H2020-FETHPC-01-2016

DEEP-EST

DEEP Extreme Scale Technologies

Grant Agreement Number: 754304

D6.4
Programming environment support report

Final

Version:	1.0
Author(s):	V. Beltran (BSC)
Contributor(s):	C. Clauß (ParTec), T. Moschny (ParTec), M. Peuten (ITWM), P. Reh (ThinkParQ), H.C. Hoppe (Intel), B. Steinbusch (JUELICH)
Date:	31.3.2021

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:	D6.4
	Deliverable Nature:	Report
	Deliverable Level: PU*	**Contractual Date of Delivery:** 31.03.2021
		Actual Date of Delivery: 31.03.2021
	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:** Programming environment support report	
	ID: D6.4	
	Version: 1.0	**Status:** Final
	Available at: `http://www.deep-projects.eu/`	
	Software Tool: LaTeX	
	File(s): DEEP-EST_D6.4_Programming_Env_Support_Report.pdf	
Authorship	**Written by:**	V. Beltran (BSC)
	Contributors:	C. Clauß (ParTec), T. Moschny (ParTec), M. Peuten (ITWM), P. Reh (ThinkParQ), H.C. Hoppe (Intel), B. Steinbusch (JUELICH)
	Reviewed by:	Estela Suarez (JUELICH) Susanne Kunkel (NMBU)
	Approved by:	BoP/PMT

Document Status Sheet

Version	Date	Status	Comments
1.0	31.03.2021	Final	

Document Keywords

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

Table of Contents

List of Figures

List of Tables

Executive Summary

This deliverable presents the programming environment developed to support the Modular Supercomputing Architecture (MSA) proposed in the DEEP-EST project. This document extends the work presented on Deliverables D6.2 and D6.3, with the latest developments done in each task and also describes the maintenance experience regarding installation, bug fixing and user feedback. The original work stated in deliverable D6.1 has been refined and adapted based on application requirements, as well as on the new ESB design based on GPUs. The goal of the programming environment is twofold: Firstly, to facilitate the development of new applications and the adaptation of existing ones to fully exploit the MSA architecture. Secondly, this work contributes with optimizations and extensions of key software components such as message-passing libraries, task-based programming models, file systems, checkpoint/restart libraries, and data analytics and machine learning frameworks to leverage specific hardware features that are available on the Cluster Module (CM), the Extreme Scale Booster (ESB) and the Data Analytics Module (DAM).

1 Introduction

Deliverable 6.1 presented the programming environment to be developed and adapted according to the requirements of the Modular Supercomputing Architecture (MSA) as described in D3.1 *System Architecture*, as well as, to the initial requirements of the applications. In Deliverable 6.2 we described the overall programming environment including the different components and features defined in the initial document. We also highlighted any deviation from the original plan as required to add better support for the new ESB design. The main goal of the programming environment is to facilitate the development of applications so that they can effectively exploit the MSA architecture including specific hardware features of each module. In some cases this can be achieved transparently, while in others, modifications of the applications will be needed. In this Deliverable (D6.4) we are updating the previous one to reflect the new developments done and the status of the different components.

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

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

2 ParaStation MPI

While the last deliverable D6.3 presented an overview of all the MSA features developed for ParaStation MPI in DEEP-EST so far, this section of this deliverable focusses on the extensions to ParaStation MPI that have been realised since then. This concerns in particular the implementation of wrapper layers for the integration of libGCE / GCE as well as libNAM / NAM with their respective features in ParaStation MPI. Nevertheless, in the following first section, the aspects already presented in the previous deliverable [1] on the support of MSA-aware programming by ParaStation MPI are also briefly revisited.

2.1 Modularity awareness

2.1.1 Modularity reflecting communicators

In order to adapt an MPI application to modularity, it is desirable to adjust also its communication patterns to the hierarchical topology that an MSA usually has. To achieve this, the use of MPI communicators that reflect the underlying topology is particularly suitable. For this purpose, ParaStation MPI provides each process with the information about its own module affiliation within the MSA in the form of a module ID, which can be queried at application level via a corresponding extension of the standardised `MPI_INFO_ENV` object. This module ID can then be used by the application as the colour parameter when calling `MPI_Comm_split()` to create modularity-aware communicators. This is because this function divides the group of processes of an existing communicator (e. g., like `MPI_COMM_WORLD`) into disjoint subgroups based on the colours passed. By querying the size of such a split communicator, the processes of MPI sessions spanning multiple modules can gain knowledge about the number of processes within the module-local subgroup in addition to the knowledge about their own module affiliation.

2.1.2 Modularity-aware MPI collectives

The information about the module affiliation is not only available to the application alone, but also ParaStation MPI itself can use this locality information to realise adapted patterns to optimise collective communication operations. Furthermore, this information is not only limited to the module affiliation, but can also be extended to the node affiliation, for example. Through a recursive use of communicators with multi-level hierarchy awareness (as described in more detail here in Section 2.2.3), adaptations of collective communication patterns can be achieved even to multi-tier topologies. For this purpose, such communicators each have so-called shadow communicators, which map the communication within or between subgroups and which may themselves have also shadow communicators. Such a chain of communicators thus represents the hierarchy, and when a collective operation is invoked, the following scheme is applied along this chain at each level of the hierarchy: (1) First do all gathering and / or reduction operations within the subgroups—if required. (2) Then perform the inter-module operation with only one process per subgroup being involved. (3) Finally, distribute the data within each subgroup in a group-local manner.

2.2 Integration of libGCE

2.2.1 The API of the libGCE

The libGCE is Extoll's library intended for the low-level use of the GCE. It therefore provides an API with a modest range of functions for the use of the accelerated collective communication patterns. This API semantically matches almost one-to-one with the MPI interface of the corresponding collectives. However, it should be noted that the collectives offered by libGCE are all of *non-blocking* nature, which means that computational tasks such as a reduce operation on the GCE can take place in parallel with computations on the host nodes. Furthermore, since the interleaving of computation and communication through the GCE promises the best performance benefits, the libGCE API covers only those counterparts of the collectives from the MPI standard, which actually include both computation and communication. These are in particular:

- `MPI_(I)Reduce()` ⟶ `gce_reduce()`
- `MPI_(I)Allreduce()` ⟶ `gce_allreduce()`
- `MPI_(I)Reduce_scatter)` ⟶ `gce_reduce_scatter()`
- `MPI_(I)Scan()` ⟶ `gce_scan()`
- `MPI_(I)Exscan()` ⟶ `gce_scan_ex()`

These libGCE functions, being non-blocking, return a handle that can be used for checking the progress or completion of the operation. The functions `gce_check()` and `gce_wait()` are available for this purpose, and the difference between these two functions is that the former is again non-blocking, while the latter waits for the started communication pattern to be completed.

2.2.2 A wrapper layer towards libGCE

In order to provide the GCE functionalities transparently to MPI applications, an interface between libGCE and ParaStation MPI was realised by an additional wrapper layer, being as thin as possible. Due to the similarity of both APIs, the actual mapping of the MPI collectives to libGCE operations is in a first instance just formed by the mapping of the MPI function parameters to their counterparts on the libGCE side plus the respective function invocation. Regarding the parameters, first the used MPI communicator is mapped to an array of Extoll RMA2 addresses. Furthermore, the MPI data types and the MPI reduction operations must be translated, which can be easily done by means of their numerical values via a mapping table. It should here be noted, however, that neither derived MPI data types nor user-defined reduction operations are supported at this point by libGCE. If such are nonetheless used by an application when calling an MPI collective with GCE support, the libGCE wrapper layer automatically performs a *fallback* to the point-to-point based communication patterns offered by the upper MPICH layer of psmpi so that the communication is then eventually handled by the pscom library. The same applies if the libGCE environment cannot be initialised by all processes of an MPI session, for example, due to a lack of GCE resources. For such a case, the wrapper layer ensures that all processes of an MPI job have a coherent view during initialization so that all processes can perform the fallback together for the entire session.

Control options

If the presumed performance loss resulting from such a fallback is not to be tolerated silently, a user can also influence fallback behaviour by setting the following environment variable accordingly:

- `PSP_LIBGCE=0` – Do not use the GCE at all (even if GCE support is available).
- `PSP_LIBGCE=1` – Use GCE if possible. Perform a fallback to point-to-point if not.
- `PSP_LIBGCE=2` – Try to use the GCE and abort with a message if not possible.

For a more detailed analysis of GCE utilization, ParaStation MPI provides the user in addition with a statistics feature that can be used to print the number of successful GCE operations for each process at the end of an MPI run. This feature can be activated by setting the environment variable `PSP_LIBGCE_STATS=1`.

It is assumed that libGCE is CUDA-aware so that GPU device memory pointers can be passed directly via its API. Nevertheless, ParaStation MPI already allows this to be switched off (for example for debugging purpose or for performance measurements), so that the CUDA-awareness of ParaStation MPI then takes over the handling of the GPU buffers by using point-to-point communication via pscom for the collectives. This can be enforced by setting the environment variable `PSP_LIBGCE_CUDA=0`.

Progress handling

Since the GCE can perform not only interleaved but true parallel progress independently in the background, the libGCE API does not require a function to trigger progress explicitly—which also simplifies its integration with the progress mechanisms of ParaStation MPI. Consequently, the implementation of `MPI_Test()` only needs to look at the result of the `gce_check()` function to decide whether a started collective has been completed by the GCE in the meantime.

However, `MPI_Wait()` cannot be implemented using `gce_wait()` internally, since otherwise the progress of any pending parallel point-to-point communication, as handled by pscom, is not guaranteed and a deadlock may occur. For this reason, waiting for communication completion is realised in the wrapper layer through alternating checks via `gce_check()` with the triggering of progress in pscom.

A mock-up for testing the wrapper

In order to test the implemented wrapper layer even before the availability of the real GCE hardware, the function prototypes of the libGCE's API have additionally been backed by a mock-up implementation. This implementation just performs a mapping of the GCE operations back to the MPI collectives as offered by the upper MPICH layers of ParaStation MPI through point-to-point communion. In this way, the intended progress semantics could already be tested even without libGCE—but, of course, with the constraint that the behaviour of the real libGCE must later match the semantics as discussed and agreed with the experts from Extoll. Therefore, when the GCE hardware becomes available, the developed mock-up layer should be easily replaceable by the actual libGCE.

2.2.3 Combining libGCE use with MSA awareness

In a first approach, GCE support in ParaStation MPI was only implemented for the non-MSA-aware case, which means that the GCE here can only be used if all processes in an MPI session are also part of the same Extoll network. This is because otherwise, some processes would report an error when initialising the libGCE environment so that then, according to the fallback mechanism described above, all processes together will choose to not use the GCE collectives.

Checking GCE usability during communicator creation

To enable MSA awareness, the initialisation of the libGCE can be delayed in ParaStation MPI until MPI communicators are created. In this way, whenever a new communicator is created (including `MPI_COMM_WORLD`), it is checked whether the libGCE environment has already been successfully initialised and whether the processes of the respective communicator are all running on nodes with Extoll / GCE connectivity. In doing so, if the libGCE environment has not yet been initialised, an attempt is made to achieve this with the set of processes in the communicator. In order to ensure that this mechanism actually covers all processes with Extoll / GCE connectivity of an MPI session, this approach is combined with the shadow communicators for MSA awareness, as they have been described in Section 2.1.1 at the beginning of this chapter. However, to combine both approaches effectively, the latter had to be extended to feature a more generic solution for multi-level hierarchies, as described in the following paragraph.

Multi-level hierarchy-aware communicators

For a generic support of multi-level hierarchy-aware communicators, the subdividing scheme of node and module affiliations has been broadened so that affiliations can now be described more flexibly via so-called *badges* for the processes and *degrees* for the topology levels. Here, badges represent the process affiliation and thus correspond to the previous node and module IDs, whereas the associated degree, being also an integer value, describes the respective hierarchy level. Therefore, typical (and hence pre-defined) degrees are those that describe the node and the module affiliation levels. However, the network affiliation, for example, can also be described via an additional degree, if this information is not already given by the module affiliation:

- `PSP_TOPO_LEVEL_DEGREE_NODES = 1024`
- `PSP_TOPO_LEVEL_DEGREE_MODULES = 4096`
- `PSP_TOPO_LEVEL_DEGREE_NETWORKS = 8192`

In this example configuration, `DEGREE_NETWORKS` could represent the affiliation to a network domain, which in turn may reflect the ability to use an accelerator such as the GCE jointly for collective communication patterns.

When arranging such levels, however, it is important that the sets of processes that can be formed by the badges at one level continue to be disjoint in the sets at levels with a smaller hierarchy degree accordingly. This has the consequence, for example, that with a DAM module comprising an Extoll (DAM-EXT) and a non-Extoll (DAM) part, this must logically be mapped on two sub-modules when using the scheme above—at least if this Extoll part of the DAM module is to be described as a joint GCE domain together with the Extoll nodes of the ESB. Using such a configuration scheme, for example, in an MPI session covering multiple modules, the MPI_COMM_WORLD communicator would then first be split into shadow communicators based on the Extoll / non-Extoll property (i. e., inside / outside the GCE domain), which are only then further subdivided on the basis of the module affiliation (e. g., like CN, ESB, DAM, and DAM-EXT, see Figure 1), and finally on node level.

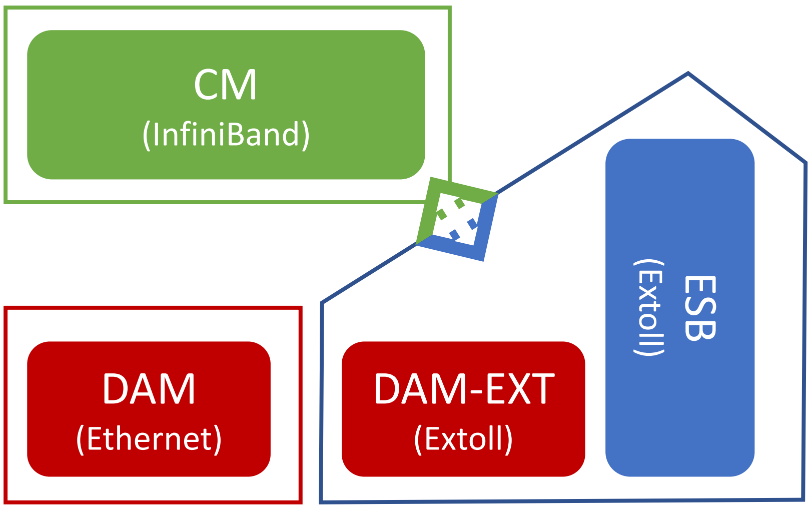

Figure 1: Division of the DEEP-EST system into networks and modules

The use of MPI_COMM_WORLD would then lead, when using a modularity-aware collective as described in Section 2.1.2, to a scenario where the pattern of the collective is being split in a first instance between processes running within the GCE domain and those being part of the rest of the network federation. In this way, the processes within the GCE domain can use the corresponding acceleration at least on their part of the split pattern for the modularity-aware collective. However, if a collective is not modularity-aware, the processes would detect that not all of them can use the GCE at MPI_COMM_WORLD level during communicator creation, and would then fallback to point-to-point communication via pscom as described above.

2.3 MPI support for the NAM

The basic concept concerning the planned integration of the NAM with the MPI environment has already been introduced and discussed in the last deliverables [1, 2, 3], and also the shared-memory based prototype developed at that time has already been presented there. In the meantime, however, a lot of work has been done regarding the NAM integration to ParaStation MPI, especially driven by the newly gained availability of the libNAM together with the Software NAM as a test vehicle. This section therefore first reviews briefly the semantics of the integration concept, in order to go subsequently into more detail about the wrapper layer newly implemented for the actual integration. In doing so, the interfaces of this wrapper layer to the applications as well as towards the NAM and its management as a shared resource are presented here.

2.3.1 Integrating the NAM into the MPI world

The key problem with mapping the MPI RMA interface to the libNAM API is that MPI assumes that all target memory regions for RMA operations are always associated with an MPI process as the owner of that memory. This means that in an MPI world remote memory regions are always addressed via a process rank (plus handle, which is the respective window object, plus offset), whereas the libNAM API only requires an opaque handle to address the respective NAM region (plus offset). Therefore, a mapping between remote MPI ranks and the remote NAM memory regions needs to be realized. The pursued concept for this mapping and thus for integrating the NAM semantically into the MPI world is to adhere to the notion of an ownership in the sense that certain regions of the NAM memory are *logically* assigned to certain MPI ranks. This in turn implies that the associated MPI window regions (although globally accessible and located on a NAM) are then to be addressed via the rank of the process to which the NAM region is logically assigned. However, it has here to be emphasised that this is a purely software-based mapping performed by the developed wrapper layer between ParaStation MPI and libNAM, and that no message forwarding is involved with a globally accessible NAM.

2.3.2 A new integration layer towards libNAM

PSNAM is the name of the developed integration layer between ParaStation MPI and libNAM, which has been rewritten from scratch during the last year of the project by incorporating all the experiences gained from the development of the early prototype. Therefore, the semantics of the MPI extensions at application level continue to follow the basic ideas as developed and documented in the early phases of the project. Furthermore, in addition to the memory regions on the NAM, persistent shared-memory regions on the compute nodes (as they were used in the early prototype for evaluation) are also still supported by PSNAM.

Design and configuration

Basically, the design of PSNAM is so generic that, besides libNAM and regular shared-memory regions, also other memory back-ends could easily be supported—provided that these memories are addressable via Put and Get operations on byte granularity. Thus, the selection of such a back-end (even if there are currently only these two available) is one of the key configuration parameters that can be specified at application level when creating PSNAM-based MPI RMA windows. This selection and all other parameters that go beyond the known MPI API have to be passed from the application to the PSNAM layer in the form of string-coded key / value pairs via an MPI info object. The following three info keys provide the basis for all further configurations:

- `psnam_manifestation`
- `psnam_consistency`
- `psnam_structure`

The *manifestation* key specifies which memory type shall be used for a region. The value for using the NAM is `psnam_manifesation_libnam`, which also corresponds to the default value. However, since shared-memory regions in host memory are also still supported, a second valid value is here psnam_manifesation_pershm, which can be used for selecting such instead. In fact, when creating an MPI RMA window with multiple regions, each process in the communicator used can even choose a different PSNAM manifestation of these two for the composition of the window.

At this point it should be noted that with shared-memory regions only those processes that run locally to them on the same host can access them directly. In contrast to this, accesses to memory regions that are not local are then to be mediated by the PSNAM layer via message-passing, but still transparently for the MPI application. However, such a forwarding of accesses is of course not necessary for NAM regions—at least as long as all processes are part of the Extoll network and can hence access the NAM directly. In the case of MPI sessions running across module boundaries, however, such a forwarding of accesses for processes outside the Extoll network could also be realised, but which would imply significantly higher access times.

The *consistency* key specifies whether the memory regions of an RMA window shall be persistent (psnam_consistency_persistent) or whether they shall be released during the respective MPI_Win_free() call (psnam_consistency_volatile). However, it should be noted here that the persistence that can be generated via this parameter only refers to the case of multiple MPI sessions *within* a Slurm job. For persistent memory regions that are to survive also Slurm job endings, the Slurm burst buffer extensions as developed in Task 5.4 must be used together with reservations.

Possible memory layouts

The psnam_structure key specifies the memory layout as formed by the multiple regions of an MPI window. Currently, the following three different memory layouts are supported:

- psnam_structure_raw_and_flat
- psnam_structure_managed_contiguous
- psnam_structure_managed_distributed

The chosen memory layout also decides whether and how the PSNAM layer stores further meta-data in the NAM regions to allow a later recreation of the structure when reconnecting to a persistent RMA window by another MPI session.

Raw and flat This layout is intended to store raw data (i. e., untyped data) in the NAM without adding meta-information. According to this layout, only rank 0 of the given communicator is allowed to pass a size parameter greater than zero during the MPI_Win_allocate() call.[1] Hence, only rank 0 allocates one (contiguous) NAM region forming the window and all RMA operations on such a flat window have therefore to be addressed to target rank 0.

Managed contiguous In this case of a memory layout, again only rank 0 allocates (contiguous) NAM space, but this space is then subdivided according to the size parameters as passed by all processes in the respective communicator. That means that here also processes with a rank greater than zero can pass a size greater than 0, and hence acquire a rank-addressable (sub-)region within this window. Furthermore, the information about the number of processes and the respective

[1]Please note that MPI_Win_allocate() does not have a root parameter. Therefore, rank 0 is used as a substitute.

region sizes forming that window is being stored as meta-data within the NAM memory. That way, a subsequent MPI session re-connecting to this window can retrieve this information and hence recreate the former structure of the window.

Managed distributed In such a window, each process that passes a size greater than zero also allocates NAM memory explicitly and on its own—at least in so-called *on-demand* cases where the Slurm resource management is not involved. Each process then contributes this memory as its “local” region to the RMA window so that the corresponding NAM allocation becomes directly addressable by the respective process rank.

2.3.3 Integration with the resource management

If the NAM memory is managed with involvement of the Slurm burst buffer plugin developed in Task 5.4, then the PSNAM layer shall not request NAM memory from the NAM manager itself, but use the pre-allocated memory space as provided by the plugin. For this purpose, the plugin passes the names of such pre-allocated NAM memory regions as a colon-separated list via a environment variable to the PSNAM layer. (`NAM_ALLOCATIONS_FOR_JOB` is the name of the variable that is used for this and that might also be parsed by the application itself). In doing so, these names of the allocations correspond to the names that the user has previously given when specifying the job’s resource requirements. This way, either the user can take reference to these allocations explicitly by passing their names during `MPI_Win_allocate()` calls via the respective info object (with `psnam_libnam_allocation_name` as the info key) or the PSNAM layer can transparently check the list passed by the plugin for suitable allocations according to a first fit policy.

However, a downside of this approach could be that an allocation might demand for a quite fine-grained pattern of repeated allocations, potentially even of unpredictable size and / or with altering releases in between. Such demands are taken into account with the concept of *segments* in the PSNAM layer: A segment is a PSNAM *meta-manifestation* that maintains a size and offset information for a sub-region within a larger allocation. This offset can either be set explicitly via `psnam_segment_offset` by the application (e. g., for splitting an allocation among multiple processes, quite similar to the managed contiguous layout described above), or it can be managed dynamically and implicitly by the PSNAM layer (e. g., for using the memory of an allocation for multiple MPI windows). Moreover, this concept of segments can also be applied recursively, which means that an allocation that has already been divided into segments can even be further sub-divided into sub-segments, and so forth. This can thus make the handling of pre-allocated NAM memory regions quite flexible—but potentially also quite complex.

2.4 Project resources and references

Unfortunately, at the time of writing this deliverable, neither the GCE nor a real hardware NAM were available for further evaluation. Although the development of the MPI integration could be significantly advanced, especially by using the Software NAM, final tests and particularly the evaluation of the use of the new features by applications are still pending.

At this point, we would therefore like to refer to the various places in the project where resources and documentation regarding the results of Task 6.1 can already be found:

- Sources of ParaStation MPI with PSNAM extension: (check branches named *psnam-integration**)
  `gitlab.version.fz-juelich.de/DEEP-EST/psmpi`
- BSCW folder with the early NAM integration proposal and the current version of the user manual:
  `bscw.zam.kfa-juelich.de/bscw/bscw.cgi/2723001`
- Wiki page of the project with documentation about the usage of ParaStation's MSA features:
  `deeptrac.zam.kfa-juelich.de:8443/trac/wiki/Public/ParaStationMPI`

As soon as further results are available, they will be added to these places accordingly.

3 The OmpSs-2 Programming Model

The OmpSs-2 programming model has been extended through the project's duration to meet the MSA architecture's specific requirements. In this chapter, we focus on the work done since the last deliverable on five particular areas. In Section 3.1, we describe runtime optimizations to deal with fine-grained tasks, in Section 3.2, we present several techniques to make our runtime more energy-efficient, in Section 3.3, we explain how we can leverage CUDA and OpenACC kernels from OmpSs-2, in Section 3.4, we describe the extensions done in TAMPI to support the NAM. Finally, in Section 3.5, we present several benchmarks to illustrate the OmpSs-2 features.

3.1 Runtime optimizations

Task-based programming models like OmpSs-2 and OpenMP provide a flexible data-flow execution model to exploit dynamic, irregular, and nested parallelism. Providing an efficient implementation that scales well with small granularity tasks remains a challenge, and bottlenecks can manifest in several runtime components.

OmpSs-2 has three main components: the dependency system, the task scheduler, and the memory allocator, which tightly interact with each other. The first stage of a task's life cycle is its creation, which involves the memory allocator. The runtime then checks its data dependencies to determine if the task is ready or blocked based on the previous tasks' dependencies. Once all its dependencies are satisfied, the task becomes ready and is added to the scheduler, which will eventually schedule it on an available core. Once the task has been executed, it releases its dependencies so that its successor tasks may become ready. Note that the three components require a synchronization mechanism as they have to deal with multiple requests simultaneously. Thus, the application developer has to strike a balance in task granularity: it has to be small enough to provide sufficient work for all available cores while being coarse enough to evade runtime system overheads.

However, as applications scale out to more cores (or nodes) and the problem size remains constant, task granularities naturally decrease. At some point in the scaling process, tasks can become so small that the application is overhead-bound, and the scalability depends on the ability of the parallel runtime to handle small tasks.

The proliferation of many-core architectures and workloads with irregular parallelism and load imbalance have shifted the focus from traditional fork-join parallelism to task-based parallelism. Nevertheless, task management costs are still an important source of overhead, especially when using fine granularities. We have enhanced two critical components of the runtime system to manage fine-grained tasks: the dependency system and the scheduler. We have combined both with a state-of-the-art memory allocator to achieve very competitive performance. We have introduced a novel wait-free approach to implementing dependency management inside a parallel runtime. We have also defined the Atomic State Machine concept and its restrictions and formalized its wait-freedom. Additionally, we proposed a novel Delegation Ticket Lock that delivers very good performance compared to other state-of-the-art locks, while keeping the simplicity in the development of scheduling internals and policies. We also identified the critical contention bottleneck caused by memory management and tackled the problem by leveraging the jemalloc state-of-the-art scalable memory allocator. Finally, we implemented highly-detailed instrumentation to provide information

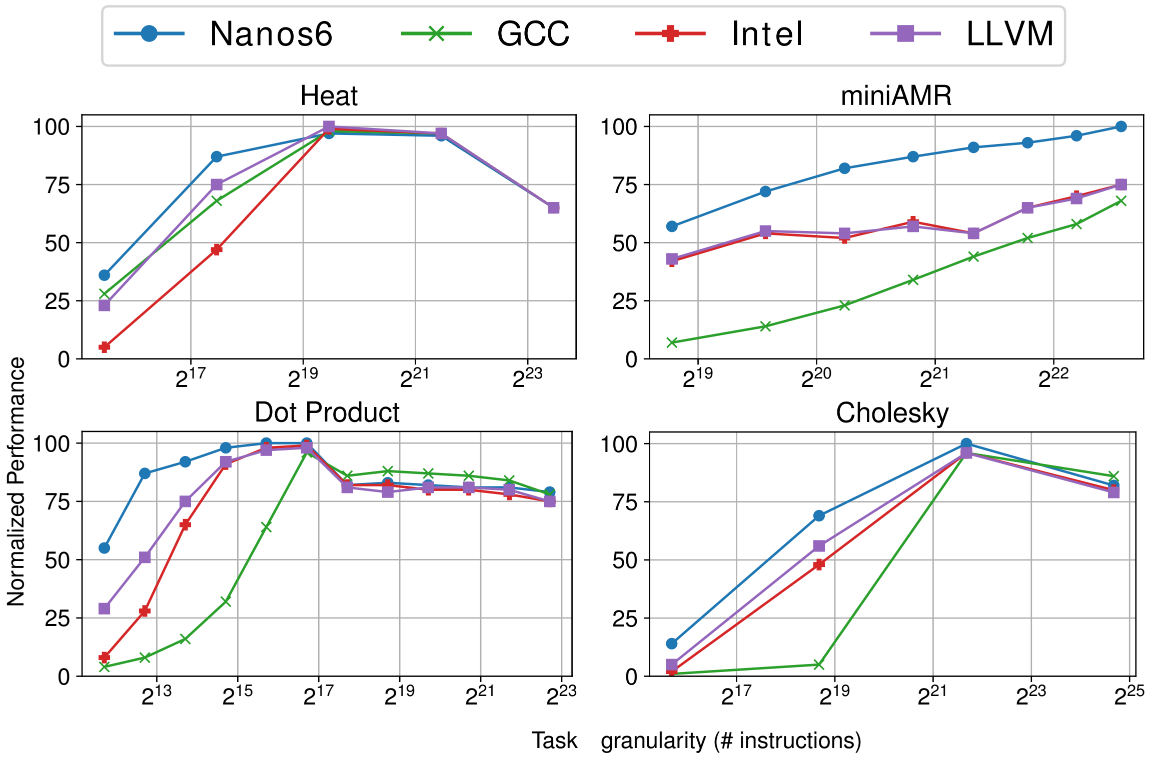

Figure 2: Comparison of performance between the optimized Nanos6 runtime and the main OpenMP runtimes on Intel Xeon (higher is better)

from both application and kernel levels, while introducing minimal overhead. Such a tool is crucial to identify and analyze bottlenecks in modern runtime systems. Our detailed evaluation presented at [13] assesses the performance of the different components separately and together, showing important performance improvements compared to (1) the previous version of the runtime system, and (2) state-of-the-art runtime systems such as Intel OpenMP, GNU GOMP and LLVM OpenMP.

Figure 2 shows the normalized performance of three state-of-the-art OpenMP runtimes and the optimized Nanos6 runtime on a Intel Xeon-based system. The problem size is fixed for each benchmark and the task granularity is measured in number of instructions. The results are really positive, as in all of the benchmarks, the best performance in small granularity tasks is provided by the Nanos6 runtime. In some cases, a higher peak performance is also achieved. This happens when the ideal block size for a specific benchmark is small enough that performs better in one runtime than another.

3.2 Energy-aware runtime system

Hardware and software techniques such as dynamic voltage and frequency scaling (DVFS) for core and uncore units or dynamic concurrency throttling (DCT) can be useful to save energy while maintaining performance. These techniques are complementary and can be used together. However, it is not easy to use them, as they have to be adjusted dynamically based on the workload, so it is even more complex to combine them properly. Ideally, these techniques should be applied transparently, without relying on application developers. To that end, we have extended Nanos6 runtime system with an infrastructure that categorizes workloads based on their computational profile: memory-bounded, compute-bounded, or balanced. This categorization is done in an on-line manner and with negligible overhead. With this additional information, we have enhanced the CPU-manager and scheduler of OmpSs-2, to automatically combine DVFS and DCT techniques at the core and uncore level based on workload characteristics.

Our analysis, submitted to [14], shows that combining these techniques can provide up to 15% better energy efficiency on average, achieving the energy efficiency of the best static configuration, and, in some situations, even improve performance in applications. Furthermore, we observed that using DCT for memory-bounded phases provides more benefits than using per-core DVFS. Finally, our findings can be summarized as an effective heuristic, which combines: (i) a DCT heuristic for memory-bounded phases, and (ii) an uncore DVFS heuristic for compute-bounded phases.

Below, we compare the vendor OpenMP implementation (IOMP), the baseline OmpSs-2 execution (OSS), a version of OmpSs-2 using core and uncore DVFS techniques (DVFS (uncore + core)), and a version of OmpSs-2 using the uncore DVFS technique in combination with the DCT technique (DVFS (uncore) + DCT). Thus, for memory-bounded phases we use DCT, while for compute-bounded phases we use DVFS. We normalize all the results to the best static configuration, having tested all the possible number of cores and every available frequency as well.

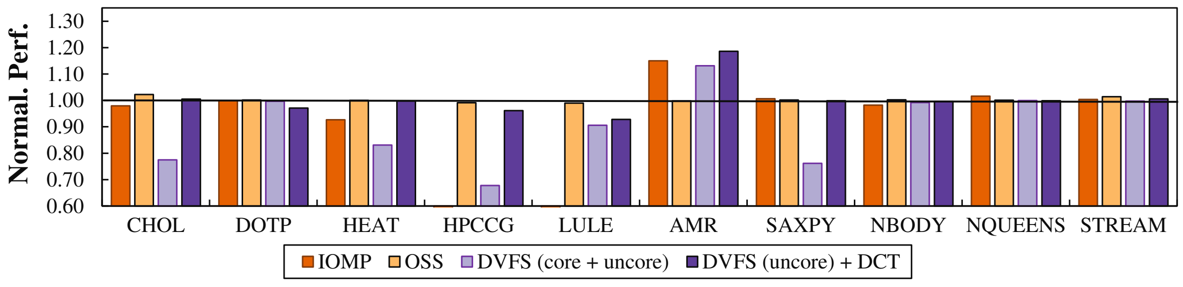

Figure 3: Performance results normalized to the best static configuration in SSF (↑ values = ↑ performance).

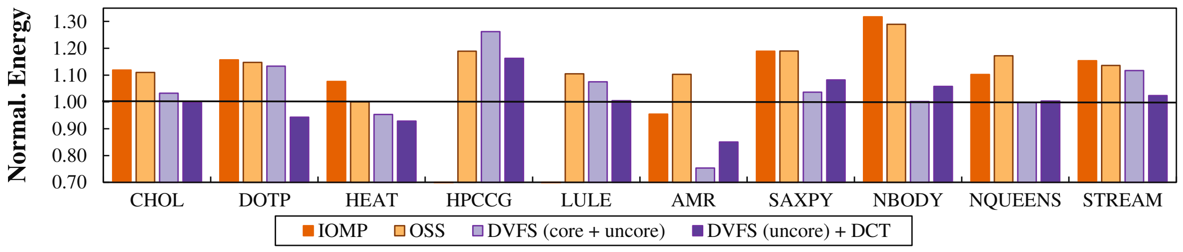

Figure 4: Energy results normalized to the best static configuration in SSF (↓ values = ↑ energy efficiency).

The overall results for all benchmarks show that performance is similar across all versions. When employing DVFS for both core and uncore units, we see that for MultiSAXPY and Cholesky, the performance worsens. Since some of these benchmarks have memory-bounded and compute-bounded phases, when decreasing the frequency of cores at a memory-bounded phase the performance suffers while executing compute-bounded kernels. For this reason, we include the last series in these plots, where we use DCT for core units and DVFS for uncore units, combining the best of both heuristics. In miniAMR we notice an increase of performance due to the previously explained reasons for the DCT heuristic. On the other hand, for the combination of DCT and DVFS (uncore) heuristics, we notice how the slowdowns for both Cholesky and MultiSAXPY disappear, as the negative effects of DVFS core are nonexistent, and for Heat and miniAMR we see improvements in performance. These are the mirrored improvements from the DCT heuristic combined with the benefits from using DVFS uncore.

Figure 4 shows the energy results of our DVFS heuristics. To ease readability we also include Table 1, which summarizes the average reduction in energy per benchmark of the DVFS (uncore) + DCT heuristic when compared to the baseline OmpSs-2 version (Energy wrt OSS), and average energy consumption reduction per benchmark when compared to the best static configuration (Energy wrt Optimal). In this last column, negative numbers indicate how close the energy consumption of our heuristics is to the energy consumption of the manually found static optimal. When obtaining the best static configurations by tweaking frequencies, we noticed that using DVFS uncore can benefit: (i) purely compute-bounded benchmarks by decreasing the frequency, but also (ii) memory-bounded phases in benchmarks by maximizing the frequency. In this last scenario, energy consumption is reduced by maintaining a static frequency in highly memory-bounded phases. As shown in the figure, for all benchmarks the energy consumption of the combination of heuristics (DVFS (uncore) + DCT) is always lower than the energy consumption of the OmpSs-2 baseline (OSS). Furthermore, the energy consumption of this version is always close to the best static configuration. As seen in Table 1, for compute-bounded benchmarks, we notice a considerable reduction of energy consumption due to DVFS uncore, more specifically 22% and 17% for NBody and NQueens, respectively. In other benchmarks where compute and memory-bounded phases are balanced (such as Cholesky, Lulesh and HPCCG), DVFS uncore also enables a reduction of the energy consumption, making it identical to the best static configuration by reducing it by 11% and 10% for Cholesky and Lulesh, or bringing it closer for HPCCG, with a reduction of 3%. Lastly, for memory-bounded benchmarks we notice a considerable reduction of energy consumption when compared to the baseline OmpSs-2 version. Specifically the energy consumption reductions are 21% for Dotproduct, 8% for Heat, 30% for miniAMR, 10% for MultiSAXPY, and 11% for STREAM.

Even though some these reductions may only account for 10% globally, it is important to notice that in almost every scenario, the energy consumption obtained using these heuristic meets the one reported by the best static configuration. In average, the energy consumption of all scenarios for the DVFS (uncore) + DCT when compared to the average energy consumption for the best static configurations is around 102%. This shows that in average, our heuristics are able to match the energy consumption of the best static configuration, and as previously shown, even improve it.

3.3 Support for Accelerators

The OmpSs-2 programming model supports accelerators by leveraging kernels written in other programming languages such as CUDA C, OpenCL C or OpenACC. These kernels are annotated like regular tasks, specifying the `input` and `output` parameters. With this information the OmpSs-2 runtime can coordinate the execution of tasks on the CPUs and kernels on the GPUs, managing the

	Energy wrt OSS (%)	Energy wrt Optimal (%)
CHOL	11.3%	0.2%
DOTP	21.7%	6.1%
HEAT	7.8%	7.7%
HPCCG	2.3%	-14.0%
LULE	10.1%	-0.4%
AMR	29.7%	17.6%
SAXPY	10.0%	-7.6%
NBODY	21.89%	-5.5%
NQUEENS	16.9%	-0.3%
STREAM	10.9%	-2.4%
AVERAGE	14.2%	0.15%

Table 1: Energy reduction improvements per benchmark

synchronization between tasks and kernels transparently. The data transfers between the host and the accelerator memory are also transparent to the application developer. There are two implementations of this feature, one that relies on specific hardware support to provide Unified Memory and another that is fully managed by the runtime system by using a directory/cache [15] to track data locations and perform the required memory transfers.

Besides normal CUDA kernels, OmpSs-2 runtime has also been extended to leverage optimized kernels provided by libraries such as cuBLAS and cuSolver, making it easier to port applications that rely on those kernels.

The OpenACC support for OmpSs-2 [16] has been developed in the context of the EPEEC[1] project. This feature allows the use of OpenACC annotations in sequential code to generate optimized kernels that can run on the GPU.

Section 3.5 describes several benchmarks modified to illustrate how OmpSs-2 applications can leverage CUDA and OpenACC kernels to exploit the ESB GPUs.

3.4 TAMPI extensions to support the NAM

The development of the Task-Aware MPI (TAMPI) library[2] was started in the context of the INTER-TWinE H2020 project. This library extends the functionality of standard MPI libraries by providing new mechanisms for improving the interoperability between parallel task-based programming models, such as OpenMP or OmpSs-2, and both blocking and non-blocking MPI operations. By following the MPI Standard, programmers must pay close attention to avoid deadlocks that may occur in hybrid applications (e.g., MPI + OpenMP) where MPI calls take place inside tasks. This is given by the out-of-order execution of tasks that consequently alter the execution order of the enclosed MPI calls. The TAMPI library ensures a deadlock-free execution of such hybrid applications by

[1]EPEEC Project funded by European Union's Horizon 2020 research and innovation programme under grant agreement №801051

[2]Task-Aware MPI [17] `https://github.com/bsc-pm/tampi`

implementing a cooperation mechanism between the MPI library and the parallel task-based runtime system. Moreover, applications that rely on TAMPI do not require significant changes to allow the runtime to overlap the execution of computation and communication tasks. TAMPI provides two main mechanisms: the blocking mode and the non-blocking mode. The blocking mode targets the efficient and safe execution of blocking MPI operations (e.g., `MPI_Recv`) from inside tasks, while the non-blocking mode focuses on the efficient execution of non-blocking or immediate MPI operations (e.g., `MPI_Irecv`), also from inside tasks. 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.

In the previous deliverable we detail the blocking and non-blocking modes of TAMPI. In this one, we will summarize the extensions done in TAMPI to support also MPI one-sided operations inside tasks. With this additional support we will be able to access the NAM memory leveraging the extensions of ParaStation MPI described in Section 2.3.

3.4.1 One-sided operations

As described in Chapter 2, ParaStation MPI has been extended to access NAM memory using the standard `MPI_Get`, `MPI_Put`, and `MPI_Win_fence` operations. Both `MPI_Get`, `MPI_Put` are non-blocking, so they can be safely used inside tasks. However, `MPI_Win_fence`, which is used to ensure that any previous `MPI_Get` and/or `MPI_Put` operation on a given MPI window has been completed, is a blocking operation and cannot be safely used inside tasks. To support `MPI_Win_-fence` on TAMPI we need a non-blocking version that, unfortunately, does not exist in the latest MPI standard. To address this situation, we have extended ParaStationMPI to support a non-blocking version that we call `MPI_Win_ifence`. This version has the same parameters as the original `MPI_Win_fence` but adds an extra parameter that is an `MPI_Request`, which can be tested using the standard `MPI_test` functions. With this additional function we have extended both the blocking and non-blocking interface of TAMPI to support `MPI_Win_fence` and `MPI_Win_ifence` inside tasks. Now it is possible to perform get, put and fence operations inside tasks safely to either access remote memory from another rank or on the NAM. Section 3.5.2 explains how we have used TAMPI one-sided operations to implement the halo-exchange in the Heat-Equation benchmark, as well as, saving the problem state on the NAM for each iteration.

3.5 Benchmarks and mini-Apps

We have used several well-known benchmarks and mini-apps to demonstrate the new features implemented in OmpSs-2. To that end, we have first ported these benchmarks and mini-apps from pure MPI or hybrid MPI+OpenMP fork-join model to a hybrid TAMPI + OmpSs-2 data-flow model. It is worth noting that moving from pure MPI or fork-join model to a data-flow model based on tasks already requires a significant effort. This is why large and complex applications have not been evaluated in the context of this project, as porting them to a task model will require a huge effort that currently cannot be justified by the expected performance improvements.

The miniAMR [12], Lulesh [11], HPCCG [11] have been ported to OmpSs-2 model to evaluate TAMPI and worksharing tasks. A distributed version of Cholesky and HPCCG benchmarks have been extended to evaluate OpmSs-2 support of CUDA and OpenACC, as well as, the MSA architecture.

3.5.1 HPCCG and Cholesky benchmarks

The original OmpSs-2 versions of HPCCG and Cholesky benchmarks have been modified to run on the ESB. To that end, all the tasks have been changed by optimized CUDA kernels from cuBLAS and cuSolver libraries. It is worth noting that for both applications its structure remains the same, and the runtime system is managing the synchronisation between CUDA kernels. The HPCCG version has also been extended to replace some of the optimized CUDA kernels with versions generated with OpenACC. Finally, an MSA-aware version of both benchmarks have also been developed. In this versions some of the kernels run on the cluster nodes, while others, run on the ESB. Both intra-cluster and inter-cluster communications have been implemented using TAMPI. All these versions, as well as, detailed instructions to execute them on the MSA are available in the project GitLab repository.[3,4]

3.5.2 Heat Equation

In this section, we exemplify the use of TAMPI one-sided primitives through the Heat benchmark. We use an iterative Gauss-Seidel method to solve the Heat equation, which is a parabolic partial differential equation that describes the distribution of heat in a given region over time. This benchmark simulates the heat diffusion on a 2D matrix of floating-point elements during multiple time steps. The 2D matrix is logically divided into 2D blocks and may have multiple rows and columns of blocks. The computation of an element at position `M[r][c]` in the time step `t` depends on the value of the top and left elements (`M[r-1][c]` and `M[r][c-1]`) computed in the current time step `t`, and the right and bottom elements (`M[r][c+1]` and `M[r+1][c]`) from the previous time step `t-1`. We can extrapolate this logic in the context of blocks so that a block has a dependency on the computation of its adjacent blocks. Notice that the computation of blocks in a diagonal is fully concurrent because there is no dependency between them.

There are three different MPI versions, and all of them distribute the 2D matrix across ranks assigning consecutive rows of blocks to each MPI rank. Note that the matrix is distributed by blocks vertically but not horizontally. Therefore, an MPI rank has two neighboring ranks: one above and another below. The exceptions are the first and last ranks since they have a single neighbor. This distribution requires the neighboring ranks to exchange the external rows (halos) from their boundary blocks in order to compute their local blocks in each time step.

This benchmark is available in the DEEP-EST public Gitlab repository.[5] The first version is based on an MPI-only parallelization, while the other two are hybrid MPI+OmpSs-2 leveraging tasks and the TAMPI library. We briefly describe each one below:

[3] `https://GitLab.version.fz-juelich.de/DEEP-EST/ompss-2-benchmarks/-/tree/master/hpccg`

[4] `https://GitLab.version.fz-juelich.de/DEEP-EST/ompss-2-benchmarks/-/tree/master/cholesky-mpi-oss`

[5] `https://pm.bsc.es/GitLab/DEEP-EST/apps/Heat`

Detailed instructions on how to compile and run this benchmark in the DEEP-EST system can be found in the DEEP Trac.[6]

1. `01.heat_mpi`: A straightforward MPI-only implementation using blocking MPI primitives (`MPI_Send` and `MPI_Recv`) to send and receive the halo rows. The computation of blocks and exchange of halos inside each rank is completely sequential.
2. `02.heat_itampi_ompss2_tasks`: A hybrid MPI+OmpSs-2 version leveraging TAMPI that performs both computation and communications using tasks with data dependencies. It instantiates a task to compute each of the blocks inside each rank and for each of the time steps. It also creates sending and receiving tasks to exchange the block halo rows for each of the boundary blocks. The execution of tasks follows a data-flow model because tasks declare the dependencies on the data they read/modify. Moreover, communication tasks call non-blocking MPI primitives and leverage the non-blocking mechanism of TAMPI (`TAMPI_Iwait`), so communications are fully non-blocking and asynchronous from the user's point of view. Communication tasks issue non-blocking communications that are transparently managed and periodically checked by TAMPI. These tasks do not explicitly wait for their communication, but they delay their completion (asynchronously) until their MPI communications finish.
3. `03.heat_tampirma_ompss2_tasks`: An implementation similar to `02.heat_itampi_ompss2_tasks` but using MPI RMA operations (`MPI_Put`) to exchange the block halo rows. This program leverages the MPI active target RMA communication using the MPI window fences to open/close RMA access epochs. It uses the TAMPI library and the new integration for the `MPI_Win_ifence` synchronization function. In this way, we use `TAMPI_Iwait` to bind the completion of a communication task to the finalization of a `MPI_Win_ifence`. Therefore, the opening/closing of RMA access epochs is completely non-blocking and asynchronous from the user point of view. We assume the calls to `MPI_Put` are non-blocking. Finally, as an optimization, we register multiple MPI RMA windows for each rank to allow concurrent communications through the different RMA windows. Each RMA window holds a part of the halo row that may belong to multiple logical blocks. Each communication task exchanges the part of the halo row assigned to a single MPI window. Finally, `MPI_Put` and `MPI_Win_ifence` are also used to save the state of each iteration on the NAM using tasks, so the state of the whole simulation is available for analysis after the program ends.

3.6 Using OmpSs-2 on the DEEP-EST prototype

To facilitate the use of the programming model including its tool chain for application developers, there are manuals and tutorials.[7] Furthermore, on the DEEP-EST TRAC[8] there is information on how to load modules, compile and trace applications on the DEEP-EST prototype. This information is periodically updated to reflect any change on the software stack or the DEEP-EST prototype system.

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

4 Data Analytics Programming Model

The previous Deliverable D6.3 did report on the set of frameworks for data analytics or AI installed on the DEEP-EST prototype, per the request of WP1 application developers. The tool chosen to manage the complex set of often interdependent SW packages is Easybuild, and this chapter gives an update on the installed frameworks in the first section. The installation status at the time of writing the document does reflect and satisfy the needs of the WP1 developers, and of third party users in the context of the Early Access Program.

The second chapter gives an update on the programming tools and frameworks used for the Intel FPGAs in the Data Analytics Module, the persistent memory toolkit and the third section summarises the activities around the new oneAPI programming interface and the Intel oneAPI toolkits.

4.1 Installed Frameworks

To accommodate requests for urgent updates or extensions of the installed set of SW packages and at the same time provide a stable platform for code development and benchmarking, three different Easybuild stages were created:

- *Production stage* (currently Stage 2019a): stable set of fully functional, usually downstream package versions, fully compatible with the base OS and supported by the WP1 support team
- *Development stage* (currently Devel-2019a): set of updated packages which are not yet considered fully stable, may require kernel updates to the base OS, and are candidates for future inclusion in the Production stage
- *Early development stage* (currently Devel-2020): upstream versions of packages which are seen as potentially unstable, introduce significant changes (maybe even in APIs), and require base OS updates or even upgrades

This staged approach is of particular importance for data analytics and AI frameworks, since the progress in package and API development is very brisk here, and changes between versions can be sweeping. A good example is TensorFlow, which in version 2.x did change over to a completely different backend architecture, breaking tools which support the backend and substantially changing performance characteristics.

Table 2 shows the updated list of packages required by the WP1 developers. Table 3 indicates the installed package versions for the above mentioned three stages.

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

Table 2: List of Data Analytics and AI frameworks required by WP1 Developers

Stage	Package	Python				Intel	ParaStation MPI		
		2.7	3.6	3.7	3.8	MPI	GPGPU	x86	x86 MT
Production	Python-GCCcore-8.3.0	Y	Y						
	Pytorch-1.1.0-GCCcore-MKL-8.3.0		Y						
	scikit-GCCcore-8.3.0	Y	Y						
	mpi4py-3.0.1	Y	Y	Y		Y	Y	Y	Y
	TensorFlow-1.13.1-GCCcore-8.3.0		Y						
	Keras-2.2.4-GCCcore-8.3.0		Y						
Devel-2019a	Python-GCCcore-8.3.0		Y		Y				
	Pytorch-1.4.0-GCCCore-8.3.0		Y						
	Horovod-0.16.2		Y			Y			
Devel-2020	Python-GCCcore-8.3.0				Y				
	PyTorch-1.7.0				Y				
	scikit-GCCcoreMKL-9.3.0				Y				
	mpi4py-3.0.3				Y		Y	Y	Y
	TensorFlow-2.3.1-GCCcoreMKL-9.3.0				Y				

Table 3: Data Analytics and AI framework installed in the three Easybuild stages

As before, the complete software stack available on the DEEP-EST prototype can be evaluated with the `module spider` command. With the current installation workflow, adding a new framework can usually be carried out without too much effort; however, certain packages like Horovod have proven to cause a lot of extra work, partly because of the dependency to the binary interfaces of all supported MPI implementations and partly because of very numerous and complex inter dependencies with other packages.

Moreover, the highly optimised, extensively threaded mathematics routines for scientific applications in HPC (Intel MKL and NVIDIA cuBLAS) along with the latest Deep Neural Network libraries (NVIDIA cuDNN) are installed on the system for the corresponding hardware. The exact library versions are listed below:

- Intel MKL/2019.5.2.8.1
- cuBLAS/10.1.105
- cuDNN/7.5.1.10-CUDA-10.1.105

4.1.1 OpenCL Heterogeneous Programming Framework

As reported before, specific interest to use the FPGA accelerators of the Data Analytics Module was expressed by the WP1 partners Astron and CERN. Both did express a clear preference of using the standardised OpenCL programming framework for their work. As reported in Deliverable D1.5, partner Astron had ported part of their applications suite to the Stratix 10 FPGAs, and they were supported closely by Intel FPGA experts to devise a new OpenCL implementation which was adapted to the specific FPGA architecture.

In this process, the installed versions of the Intel FPGA SW stack were upgraded several times to make improvements to the OpenCL compiler, the FPGA backend and the general FPGA development framework available to all users. The officially installed version of these tools is now 2.0.1.

Initially planned work to support the inference of TensorFlow-produced models on the FPGAs by way of the Open Neural Network Exchange (ONNX) model format were not pursued, since the TensorFlow framework development switched to a new backend API and implementation with version 2.x which does not rely anymore on a static computation graph to capture a trained model and is therefore not compatible with ONNX.

4.1.2 oneAPI Programming Model

Intel publicly announced the new oneAPI programming model at the SC19 conference in Denver. oneAPI is a unified standards-based programming model [32], with the main objectives of simplifying application programming for heterogeneous platforms and achieving portability across multiple CPU and accelerator architectures. Functional portability is the first goal, yet the initiative aims higher and wants to achieve a degree of performance portability, too.

To this end, oneAPI has extended the SYCL[1] programming model into the Data Parallel C++ (DPC++) language, which is fully embedded into C++ V14 and later, and enables developers to code computational kernels suitable for execution on accelerators using C++ templates, to specify the required data transfers, and to manage offload to multiple accelerators. Care has been taken to enable an intelligent middleware to dynamically optimise DPC++ applications and to orchestrate kernel offloads, execution and data transfers in an efficient way. oneAPI is an open industry consortium, and implementations are becoming available for a variety of Intel and non-Intel platforms, including NVIDIA's GPGPUs.

The central place to learn about oneAPI is `https://www.oneapi.com`, with the specifications available at `https://spec.oneapi.com/versions/latest/index.html`.

Intel itself has developed a highly optimized implementation of DPC++, which targets Intel's line of CPUs, FPGAs and GPGPUs. This is contained in the latest releases of Intel's SW development tools, now refactored as "oneAPI toolkits" free of charge at `https://software.intel.com/content/www/us/en/develop/tools/oneapi/all-toolkits.html`. These comprise the full functionality of the compilers, performance analysis and optimisation tools, mathematical and communication libraries and debugger, plus domain-specific extensions.

Intel and JSC collaborated closely in evaluating the applicability of DPC++ and the use of the Intel toolkits for HPC codes; since early releases did require a different base OS installation from the one available on the DEEP-EST prototype, this work was first done on small systems outside of the DEEP-EST prototype, and then on a specially configured node of the ESB. Experiments included the use of the Intel compilers for a combination of an Intel CPU and its integrated GPGPU, and the use of a third-party DPC++ compilation system to target the Intel CPU plus NVIDIA Tesla V100 card on the one ESB node. Codes used included DPC++ examples, and more interestingly, CUDA codes which were translated to DPC++ via an Intel-supplied, third-party code compatibility tool. These experiments did establish the viability of DPC++ and the tools used regarding code functionality; performance results on Intel CPUs were (as expected) good, and the integrated graphics units targeted were

[1]A C++ single-source heterogeneous programming model for acceleration offload developed as part of the Khronos standardisation effort, see `https://www.khronos.org/sycl`

driven to the limits of their performance potential. Performance of codes on NVIDIA GPGPUs is still under analysis.

Testing the oneAPI functionality included the toolkit examples (such as DGEMM, SAXPY, and random number generators), writing & debugging Lamdba functions to be offloaded, handling of buffers in local and in unified shared memory, and using device selectors. On the NVIDIA V100 GPGPUs, matrix multiplication codes were tested, and work is continuing to evaluate calling of NVIDIA cuBLAS routines and using the new CUDA backend for MKL. An Intel engineer not funded by the project has worked on a oneAPI port of the GROMACS code, and JUELICH has contributed a simple n-body simulation example useful to test different schemes for handling buffers and leveraging unified shared memory.

Problems persist with installing the FPGA oneAPI tools, due to the fact that CentOS 7 is not supported at this time. Intel will continue to work with JUELICH to address this problem.

As reported in D1.5, partner CERN also performed experiments with oneAPI, porting a CUDA implementation of an electromagnetic calorimeter reconstruction code (which was produced earlier in the project) to oneAPI. Three steps were involved: first, the DPC++ code compatibility tool created a rough first DPC++ version; in a second step, the issues found by the tool during conversion were addressed manually, and in the final step, DPC++ compiler/linker errors were corrected. For the test code, the second step was very easy – it was, on the other hand, noted that with the current oneAPI tool versions, code which uses the NVIDIA tensor cores needs to be re-implemented. The third step involved more work, since the primitives used by the Eigen library had to be modified to fit in with the restrictions of kernel code in DPC++. This could be addressed in the future by libraries like Eigen becoming available in oneAPI-compatible forms.

Due to time constraints, performance could only be checked against a pure C++ code on CPU nodes—here, the rough oneAPI implementation did show very similar performance.

4.1.3 Persistent Memory Toolkit

The Data Analytics Module is composed of 16 nodes with 384 GB RAM plus 3 TB of Intel Optane Persistent Memory. Compared to DRAM, Intel Optane Persistent Memory has higher latency and lower bandwidth , yet offers much higher affordable capacities than DRAM and data persistence. It can be configured in two principal modes: Memory Mode and App Direct Mode.

In Memory Mode, no changes to the application are required: the installed DRAM acts as a memory cache and the Intel Optane Persistent Memory transparently offers its full memory capacity to the OS and to applications. However, memory contents is volatile. In DEEP-EST, the WP1 partner Astron has made use of this mode for applications running on the CPU and the FPGA of the DAM nodes. No specific changes or adaptations were required to the base OS of the DAM or other SW packages – Memory Mode is enabled via UEFI/BIOS settings, and requires a node reboot.

In App Direct Mode, DRAM and persistent memory are mapped onto separate memory address spaces (seen as memory nodes by Linux) , and applications have to be modified in order to exploit the different characteristics of the two memory technologies. Access to the persistent memory occurs through regular load and store operations. Intel has released the Open Source Persistent Memory Development Kit (PMDK, see [33]).

A special use case of App Direct mode is to map a file system onto a non-volatile memory partition; for this, the fs-dax layer provided by PMDK enables file system access while avoiding the need to go through a block device chain. For I/O-heavy applications, this usage mode can provide significant speed-ups, as for instance reported by the NextGenIO project. With WP6 partner ITWM, the BeeOND parallel file system was adapted to use the persistent memory as a storage target, enabling a job running on n DAM nodes to use a transient BeeGFS file system placed onto the $n \times 3$ TB of persistent memory at bandwidths significantly exceeding those achievable for the NVMe SSDs.

App direct mode and PMDK are in principle supported by the base OS of the DAM (CentOS 7), which runs the 3.x Linux kernel. Newer OS versions (such as CentOS 8 with kernel 4.x) provide significantly better performance, and experiments were run with a back-ported 4.19 kernel to establish whether the DAM nodes would be fully functional with a combination of CentOS 7 and such kernel. Since the results were positive, such a combination was used for BeeOND bechmarking.

5 I/O, file system, and storage

5.1 BeeGFS Storage Plugin Infrastructure

The prototype BeeGFS Storage Plugin Infrastructure allows BeeGFS to use a non-POSIX backend to store the user data into it, allowing new technologies like NVDIMMs, DAOS, CORTX or even Amazon S3 buckets to be used with BeeGFS. The concept of the BeeGFS Storage Plugin Infrastructure was already presented in D6.3, as were the two example storage backends: The heap plugin stores all data in system RAM and provides a lightweight alternative to tmpfs for volatile storage. While the pmem is a prototype for BeeGFS chunk data storage on non-volatile memory NVDIMMs using Intel's Persistent Memory Development Kit, allowing for fast storage which survives system crashes and restarts.

Having NVDIMM devices installed, the BeeGFS Storage prototype with the pmem prototype plugin was deployed on several DAM nodes and successfully tested. It is now also available for testing to the regular DEEP-EST user through the BeeOND facility (see next section). However, as with any developing technology, the NVDIMM support of the prototype pmem plugin still can be optimised further. Also recent changes in the upstream BeeGFS code lead to the needed for further adaptations of the plugin infrastructure to archive better performance. As there is also some interest from business costumers as well, an integration into the official BeeGFS repository for the next major release, and the needed steps therein are currently discussed.

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 can be integrate into any resource manager / job scheduler. This makes it ideal as a per job scratch file system or cache layer.

However, the widespread successful hacking of HPC Systems in 2020, lead the Jülich Supercomputer Center to change its security policy, which also effects the integration of the BeeOND framework: Normally BeeOND is run with root privileges on one of the participating nodes, starting the needed services on the other nodes using root ssh connections. The need for having elevated privileges stems from the fact that the actual mounting of the BeeOND filesystem needs root privileges. This behaviour is now forbidden, as is the usage of sudo for the same task, and a new approach was found to get BeeOND working: It was decided to use SLURMs prologue/epilogue facility instead, which can run a script on job start and end on each node with the needed privileges. This unfortunately meant a complete rewrite of BeeOND: instead of having one central process which coordinates the start-up and shutdown of the filesystem, we now have a collection of independent processes which have to coordinate their work with one another. For this, each agent determines the full configuration of the filesystem and starts/stops the services needed on the local node. Furthermore, using the included BeeGFS facilities, each of the agents oversees the startup/shutdown of the BeeOND filesystem as a whole and is therefore able to detect remote failures, as well. This allows each instance of BeeOND to bring down the filesystem in a consistent way in case of a failure in another node.

For this special version of BeeOND, a SLURM SPANK plugin was created which provides a BeeOND option to the cluster users and which pass its content to the BeeOND script on the individual nodes. Compared to the previous implementation to other job schedulers this allows the user to also pass options to the BeeOND script. It allows for finer control of the resulting temporal filesystem, instead of the user relying on that the default options fits her or his needs. Furthermore this makes integrating the BeeGFS Storage Plugin Infrastructure easy, as it is now an additional option to the BeeOND parameter. Users who want to test this feature therefore only need to change one line in their job description.

This new BeeOND version was successfully deployed earlier this year and is now in active use. Furthermore, this development has already gained interests among commercial BeeOND users, as similar security restrictions are imposed on other HPC clusters too. Also the better integration into SLURM is of interest by some commercial clients, too.

5.3 BeeGFS Monitoring

The target of the BeeGFS Monitoring server is to stream real-time statistical data of a BeeGFS filesystem into either a InfluxDB or a Apache Cassandra Database, be it for further analysis and/or graphical presentation using products such as Grafana. This monitoring service is part of the BeeGFS filesystem since version 7.0 and its inner working was already presented in D6.3. Since then the software has undergone only minor changes to work with DCDB, the monitoring framework implemented by BAdW-LRZ, as describe in D5.3: Instead of the original plan to directly write into the Apache Cassandra Database, it was decided to use the https interface of DCDB, which can mimic the InfluxDB protocol. This has the advantage that we now have one defined interface between the two products, saving both projects to adapt its product whenever the other changes its database structure. Furthermore as the InfluxDB protocol is already supported by the BeeGFS monitoring, only minor adaptations to it were needed. The greatest addition was the support for encrypted connections, a feature which will be shipped in the upcoming v.7.2.1 release of BeeGFS.

The BeeGFS Monitoring itself was installed on the DEPP-EST BeeGFS installation and feeds its data to the DEEP-EST DCDB instance. Their status can now be monitored using Grafana Panels like the ones shown in Figure 5.

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

5.4 SIONlib

Since the last report in D6.3, no optimisations were performed on the features added to SIONlib, because the storage hardware in the prototype system was deemed insufficient for the purpose of performing I/O performance measurements and the installation of an improved flash-based storage system is still outstanding.

The installation of SIONlib on the prototype system was already completed in D6.3 and since no new versions were released in the mean time, there is no installation activity to report on.

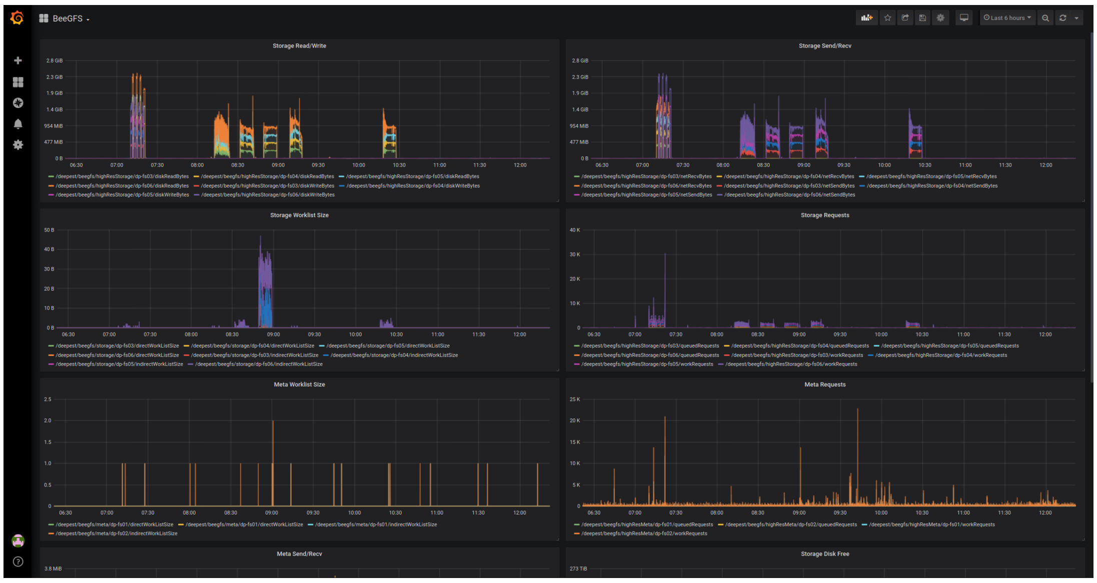

Figure 5: Grafana Panels showing different properties of the SSSM BeeGFS installation.

Recently, we identified a bug in SIONfwd which made the I/O forwarding server, `sionfwd-server`, abort when receiving a `pread` or `pwrite` message with I/O size zero. A fix for the bug is included in SIONfwd version 1.0.1.[1] This bug is not triggered by the stable version of SIONlib used in the DEEP-EST project, 1.7.6, and thus the new version of SIONfwd was not installed on the prototype system.

We received user feedback from the xPic developers early on in the project (July 2018), when a refactoring of C pre-processor logic in the public header files of SIONlib 1.7.2 caused a compilation error for xPic. The xPic developers were given instructions on how to correctly use the `sionconfig` utility program to help with compilation. Additionally, a new version of SIONlib, 1.7.3-rc.2, which restored the old pre-processor logic was prepared and installed on the DEEP-EST prototype system for the xPic developers.

[1] `https://gitlab.version.fz-juelich.de/SIONlib/SIONfwd/-/releases/v1.0.1`

6 Resiliency

In deliverables D6.2 and D6.3, we presented the advanced features of FTI and the OpenCHK checkpoint/restart model based on pragmas. This task, as originally planed, completed the work described in the project proposal, so there are no further development to report.

7 Summary

This document describes the programming environment developed to efficiently exploit the MSA architecture. The programming environment covers the most relevant parts of the software stack required to run on a supercomputer and it includes these 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)
- SIONlib and BeeGFS filesystem (Section 5) to exploit the storage subsystem
- FTI/SCR multi-level checkpoint/restart libraries (Section 6) to enhance the application resiliency to system faults

In the following paragraphs we summarize the main developments completed for each of these software components.

We have enhanced ParaStation MPI with modularity awareness at various levels of the software stack. In doing so, recent developments have focused in particular on the integration of libGCE / GCE as well as libNAM / NAM with their respective features in ParaStation MPI. Therefore, on the one hand, ParaStation MPI has been extended by a corresponding infrastructure to integrate the functions of libGCE and to support the GCE as an accelerator for MPI collectives also in MSA-aware sessions. On the other hand, for supporting the NAM, a corresponding wrapper layer has been implemented for ParaStation MPI towards libNAM and its related management API. This wrapper layer, called PSNAM, establishes the actual linkage between the MPI RMA interface and the access to persistent memory regions such as those of the NAM. In addituion, an integration with the resource manager has been implemented so that PSNAM can also handle pre-allocated NAM regions, e. g., those provided by the NAM burst buffer plugin as developed in Task 5.5 for Slurm.

The OmpSs-2 programming model has been enhanced to improve programmability and exploit specific hardware features of each module. We have extended OmpSs-2's runtime system with a new scalable scheduler and a wait-free dependency system to mitigate tasks management overheads. The runtime has also been extended to dynamically apply both dynamic voltage and frequency scaling (DVFS) and dynamic concurrency throttling (DCT) to exploit hardware resources efficiently, by saving energy while maintaining performance. We have extended the Task-Aware MPI (TAMPI library) to support one-sided operations inside tasks, which can be used to exchange data between MPI ranks but also to access the NAM. To address the new ESB design we extended OmpSs-2 with support for CUDA C kernels. Now, we have extended the runtime to also support optimized CUDA kernels provided by libraries such as cuBLAS, cuFFT or cuSolver. Moreover, OmpSs-2 has also been extended to support OpenACC as an alternative to generate kernels by annotating sequential code with pragmas. The runtime system has also been enhanced with a directory/cache to transparently manage data transfers between host and devices.

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, supported and regularly updated on the DEEP-EST prototype.

For the integration of BeeOND into the DEEP-EST cluster, we had to reprogram it, as the previous version was in conflict with the new security guidelines imposed by the Jülich Supercomputer Center. The opportunity was taken, to also create a solution that fully integrates BeeOND into the SLURM job manager. This now gives the users the possibility to fine tune their temporary parallel file system to their needs. With this new BeeOND version the usage of the BeeGFS PMEM storage plugin prototype is now also possible by only changing one line in the job description. Concerning the BeeGFS monitoring, smaller adaptations where needed, but it is now deployed and is feeding its statistic into DEEP-EST DCDB instance, where users can now inspect current and past performance of the DEEP-EST BeeGFS filesystem.

With all these new software developements, installed on the DEEP-EST prototype, a comprehensive, production ready programming environment is available for this and future MSA systems.

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
DAM-EXT	Part of the Data Analytics Module featuring Extoll connectivity
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] V. Beltran et al.: *DEEP-EST Deliverable 6.1: Design and specification of programming environment*, March 2018

[2] V. Beltran et al.: *DEEP-EST Deliverable 6.2: Prototype Programming Environment Implementation*, March 2019

[3] V. Beltran et al.: *DEEP-EST Deliverable 6.3: Complete Programming Environment Implementation*, December 2019

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

[5] 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

[6] 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`

[7] 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/`

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

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

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

[11] M. Maroñas, X. Teruel, J.M. Bull, E. Ayguadé, V. Beltran. "Evaluating Worksharing Tasks on Distributed Environments." CLUSTER 2020: 69-80

[12] K. Sala, A. Rico, V. Beltran. "Towards Data-Flow Parallelization for Adaptive Mesh Refinement Applications." CLUSTER 2020: 314-325

[13] D. Alvarez, K. Sala, M. Maroñas, A. Roca and V. Beltran. "Advanced Synchronization Techniques for Task-Based Runtime Systems." 26th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP'21)

[14] "Paper submitted to a conference with double-blind review process"

[15] R. Cano. Communication in Task-based Runtimes for Heterogeneous Systems. Master Thesis. `https://upcommons.upc.edu/handle/2117/334936`

[16] O. R. Korakitis. Towards supporting Composability of Directive-based Programming Models for Heterogeneous Computing. Master Thesis. `https://upcommons.upc.edu/handle/2117/335903`

[17] Kevin Sala, Jorge Bellon, Pau Farré, Xavier Teruel, Josep M. Perez, Antonio J. Pena, Daniel Holmes, VicenȘBeltran, 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.

[18] 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.

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

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

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

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

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

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

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

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

[27] 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

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

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

[30] 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

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

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

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

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

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