H2020-FETHPC-01-2016

DEEP-EST

DEEP Extreme Scale Technologies

Grant Agreement Number: 754304

D5.3

Prototype Software Implementation

Final

Version: 1.0
Author(s): N. Eicker (JUELICH), Th. Moschny (ParTec), C. Clauß (ParTec)
Contributor(s): J. Corbalan (BSC), A. Jokanovic (BSC), R. Fischer (BSC), N. Burkhardt (EXTOLL), M. Nüssle (EXTOLL), S. Krempel (ParTec), A. Netti (BAdW-LRZ), M. Ott (BAdW-LRZ), D. Tafani (BAdW-LRZ), Z. Ul Huda (JUELICH), J. De Amicis (JUELICH), I. A. Comprés Ureña (Intel)
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:	D5.3
	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 Software Implementation	
	ID: D5.3	
	Version: 1.0	**Status:** Final
	Available at: http://www.deep-projects.eu/	
	Software Tool: LaTeX	
	File(s): DEEP-EST D5.3 Prototype Implementation.pdf	
Authorship	**Written by:**	N. Eicker (JUELICH), Th. Moschny (ParTec), C. Clauß (ParTec)
	Contributors:	J. Corbalan (BSC), A. Jokanovic (BSC), R. Fischer (BSC), N. Burkhardt (EXTOLL), M. Nüssle (EXTOLL), S. Krempel (ParTec), A. Netti (BAdW-LRZ), M. Ott (BAdW-LRZ), D. Tafani (BAdW-LRZ), Z. Ul Huda (JUELICH), J. De Amicis (JUELICH), I. A. Comprés Ureña (Intel)
	Reviewed by:	H. Cornelius (Megware) E. Suarez (JUELICH)
	Approved by:	BoP/PMT

Document Status Sheet

Version	Date	Status	Comments
1.0	31.03.2019	Final version	

Document Keywords

Keywords:	DEEP-EST, HPC, Exascale, Software, specifications, job scheduling, resource management, network management, network bridging, system monitoring, Slurm

Table of Contents

List of Figures

List of Tables

Executive Summary

This deliverable provides a comprehensive overview of the current status of WP5 and summarises all implementation work performed so far. In doing so, prototype implementations of the related software components and products are presented and their integration status, functionality and capabilities are discussed. For instance, regarding interconnect management, an enhanced software stack for the EXTOLL fabri^3 hardware based on the EXTOLL Management Program (EMP) has been developed. With respect to network bridging, both the ParaStation MPI gateway framework as well as IP forwarding have by now been set up and first performance evaluations have been done on the Network Federation Gateway SDV. In the area of resource management, the handling of heterogeneous jobs by means of Slurm's job packs features has been realized for ParaStation Management together with the handling of gateway nodes as global resources. In addition, with regard to scheduling, better support for modularity and workflows has also been tackled, for example, by implementing a new delay switch and a new dependency type for Slurm. Finally, for a comprehensive and scalable system monitoring, a data visualisation module and data analytics engine have been developed for the Data Centre Data Base (DCDB).

1 Introduction

The operation of a Modular Supercomputer requires a system software stack with new features on top of the ones utilised on today's monolithic supercomputers. Deliverable D5.1 presented a detailed analysis of the corresponding requirements in order to set the stage for further work in WP5. Moreover, a rough design sketch of the software stack was presented. This sketch was updated to a design with more details in Deliverable D5.2 taking the progress on the design of the hardware modules made in the meantime into account. Due to the re-investigation of the design of the Extreme Scale Booster (ESB) module after the review at M12, D5.2 had to undergo yet another update recently, now reflecting the decision to have an GPGPU-based ESB.

Nevertheless, not all parameters of the hardware design were fixed at the time of writing D5.2. As an example, the actual implementation of the fabri^3 determines largely the design of the EXTOLL interconnect management program developed by Task 5.2. Therefore, concurrent to the hardware design of the fabri^3 the design of the management software was updated. The achieved results are presented in Chapter 2.

Beyond that, the re-design of the ESB changed the requirements on the software design. Due to the fact that the ESB will employ GPGPUs, the EXTOLL drivers have to support this type of devices. As an enhancement, Section 2.3 sketches the design and implementation of the according functionality.

For IP-bridging a first evaluation of the implementation – which fully relies on the Linux kernel's IP-forwarding functionality – is presented in Section 3.3. For this, the network federation gateway software development vehicle (NFGW-SDV) is utilised. The results achieved by these efforts will help WP4 to decide on the number of gateway nodes to be employed in the DEEP-EST prototype.

Section 5.2 describes the status of the Slurm extensions implemented to support heterogeneous workflows and a first rough performance model of this feature. The model covers the I/O requirements needed in order to switch from one step of the workflow to its successor and identifies the possible benefits of overlapping workflow steps in order to transfer data directly between them. This performance model will be used in the further course of this project in order to evaluate the new workflow model.

Further updates showcased in this document reflect the fact that in the meantime the implementation of the proposed software design has started. Therefore, for each software product to be produced by WP5 a brief status summary is provided.

This document presents all design decisions taken in WP5 since Deliverable D5.2 and summarises all implementation efforts performed so far giving a thorough overview of the current status. Similar to its predecessors, this document is organised following the structure of WP5 and its tasks. Each chapter contains several sections – one for each software product to be developed.

2 Interconnect Management

2.1 EXTOLL Management Program

The EXTOLL Management Program (EMP) is the software suite that manages the EXTOLL interconnection network. An introduction and description of EMP was outlined in D5.2 [14]. EMP does the network topology discovery, as well as the configuration and surveillance of a EXTOLL interconnection network. The software suite consists of two main components: master and slave daemons. This section provides a further detailed description of the master daemon and its improvements for the DEEP-EST project.

The EMP master daemon is written in the programming language Scala [1]. It combines object-oriented and functional programming into one concise programming language. The object-oriented part helps in building a clean software architecture, which can handle the complexity of a software suite configuring a modular interconnection network. The functional programming statements provide a natural interface for describing graph algorithms like network discovery, routing computation and network graph comparisons.

The master daemon is build from the following software components: First, there is the *EMP core*. EMP core provides the data structures and algorithms for network configuration. Second, there is the *EMP server*, which uses the data structures and algorithms from EMP core to configure the network. It also provides the user interface for interacting with EMP core.

The basic class diagram of EMP core is shown in Figure 1. EMP core uses a hardware abstraction layer to model the different control and status register file (CSR) layout of the different EXTOLL node types like Tourmalet, NAM or GCE. The node type is encoded in bits 23:20 of the Globally Unique Identifier (GUID) of each node. Tourmalet uses `0`, NAM `0xf` and GCE uses `0xe`.

The read and write access to the hardware is modelled by the device class. It provides an interface for reading and writing hardware registers in an EXTOLL node. There are different device implementations available. Firstly, there is the RMA-based device. It provides an in-band remote CSR access using the RMA functional unit. This access can be blocked in case of a fatal network error event. To enable a more reliable network recovery, a new device implementation was developed. This device uses the new side-band EXTOLL management capabilities of fabri[3]. It uses the USB and I^2C access described in D5.2, Chapter 2.

EMP server is based on the *play framework* [2]. The play framework enables web applications in Java and Scala. It provides the web interface as well as the RESTful API for EMP server. EMP server runs as the EMP master daemon on node in the EXTOLL interconnection network.

The RESTful API provides a complete set of endpoints to manage an EXTOLL interconnection network. The following endpoints are available:

GET emp/getTD	Retrieve the discovered network topology as topology definition file
POST emp/run	Start the network configuration
GET emp/status	Retrieve the status of emp/run

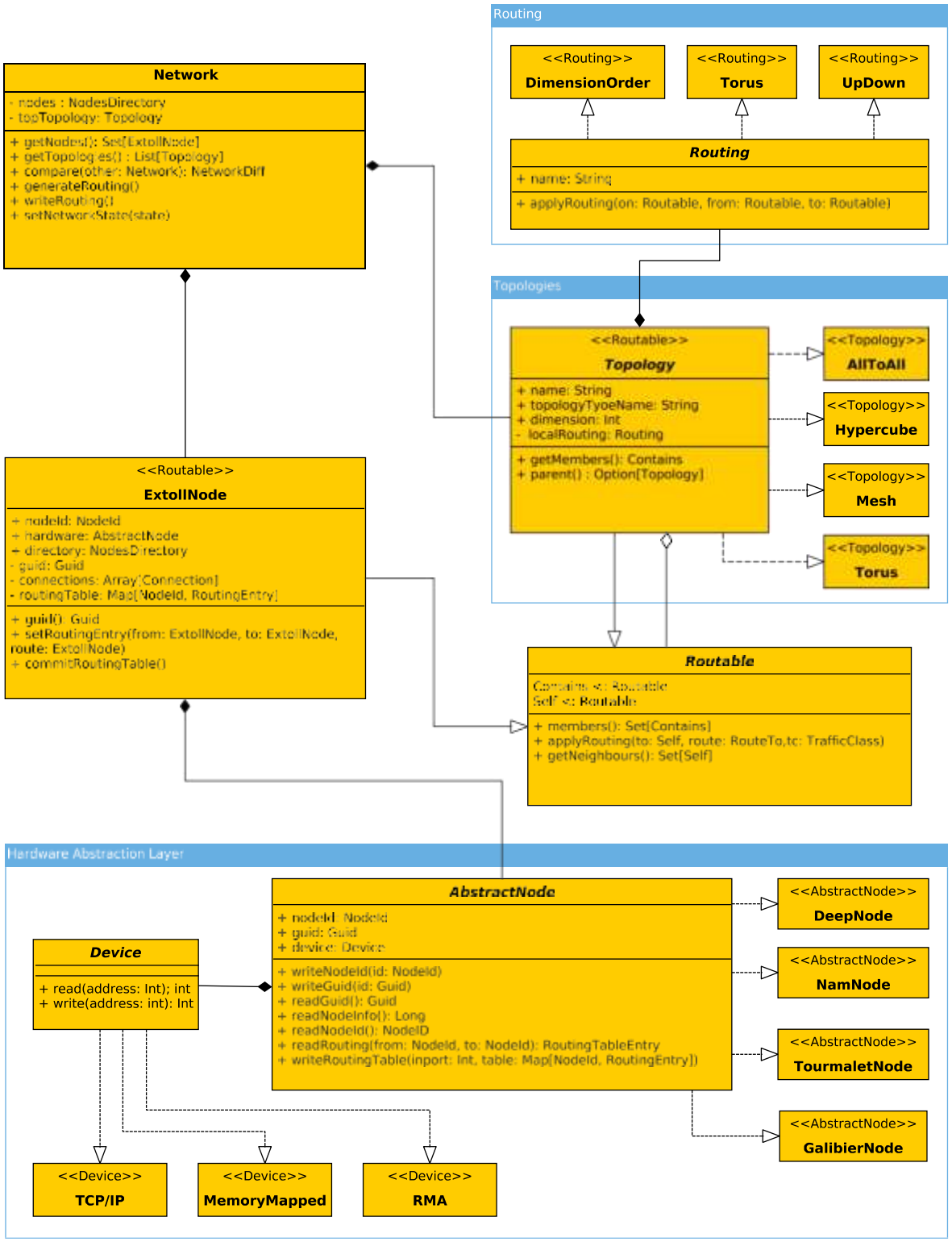

Figure 1: Class-diagram of important EMP classes.

POST emp/discover Start a network discovery

GET log Retrieve server log file

GET network/status Retrieve network status

POST network/stop Stop network

POST network/start Start network

GET network/info List network information

GET nodes List all available nodes

GET nodes/:id Retrieve information about node id

GET nodes/:id/link Retrieve link information for node id

GET nodes/:id/xbar Retrieve cross bar status information for node id

GET nodes/:id/pcie Retrieve PCIe link status information for node id

GET nodes/:id/velo Retrieve VELO functional unit status information for node id

GET nodes/:id/rma Retrieve RMA functional unit status information for node id

As mentioned before, EMP server uses EMP core for the network configuration. EMP server can either run in auto-configuration or in manual mode. In manual mode, which is the default mode, the server waits for administrator interaction. E.g. the administrator can trigger a network topology discovery, set up the routing tables in each network node or activate the network after a successful configuration. Therefore, the administrator can use either the web interface or a command line tool called `emp-ctrl`. `emp-ctrl` is a command line wrapper around the RESTful API, in order to provide a nice administration interface for the user, without dealing with RESTful API details.

In auto-configuration mode, the server tries to configure the EXTOLL network when the server gets started. If no network topology definition file is provided, the server does a network discovery and configures all nodes, which are available when the discovery was started. As boot times for the nodes in a cluster can differ from each other, it can happen that not all nodes in the cluster are available at discovery time, leading to an irregular network topology. Therefore, this configuration mode uses up/down routing as routing algorithm, as up/down routing is deadlock-free on any given network topology.

To select a different routing algorithm and to wait for all nodes in the cluster, a network topology definition file can be provided to the EMP server. This definition file describes which nodes should be available in the cluster, in which topology they are connected, and which routing algorithm should be used for the given topology. The server reads the topology definition file on start up, and creates an EMP core network graph data structure. Afterwards, a network discovery is started. The discovery also returns an EMP core network graph. These two network graphs get compared. If they don't match, the network discovery is repeated at a regular interval, until all nodes defined in the topology definition file are available. Thereafter, the routing tables for all nodes are generated, based on the routing algorithm specified in the topology file. This is followed by writing the routing for each node and the activation of the network, so that the EXTOLL Linux Ethernet emulation layer (EXN) or user-space software can

use the EXTOLL network. Before network activation, the EXTOLL driver forbids access to the EXTOLL network and devices.

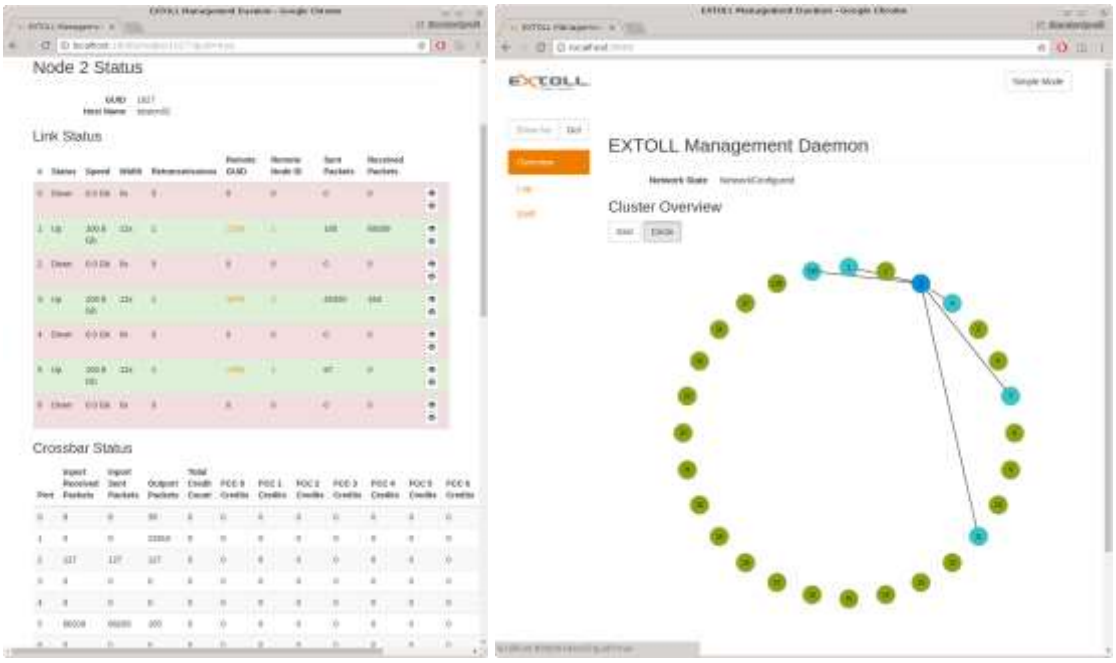

Figure 2: EMP Webinterface: status of a single node (left), connectivity of a single node in a larger network (right).

For administration purposes, two ways are implemented to interact with EMP. The first tool is a web frontend that the administrator can open in a web-browser window. Two screenshots of this interface are shown in Figure 2. Here, information about the overall network status, status of each node, and more can be displayed. It is also possible to trigger certain actions (re-discovery, new routing, . . .). The other tool for the administrator is the command line tool `emp-ctrl`. `emp-ctrl` uses the RESTful API of EMP and provides the administrator with all of the possibilities also present in the web-interface. Some important commands are:

emp-ctrl emp run	run emp on the current network
emp-ctrl emp status	show the current status of emp
emp-ctrl emp discover	let emp auto-discover the current network
emp-ctrl emp getdot	download the current network graph as . `dot` file suitable for processing with GraphViz
emp-ctrl network status	Show network status (configured, un-configured, up, down)
emp-ctrl network info	Show network information (topology, routing used, . . .)
emp-ctrl network start	Start the network (software can use the network)
emp-ctrl network stop	Stop the network (access to the network is denied)
emp-ctrl network listnodes	List all nodes present (nodeIDs)
emp-ctrl network missinglinks	List all missing links (in respect to the topology definition)

emp-ctrl network missingnodes List all missing nodes (in respect to the topology definition)

emp-ctrl node status Show complete node status

emp-ctrl node link Show detailed link status

emp-ctrl node pcie Show detailed PCIe status

emp-ctrl node coordinate Show coordinates

There are more commands, and most commands also accept further parameters. For example, the `node` commands accept a `--i` parameter to specify the node for which to run the command. On the command-line running `emp-ctrl help` will also display a comprehensive overview of all commands and parameters.

EMP also uses a number of configuration files which are placed in `/etc/extoll` by default. The main configuration file for the EMP daemon called `master.conf` holds lists key-value pairs to configure the daemon. There is also a `slave.conf` to configure the slave daemons. One important configuration value of the slave daemons is the domain name or IP address under which the master daemon can be found.

While not required, it is generally good to rely on topology files when working with EMP. A topology file basically describes a network topology (a graph) and also specifies thinks like the preferred routing to be used. It also lists the GUIDs of the network chips of the network (inventory). EMP discovery can then detect any anomaly in the network.

Figure 3 shows a `.td` file derived from the real DEEP-ER SDV. It was shortened, nodes taken out, to make it fit on single page. Nevertheless, all important parts can be seen. The first section is the inventory specifying the GUIDs of the Tourmalet devices and the node id these devices will have in the configured network. Note also the special parameters to denote NAM devices. Following the inventory are several sections describing the hierarchical topology of the SDV, with each (sub-)topology consisting of other sub-topologies or leaf nodes. Each topology has a type, a dimension, and it is specified which routing algorithm is to be used. The final section (links) specifies *how* the different nodes are connected by links (for example cables) in the physical world. When the file is read and processed by EMP, it will be checked syntactically, semantically and then applied to the actually discovered network. Each of these steps may yield errors that are reported in the log.

2.2 NAM/GCE Manager

The NAM and GCE in DEEP-EST already share the same board and therefore their base hardware configuration. They differ of course in their individual FPGA firmware implementation. Nevertheless, it becomes apparent that when looking at the board and the firmware implementations the important blocks regarding management of these components are the same:

- Direct loading of an FPGA image onto the FPGAs is performed the same way using an USB interface from the fabri[3] management controller to the FPGA using a FTDI chip on the FPGA board.
- Flashing the boot ROM of the FPGA board for automatic configuration after power-up uses the same infrastructure.

```
inventory
    0x649 1
    0x5e1 2
    0x63a 3
    0x5d6 4
    0x5d5 5
    0x63e 6
    0x627 7
    0x653 8
    0x5f5 17
    0x679 18
    0x615 19
    0xf00001 129 t=NAM1
inventoryend

topology top
type Mesh
dimension 1
routing ShortestPath
(0) NAM1
(1) SUB
topologyend

topology SUB
type AllToAll
dimension 1
routing ShortestPath
(0) SDV
(1) FS
topologyend

topology SDV
type Hypercube
dimension 3
routing DimensionOrder
    (0,0,0) 1
    (0,1,0) 2
    (1,0,0) 3
    (1,1,0) 4
    (0,0,1) 5
    (0,1,1) 6
    (1,0,1) 7
    (1,1,1) 8
topologyend

topology FS
type Mesh
dimension 1
routing DimensionOrder
(0) 17
(1) 18
(2) 19
topologyend

topology NAM1
type AllToAll
dimension 1
routing ShortestPath
 (0) 129
topologyend

links
1 0 <-> 0 2
1 1 <-> 1 3
1 2 <-> 2 5
1 4 <-> 0 129
2 1 <-> 1 4
2 2 <-> 2 6
3 0 <-> 0 4
3 2 <-> 2 7
3 4 <-> 0 129
4 2 <-> 2 8
5 0 <-> 0 6
5 1 <-> 1 7
6 1 <-> 1 8
7 0 <-> 0 8
17 2 <-> 0 18
18 2 <-> 0 19
17 1 <-> 5 6
18 1 <-> 5 7
19 1 <-> 5 8
linksend
```

Figure 3: Topology definition file for simplified DEEP-ER SDV including inventory.

- Accessing the status & control register file of the FPGA configurations uses the same serial bus interface from FTDI to FPGA. Furthermore, the protocol, and registerfile hardware resembles very much since they are based on a register file hardware created from an in-house register file generation tool using a domain-specific language.
- Configuring the EXTOLL network part of the two configurations is exactly the same.

Also, access to manage the NAM and GCE from processes within the system, be it administration level software, batch system, or MPI processes will use TCP/IP and the implementation inside the manager is done in Scala and the *play framework*, very much like EMP.

It was thus decided to have a single instance of "manager" being run on the management CPU of each fabri[3], which is responsible to provide the management facilities for all NAM and GCE boards present in the system. Again, internally there is a RESTful API, which is used to access the services of the manager. It is not necessary for other software packets to directly use this API, but access is encapsulated in functions of *libnam* and *libgce*. A web-interface for the administrator, again like EMP, is also provided. Within the end-points of the API, two major nodes are present, one for the NAMs and one for the GCE, which reflects the differences of the two. The following sub-sections give an overview of the NAM and GCE management tasks.

2.2.1 NAM specific management

NAM provides memory resources in the network, both fast volatile as well as a little slower non-volatile ones. Software can request allocations from this NAM memory pool. Later the allocation can be attached to a specific process, which can then access the memory very fast based on an RMA protocol. On top of this, RMA protocol upper-level protocols can be implemented.

The management of the NAM is most importantly responsible to manage allocations and attachments. An allocation request with a specific size and further meta-data reaches the manager via the RESTful API. An unused memory buffer is searched in the free table, potentially divided up into a now used part and a still unused part. The free segments as well as the allocations are stored in an SQL database on persistent storage. After each allocation or de-allocation, the manager iterates over the free segments and combines adjacent free segments to form larger segments. This implements a limited de-fragmentation of the base memory pool. An allocation returns a unique allocation identifier, which can be later used to request attachments. Of course, memory can also be freed again. An administration interface via the API exists to forcefully free memory, even if the unique identifier is no longer known (for example if an application that used it has crashed). Flags are supported when allocating memory to give a hint about whether the memory should be volatile or non-volatile. Also, allocation of a memory segment may fail, if memory resources are exhausted, i.e. no large enough contiguous segment of memory can be found to satisfy the size of the request.

Attach requests also reach the manager via the API. The attachment is marked in the database and in this case, an entry is stored in the AAT (Attached Allocation Table) in the NAM hardware via the USB device interface. After this has been performed, user software can access the memory using RMA. After a successful attachment, the necessary address and id values are returned to the user process, which `libNAM` then uses to create the correct RMA operations

for access. Note, that these addresses, IDs plus the node and VPID ID of the issuing process are used to protect the memory from access from other processes, that are not attached to this memory segment. It is possible to have several processes to attach to the same segment; synchronisation primitives are implemented by the NAM hardware and are available via `libNAM`. The management software only supports monitoring and (if needed) forceful freeing of resynchronisation primitives. The hardware AAT table is a limited resource, thus attaching to a segment may fail if all entries are already in use. Again, memory can of course also be detached again.

2.2.2 GCE specific management

The Global Collective Engine, based on the same hardware platform as the NAM, and also sharing the software platform for management, enables efficient offloading of global, collective operations, typically MPI collective operations like `MPI_Reduce()`. While all of the actual collective operations are performed by the actual user-space processes comprising the parallel job (i.e. the MPI processes) accessing the GCE using direct, fast-path RMA transaction over the EXTOLL network for highest performance, there is a number of management operations that are performed through an out-of-band interface, that shares all of its protocols with the above described NAM solution.

In the case of the GCE, the main job of GCE management is to create and release groups of processes that should use the GCE. These groups are commonly called *communicators* in the MPI world. Thus, whenever an MPI communicator is created by an MPI job, and this communicator should be able to use GCE services, the communicator has to be created also on the GCE. To this effect, the root process of the new communicator has to deliver all of the communicator information to the GCE. The GCE management software then writes the necessary configuration tables in GCE memory. In the same way, release of a communicator is handled.

The GCE management software also checks the resource limits of the hardware implementation of the GCE. All of the communicator configuration as well as the operation status and data scratch buffers for all active communicators and active communication operations have to fit in the DRAM memory of the GCE. The current version of the GCE chose to use 4 to 16 GB of memory, thus there is an upper limit for the number of communicators, as well as for the active operations per communicator. There are some more limits, for example maximum size of a collective operation, maximum communicator size, etc. The limits of the hardware prototype implementation in DEEP-EST are shown in Table 1.

Figure 4 shows the DRAM layout of the GCE. The Comm data structure (in blue) is directly managed from the GCE management. The other memory structures are used by GCE itself when processing operations for a configured communicator.

To actually use the GCE, a new user-level library is introduced, `libGCE`. `libGCE` basically offers functions to create and release communicators (which are slow-path functions and interact with the GCE management process) and collective operations (like `gce_reduce()`) which resemble the widely known MPI collective operations. An MPI implementation that wants to leverage GCE then needs to link against `libGCE`, call the communicator create and release functions in the respective functions in the MPI library, and call the `libGCE` collective operation functions instead

Description	Value	Comment
Max. collective operation data size	8 MB	Number of vector elements (count) varies with data type size
Max. number of communicators in the system	256	Can be spread over multiple MPI jobs. Corresponds to a Comm ID length of 8 bits
Max. number of collective operations per communicator	8	Corresponds to an ICID length of 3 bits
Max. number of processes per communicator	1024	Corresponds to a rank length of 10 bits
Memory address length	64 bit	
Arithmetic-logical operation encoding length	4 bit	16 different operations possible
Data type and size encoding length	4+2 bit	64 different data types possible

Table 1: GCE hardware parameters and limits.

of the legacy software implementation of the collective operation. Of course, the software implementation shall be retained, and has to be used in case that the GCE cannot perform the operation (for example: out of resources).

2.3 GPGPU support implementation for the EXTOLL driver

In Section 2.4 of D5.2 [14], the support for GPUDirect for RDMA on the EXTOLL network stack was introduced. To actually implement this, a kernel module will be added to the set of EXTOLL kernel modules already present. The new module handles the case of registering CUDA (i.e. GPU) memory buffers, while the existing modules are used to handle registration of CPU buffers.

The `extollgpudirect` module has to be linked against the actual CUDA kernel driver used in the system. It can then call the functions of the NVIDIA kernel API. Basically `nvidia_p2p_get_pages()` has to be called instead of the Linux kernel standard `get_user_pages()`. The same holds true for de-registration which is done by calling `nvidia_p2p_put_pages()` instead of `put_pages()`. Of course, there is actually some more code needed, for example GPU buffers have an alignment of 64 kB, while CPU memory registration is typically performed on a 4 kB granularity. A good introduction to this topic is given in [3]. To complete support of registration of GPU buffers, also changes in the user level software are necessary. Here, the changes for RMA can be entirely hidden in `librma`. A compile-time option can turn on/off CUDA support. If CUDA support is needed, upon a registration request it is first determined if the buffer is a GPU buffer. If so, the CUDA runtime has to be informed about this (using `cuPointerSetAttribute()`) and then the new kernel registration function for GPU buffers is invoked. If the memory is a CPU buffer, the code path remains unchanged. De-registering works in an analogous way.

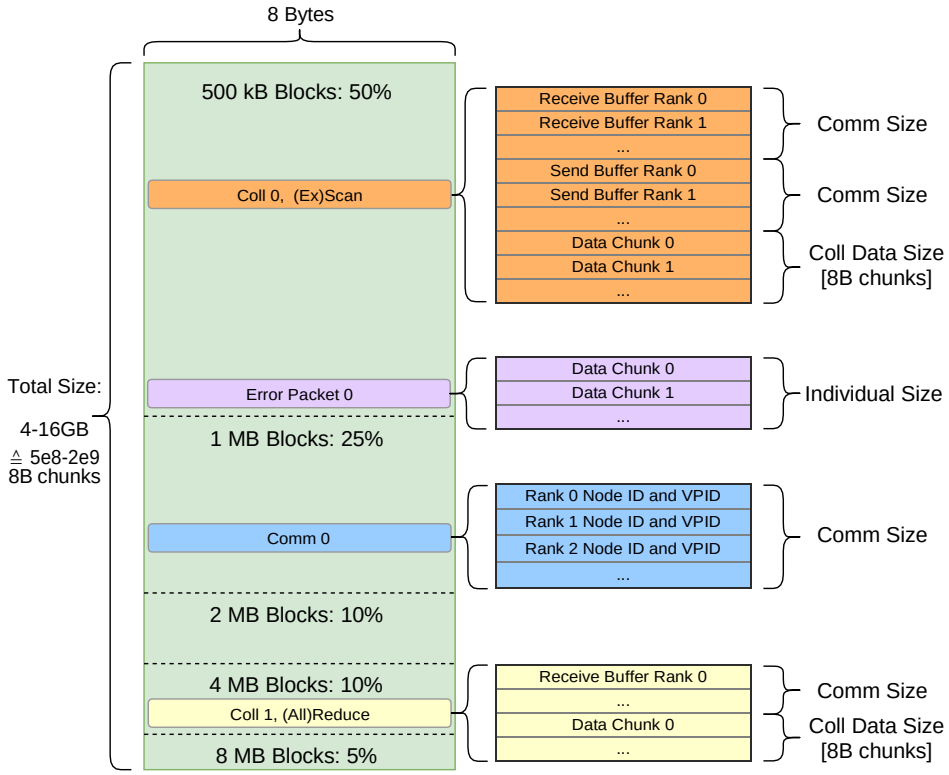

Figure 4: Memory map of GCE.

Note that the whole process is also transparent and amenable for registration caches, as already implemented and used by user-space components in the DEEP-EST software stack (MPI, librma).

The same handling of GPU buffers that is added to `librma`, can also be used to enable direct CPU access to a GPU buffer, for example by using `memcpy()`. This could be used to implement CPU copy implementations of `MPI_Send()/MPI_Recv()` for small messages, where setting up a rendez-vous RMA communication is too costly. It will be investigated, whether the overhead incurred by registering and handling the GPU buffers in this way, is actually offset by the potential lower latency gained by circumventing costly CUDA `memcopy()` operations. This is a topic of research that will be investigated once real hardware with V100s is available in the project.

3 Inter-Module Network Bridging

This section describes the implemented prototype of the MPI bridging framework. In doing so, it summarises the aspects regarding its software design that have been discussed in Deliverables D5.1 [13] and D5.2 [14] before. The actual implementation eventually follows the specifications made previously. However, regarding the NAM bridging, no implementation work has yet been possible due to the current lack of appropriate NAM hardware and/or suitable low-level software.

3.1 Architectural Overview

As already described in detail in D5.2, the basis for the MPI-related network bridging is the ability of the pscom library to control different network technologies simultaneously by using multiple of its pscom plugins. In this way, nodes equipped with two or more network interfaces can act as *gateways* between different domains of interconnect technologies and thus between different modules of the MSA.

The task of these pscom gateways is then to receive MPI messages from one network domain and to re-inject them into the other domain. Since Task 5.3 is primarily about forwarding MPI traffic, it will only be necessary to bridge between the InfiniBand Cluster Module (CM) and the EXTOLL Extreme Scale Booster (ESB) via the pscom gateways as shown in Figure 5.

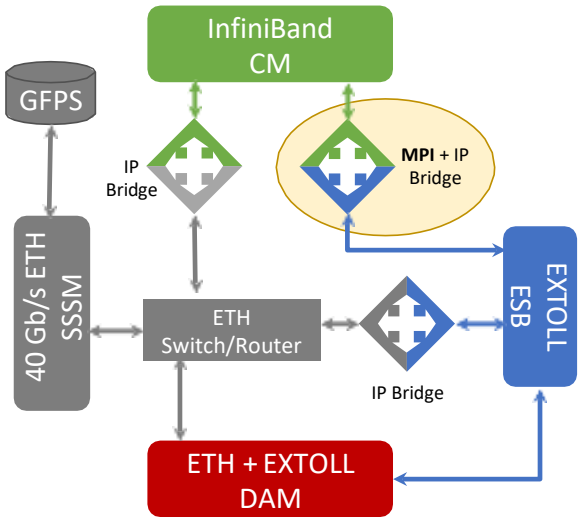

Figure 5: Abstract topology overview of the DEEP-EST system.

To avoid bottlenecks with respect to the transmission bandwidth of the CM-to-ESB communication, the gateway interface might comprise *multiple* gateway nodes. Therefore, the MPI bridging framework must be likewise able to handle multiple gateway nodes from a software point of view in such a way that load balancing can be achieved between them. For this purpose, a routing scheme is applied mapping each node within one network domain to a certain gateway node in such a manner that the communication load is spread across the interface.

However, since the number of gateway nodes is much smaller than the number of compute nodes, this resource has to be divided among all the MPI processes that want to communicate across the interface—and this applies to both the processes *within* a job and to the mapping of processes from *different* jobs. In doing so, the mapping scheme realised so far assumes (analogous to the allocation of compute nodes) a *job-exclusive* assignment. This means that a fixed number of gateway nodes is always reserved for each distributed compute job (or rather for each *Job Pack* (cf. Sec. 4), so that usually only MPI processes within a job or session have to share these nodes, i. e., this results in an implicit multiplexing of messages routed through the gateways.

3.2 Implementation Details

In accordance with the discussion in D5.2, the implementation of the gateway interface has been realised in terms of daemon processes (the so-called *psgwd* daemons). There is a one- to-one mapping between active gateway nodes and psgwd daemons and serve as forwarding instances transparently connecting the MSA modules.

3.2.1 Connection Setup

Two MPI processes of a single job located in different MSA modules with different network technologies will use a dedicated gateway plugin (the so-called *pscom4gateway*) from the pscom communication library for their interaction. This gateway plugin, which has been developed exactly for this task, now serves as the mediating instance between the two modules on the MPI level.

For choosing the correct plugin regarding the communication within a module and between modules, respectively, the gateway mechanism relies on the pscom-intrinsic priority / fallback scheme for the transport selection. In doing so, this scheme favours those interconnects as transports that promise faster communication. The slowest transport is here considered to be the TCP/IP protocol being also used for the general connection setup via a kind of *pseudo* plugin, which is firmly integrated into the library and always available.

Obviously, for not falling back to plain TCP/IP communication also for payload, the gateway plugin requires a higher priority than this pseudo plugin, but at the same time a lower priority than the domain-internal plugins. Table 2 shows an excerpt from this priority scheme for the pscom plugins.

However, as already denoted above, the connection establishment is still to be conducted via TCP/IP as an auxiliary transport for negotiating the actual payload connection. In case of the gateway plugin, this naturally means that the processes have to connect to the gateway daemons instead of directly contacting the target processes within the other module. This detour is realised by means of a *routing file*, which is generated by a script and that contains the mapping scheme and the addresses the gateway daemons are listening on. (See Section 4.3 for an example.)

Transport	Priority
TCP	2
SHM	90
OPENIB	20
EXTOLL	30
PSM	30
GATEWAY	10

Table 2: Priority/fallback scheme for the transport selection of pscom.

3.2.2 Payload Transport

At this point it shall again be emphasised that the subsequent payload connections between the MPI ranks and the gateway daemons are not based on TCP/IP but rather on the fast module-internal interconnects such as InfiniBand and EXTOLL. (See Figure 6, as it has also been shown and detailed in Deliverable D5.2.) For example, messages to be sent from the CM to the ESB are first forwarded via InfiniBand to the respective gateway daemon, which stores and instantly forwards them via EXTOLL to the actual receiver.

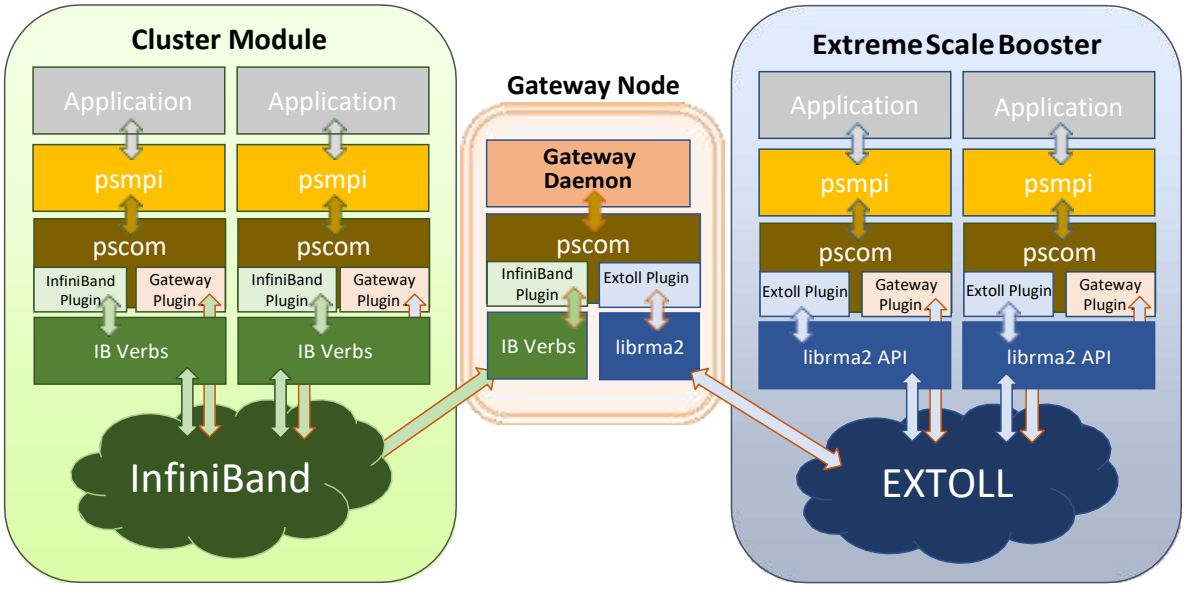

Figure 6: Layer model of pscom with plugins and gateway daemons.

For this forwarding of inter-module messages, their headers need to be extended by additional information such as the MPI rank and the network address of the final destination. The gateway daemon evaluates this meta-data to decide which connection has to be used for sending the message to the final recipient within the other module. This information is contained within the so-called *gateway envelope*. Conversely, the gateway plugin of an MPI processes accepts the messages sent by the daemon via the local interconnect and eventually matches them to the posted receive requests.

This scheme is implemented as a *store-and-forward* mechanism where the messages are copied in terms of chunks into intermediate buffers. Although this temporary buffering implies an overhead that could be avoided at a first glance, these intermediate copies are very well necessary–at least without further measures. The reason for this is that one MPI-process-to-daemon connection may carry payload for different receivers so that without buffering deadlocks may occur.

According to this, the communication path from a sender in one module to a receiver within another module is as follows for RDMA/RMA-style interconnect technologies like InfiniBand and EXTOLL: The message within an application buffer is copied chunk-wise into registered ring-buffers for outgoing messages by the respective plugin. The interconnect then transports these chunks via RDMA into likewise registered receive buffers at the daemon. The daemon then reassembles the chunks coming from this plugin in terms of *message fragments* and evaluates the gateway envelope. After storing the data within the intermediate buffers, the daemon re-injects the message into the respective pscom connection for forwarding it via the second plugin, which internally may also makes use of ring-buffers.

At first glance, the multiple copy operations may seem to have a negative impact on performance, but note that they can be interleaved. So for instance, the fragmentation size, which is the maximum size for the above mentioned message fragments, can be adjusted (by setting `PSP_GW_MTU`) as a tuning parameter for gaining an improved throughput (cf. Fig. 7).

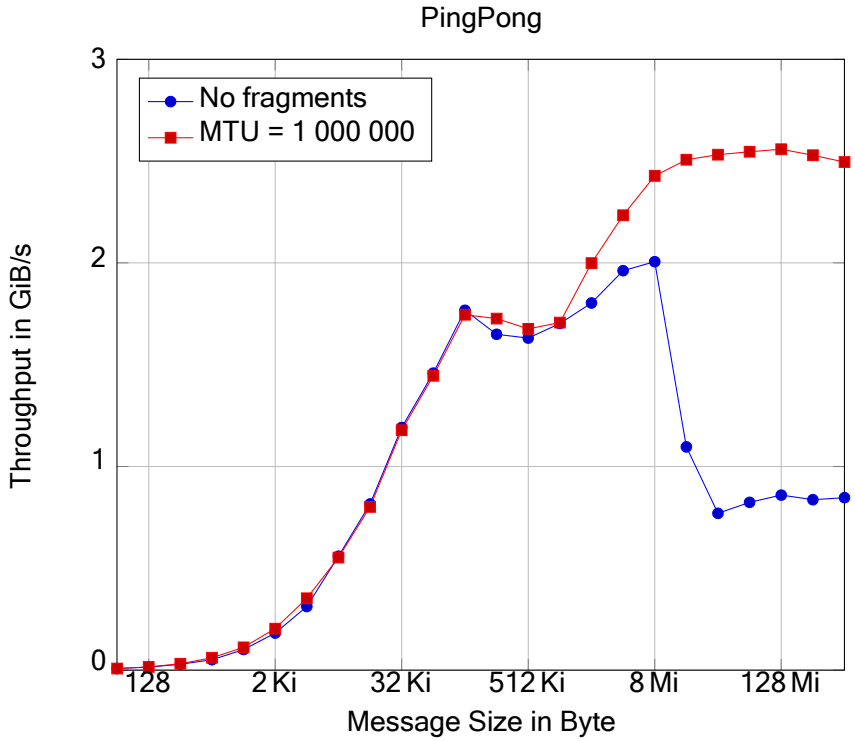

Figure 7: Impact of the fragmentation size on the throughput (as measured on the JURECA Cluster-Booster system at Jülich).

3.3 IP bridging

Beyond bridging of MPI traffic further traffic classes have to be forwarded between the different fabrics. As sketched in Figure 5 this most importantly includes I/O because the SSSM is only connected via 40 Gb/s Ethernet while other modules like the CM or the ESB do not feature this type of interconnect technology. Since BeeGFS utilises IP as the transport layer, IP bridges are required to forward the corresponding traffic.

Setting up IP bridges is straightforward: they might be realised by plugging network interface cards of the corresponding technologies into a single gateway node. The actual transport utilises the IP forwarding capabilities of the Linux kernel. Therefore, it is sufficient to setup the corresponding routes pointing to the gateway nodes in the endpoints on both sides and to enable IP forwarding on the gateway nodes itself.

While setting up the IP gateway functionality can be done by just pulling the right triggers in the Linux kernel, the sizing of the bridge remains an open question. In order to identify the necessary number of gateway nodes, on the one hand the bandwidth requirements for the specific traffic class has to be identified and on the other hand the achievable bandwidth through a single gateway node has to be determined.

This section reports on the analysis of the IP bridging between the CM's InfiniBand and the SSSM's 40 Gb/s Ethernet networks. All measurements were conducted on the three nodes of the NFGW-SDV. The node equipped with a Mellanox ConnectX-5 InfiniBand HCA and a Mellanox ConnectX-5 Ethernet NIC acted as the gateway node. A second node hosting a ConnectX-5 InfiniBand HCA (and an EXTOLL NIC) was utilised as the InfiniBand endpoint for the measurements. In the same manner the third node being equipped with a ConnectX-5 Ethernet NIC (and an EXTOLL NIC, too) acted as the Ethernet endpoint. All pairs of cards of the same types are directly connected without any switch in between. EXTOLL NICs were not used during the measurements.

Due to the direct connection of the ConnectX-5 Ethernet NICs they would auto-negotiate a 100 Gb/s Ethernet connection. Since the DEEP-EST prototype will utilise a 40 Gb/s switch, for the measurements on the NFGW-SDV the capabilities of the NICs were artificially reduced to the corresponding actual speed (40 Gb/s) using the `ethtool` utility.

In a first step the capabilities of the two fabrics in use were identified ignoring the gateway itself. For this, measurements using the `iperf3` tool were conducted. The results turned out to be highly dependent of the CPU cores being used on both ends of the connections. Therefore, a thorough analysis of bandwidth results depending on the utilised cores on `iperf3`'s client- and server-side was made. Two examples for the resulting heat-maps are presented in Figure 8. The experiments were conducted with two clients streams running in parallel utilising `iperf3`'s `-P` option. The threads where pinned by the `-A` of the `iperf3` benchmark. The numbers on the abscissa denote the client's local core number, the ones on the ordinate indicate the remote core number, i.e. the ones of the server. The results clearly indicate that the measured bandwidth is significantly better when utilising cores 4–7 on the server side, i.e. cores located in the same processor-socket providing the PCIe link the network interface cards are attached to[1].

[1] Cores 12–15 represent the second SMT-thread of the same physical core.

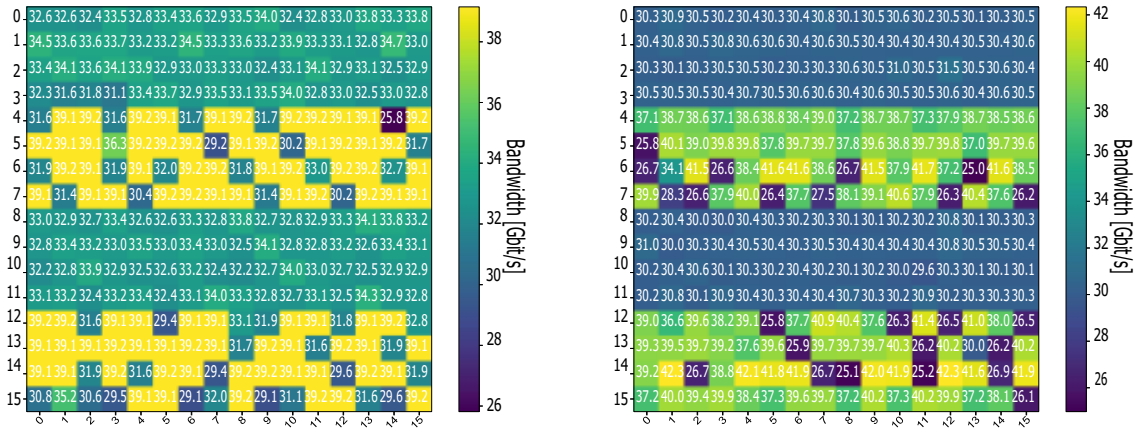

Figure 8: Bandwidth heat-map for different cores on `iperf3`'s client and server side. Results for 40 Gb/s Ethernet (left) and EDR InfiniBand (right).

In order to prevent `iperf3` processes from fluctuating between cores and, thus, suffering from lost cache content, further experiments were conducted with processes pinned via `numactl`. For this test up to four pairs of `iperf3` processes running in parallel were started. Each measurement was run for 40 seconds omitting the first 10 seconds in order to exclude startup effects of the TCP connections. Four classes of tests are presented in Figure 9: EDR InfiniBand only (marked as IB in the diagram), 40 Gb/s Ethernet only (40GbE), gateway traffic with EDR InfiniBand on the server side (40GbE → IB) and gateway traffic with EDR InfiniBand on the client side (IB–40GbE). Each experiment was conducted 5 times and averaged, the error-bars of the results indicated the standard deviation and, thus, the fluctuations of the results.

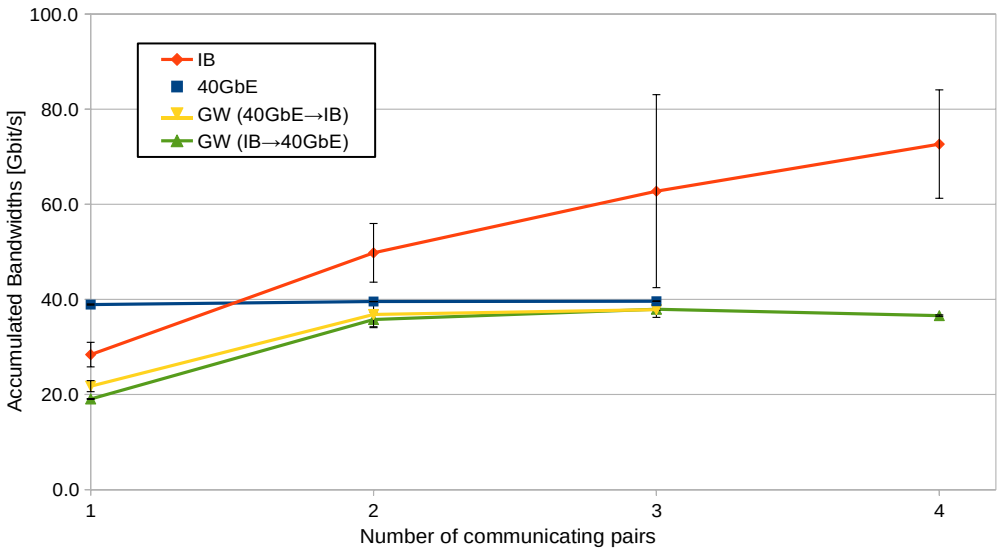

Figure 9: Accumulated bandwidth for different number of pairs of communicating `iperf3` processes using the IP over InfiniBand (IB), IP over 40 Gb/s Ethernet (40GbE) or IP forwarding protocol (40GbE→IB and IB→40GbE).

It is clearly visible that results for traffic on the InfiniBand connections are less stable. This might be due to the fact that these connections were setup utilising the default kernel drivers coming with CentOS 7.5 and no specific parameters for these kernel modules. Further tuning of the InfiniBand setup might stabilise the results and even improve the available bandwidth. This is work in progress and will become important when bringing the gateways for the Cluster Module into operation.

While for the Ethernet results it is evident that already a single pair of processes is capable to fill the link, for EDR InfiniBand even four pairs are not sufficient the fully saturate the higher capacity of the 100 Gb/s link. Nevertheless, already two pairs create 50 Gb/s bandwidth which is sufficient to fill the Ethernet link when being forwarded.

For the IP-forwarding results no clear dependence of the measured bandwidth on the direction of forwarding can be deduced. Furthermore, two pairs of traffic streams are sufficient to fill the bandwidth capacity of the gateway node which is due to the Ethernet link limited to 40 Gb/s. Furthermore, Figure 9 clearly indicates that the capacity of a single traffic stream forwarded by the gateway node is limited to 20 Gb/s.

4 Resource Management

4.1 Summary of the Requirements

As briefly described in D5.1 [13] and revisited in D5.2 [14] most applications will go more and more into using different types of accelerators for different tasks inside their workflow and thus become more and more heterogeneous. The Resource Management System needs to be able to manage these kinds of resources (including hardware accelerators, memory class and capacity, and storage system) and provide them to the jobs.

In the DEEP-EST prototype architecture, the ESB-nodes and the DAM-nodes are connected via gateway nodes to the CM-nodes (cf. Sect. 3). These gateway nodes manage the data transfer between Cluster and Booster nodes and the Resource Management System has to manage them as a kind of global resource.

In addition, further global resources will be embodied by the Network Attached Memory as well as the Global Collective Engine and similar management functionalities will be required. However, since NAM and GCE are not yet available, the remainder of this chapter focusses on the implementation work done so far with respect to the handling of gateway nodes.

4.2 Resource Allocator

Starting from version 17.11, Slurm supports the ability to submit heterogeneous jobs. From a user's perspective, this is implemented by using a colon notation for `salloc`, `sbatch`, and `srun`. With each of the commands a user can request multiple sets of resources at once, separated by a colon, forming all together one *pack* job allocation. At the same time (in case of `srun` and `sbatch`) or later on (in case of `salloc`) different executables can be started forming multiple jobs inside the pack job allocation. As part of this project, the ParaStation Management Daemon's plugin `psslurm` has been extended to provide full support for this Slurm feature.

For the allocation of dynamically determined resources such as the gateway nodes or starting additional daemons as the required gateway daemon on these nodes, Slurm currently does not provide any support. So, extensions of such functionality are needed. As a proof of concept, we made these extensions to the ParaStation Management Daemon by implementing the additional plugin `psgw` using several existing capabilities. In the long run it would be desirable to bring at least parts of this mechanism into Slurm itself.

4.2.1 Usage

Whenever a user wants to start a heterogeneous job using Cluster nodes and Booster nodes together and processes on both node types shall communicate with each other, an additional mechanism is needed to specify that gateway nodes are needed to connect the cluster with the booster nodes (cf. Sect. 3.2.1). To support this, we implemented a new Slurm SPANK plugin, adding some options to the frontend commands `salloc`, `sbatch`, and `srun`. The SPANK plugin

has little functionality itself, just setting a couple of environment variables, leaving the smart part to the ParaStation Management Daemon.

This plugin provides several options to the user:

--gw_file=path Path to the gateway routing file

--gw_plugin=string Name of the route plugin

--gw_num=number Number of gateway nodes

--gw_env=string Additional gateway environment variables

--gw_cleanup Automatically cleanup the route file

--gw_binary=path Debug psgwd

The only mandatory option is `--gw_num` to request gateway nodes for a pack job.

To start a simple interactive pack job using two gateway nodes the command may look like

```
srun --gw_num=2 -N 1 -C cluster ./hello_cluster : -N 2 -C booster ./hello_booster
```

This leads to the allocation and the setup of the required gateway nodes. A routing file is generated and stored in the users home directory or at the location specified by the `--gw_file` option. This routing file then is used by the MPI communication layer to set up the connections to the gateway nodes and thus between processes running on the cluster and booster nodes. The option `--gw_plugin` influences the generation of the routing file by choosing the used routing plugin. Using routing plugins, site administrators can provide different routing strategies satisfying the needs of different communication patterns. The routing file is automatically removed when the allocation is revoked and the option `--gw_cleanup` was given. The remaining options `--gw_env` and `--gw_binary` are for debugging purposes.

4.2.2 Background

Slurm currently does not provide a mechanism to handle allocations of global resources as needed to manage the gateway nodes and it turned out that adding such support into Slurm is not feasible in this project. So we decided to work around this limitation by managing the gateway nodes inside the ParaStation Management Daemon, which provides a more general resource management capability. This approach slightly softens the border between Resource Allocator and Process Manager, since it is necessary to late-allocate the gateway nodes as additional resources after the job has been scheduled in the prologue phase and eventually requeue it if not enough gateway nodes can be allocated at that moment (see the next section for details about the allocation of gateway nodes).

4.3 Process Manager

Once the scheduler decides to run a heterogeneous job as described in the previous section, the process manager needs to setup the infrastructure by starting the required gateway dae-

mons. Subsequently it starts the processes on both, the compute nodes and the booster nodes and provides them the information needed to communicate to each other.

4.3.1 Technical Overview

In the prologue phase of the pack job the `psgw` plugin of the mother superior `psid` requests gateway nodes from the master `psid`. If the request is successful, a prologue is executed on the gateway nodes and a ParaStation Gateway Daemon called `psgwd` is started for the pack job on each allocated gateway node. The addresses and ports on which the gateway daemons are listening for requests are forwarded to the route script. This script generates the routing file, considering the options given by the user (see “Usage” in section 4.2).

The resulting routing file may look like this:

```
192.168.12.77:40158 cluster015 booster003
192.168.12.77:40158 cluster016 booster003
192.168.12.78:40889 cluster015 booster010
192.168.12.78:40889 cluster016 booster010
```

The routing file above shows that the Cluster nodes cluster[015-016] use the gateway with address 192.168.12.77 and port 40158 to talk to the booster003 node. For the Booster node booster010 the gateway with address 192.168.12.78 and port 40889 is used.

4.3.2 Technical Details

Compute Nodes

At the compute nodes, beside the plugin `pelogue` to manage a pack job global prologue phase, the new plugin `psgw` for the ParaStation Management Daemon `psid` is required. It only takes action if a node becomes the mother superior of the pack job. If so, during the prologue phase it allocates the required number of gateway nodes, triggers the start of a prologue script at the allocated gateway nodes and then the start of the ParaStation Gateway Daemons on that nodes. Finally, it takes care of cleaning up the routing file if requested by the user. If something went wrong, it additionally starts the error script `psgw error` (see “Error Handling” below).

Gateway Nodes

On the gateway nodes, the psid plugins `pelogue` and `psexec` are used and thus have to be loaded. `pelogue` is needed to execute a separate prologue on the gateway nodes which are not part of the pack job from the Slurm and psslurm perspective since they are not allocated by the scheduler yet, but managed by the master ParaStation Management Daemon. The prologue is meant for example to check the nodes to be in a healthy state. `psexec` is used by the `psgw` plugin on the pack job mother superior to actually start the required instances of the ParaStation Gateway Daemon `psgwd`.

Error Handling

If not enough gateway resources are available or other fatal errors occur during startup, a script called `psgw_error` is called on the Slurm head node. This is triggered by `psgw` on the pack job mother superior using `psexec`. The current purpose of the `psgw_error` script is to re-queue batch jobs and set the eligible time. This is necessary since the Slurm scheduler is not aware of the gateway nodes and so a simple re-queueing could restart the job immediately again. The solution to set an eligible time (currently at 10 minutes) is meant provisional and subject to be replaced by a better mechanism.

5 Job Scheduler

5.1 Efficient Job scheduling for modular architectures

As we reported in D5.1 [13], there are two main scenarios that the scheduler should be enabled for: 1) the application can be executed in different modules and the scheduler should choose the module that gives the lowest slowdown, and 2) the application is executed simultaneously in multiple modules. Here we will describe the work that has been done in order to enable efficient scheduling considering the first scenario.

5.1.1 Investigating the use of Slurm's partitions mechanism for definition of modules

First, we investigated the possibility of using the Slurm's partition mechanism for the definition of modules. Currently, the concept of partitions is used in Slurm to separate resources in different subsets, in order to specify different limitations in each of them in terms of maximum wall clock time, maximum number of nodes requested, etc. These resources are supposed to be homogeneous among partitions, since users are submitting for the same job the same set of requirements to all the partitions specified. It is possible that a user specifies the list of partitions on which its job can be executed [29]. Since the resources among partitions are homogeneous, the requested amount of resources and wall clock time will be the same across different partitions. As long as the requested resources and time are within the per-partition limits, Slurm can choose any of the partitions in the user specified list from which to allocate resources for user's job. Slurm creates one register job queue rec per partition, and once a job is allocated resources, other records for that job in the queue will be ignored.

We conducted the following experiments on Slurm's simulator. We wanted to evaluate the benefit of having a second partition for smaller partitions. Since small partitions saturate quicker, allowing jobs to run in secondary partitions can translate into a dramatic reduction on the slow down metric. For this following experiment we considered three different partitions, based on the initial configuration of the DEEP-EST prototype. Although the plan for the number of nodes has been changed in the meantime, the considered configuration is still valid for theorethical analysis:

- Partition CM with 50 nodes, Sockets=2 CoresPerSocket=12 ThreadsPerCore=1
- Partition ESB with 150 nodes, Sockets=2 CoresPerSocket=28 ThreadsPerCore=1
- Partition DAM with 25 nodes for DAM, Sockets=2 CoresPerSocket=20 ThreadsPerCore=1

For the jobs, we based our trace on the RICC-2 [30]. We shortened it to 1,000 jobs (for feasible deterministic runs). We also filtered it to list the first 1,000 jobs which ask for a maximum number of 128 cores (to avoid that a huge job would never run on DAM for instance). We used a converter to update the jobs on the trace so each partition would receive about 33% of the jobs. Finally, we "submit" jobs requesting the DAM partition to also request the ESB. Finally, we did a sweep on the amount of jobs that would receive (DAM, ESB) as the input for

partition, which made it possible to verify the benefit of having a second partition option for smaller partitions.

The Figure 10 shows the reduction in average slowdown when the percentage of jobs that can be executed in either DAM or ESB increases. The base case is the workload where each job can be executed on one strictly specified partition (only DAM). Finally, we present the sweep across the x-axis which is the percentage of jobs requesting DAM which also request ESB.

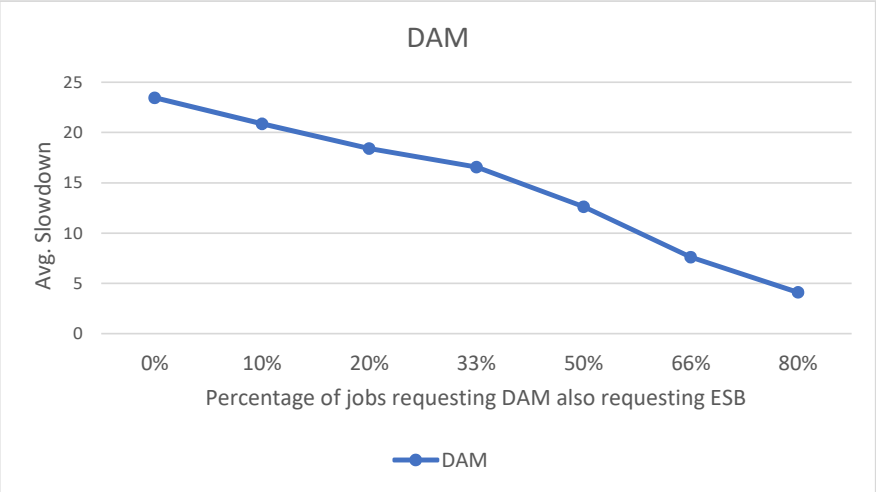

Figure 10: Benefits of requesting DAM module's resources as a first choice and ESB module's resources as a second choice.

The Figure 11 presents the same sweep across the x-axis which is also the percentage of jobs requesting DAM which also request ESB. Besides, in this Figure it also correlates the reduction in average slowdown to the reduction in number of jobs executed in this partition. As more jobs are allowed to run on ESB partition, less jobs are executed in DAM, translating into the reduction seen in the average slowdown.

5.1.2 The approaches considered to support heterogeneous modules

However, in modular architectures, the resources among modules are heterogeneous which makes it necessary to specify time and resources per-module. We have contemplated several approaches, and describe in the following their advantages and disadvantages. The criteria that we had in mind when considering different approaches are portability, cost of implementation, and user experience.

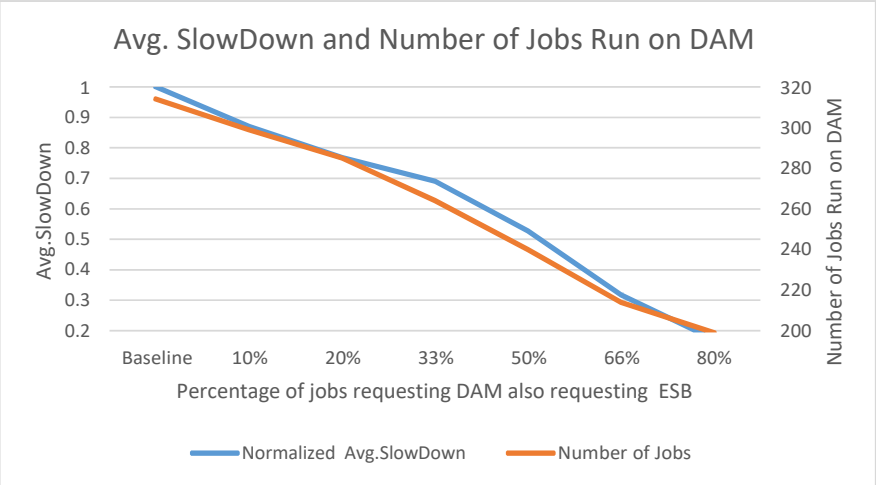

Figure 11: Correlation between the reduction in average slowdown time and the reduction in jobs run in the partition.

First, we assume that the user employs only a single `sbatch` requesting the list of modules, and the requested computing resources and wall-clock time are specified having in mind the preferred, i.e., first-in-list module.

We assume that in MSA the computing resources across modules are of different type. Thus, to support requests for different modules we need to enable Slurm to store the necessary amount of computing resources and wall-clock time per module. Moreover, we would require Slurm to automatically calculate the per-module amount of resources to make it easier for the user. There are two tasks here.

- First, we need to define in which job structure we will store the per-module resource and time description.
- Second, we need to define how to automatically recalculate resource and time per-module based on the resource and time given for the preferred module.

We have investigated which changes to the job structures need to be done to accommodate these new per-module resource and time description. Since a single `sbatch` is done, there will be a single job record generated. Also, there will be per module job_queue_rec created, each containing a pointer to the same job record (Figure 12).

One option was to replicate the original job record and change the amount of resources (min nodes, max nodes, req nodes) and time limit field. The issue that we encountered is that there exists a hashtable in Slurm that does a unique mapping between job id and job record pointer. Thus, having one job id for multiple job record pointers would not be compatible with the current implementation of Slurm and changing this design might significantly impact the

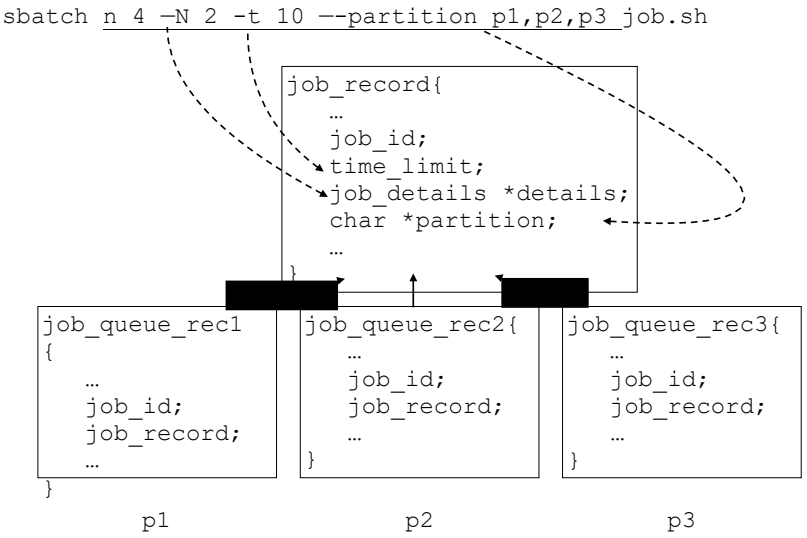

Figure 12: Slurm's job structures used in the case of multiple modules.

portability. The conclusion from this research is that the job records per module should be created at submission time, having a unique job id per job record.

Therefore, as a second approach, we considered using the mechanism of heterogeneous jobs which would create a sub-job for each module with a direct definition of time and resources per module. In this case, the job structures would be created per user request per module, and each will have its own job_id. The idea would be to allow only one sub-job to execute while cancelling the rest. Since the mechanism of heterogeneous jobs is rather new in Slurm and parts of job scheduling code, such as backfill, do not support well this mechanism, we decided that it might be too costly to adapt this mechanism for the new purpose of choosing the best module. Also, note that this approach would require the user to define the resources and time per-module and our final goal is to enable Slurm to do it automatically. Moreover, the concept of heterogeneous job is not created for that purpose and selecting this option could lead to a conflict when using heterogeneous jobs to run in multiple modules at the same time.

Finally, we considered using the mechanism of dependencies. The idea would be to do multiple `sbatch` submissions of the job using a newly defined dependency. Our starting point is an already existing dependency option *singleton*. All the jobs with this option having the same job name and user id will be executed one at a time. Our idea is to adapt this option and create a new one called *plussingleton*. The adaptation consists in allowing that out of all the jobs with this dependency type and having the same job name and user id only one, namely the one that gives the lowest slowdown, will be executed, and the rest will be cancelled. At the moment, this would require that the user submits as many `sbatch` as the modules he wants to be considered, specifying the same `--job-name` for each of them and specifying the `--dependency=plussingleton` option and the `--kill-on-inv-dep` flag. The flag is an existing flag in Slurm, and it will enable cancelling all the jobs that have an invalid dependency, i.e., that are not chosen for the execution. Within each `sbatch` a single module and time and re-

sources for that module will be specified. Besides, the jobs with lower requested time are given higher priority. Note that our current implementation does not allow for automatic translation of resource and time requests, as shown below.

```
sbatch -n 8 -N 4 --time 5 --priority=3 --partition=esb --dependency=plussingleton
    --kill-on-invalid-dep=yes --job-name=myjob job.sh
sbatch -n 8 -N 2 --time 7 --priority=2 --partition=cm --dependency=plussingleton
    --kill-on-invalid-dep=yes --job-name=myjob job.sh
sbatch -n 8 -N 1 --time 14 --priority=1 --partition=dam --dependency=plussingleton
    --kill-on-invalid-dep=yes --job-name=myjob job.sh
```

5.1.3 The evaluation of the first prototype based on new dependency type

We present the evaluation of our first implementation of a scheduler prototype that supports the first scenario. The experiments are conducted on a real machine, i.e. the MareNostrum supercomputer, using our Slurm-over-Slurm environment and a synthetic 50-jobs workload on a 9-nodes cluster. In the 9-nodes cluster, we configured Slurm with three modules and called them *CM*, *ESB*, and *DAM*. There are three CM nodes, five ESB nodes, and one DAM node. These numbers of nodes are chosen to keep the proportion among modules similar to that one in the DEEP-EST prototype. The base workload was created such that the user of each job specifies strictly one of three modules. Each module is requested by approximately one-third of the jobs. We did not execute real applications but sleeps of different number of seconds. The derived workloads are created to evaluate our prototype. They are created by adding to 10%, 20% and 30% of job requests two additional requests for the other two modules. These two additional requests have *plussingleton* dependency with the original one and only one out of the three dependent job requests will be executed and the rest will be cancelled. Also, we create a simple model that converts the number of resources and wall-clock time requested from one module to another. We assumed that the nodes in different modules have different CPU frequencies and memory sizes and time for sleeps are calculated proportionally. This is not the case in MareNostrum machine since all the nodes were homogeneous, but in this way, we model heterogeneity of resources. Even though there are two additional jobs in the derived workload, only one of them is actually executed, thus the amount of load remains the same. We execute all workloads on the same system and configuration and compare metrics in Figure 13. The figure shows that, as we enable more jobs with the choice of more modules, the average wait time and average slowdown reduce by 11-40%. The purpose of the experiment was to actually validate the implementation of our prototype. We will further implement the prototype in the Slurm simulator and do more large-scale experiments on bigger systems and workloads. This will allow us to better evaluate the improvements in the system performance when using our prototype.

5.1.4 Next steps for prototype refinement

In the next steps, we will focus on improving the user experience. The idea would be to create a new or adapt the existing `sbatch` command such that all the described process is done automatically. For example, the user would use a new `sbatch` command's option (e.g., `--module-list`) specifying the list of modules. This Slurm implementation of the new option will trigger multiple

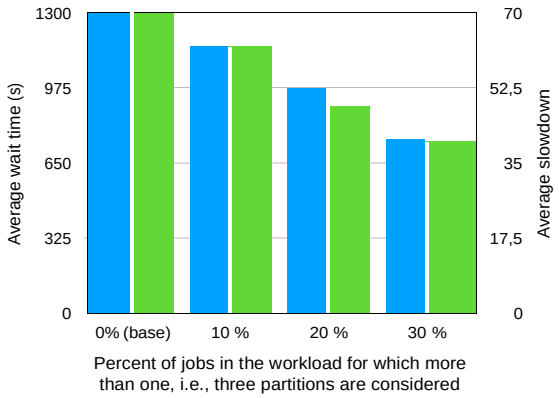

Figure 13: Average wait time and average slowdown of the jobs in the workload. On the x-axis we show percent of jobs in the workload for which all three modules are considered when making resource selection decision. 0% is the base case workload where for each job strictly one module is considered for resource selection.

job submissions that have the *plussingleton* dependency among them. Also, we will work on enabling Slurm for automatic calculation of the resources and time per module. The job_name option will be replaced with a new option, not to create a conflict with other scenarios in which the same user can use the same job_name. The priority of the jobs will be calculated by Slurm based on the wall-clock time values. Besides, options `--dependency=plussingleton` and `--kill-on-inv-dep` will be used only internally by Slurm, the user will not have to specify them. The user will have to do a single `sbatch` and specify nothing more than a list of modules as shown below.

```
sbatch -n 8 -N 4 --time 5 --module-list=esb,cm,dam job.sh
```

5.2 Efficient support for coupled workflows using DEEP-EST features

The heterogeneous job pack support introduced in Slurm 17.11 [24] provides features needed for applications using modular architectures. In such implementation, multiple jobs can be run on different modules simultaneously, with the possibility for the user to specify the job parameters of each component individually. All the components of a heterogeneous job are expected to start and end at the same time, with the caveat that it is possible to allow for different durations for each component by specifying different values for the `--time` option. Nevertheless, the heterogeneous job pack leader (i.e. the first component of the heterogeneous job) must remain in running state for the entire duration of the heterogeneous job.

This behaviour does not necessarily reflect the workload of some applications, which might not need all the resources at the same time. In particular, one part of a heterogeneous application might need to run only after the completion of another part due to data dependency. Another possible scenario is a heterogeneous job pack where multiple components need to be used at

the same time, but resources in some components might be actually used only after an initial (or before a final) *homogeneous* section of the application. While the first scenario would already be addressable by using dependencies for jobs running on different modules, the second one would require the possibility of introducing a delay between the start time of different jobs (or different components of a heterogeneous job). Though, the introduction of a delay between jobs could also be exploited in the first scenario (and more in general with any type of job dependency), allowing the establishment of network communication between two consecutive jobs, potentially minimising the time required to transfer data across the jobs and therefore minimising the waste of computational resources.

5.2.1 Evaluation of the data transfer times for workflow applications

A possible application of the delay switch previously described concerns its usage with workflows, to allow jobs to exchange data over the network. This solution might be preferable to dumping the data to the filesystem, and then retrieving it at the beginning of the following job.

In order to investigate the potential benefit in using the delay switch for modular workflows in the context of the DEEP-EST project, an analytical model of the expected performance of data transfers across the envisaged system was developed. This model was made necessary given the current unavailability of the DEEP-EST system, which is planned to be ultimated by the end of 2019 (as after the third amendment to the DoA). To build the model, the envisaged system described in [17] and [18] was considered, using the hardware details provided in [15] and [16]. The main assumptions used in building the model are listed below:

- Only transfers between pairs of nodes, either of the same module or of different modules, were considered. It's assumed that it's possible to base estimations of transfer across larger sets of nodes on the node-to-node calculations here shown.
- All transfers of data were modelled using the following functional form, taken from [28]:

$$T_{tr}(N) = T_l + \frac{N}{B} \tag{5.1}$$

  where T_{tr} stands for the transfer time of an amount of data N , T_l being the latency in the communication and B a theoretical bandwidth.
- The transfers of data across multiple devices were modelled in the following manner:

$$T_{tr}(N) = \sum_i T_{li} + \frac{N}{min(B_i)} \tag{5.2}$$

  where the i index refers to the i-th device. This simplified model implies that each device imposes an additional latency to the transfers, while the bandwidth is capped by the minimum value of all devices involved.
- When crossing gateways, each pair of nodes can make use of one gateway node at most (multiple pairs can use the same gateway).
- When using the NAM, only one NAM device is supposed to be used at a time.
- The ideal maximum bandwidths were used for each device. For the gateways, this is equal to the ideal bandwidth of the slowest of the two networks they bridge. For the

filesystems, it is assumed that the disk operations are the bottleneck for the data transfer. For the NAM, it is assumed that the bottleneck lies in the read and write operations to the NVMe SSDs attached to the NAM board, as described in [16].

- The latencies used here are first guesses, since the real values heavily depend on technology and topology and the final system is not available for measurements. In general it is expected that latencies dominate only very small transfers, which are not considered to be relevant for transfers across components of a workflow.
- No caching effect was considered in the model for the filesystem and NAM.
- Correction coefficients are used to obtain more realistic final values for the bandwidths on the different devices.
- Data transfers via network occur only once, while transfers via filesystem and NAM are taken into account for separate write and read phases, with possibly different bandwidth used for read and write operations to disk and NAM.

The network federation layout depicted in Figure 5 shows that MPI bridging across networks will be supported only for the InfiniBand – EXTOLL interconnection. Moreover, it is expected that any transfer between CM, ESB and DAM will occur across this interconnection, to avoid the bottleneck of the low bandwidth of the Ethernet network (given that the DAM nodes will be equipped with EXTOLL as well). Therefore, in the model only three scenarios for data transfers in workflows were considered:

- InfiniBand-InfiniBand;
- EXTOLL-EXTOLL;
- InfiniBand-EXTOLL.

In addition to this, since it is envisaged that in the future more than one InfiniBand - EXTOLL gateway node may be available, two additional scenarios were accounted for:

- 1 gateway node;
- 4 gateway nodes.

The values of the parameters used in the model are reported in Table 3. Even if only InfiniBand and EXTOLL were considered for the workflow transfers, the parameters for the Ethernet devices are necessary for the evaluation of the transfer via filesystem.

The results obtained by the model, reported in Figure 14, generally show that the network option is always faster than the others. This is firstly due to the fact that filesystem and NAM transfers are doubled for the write and read operations. In addition to this, such transfers might need to cross more gateways with respect to network transfers. Finally, especially in the case of the filesystem, the write and read operations to the device might provide the main bottleneck to the whole transfer process.

For intra-modular transfers, the network option is considerably faster than the others by at least one order of magnitude. This is due to the absence of gateways, which provide an important slowdown according to the parameters used. Some differences arise for the NAM options between the InfiniBand – InfiniBand and the EXTOLL – EXTOLL transfers due to the fact that the NAM devices are on EXTOLL, so in the second case the gateways do not provide a potential bottleneck.

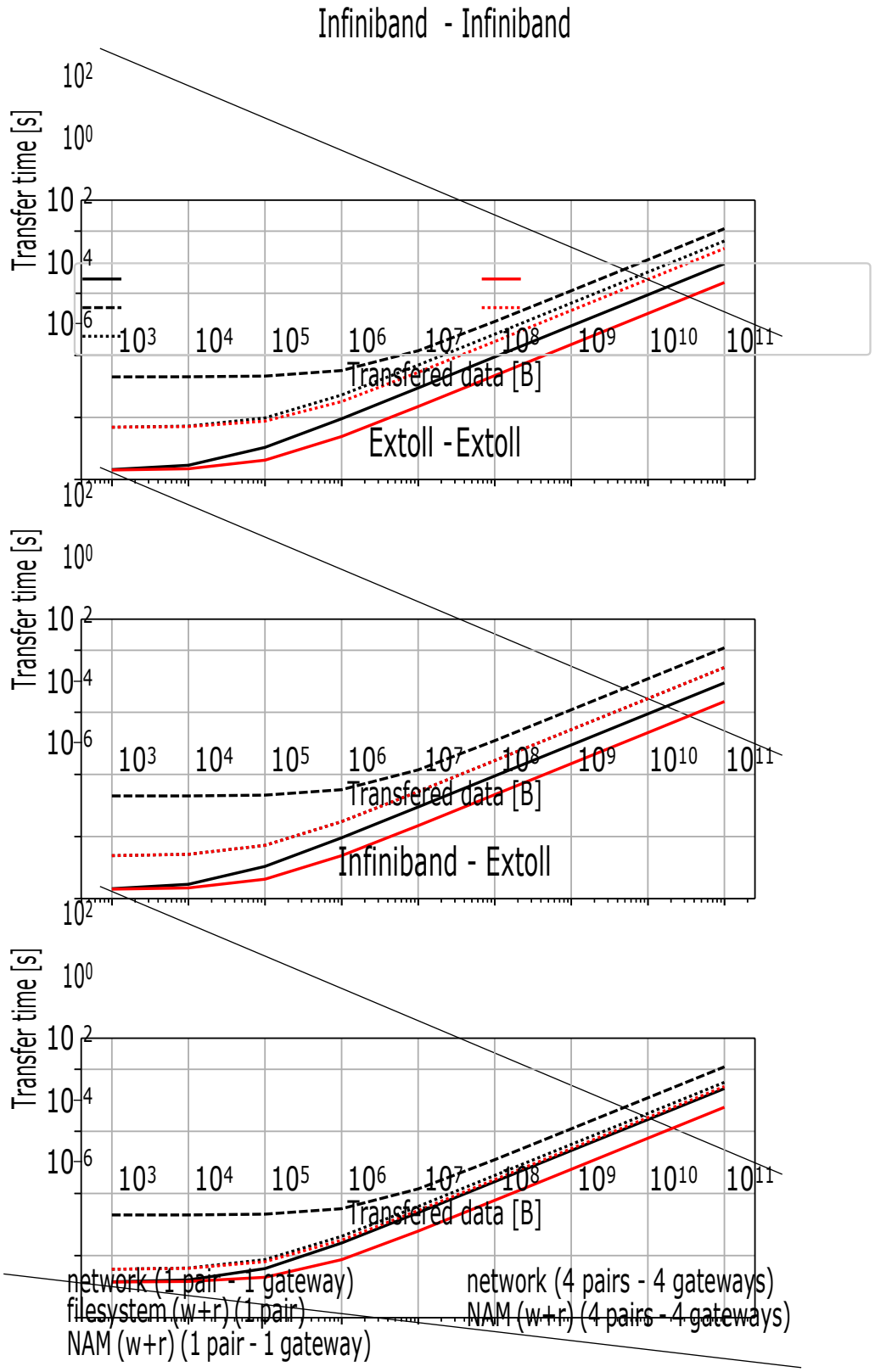

Figure 14: Results obtained by the data transfer module for different combination of networks.

Network option	Latency	Bandwidth		Corr. coeff
		read	write	
InfiniBand	2 µs	12.5 GB/s		0.9
EXTOLL	2 µs	12.5 GB/s		0.9
Ethernet	2 µs	5.0 GB/s		0.9
InfiniBand-EXTOLL GW	10 µs	12.5 GB/s		0.33333
Ethernet-EXTOLL GW	10 µs	5 GB/s		0.33333
InfiniBand-Ethernet GW	10 µs	5 GB/s		0.33333
Filesystem	1 ms	5 GB/s	4 GB/s	0.5
NAM	10 µs	8 GB/s		0.9

Table 3: Parameters used in the model for the data transfer performance.

For the inter-modular cases, the performance of network transfers is reduced by the gateway bottleneck. For transfers between one pair of nodes, this results in comparable transfer times between network and NAM (less than a factor 2).

In all scenarios, considering multiple gateways between InfiniBand and EXTOLL results in a strong improvement in the performance of network transfers. Four pairs of nodes where used in this case, since with the current implementation of the inter-modular bridging described in Section 3, one pair of nodes can use at most one gateway node. This also causes an improvement in the NAM transfers for the InfiniBand – InfiniBand and InfiniBand –EXTOLL scenarios.

In general, the following considerations may be made after the analysis of the results of the model:

- larger workflows that span multiple nodes may make use of the combined network bandwidths across the nodes when transmitting data via network. This is particularly relevant for homogeneous workflows, since in the case of heterogeneous workflows the limited number of gateway nodes would cause a bottleneck for the data transmission. Still, increasing the number of gateway nodes between InfiniBand and EXTOLL would relieve this constraint.
- For large amounts of data to be transmitted, the absolute difference in transfer times among the different options could become considerable, in favour of the network option that would be exploited with the delay switch.
- NAM is a limited shared resource, the total capacity of which is expected to be in the order of 12 TB. High utilisation of such resource might result in limited availability for the user running workflows, making it more beneficial to adopt the delay switch. Also, access to NAM is currently supported only within the EXTOLL network: the delay switch could become the only alternative to file system transfers for InfiniBand-related workflows in case NAM access from InfiniBand is proved infeasible or non-performing.

5.2.2 Implementation of the delay switch

In D5.2 [14], we elaborated the need for workflows with overlapping jobs to enable direct communication between the jobs. This is not available in standard Slurm. For this purpose, we

planned to change the heterogeneous job pack feature of Slurm. A thorough investigation of the job pack feature was done. Job packs are currently only scheduled using the backfill scheduler plugin available in Slurm. Algorithm 1 shows the original working of the backfill scheduler.

```
foreach job in JobsList do
    if canStart(job) then
        if isJobPack(job) then
            saveStartTime(job);
        else
            start(job);
        end
    end
end
 foreach jobPack in jobPackList do
    flag = false;
    foreach job in jobPack do
        if canStart(job) then
            flag = true;
        else
            flag = false;
        end
        if flag == false then
            break;
    end
    if flag then
        startAllJobs(jobPack);
    end
end
```

Algorithm 1: Abstract working of backfill scheduler of Slurm.

To enable overlapping workflows in Slurm, we proposed a new *delay* switch. A user provides the amount of time for each component (a job in Slurm) in a job pack to be delayed from the start of the first component. We used this information to find appropriate scheduling for the whole job pack. Once the whole job pack can be scheduled with the desired delaying times for each component, we reserve the nodes using the standard Slurm reservation system to ensure starting of respective components at the desired time. Algorithm 2 shows the changes we made to the backfill scheduler.

The modified backfill scheduler checks if the resources for all the components in a job pack where the *delay* switch has been used can be reserved at the required times. The newly created reservations are linked to the respective components of the job pack. Slurm allocates the resources automatically to each component when the reservation time starts. In case that any component's resources could not be reserved at the required time, we get a new time when this component could be allocated the required resources. In that case, this new time difference is added to the start times of all the components. All of the already made reservations are deleted and the whole process of reservations starts again. The whole process continues until

```
foreach job in JobsList do
    if canStart(job) then
        /* including the delay switch value                    */
        if isJobPack(job) then
            saveStartTime(job);
        else
            start(job);
        end
    end
end
foreach jobPack in jobPackList do
    if ! isDelaySwitchUsed(jobPack) then
        /* No changes to original backfill strategy for normal job packs
                                                                */
    else
        foreach job in jobPack do
            if canResourcesBeReservedAtStartTime(job) then
                reserveResources(job);
            else
                deleteAllReservations(jobPack);
                t = TimeWhenReservationPossible(job) - currentStartTime(job);
                addDelayTime(jobPack, t);
                if StartTimeOfFirstJob(jobPack) < backfillWindow then
                    /* reset the job to the first element of jobPack
                    resetCounter(jobPack);                      */
                else
                    /* This job pack cannot be allocated in current backfill
                       window. Goto the next job pack
                    break;                                      */
                end
            end
        end
    end
end
```

Algorithm 2: Changes done to backfill scheduler.

all the components have reservations or the starting time of the first component in the job pack becomes greater than the backfill window (a parameter to the backfill scheduler).

Initially, we added the *delay* switch to the parameter list of the `srun` command of Slurm. However, `srun` has been designed to start all the components of a job pack simultaneously. We tried to change this design as not all the components in our proposed workflows start at the same time. Due to the complex design of `srun`, our attempts to change it were not successful. A to- tally new design could have been implemented for `srun` but our aim is to make as few changes

as possible to the standard Slurm implementation itself, since this will facilitate the adoption of our extensions by future Slurm distributions.

Our current implementation has the *delay* switch added to the `sbatch` command of Slurm. `sbatch` is used to just submit the jobs to Slurm, which in turn puts the new jobs into a queue and schedules them accordingly. The backfill scheduler allocates the job packs with delays as described in the Algorithm 2. As the heterogeneous job pack feature in Slurm also creates new environment variables, these can be used in a job script to determine which application to run in which component. The following listing provides an example `sbatch` batch script.

```
$cat my_job.cmd
#!/bin/bash
#SBATCH --mem-per-cpu=16g -t 60 -N1 -C skylake
#SBATCH packjob
#SBATCH --N32 --exclusive -C NAM --mem_nam=100GB --delay 55

if [ $SLURM_JOB_ID == $SLURM_JOB_ID_PACK_GROUP_0 ]
then
    srun app1
elif [ $SLURM_JOB_ID == $SLURM_JOB_ID_PACK_GROUP_1 ]
    srun app2
fi
$sbatch my_job.cmb
```

This script runs only `app1` when executed in the first component of a job pack and `app2` in the second component. Currently, the batch script is only run by the root node of the first component in a job pack. Work is ongoing to change this behaviour in the case of job packs where the *delay* switch has been used. In that case, the script will be run on the root nodes of all the respective components in the job pack.

5.3 Efficient Scheduling and Management for Shared Global Resources

The Extreme Scale Booster (ESB) module will be configured with the EXTOLL fabri[3] network. This network allows for the attachment of resources other than the compute nodes in its multi-dimensional torus topology.

Extensions to the different system software packages have been planned and some already developed in order to represent, manage, and schedule these resources. Additionally, batch script parameters to request these resources have been added. Software interface extensions to allow the use of these resources from applications have been planned.

5.3.1 Shared Global Resource Types and Expected Configuration

There are two types of shared global resources planned for the Extreme Scale Booster (ESB) module in the DEEP-EST prototype system: Network Attached Memory (NAM) and Global Collective Engine (GCE).

Network Attached Memory (NAM)

The ESB will have multiple Network Attached Memory (NAM) instances attached as peers in its EXTOLL network. Each instance will have its own storage capacity and a limited number of simultaneous connections.

There will be an independent NAM manager at each instance, with NAM managers not being aware of each other. The current plan is to have the distributed view of NAM resources only at the Slurm instance configured for the DEEP-EST prototype.

A portion of the available NAM capacity will be reserved for exclusive allocations performed manually by system administrators based on user requests.

Global Collective Engine (GCE)

Similarly to NAM resources, the Global Collective Engine (GCE) instances will be attached as peers in the EXTOLL network. The GCEs allow for the offload of common blocking and non-blocking collective operations. The ParaStation MPI library will be adapted to make use of this resource transparently for the benefit of MPI applications running in the ESB.

5.3.2 Requesting Shared Global Resources with sbatch

Shared global resources for a job or job pack will be requested with the `sbatch` command in the final prototype. New options have been added to the `sbatch` command, based on current development plans and design of the NAM and GCE resources.

Requesting Network Attached Memory (NAM)

As mentioned earlier, Network Attached Memory (NAM) instances have two limited resources: capacity and connections. Only the capacity resource is exposed to the users. The connections will be managed by Slurm and the ParaStation process manager based on node and process counts. Users can specify the NAM capacity requirement of a job with the following `sbatch` option:

```
--network-attached-memory=<megabytes>
```

The specification of this requirement triggers only a bound check in the current prototype, based on the configured total capacity of the NAM in the Slurm system. There is no other effect and the option is otherwise ignored. Functionality for this resource will be developed initially against a mockup of the NAM library, and later with its actual implementation when available.

The visibility of the NAM from partitions other than the ESB is still under discussion. This will determine whether NAM requirements will only be valid on jobs that target the ESB.

Enabling or Disabling the Global Collective Engine (GCE)

It is currently too early in the development of this resource to have a final solution. The current expectation is that the Global Collective Engine (GCE) will be either enabled or disabled. The following option was added to `sbatch`:

```
--global-collective-engine=<enabled|disabled>
```

This will likely change as the design of the GCE resource develops. It is possible for this feature to be hidden from users entirely, and as a consequence this `sbatch` option could be removed. For example, ParaStation MPI could take advantage of the GCE always when it runs in the ESB module, making this option redundant.

5.3.3 Shared Global Resource Configuration

The modules of the Modular Supercomputing Architecture (MSA) are represented as separate partitions in the Slurm configuration file. In addition to the nodes, the shared global resource instances will need to be defined as parts of the Extreme Scale Booster (ESB) module definition. In the DEEP-EST prototype system, shared global resources will only be available in the ESB module, since these are hardware features of the EXTOLL network only.

The partition section of Slurm's configuration file (*slurm.comf*) has been extended with two new keywords:

PartitionNetworkAttachedMemory: Total NAM capacity in megabytes.

NetworkAttachedMemoryInstances: Host names of the NAM instances separated by comma.

GlobalCollectiveEngine: Specifies if the partition has one or more GCE instances.

GlobalCollectiveEngineInstances: Host names of the GCE instances separated by comma.

The internal partition information data structure has been updated to include matching members. The NAM capacity is stored as an integer, with zero as default. The NAM instances are stored as a string and follow the same implementation as the nodes definition, with a NULL value as default. The GCE is a boolean value that specifies if the GCE feature is present in the partition, and defaults to false. Its list of instances follow the same hostname list format NAM and nodes strings. These data structures will be further extended as necessary when the NAM library is made available and the GCE design is finalised. Additional data will be added, such as a NAM and GCE node index array, if it is decided that Slurm will monitor these like regular nodes. Monitoring functionality will be determined later, once the API towards the NAM and GCE managers over the network is finalised; monitoring these will be perform in a different manner when compared to regular nodes, since there will be no `slurmd` instance running in these resources.

6 System Monitoring and RAS Plane

In this chapter we describe the development status and present novel features of DCDB [4], the holistic sensor monitoring tool that will be deployed on the DEEP-EST prototype. An initial system design of the tool can be found in deliverables D5.1 [13] and D5.2 [14]; additionally, an indicative list of sensors and performance metrics that will be collected on the prototype is listed in [14]. Section 6.1 describes the current software architecture of the tool, providing details on recent changes and improvements introduced since our last report. In Section 6.2, we present the DCDB data source plugin for Grafana, which will allow for a comprehensive and effective visualisation of the sensor data of the monitored DEEP-EST prototype. Section 6.3 introduces the DCDB Data Analytics Framework, a plugin-based software component tightly integrated in DCDB that allows to perform asynchronous analytics tasks in a streaming fashion. Finally, Section 6.4 concludes the chapter presenting the direction of our future efforts in the project.

6.1 Current Status

The architecture of DCDB (depicted in Figure 15) did not change much since we reported on it in the previous Deliverable D5.2 [14]. The most notable changes are the addition of the Data Visualisation module and the Data Analytics engines in the Pusher and the Collect Agent that are described in more detail in Sections 6.2 and 6.3, respectively. Other than that we continuously improved the DCDB code base in terms of stability and performance and added

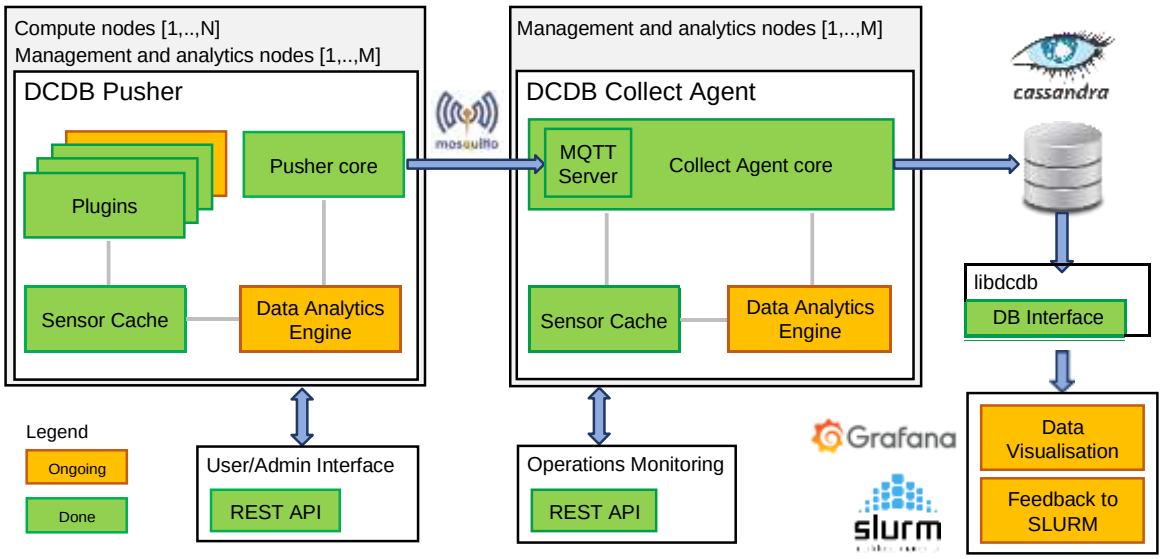

Figure 15: Architecture of DCDB.

minor features such as auto-publishing of sensors, new plugins (e.g., for Linux procfs, BACnet, SNMP, and MSR registers), and support for wildcard queries in libdcdb. Most recently we started work on integration with Slurm such that DCDB can also be queried for sensor data relevant for a particular Slurm job.

6.1.1 Improvements to MQTT Communication

During production testing of DCDB on one of the Linux Cluster systems at BAdW-LRZ we came across a limitation in the mosquitto MQTT library that DCDB employs for communication between the Pusher and the Collect Agent: it only allows for 1024 open file handles. While this is typically not an issue in most setups, it becomes problematic in combination with DCDB's PerfEvents plugin that leverages the Linux perf_events API for accessing performance counters. With perf_events, a new file handle is opened for each core and each metric to be measured. On systems with high core counts, it is therefore easily possible to exceed 1024 open file handles which then prohibits any MQTT communication. This limitation is due to the fact that mosquitto utilises the `select()` system call to wait for inbound packets on its communication socket which, in Linux, has a hard-coded limit of 1024 file handles it can monitor.

To overcome this issue we briefly switched to the Eclipse Paho library for MQTT communications [6] that also promised other interesting features such as auto-reconnect, support for high availability configurations, and tighter control over allocated memory buffers and therefore its memory footprint. However, already during initial testing it turned out that Paho also employs the `select()` system call and hence has the same limitations in terms of open file handles as mosquitto. On top of that, MQTT communication with Paho proved as unreliable at high message rates which effectively rendered it useless for use in DCDB.

We finally decided to stay with mosquitto and patched its source code to use the `poll()` system call instead of `select()` as `poll()` has no limitations in the number of file handles it can monitor. As we were particularly intrigued by Paho's feature to control the size of the message buffers it uses, we decided to also implement a similar functionality in mosquitto: specifically, we introduced a new library call `mosquitto_max_queued_messages_set()` that allows for setting a limit on the number of messages that will be queued in case they cannot be delivered to the MQTT broker. Since sensor data is usually read at high frequency by the Pusher and hence sent to the Collect Agent at high message rates, longer outages of the Collect Agent (e.g., due to updates or network failures) would otherwise result in large amounts of memory being occupied to buffer those messages until the Collect Agent becomes available again. As the Pusher will often run on compute nodes to collect in-band data, such behaviour could cause scientific applications running on the compute nodes to crash due to memory shortage. All patches to the mosquitto source will be committed upstream for other users of mosquitto to benefit from our improvements.

6.2 Visualisation of Sensor Data with Grafana

As previously reported in Deliverables D5.1 [13] and D5.2 [14], several visualisation tools for plotting the sensor data monitored by DCDB have been considered and thoroughly evaluated.

Amongst all of them, the Grafana visualisation tool from GrafanaLabs [7] provided the best solution for our task, primarily because of its many advantages over other apps, some of which are listed as follows:

- Grafana provides a comprehensive set of visualisation options, all very useful and effective for visualising metrics in HPC administration setups (e.g., graphs, heatmaps, histograms, tables, etc.).
- It is possible to define alerts, allowing to set specific metric thresholds and to receive notifications via multiple options (e.g., email, Slack, PagerDuty, etc.).
- Grafana is designed following an extensible architecture, allowing for the implementation of plugins supporting different dashboards and features.
- It is a completely open-source project, backed by a strong user and developer community.

Grafana additionally provides support for several databases, some of which already benefit from a dedicated plugin directly shipped with the tool (e.g., InfluxDB [8]). However, at the time of writing, there is no plugin available for Apache Cassandra [5], the database upon which DCDB is built, and potential development efforts in this direction are not included in the road map of the GrafanaLabs developers. This lack of support motivated us to write our own plugin which, in addition to retrieving data from the Cassandra key-value store, is also designed with the goal of profiting from current and future features offered by DCDB.

6.2.2 The Grafana DCDB Data Source Plugin

The DCDB data source plugin for Grafana is built on top of the Simple JSON data source plugin [9] and most of its methods to retrieve data are based on query mechanisms that the JSON plugin offers. Every client request is performed via RESTful APIs and a dedicated HTTP server is responsible for processing them and for sending back the query results. In the Grafana Simple JSON data source plugin, the URLs that the HTTP server needs to support for processing HTTP requests are mainly three, specifically:

- `/`: this URL sends a GET request only once and is used to setup the data source. It makes sure that the connection to the database has been established and checks user credentials via HTTP basic authentication.
- `/search`: this URL is used to send a POST request to retrieve the list of metrics (sensors) available from the database. The returned data is formatted as a JSON file.
- `/query`: this URL sends a POST request and issues the query to acquire monitored data for the selected sensor over a specified interval of time. Similarly to the `/search` request, the sensor data and the associated sampling timestamps are returned in a JSON format.

The DCDB data source plugin utilises the same URLs for the same purposes, with the exception of the `/search` request which has been modified to support the selection of multiple levels of hierarchy in a data centre, as explained in the next section.

6.2.3 Supporting Hierarchical Queries

To this day, a missing feature in Grafana and in all of its plugins supporting different database backends is the possibility of building a hierarchical query, and select metrics at a certain level of the hierarchy. This functionality is particularly useful for sensor monitoring set-ups in HPC or data centre environments, where a system administrator could "navigate" through different hierarchical levels of a system (e.g., a rack, a chassis, or a server) and query data from specific sensors available at the selected level. This feature becomes almost necessary if the monitored system comprises a very high number of sensors (i.e., potentially in the order of millions for a flagship supercomputer), a scenario which would be extremely inconvenient to manage with a flat query mechanism. Under these premises, the DCDB data source plugin for Grafana provides exactly this feature by extending the `/search` POST request supported by the JSON plugin with hierarchical levels.

The resulting Grafana panel with the DCDB data source plugin is depicted in Figure 16. As illustrated, the query tab of a Grafana panel comprises two dropdown menus, one for selecting the hierarchical level of the system and a static one for selecting a sensor at a given level. When the user selects a specific level, the next one in the hierarchy (if present) becomes dynamically available for selection with a new dropdown menu. When selecting a specific level or sensor, the `/search` POST request is sent with the following URL format:

```
/search/<level 1>/.../<level n>/[sensor]
```

where:

- `<level i>` specifies the `i`-th level of the hierarchy (e.g., rack, chassis, node, etc.)
- `[sensor]` is a keyword that is appended at the end of the URL to distinguish between POST requests related to hierarchical levels and POST requests associated with sensors at a specific level.
- `n` represents the maximum number of hierarchical levels of the system.

Figure 16: Grafana DCDB data source plugin with hierarchical query support.

Once the request has been received, the HTTP server correctly selects the required level of hierarchy and queries the database for the selected sensor values to be plotted, returning the timeseries data in a JSON format. Figure 16 illustrates a visualisation example of four sensors, specifically the power consumption of four different nodes during the execution of the Kripke benchmark on the CooLMUC-3 system of BAdW-LRZ, a 148-node cluster of Intel Xeon Phi 7210-F processors. It should be noted that Grafana already provides convenient visualisation features such as stacking of timeseries data or comprehensive formatting of axes and legends (i.e., displaying useful information like current average or maximum values of plotted metrics).

6.3 The DCDB Data Analytics Framework

In this Section we describe the architecture of the data analytics framework that was developed on top of the DCDB monitoring solution, and introduce its most important features.

6.3.1 Investigation and State of the Art

Before starting development of the DCDB Data Analytics Framework, we investigated the state of the art and identified the most feasible design path. Our framework must be able to tackle problems in the streaming, real-time domain that are common in HPC system management, such as energy efficiency optimisation or anomaly detection. Moreover, the tool should be flexible, scalable and suitable for deployment in a wide variety of use cases.

Existing open-source monitoring solutions such as the *Distributed Lightweight Metric Service* (LDMS) [10], *Examon* [11] or *Ganglia* [12] do not offer any form of built-in streaming data analytics functionality, and rely on external tools to perform such tasks. Examon, in particular, relies on using the *Apache Spark* framework to perform data analytics. On the other hand, commer- cial and closed-source products such as *Zenoss*[1] or *Splunk* [2] offer data analytics capabilities to some extent. However, due to their commercial nature, these products are not suitable for use in the context of the DEEP-EST project and in many HPC environments as well.

Given the above, we decided to design a custom data analytics framework for DCDB. We considered using tools such as Apache Spark to support our data analytics infrastructure, similarly to Examon. The architecture for such a framework is sketched in Figure 17. This design choice would have reduced development effort considerably, leading to a framework loosely-coupled with DCDB. However, this would have resulted in the following limitations:

1. Spark is based on the *map-reduce* programming model, which is limiting;
2. Such tool needs to be setup on a independent, dedicated cluster, increasing configuration and maintenance efforts considerably;
3. The resulting data flow is inefficient, since DCDB CollectAgents need to send the same sensor data to both the Cassandra key-value store and to the Apache Spark cluster, in order to perform streaming data analytics;

[1] https://www.zenoss.com/
[2] https://www.splunk.com/

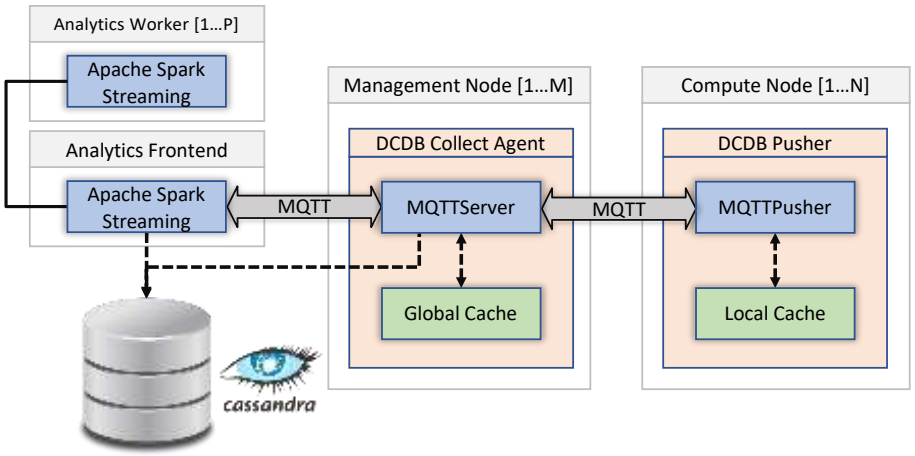

Figure 17: DCDB Data Analytics Framework's architecture if Apache Spark was used.

4. Native DCDB features such as sensor caching are unused.

As such, we decided to pursue the development of a tightly-integrated streaming data analytics framework in DCDB, not based on any external tools and resulting in a highly-efficient architecture. This does not limit the scope of our framework, since tools like Apache Spark can still be used to perform offline analysis by accessing the DCDB Cassandra key-value store directly.

6.3.2 Architecture Overview

Our data analytics framework is plugin-based, similarly to the DCDB Pusher itself. Its architecture is described in Figure 18. Each plugin contains algorithms to deal with specific problems (i.e. regression, classification or clustering) which can be independently configured and instantiated. Plugins are instantiated in the form of *analysers*, which are the actual entities performing asynchronous analysis operations. Each analyser, in turn, operates on a series of *units*, which are basic operational blocks identified by a series of input and output sensors. Units represent entities in an HPC system carrying specific semantic value (i.e. CPU cores, nodes or racks). The *Sensor Navigator* component provides abstractions to navigate the current *sensor space* and identify units using a hierarchical approach, similarly to Grafana hierarchical queries, as described in Section 6.2.2.

A central entity defined as the *Analytics Manager* is tasked with loading the plugins at startup and instantiating the related analysers according to the current configuration. Such entity also allows to retrieve all sensor data generated by the analysis processes and exposes the underlying analysers to the RESTful API, which can be used to perform a variety of management operations. A *Query Engine* entity, instead, provides the analysers with access to all available sensor data that should be used as input. These two functional entities provide an abstraction layer to the implementation of data analytics plugins, which can be subsequently deployed on instances of DCDB Pusher or CollectAgent indifferently.

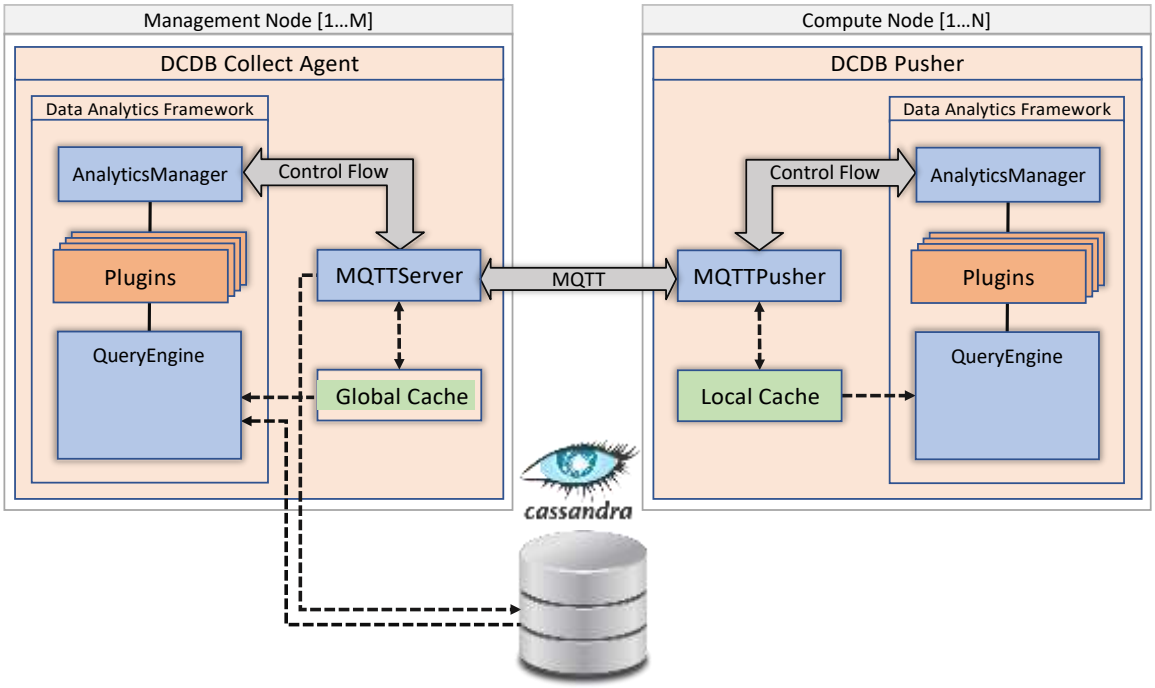

Figure 18: Architecture of the DCDB Data Analytics Framework.

6.3.3 Feature Set of the Framework

Here we supply an overview of the DCDB Data Analytics Framework's main feature set, and discuss how the core algorithms implemented in plugins can be configured to cope with specific necessities, so as to cover most realistic use cases for data analytics in HPC systems.

Configuration Abstractions. We implemented a series of abstractions to ease the instantia- tion of units and analysers. These abstractions are encapsulated in the Sensor Navigator entity, which uses hierarchical relationships to create a tree-like representation of the sensor space, in which leaf nodes represent actual sensors, and inner nodes represent grouping entities such as CPU cores, nodes or racks. We then created appropriate configuration constructs to exploit such representation and intuitively instantiate units in analysers (i.e. creating an analyser with one unit for each CPU core in a node automatically). Furthermore, this abstract tree repre- sentation enables identification of sensors that are connected by hierarchical relationships, and decouples configuration files from the actual deployment scenario and sensorspace.

Deployment. Analysers can be deployed in both, Pusher and Collect Agent instances. Analysers instantiated within Pushers only have access to sensor data in the local cache: this allows for quick turnaround times, and enables the deployment of models at a large scale, by pushing data analytics to the compute nodes. All sensor data produced in the analysis process is then published over MQTT. Such data can be optionally exposed to the local file system, allowing for use by node-level frameworks (i.e. for runtime optimisation) with lowlatency.

Analysers instantiated within DCDB CollectAgent instances, on the other hand, have access to sensor data both in the global cache and in the Cassandra key-value store. As such, this type of deployment allows to combine data from different Pusher instances and is more suitable for complex models (i.e. clustering-based anomaly detection).

Operational Modes. Analysers can be configured to operate in *streaming* and *on-demand* modes. Streaming analysers perform computation periodically and asynchronously, thus producing time series sensor data that is pushed to the Cassandra key-value store. This is desirable, for example, when doing regression of performance metrics. On-demand analysers, instead, operate only when explicitly queried over the RESTful API, and return the results of their computation through such interface. This second approach is preferable for applications requiring data analytics at specific and unpredictable times, such as at job submission or scheduling.

6.4 Future Work

We plan to integrate job information, as supplied by Slurm, in the DCDB Data Analytics Framework so as to broaden its scope and allow for the implementation of models that require such data. Moreover, we plan to identify a series of concrete data analytics use cases for the DEEP-EST prototype and implement appropriate corresponding plugins for our framework. The Grafana DCDB data source plugin will further be improved to support transformations on the plotted data (like derivative, moving average, standard deviation, etc.). Finally, DCDB will be deployed on the DEEP-EST prototype system and made available to system administrators as well as application developers and users.

7 Summary

This deliverable provides a comprehensive overview of the current status of WP5 by presenting the design decisions made since Deliverable D5.2 and summarising all implementation work performed so far. As the major design decisions resulting from the change of the ESB architecture were already discussed in an update to D5.2, this deliverable can directly follow-up and focus on the first prototype software implementations.

The most important implementation advancements with regard to *interconnect management*, *network bridging*, *resource management*, *job scheduling* and *monitoring* are the following:

Enhancement and employment of the EXTOLL Management Program (EMP). Here, in particular the EMP master daemon has experienced significant improvements in the recent project phases. The master is based on the two components EMP *core* and EMP *server* with which the network is configured. In doing so, the EMP server supports both auto-configuration and administrator-based configuration via a user interface, which can be operated either via a web frontend or via the command line. In auto-configuration mode, a network discovery can be performed. In addition, by using topology files, network anomalies can also be detected. Moreover, for the administration of NAM and GCE, a very analogous approach was chosen, in which the said user interface is also employed, while under the hood the then implemented libNAM and libGCE will act as the lower-lever interface to the hardware.

MPI gateway implementation and IP forwarding for inter-module communication. By now, a prototype implementation of the MPI framework, which will be responsible for forwarding of MPI traffic between Cluster (InfiniBand) and ESB (EXTOLL), has been implemented — in accordance with the specifications as detailed in Deliverable D5.2. A load-balancing scheme by means of routing file is applied that ensures a job-exclusive assignment of one or more gateway nodes to each distributed MPI session. In addition, for forwarding other traffic classes than MPI payload, the configuration for IP bridging has been prepared and tested on the Network Federation Gateway Evaluator (NFGW-SDV) and early performance results are presented in this deliverable.

Handling of heterogeneous jobs by the resource manager. For handling heterogeneous jobs via *job packs*, where parts of a session run in different modules, the SPANK plugin of Slurm has been modified to pass the required information to the ParaStation Management framework. This way, it is now possible for the user to specify, for example, the number of desired gateway nodes for the distributed job pack by using additional parameters for the `srun` command. If the desired resources could be allocated, the ParaStation Resource Manager starts one instance of the MPI Gateway Daemon on each assigned gateway node so that MPI traffic can be forwarded between the different modules of the job pack.

Scheduling support for modularity, workflows and shared resources. To further improve modularity support, a first prototype of a job scheduler with a new dependency type has been implemented that adapts the scenario in which an application can run in different modules and the scheduler should choose the module that offers the least slowdown. Initial evaluation results show that this prototype can actually yield improvements in terms of shorter waiting times and less slowdowns. In addition, for the scenario of workflows that move in the form of successive sub-jobs through parts and modules of the system, different solutions were evaluated.

In doing so, in particular a new *delay switch* for Slurm has been proposed and prototypically been implemented, which allows users to provide a certain time value for each sub-job to be delayed in relation to the start of the workflow. Last but not least, means for requesting shared global resources like NAM and GCE via Slurm have been considered and related configuration options and command line parameters have been proposed.

Data visualisation module and a data analytics engine for DCDB. For the Grafana visualisation tool from GrafanaLabs, a data source plug-in has been implemented for interfacing the Data Centre Data Base (DCDB) so that a comprehensive and effective visualisation of sensor data as gained by monitoring the DEEP-EST prototype will be possible. In addition, a data analytics framework has also been developed and integrated with DCDB that allows for asynchronous data analytics of monitoring data. In doing so, both streaming as well as on-demand analysis is enabled in particular with respect to the peculiarities and characteristics HPC environments.

Even though the features described above and their corresponding software products are currently still in a prototype phase, we are confident that we have already reached a level of maturity that will enable their unrestricted employment in the near future. The related complete software stack with the fully implemented set of all required functionalities is then to be presented in the subsequent Deliverable D5.4.

List of Acronyms and Abbreviations

A

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

B

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

C

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

CNN	Convolutional Neural Networks
COTS	Commercial off-the-shelf
CPU	Central Processing Unit
CSR	Control and Status Register
CSIC	Spanish Council for Scientific Research

D

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

E

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

F

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

G

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

H

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

I

IC	Innovative Council
I^{2}C	Inter-Integrated Circuit computer bus
IB	see InfiniBand
IDC	International Data Corporation
InfiniBand	A networking communication standard for HPC clusters
Intel	Intel Germany GmbH, Feldkirchen, Germany

I/O	Input/Output. May describe the respective logical function of a computer system or a certain physical instantiation
IP	Intellectual Property
IPMI	Intelligent Platform Management Interface
iPic3D	Programming code developed by the KULeuven to simulate space weather
ISO	International Organisation for Standardisation

J

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

K

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

L

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

M

Megware	Megware Computer Vertrieb und Service GmbH, Chemnitz, Germany
MHD	Magneto-hydrodynamics
Mont-Blanc	European scalable and power efficient HPC platform based on low-power embedded technology
MoU	Memorandum of Understanding

MPI	Message Passing Interface, API specification typically used in parallel programs that allows processes to communicate with one another by sending and receiving messages
MPICH	MPI implementation maintained by Argonne National Laboratory
MSA	Modular Supercomputer Architecture
MUSIC	Multisimulation Coordinator (MPI-based library for coupled codes)
MQTT	Message Queuing Telemetry Transport (a publisher/subscriber-based messaging protocol)

N

NAM	Network Attached Memory
NCSA	National Centre for Supercomputing Applications, Bulgaria
NEST	Widely-used, publically available simulation software for spiking neural network models developed by NMBU
NF	Network Federation within the DEEP EST prototype
NMBU	Norwegian University of Life Sciences, Norway
NN	Neural Network
NUMA	Non-Uniform Memory Access
NV-DIMM	Non-Volatile Dual In-line Memory Module
NVM	Non-Volatile Memory. Used to describe a physical technology or the use of such technology in a non-block-oriented way in a computer system
NVRAM	Non-Volatile Random-Access Memory

O

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

ORTE Open MPI Runtime Environment (i.e. a Process Manager)

P

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

Q

R

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

S

SCR	Scalable Checkpoint/Restart. A library from LLNL
SDV	Software Development Vehicle: HW systems to develop software in the time frame where the DEEP-EST prototype is not yet available
SIMD	Single Instruction Multiple Data
SIONlib	Parallel I/O library developed by Forschungszentrum Jülich
SKA	Square Kilometer Array
Slurm	Job scheduler that will be used and extended in the DEEP-EST prototype
SME	Small and Medium Enterprises
SNMP	Simple Network Management Protocol
SPANK	Slurm Plug-in Architecture for Node and job (K)control
SRA	Strategic Research Agenda prepared by ETP4HPC
SSSM	Scalable Storage Service Module
STEM	Science, technology, engineering and mathematics
STS	Satellite time series
SW	Software

T

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

U

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

V

W

WLCG	Worldwide LHC Computing Grid
WP	Work package

X

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

Y

Z

Bibliography

[1] *The Scala Language*, [Online], Available: http://www.scala-lang.org

[2] *The High Velocity Web Framework For Java and Scala* [Online], Available: http://playframework.com

[3] *Developing a Linux Kernel Module using GPUDirect RDMA* [Online], retrieved 04.03.2019, Available: https://docs.nvidia.com/cuda/gpudirect-rdma/index.html#pinning-gpu-memory

[4] *DCDB – DataCentre DataBase* [Online], Available: http://gitlab.lrz.de/dcdb

[5] *The Apache Cassandra Database* [Online], Available: http://cassandra.apache.org

[6] *The Eclipse Paho Project* [Online], Available: https://www.eclipse.org/paho/

[7] *Grafana: the Open Platform for Analytics and Monitoring* [Online], Available: https://grafana.com

[8] *InfluxDB, the TIme-Series Database* [Online], Available: https://www.influxdata.com/time-series-platform/influxdb

[9] *The Grafana Simple JSON Data Source Plugin* [Online], Available: https://github.com/grafana/simple-json-datasource

[10] A. Agelastos et al.: *The lightweight distributed metric service: a scalable infrastructure for continuous monitoring of large scale computing systems and applications*. SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2014.

[11] F. Beneventi et al.: *Continuous learning of HPC infrastructure models using big data analytics and in-memory processing tools*. Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017. IEEE, 2017.

[12] M. Massie et al.: *The ganglia distributed monitoring system: design, implementation, and experience*. Parallel Computing 30.7 (2004): 817-840.

[13] N. Eicker et al.: *DEEP-EST Deliverable 5.1: Collection of Software Requirements*, December 2017

[14] N. Eicker et al.: *DEEP-EST Deliverable 5.2: Software Specification*, June 2018

[15] H. Cornelius and A. Auweter: *DEEP-EST Deliverable 4.1: Prototype Hardware Design*, June 2018

[16] M. Nuessle et al.: *DEEP-EST Deliverable 4.3: Network Federation, Fabri^3, NAM and GCE designs*, June 2018

[17] H. C. Hoppe, H. Cornelius et al.: *DEEP-EST Deliverable 3.1: System Architecture*, December 2017

[18] N. Eicker et al.: *DEEP-EST Deliverable 3.2 - Update: High level system design*, January 2019

[19] S. Pickartz, C. Clauss, S. Lankes, S. Krempel, T. Moschny, and Antonello Monti: *Non-Intrusive Migration of MPI Processes in OS-bypass Networks*, IEEE Parallel and Distributed Processing Symposium Workshops (IPDPSW), IPRDM Workshop, 2016, http://dx.doi.org/10.1109/IPDPSW.2016.134

[20] J. Schmidt and Andreas Galonska: *DEEP-ER WP3: Network Attached Memory (NAM)*, 2016

[21] Morris A. Jette, Andy B. Yoo and Mark Grondona: *SLURM: Simple Linux Utility for Resource Management*, in Proceedings of the 9th International Workshop Job Scheduling Strategies for Parallel Processing (JSSPP), Springer, Lecture Notes in Computer Science (LNCS), volume 2862, pages 44-60, http://dx.doi.org/10.1007/10968987_3

[22] Navarro A., Mateo S., Perez J.M., Beltran V., Ayguadé E. (2017) Adaptive and Architecture-Independent Task Granularity for Recursive Applications. In: de Supinski B., Olivier S., Terboven C., Chapman B., Müller M. (eds) Scaling OpenMP for Exascale Performance and Portability. IWOMP 2017. Lecture Notes in Computer Science, vol 10468. Springer, Cham Available: https://link.springer.com/chapter/10.1007/978-3-319-65578-9_12

[23] *Heterogeneous Resources and MPMD*, Presentation at the Slurm 2015 User Group Meeting [Online], Available: https://slurm.schedmd.com/SLUG15/Heterogeneous_Resources_and_MPMD.pdf

[24] *Heterogeneous Job Support in Slurm* [Online], Available: https://slurm.schedmd.com/heterogeneous_jobs.html

[25] *Consumable Resources in Slurm* [Online], Available: https://slurm.schedmd.com/cons_res.html

[26] *srun man page* [Online], Available: https://slurm.schedmd.com/srun.html

[27] *sbatch man page* [Online], Available: https://slurm.schedmd.com/sbatch.html

[28] Hager G., Wellein G., (2011) *Introduction to High Performance Computing for Scientists and Engineers*. Boca Raton: CRC Press, https://doi.org/10.1201/EBK1439811924

[29] *Jobs Submitted to Multiple Partitions, p6* [Online], Available: https://slurm.schedmd.com/SUG14/sched_tutorial.pdf

[30] *The RICC log* [Online], Available: http://www.cs.huji.ac.il/labs/parallel/workload/l_ricc/index.html