the most popular frameworks and libraries used for data analytics and machine learning (Section 4)
- BeeGFSfilesystem (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 Shared Memory Processors (SMPs). We have reused the same infrastructure to provide topology-aware MPI communicators, so interested application developers can also optimize application communication patterns based on the topology of the system. Moreover, the MPI RDMA API has been extended to support one-sided communications on the Network Attached Memory (NAM). A prototype implementation of this feature based on shared memory is already available. 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 have been enhanced to improve programmability and exploit specific hardware features of each module. We have extended OmpSs-2 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 are also working on malleable tasks to improve very-fine grained dynamic parallelism. Although the above-mentioned features were designed with the original ESB design based on a many-core processor in-mind, we plan to extend or adapt 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 support both blocking and non-blocking MPI operations inside tasks. 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. This feature greatly simplifies the orchestration of complex applications that perform computations on both CPUs and GPUs, as well as, (MPI-) communications. Finally, we have already started to work on locality-aware scheduling in order to leverage the memory hierarchies present on the different modules, but no prototype is available yet with this feature.

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 Tolerant 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 checkpoiting (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.

2 ParaStation MPI

2.1 MPI Support for Modularity-awareness

2.1.1 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 realizing an appropriate modularity-awareness.

For doing so, the knowledge of faster and slower communication links is mandatory since this information actually represents the different levels of the system hierarchy. In the case of SMP systems, this knowledge can be obtained by simply comparing the hostnames. In contrast, the determination of the module affinity has been realized differently: Following a generic approach, we have decided to avoid a direct use of the network or the node information within ParaStation MPI. Instead, the resource manager (to be implemented in Task 5.4) undertakes this task at a higher level possessing a more global view. In doing so, it might query a configuration database such as psconfig or evaluate some other kind of globally available topology information. Subsequently, it can pass this information in terms of *module identifiers* to the individual MPI ranks.

The related code within MPICH can then use this information 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 different communication domains as well as the communication links among them. This approach supports the beneficial mapping of the collective communication patterns onto the hierarchy using a multi-staged approach (cf. Deliverable D6.1).

2.1.2 Topology-aware MPI Communicator Creation

The above-mentioned topology awareness happens *under the hood* of the MPI interface and hence transparently to the application. However, there may also be use cases where the MPI application itself wants to adapt its communication pattern and / or its algorithmic behaviour to the modularity of the system. Therefore, the knowledge of the respective module affinity needs to be made available at the application level as well. Here, the MPI interface is the natural point for querying this information by the application.

For the realization of such a feature, we have leveraged the module identifiers from above in two ways:

1. We allow to query the module identifier directly via the `MPI_INFO_ENV` object.
2. We have added a new split type for `MPI_Comm_split_type` that makes internally use of the module identifier.

While in particular the latter is quite handy for creating new sub-communicators with process

groups representing the different system modules, the former is more generic and particularly more portable since the symbol `MPIX_COMM_TYPE_MODULE` for the new split type is not standard compliant. Figure 1 shows how both mechanisms can be used to split communicators into sub-communicators, each of which then represents an intra-module communication domain.

```
1 int module_id;
2 char value[MPI_MAX_INFO_VAL];
3
4 MPI_Info_get(MPI_INFO_ENV, "deep_module_id", MPI_MAX_INFO_VAL, value, &flag);
5
6 if (flag)
7         module_id = atoi(value);
8 else
9         module_id = MPI_PROC_NULL;
10
11 MPI_Comm_split(oldcomm, module_id, 0, &newcomm);
```

(a) Creation of a sub-communicator by querying the module identifier via an info object.

```
1 MPI_Comm_split_type(oldcomm, MPIX_COMM_TYPE_MODULE, 0, MPI_INFO_NULL,
2                     &newcomm);
```

(b) Creation of a sub-communicator by using a new split type.

Figure 1: Two different ways for the creation of sub-communicators based on the module affinity.

2.2 MPI Support for NAM-related RMA Operations

One distinct feature of the DEEP-EST prototype will be the Network Attached Memory (NAM): Special memory regions that can directly be accessed via Put / Get-operations from every node within the EXTOLL network. A new generation of the NAM library (called libNAM2) will be available that features a low-level API offering an interface to such RMA operations. This API provides the access to the NAM from within an application.

However, for making the programming more convenient and / or familiar, Task 6.1 strives for integrating an interface for accessing the NAM also in the common MPI world. That way, application programmers shall be 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 standardized (or at least harmonized) fashion under the single roof of an MPI world. In doing so, we follow 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 only available as an API skeleton by now, Task 6.1 has already prepared a draft for a user manual describing the proposed functions and semantics for accessing NAM via an extended MPI interface. Additionally, we developed a shared-memory-based prototype implementation *emulating* the persistent NAM by using persistent shared-memory segments on the compute nodes. At this point it should be emphasized that this prototype is *not* intended as an actual emulation of the NAM. It shall rather offer a possibility for the later users and programmers to evaluate the proposed semantics and API

extensions from the MPI application's point of view.

Both the API proposal as well as the prototype implementation are detailed in a separate document (called "*An API Proposal for Including the NAM into the MPI World*") and only parts thereof are briefly recapped below.

2.2.1 General Semantics

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 will most probably merely require 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. In accordance with our proposal, this correlation can be achieved by sticking to the notion of an *ownership* in a sense that definite regions of the NAM memory space are logically assigned to particular MPI ranks.

2.2.2 Interface Specification

For the allocation of memory regions on the NAM, we envisage semantic extensions to the well-known `MPI_Win_allocate` function. 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 the acquisition of NAM regions instead, we leverage the `info` argument for telling the MPI library to do so: 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 (cf. Fig. 2). This region 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 2: The allocation of a NAM region by means of an MPI window object.

A central use-case for the NAM in DEEP-EST will be 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*. Otherwise, their content would be wiped when freeing the window. Our approach supports persistent NAM-based MPI windows by means of a special info object. 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. Using our approach, 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` (cf. Fig. 3), this function will return an inter-communicator that then serves for accessing the remote NAM memory regions.

```
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);
```

Figure 3: The connection to a NAM region by an MPI session different to the session that has been creating it.

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 propose to alienate the `MPI_Win_create_dynamic` function with the inter-communicator as the input and the window handle as the output parameter (cf. Fig. 4).[1]

```
1 MPI_Comm_remote_size(inter_comm, &group_size);
2 MPI_Win_create_dynamic(MPI_INFO_NULL, inter_comm, &win);
```

Figure 4: Creation of the local representative for a memory window that the MPI session recently connected to.

2.2.3 Prototype Implementation

As already stated above, the prototype implementation is currently based on persistent shared-memory segments that are located on the compute nodes — and not on the libNAM2 which is only available as a mockup by now. Moreover, the current prototype does not actually emulate the NAM but shall rather offer a possibility for the later users and programmers to evaluate the envisioned semantics from the MPI application's point of view already at this stage of the project. In doing so, the focus was *not* put on the question of how remote memory is managed at its location (currently by MPI processes running local to the memory — later by the NAM

[1] 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 without introducing new API functions in addition to MPI.

manager or the NAM itself), but rather on the question of how process-foreign memory regions can be exposed locally. That means that (at least currently) for accessing a persistent RMA window, it has to be made sure that there is at least one MPI process running locally to each of the window's memory regions.

On the DEEP-EST SDV, there is already a special version of ParaStation MPI installed that features all the introduced API extensions. When allocating a session with N nodes, one can run an MPI session, e. g., with n processes distributed across the N nodes, in which each of the processes contributes its local and persistent memory region to an MPI window. This is detailed in Figure 5.

```
1  > salloc --partition=sdv --nodes=4 --time=01:00:00
2  salloc: Granted job allocation 2514
3  > srun -n4 -N4 ./hello 'Have fun!'
4  [0] Running on deeper-sdv13
5  [1] Running on deeper-sdv14
6  [2] Running on deeper-sdv15
7  [3] Running on deeper-sdv16
8  [0] The port name of the window is: shmid:347897856:92010569
9  [0] Put to local region: Hello World from rank 0! Have fun!
10 [1] Put to local region: Hello World from rank 1! Have fun!
11 [2] Put to local region: Hello World from rank 2! Have fun!
12 [3] Put to local region: Hello World from rank 3! Have fun!
13 ...
```

Figure 5: A usage example for the creation of a NAM region within ParaStation MPI by leveraging the introduced API extensions.

Afterwards, one persistent memory region has been created by each of the MPI processes on all the involved nodes (and later on the NAM). In this example, the "port name"' being required for re-accessing the persistent window is: shmid:347897856:92010569

By using this port name (here to be passed as a command line argument — later to be passed via the MPI name publishing mechanism), all the processes of a subsequent MPI session can access the persistent window — provided that there is again at least one MPI processes running locally to each of the persistent but distributed regions[2] as shown in Figure 6.

As one can see, in this example, the four processes of the first MPI session create one persistent memory window distributed across four compute nodes of the cluster. Each of the processes then puts a message into its part of the window and afterwards, the first session is being finished. By passing the port name as an argument to the second session, the four processes of this session (running on the same set of nodes as the previous one) then can attach to the persistent regions in the host memory and can get the messages the processes of the first session have put there.

[2] As said, the latter requirement is just a current "crutch" that cease to exist for the real NAM.

```
1  > srun -n4 -N4 ./world shmid:347897856:92010569
2  [0] Running on deeper-sdv13
3  [1] Running on deeper-sdv14
4  [2] Running on deeper-sdv15
5  [3] Running on deeper-sdv16
6  [0] The port name of the window is: shmid:347897856:92010569
7  [1] Connection to persistent memory region established!
8  [3] Connection to persistent memory region established!
9  [0] Connection to persistent memory region established!
10 [2] Connection to persistent memory region established!
11 [0] Number of remote regions: 4
12 [0] Get from region 0: Hello World from rank 0! Have fun!
13 [0] Get from region 1: Hello World from rank 1! Have fun!
14 [0] Get from region 2: Hello World from rank 2! Have fun!
15 [0] Get from region 3: Hello World from rank 3! Have fun!
16 ...
```

Figure 6: The output of an MPI session connecting to the NAM region that has been created by a previous MPI session (cf. Fig. 5).

2.3 Implications of the new ESB Design: GPGPU-Awareness

GPGPU-awareness (or in particular CUDA-awareness in the NVIDIA case) refers to the idea that MPI applications may directly pass pointers to GPU buffers[3] to MPI functions such as `MPI_Send` or `MPI_Recv`. A non GPGPU-aware MPI library would fail in such a case as the GPU-memory cannot be accessed directly, e. g., via load / store operations or `memcpy()`. Instead, these memory regions have to be transferred via special routines to the host memory beforehand, e. g., by using the `cudaMemcpy()` call in the NVIDIA / CUDA case. In contrast, a GPGPU-aware MPI library recognizes that a pointer is associated with a buffer within the device memory. In this case, it would automatically copy this buffer into a temporary buffer within the host memory prior to the communication, i. e., performing a so-called *staging* of this buffer. Additionally, a GPGPU-aware MPI library may also apply some kind of optimizations, for example, by means of exploiting so-called *GPUDirect* capabilities. These are features provided by some NVIDIA GPUs enabling the direct RDMA access via the interconnect or a peer-to-peer communication between multiple GPGPUs within a single node.

The new design of the Extreme Scale Booster (ESB) obviously demands for the support of a distributed GPU programming. This is not only related to the MPI interface but in particular to exploitation of the capabilities provided by the underlying hardware. The envisioned GPU-centric programming 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. For doing so, the implementation of the above-mentioned GPGPU / CUDA-awareness for the MPI programming interface is a first but essential step towards the MPI support for GPU-centric programming on the ESB.

In fact, ParaStation MPI already supports CUDA-awareness — but just from the semantic-related point of view: The usage of device pointers as arguments for send and receive buffers when calling MPI functions is supported. However, there has yet been no effort to apply any further performance optimizations such as the exploitation of GPUDirect capabilities. This first

[3] These are memory regions located on the GPGPU, the so-called *device* memory.

degree of CUDA-awareness is achieved by applying an internal *staging step* within ParaStation MPI in case a device pointer is detected as a buffer argument.

However, for the efficiency and the scalability of the GPU-centric programming model, this simple (though comfortable — at least from the application programmer's point of view) degree is not sufficient. This is because according to the envisioned paradigm, where the MPI program on the host is just responsible for *controlling* the communication, all the actual data transfers (at least for larger messages) shall happen directly between the GPGPU and the EXTOLL NIC.

With respect to NVIDIA GPUs and the CUDA programming API, this requires an extension of ParaStation MPI by means of an optimized and EXTOLL-tailored GPUDirect support for accommodating the GPU-centric offload. In addition, the communication patterns for collective MPI operations is required to be optimized and enhanced within ParaStation MPI by leveraging the EXTOLL Global Collective Engine (GCE) and other means of topology-awareness.

3 The OmpSs-2 Programming Model

This chapter presents the implementation status of different features of the OmpSs-2 parallel programming model.

The OmpSs-2 programming model prototype 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 *"Improved Tasking"* describes features of the tasking model to improve performance and programmability. Further we discuss features that leverage data-flow programming to simplify MPI and GPU programming on the MSA in sections *"Support of MPI and the MSA"* and *"Support for Accelerators"*. Lastly we provide pointers 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 is computed relative to the time step interval and the particle mass. Finally, acceleration is multiplied by the time step interval to obtain a displacement which is added to the current particle position. Both, computing acceleration and displacement are implemented in the *update_particles* method.

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)%ranks*, implementing a circular shift). The MPI send and receive operations are implemented in the *exchange_particles* method. This method is called *rank-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.

[1] `https://pm.bsc.es/ftp/ompss-2/doc/spec/`

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 model requires ordering to guarantee dead-lock free execution. We discuss this further in section 3.3, "Support of MPI and the MSA".

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, we have *taskified* the three principle methods of the simulation 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 depen-

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

```
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, ...){
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.

```
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.

dencies, 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 task loops and support for array-type reductions.

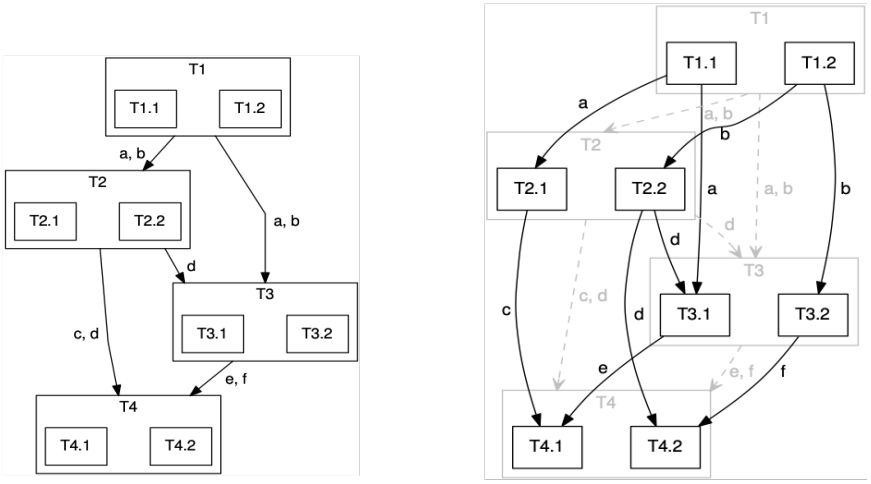
3.2.2 Task loop for malleability

The *loop* construct allows to express a malleable tasks. Such tasks follow the idea of dynamic parallelism where the runtime system decides on the best degree of parallelism and implicitly creates the right amount of work for all available compute resources within that task. This typically results in faster work instantiation. Figure 10 shows the *update_particles_block* function of the NBody code as shown previously using the loop construct to define a malleable task for adaptive, fine-grained parallelism.

The support for task loop is implemented in the OmpSs-2 prototype. Further experimentation and evaluation of the impact on many-core architectures is currently in progress.

3.2.3 Reduction support

Reductions support in OmpSs-2 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 scope of a reduction starts at the first encounter of a task and finishes at either a task synchronization point or at an unspecified point in time but before a data access of a non-participating task. The underlying runtime creates thread-local private copies and maximizes



(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 execution concurrently.

```
1 void calculate_forces_block(forces_block_t *forces,
2         particles_block_t *block1, particles_block_t *block2){
3   ...
4   #pragma oss loop 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 loop pragma allows the creation of malleable, concurrent work.

reuse of 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.

This feature is implemented in the OmpSs-2 prototype.

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 synchronization on process-level. However, strict ordering of work, manual synchronization and a fork-join parallelism originating historically from threading control make achieving good performance with traditional MPI+X programming difficult.

In the INTERTWinE H2020 project we started the development of the Task-Aware MPI (TAMPI) library[3]. 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 relay 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.

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.,

[3]Task-Aware MPI[7], `https://github.com/bsc-pm/tampi`

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 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.

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.

```
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 exascale booster with GPUs and the compute 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 and relies on efficient communication through the EXTOLL GCE (see Chapter 2.3). 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.

3.4 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 reffig:sec:ompss2:cuda:kernel). 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, we have added the definition of the *forces_block_t* 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. Support for CUDA kernel offload is developed and tested with example applications.

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

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

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

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

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

(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.

3.5 Using OmpSs-2 on the SDV

To facilitate the use of the programming model including its tool chain for application developers, we have published a manual and tutorials [4]. Examples show-case the use of all aforementioned programming features and include different examples including the presented NBody code. Further, on the DEEP-EST TRAC [5] we have are progressively adding information on how to load modules, compile and trace applications on the SDV.

[4] https://pm.bsc.es/ompss-2

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

4 Data Analytics programming model

The initial co-design phase of the project identified the frameworks and libraries used by the WP1 applications. We updated the list of software requirements at M18. In addition we identified Easybuild as a good candidate to build, install and manage the frameworks on the DEEP-EST prototype. Support with the WP1 application developers started with Intel and EPCC. The goal is to further optimize the applications for the underlying hardware.

4.1 Updated list of framework requirements

The list of required frameworks defined in deliverable D6.1 has been updated. The list is shown on Table 1. In comparison to the previous list, Pytorch, Horovod, scikit-learn, mpi4py and Dist-Keras have been added to the requirement. The underlying version of Python will be installed from the Intel Distribution for Python. This package delivers faster application performance on Intel platforms.

PyTorch [16] is now used by the DLMOS application and replaced the TensorFlow implementation. The main difference relies on the computational graphs. The Tensorflow graph is static whereas the Pytorch graph is dynamic. The dynamic approach facilitates the debugging step. The classical Python debugging tools such as pdb, ipdb, or any print statements can be used. In Tensorflow, one must use a special tool called tfdbg to evaluate the expression in the session scope.

Scikit-learn [17] provides a range of machine learning algorithms. It is built on NumPy, SciPy, and matplotlib. It is more generic than Tensorflow or Pytorch. It offers algorithms for classification such as SVMs, Random Forests, Logistic Regression etc... Scikit-learn is a higher-level library that includes implementations of several machine learning algorithms.

Horovod [18] is the Uber's[1] open source distributed deep learning system for TensorFlow, Keras, PyTorch, and MXNet. The Horovod framework can be deployed on CPU or GPU. Horovod uses the data parallelism approach. Each execution unit operates on the same model, only the training data is splitted up among the units in parallel. The averaging gradients are communicated among the nodes using the ring-allreduce approach.

Dist-Keras [19] is built on top of Apache Spark and Keras to distribute the training. The framework is developed by the Database Services Group (IT-DB) at CERN. The CERN Deep-Learning application uses the Dist-Keras framework to achieve data parallel model training.

All the frameworks will be installed on the DEEP-EST prototype system, including the CM, DAM and ESB. To manage the software installation on the modules we will use Easybuild.

[1] Uber Technologies, Inc. provides e-commerce services for car hire and develops technologies for autonomous driving.

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

4.2 Framework installation

The machine learning software installation will be done with Easybuild [20]. The goal is to build the software from source on the HPC module to have highly optimized tools. We want to avoid using readily available binary packages that were built in a generic way. In addition, we want to provide easy access for the users via the module system. Traditionally on HPC systems, the users request the installation of software requiring several dependencies and building steps from the support team. The Easybuild framework simplifies the process by autonomously building and installing the scientific software. It provides automatic dependency resolution and automatic module file generation. It allows for automated and reproducible builds of software.

The support team at JSC started to deploy the new software stack on the existing software development vehicle (SDV) managed via Easybuild. Tensorflow has been successfully built and made available to the users using the module system. Environment Modules are a standard and well-established technology across HPC sites. The installation would be carried out by the administrator of the system and not by the application developers to avoid redundant installation.

The new software stack is arranged hierarchically where the standalone tools are shown at first. Only modules that are compatible with each other are available for loading. With this organization, loading different incompatible software is not allowed. The users cannot mix different software together. As an example, to load the TensorFlow module the user has to load the Intel Compiler and the MPI module first. The module workflow generated by Easybuild is shown on Figure 14 (the number of module has been simplified).

To detail the full hierarchy of modules, the command `module spider` is available.

4.3 ESB design update

The Extreme Scale Booster (ESB) proposal has been modified to use Nvidia V100 Tesla GPGPU accelerators as Booster Nodes. The same GPGPU will be available in the DAM. We modified the Data Analytics programming model in the ESB to install the frameworks with the Nvidia proprietary optimization libraries. At compile time, the build script will have to specify the GPU flag. We will reuse the Easybuild installation script of the DAM for GPGPU on the ESB.

The WP1 application updated the partitioning scheme among the modules with the new ESB design. We summarized the usage of the DAM and ESB for the machine learning application. University of Iceland proposed different mapping for training and inference using the ESB and the DAM. They are covering different machine learning disciplines, namely: Clustering, with

```
$ module avail
----------------------------usr/local/software--------------------------------
CMake/3.12.2    Python/2.7.14    Python/3.6.5    Intel/2018.2.199-GCC-5.5.0

$ module load Intel/2018.2.199-GCC-5.5.0
----------------------------usr/local/software--------------------------------
CMake/3.12.2    Python/2.7.14    Python/3.6.5    Intel/2018.2.199-GCC-5.5.0
mkl-dnn/0.13    ParaStationMPI/5.2.1-1 (D)

$ module load ParaStationMPI/5.2.1-1
----------------------------usr/local/software--------------------------------
CMake/3.12.2    Python/2.7.14    Python/3.6.5    Intel/2018.2.199-GCC-5.5.0
mkl-dnn/0.13    ParaStationMPI/5.2.1-1 (D)     TensorFlow/1.8.0-Python-3.6.5
imkl/2018.2.199    SciPy-Stack/2018a-Python-3.6.5    PETSc/3.9.0_int8

$ module load TensorFlow/1.8.0-Python-3.6.5
```

Figure 14: Easybuild and environment module workflow

HPDBSCAN; Support Vector Machines (SVMs), with PiSVM; and deep learning. Only the deep learning application will use the DAM frameworks (Tensorflow with Keras). The DAM and ESB modules will be used concurrently, the training and inference mapping will be tested and compared. KuLeuven-DLMOS proposed to map the training step on the DAM and inference step on the ESB GPU. In fact, the xPic application will be running on the ESB and the inference output is used by the xPic code. CERN Deep-Learning is used for event classification. The initial feature engineering step will be performed on the DAM, the training will be executed as well on the DAM and the inference on the redesigned ESB.

4.4 Application support

In order to achieve high performance and energy efficiency, we are supporting the application developers in their application development. We started the collaboration between Intel, EPCC and the WP1 developers. EPCC will provide support to KU Leuven and CERN starting in M22. Intel provided support to CERN regarding the Intel OpenCL SDK for Intel FPGA. In particular matrix operation kernels (as Cholesky decomposition) for non-negative least squares (NNLS) were implemented and optimized using OpenCL.

In addition, Intel provided two Intel FPGA Arria 10 PCIe cards (Intel Programmable Acceleration Card) at JSC for testing purpose. Intel provided support to install the software stack, especially the Intel Acceleration stack. We plan to give access to the WP1 application developers starting in M22.

4.5 Future work

In the next months, we will continue the software installation in all the modules at JSC. The plan is to first install the frameworks (see section 4.1) at the CM starting in M20, at the DAM in M24 and finally at the ESB in M30 after the module is installed.

The Easybuild installation will be tested on each module and the different framework specific optimizations will be evaluated. We will work closely with the WP1 developers to assure the quality and integrity of the frameworks modules.

To support the application, the Intel FPGA Arria 10 cards will be accessible to the WP1 application developers in M22. EPCC will work directly with the developers to port and optimize the 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, NAM, memkind, erasure coding or Amazon S3.

When finished, the storage plugin interface will be obtainable together with the published BeeGFS source code. This implies that, besides the plugins provided by the BeeGFS team, interested third-party developers may write and publish custom plugins optimized for their architecture or technology.

5.1.1 Architecture outline

About the context inside 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 as an **object store**, which requires the handling of operations such as read, write, getting and setting of attributes for chunks identified by specific keys.

Optionally, features such as quota accounting and **resync** as well as **fsck** support may be implemented by a storage plugin.

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 plugins.
- A ”dummy” plugin which keeps all information on the heap.
- 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 Integration into storage server

The storage plugin currently provides functionality of a minimal prototype, the integration is ongoing. As a preparation for the storage plugin API, ChunkBuckets have been introduced to supersede a combination of different attributes that described the path of a chunk on the storage server. Chunk buckets are a 32 bit key to group chunks and improve locality and thus latency for accesses by individual jobs. Also, they simplify the storage plugin API and allow the later adaptation of new placement algorithms.

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 integratable 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, let us 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 from 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 and documented and easiest to use 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. Some default Grafana panels come with beegfs-mon.

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 straight forward 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) is 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) is collected by each server node on a per host or per system user base.

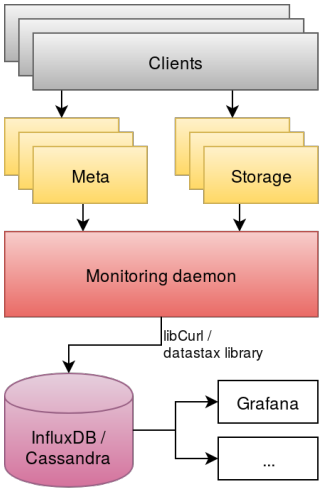


Figure 15: BeeGFS-mon data flow: The daemon pulls data from the server nodes in regular intervals and puts them into the database, where they are available for monitoring/evaluation.

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 partake in these operations. This enables an exchange of I/O data between the processes, allowing a subset of all processes, the *collector* processes, to perform the actual transfer of data to the storage on behalf of other processes. Collector processes should typically be those processes that are placed on parts of the MSA with a high bandwidth connection to the parallel file system. The mechanism has been extended with a new MSA-aware algorithm for the selection of collector processes. The algorithm is portable and relies on platform specific plug-ins to identify processes which run on parts of the system that are well suited for the role of I/O collector. So far, a specific plug-in for the DEEP-EST SDV has been implemented as well as a test plug-in. In the future, further plug-ins for new systems of the MSA type can be added.

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

```
./configure --msa=deep-est-sdv # ... 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 keys and key value pairs. The word collmsa must appear in that list to select MSA aware collective I/O.

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

Additionally, as is the case when using regular collective I/O, the size of collector groups has to be secified, 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 the programming interface offered by other libraries, such as ParaStation MPI (section 2.3), more closely, 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 needing 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

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

- 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

Our experiments with I/O forwarding have so far not reached a state that is mature enough to be made into a public API for user consumption. It will thus not be part of the software environment installed as part of this deliverable.

6 Resiliency

In this chapter we will comment on the progress of T6.5 - Resiliency. This includes enhancements implemented into FTI, new directives that form the specification of OpenCHK - a new pragma based checkpoint and restart interface - and a short discussion on ports of the newly developed technologies into the project applications.

6.1 Differential Checkpointing (dCP)

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 a following checkpoint, the checkpoint data is merely updated by the differences. The application of this technique leads to a significant reduction of checkpoint overhead. The actual degree of reduction depends on the application type as well as on the checkpointing frequency.

As we proposed in deliverable D6.1, we have implemented dCP into FTI. 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.

The current implementation is focused on keeping the stress on the meta data server and memory/storage consumption low. Thus, besides minimizing the storage and memory consumption, we aim to minimize the number of files that constitute the checkpoint. We developed a file format that is capable of keeping immutable locations of protected datasets inside the checkpoint file, but, still allows protected datasets to change their size. With this we can update checkpoint files directly with the differences without creating additional storage needs. The new file format also incorporates common FTI meta data in the checkpoint file[1]. With this, every checkpoint consists of merely one file per rank.

A study, performed by our group, shows that dCP is beneficial for most of the scientific applications, however, the method we have used reveals flaws in particular situations. As we have stated above, we update the data differences directly inside the checkpoint file. We have observed that this leads in some cases to many small non-contiguous updates which result into many low level system I/O calls. This leads to significant overhead in some cases.

We can avoid this effect by writing the data differences contiguously into a file stack and using buffered I/O. On recovery the most recent checkpoint data is fetched from the file stack and loaded to the application datasets. Another advantage of this approach is that it simplifies the port of dCP to the other reliability levels in FTI. For partner checkpoints we simply have to transfer the top file of the stack to the partner node and for the encoded checkpoint level, we

[1] FTI has to keep some meta information regarding the checkpoint files. This meta data is kept in ascii files on the PFS. The amount of those files corresponds to the number of ranks devided by the number of groups. A group in FTI is comprised by a user defined number of nodes.

just need to encode the top files of the stack.

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 16).

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

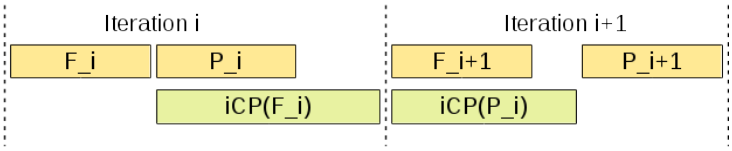


Figure 16: Overlapping checkpoint I/O with computation. In yellow the computations of the forces and particle displacements and in green the incremental checkpoints of the respective quantities.

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 prag-

mas is the portability of the code and the reduction in complexity. The developer does not need to know about the specific API of any of the underlying CR libraries.

The mechanism is represented essentially by four directives (see figure 17). 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 17: 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 and the second mode, one file which is shared among all processes. The aim of the second mode is to enable the restart from a checkpoint 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 18).

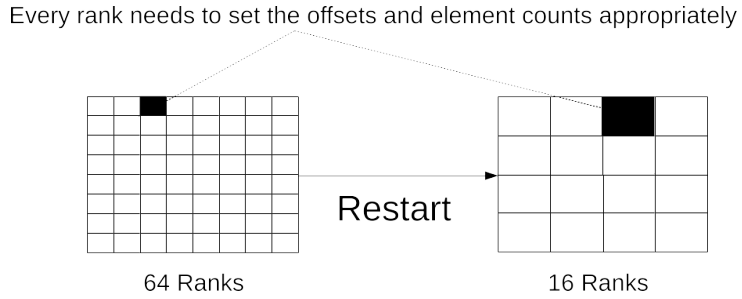


Figure 18: Variant processor recovery.

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 has 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 restarts from checkpoints with variate topologies, due to this need, we started developing the HDF5 variate processor restart feature.

6.6 Future Work

In the next months, we will focus on the proper integration of FTI into xPic and NEST and possibly on other applications of the project. The goal is to implement all the developed features and test them in large cluster runs. Beside this, we want to improve the dCP mechanism by adding the second approach (see file stack approach in section 6.1) to the existing implementation.

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. Ayguad, "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] Kevin Sala, Jorge Belln, Pau Farr, Xavier Teruel, Josep M. Perez, Antonio J. Pea, Daniel Holmes, Vicen Beltran, 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.

[8] 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.

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

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

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

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

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

[14] 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 Com-

puting (HiPC), IEEE Computer Society, pages 376-385, December 2015
http://ieeexplore.ieee.org/document/7397653/

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

[16] Paszke, Adam et al. “Automatic differentiation in PyTorch.” (2017).

[17] 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

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

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

[20] 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