H2020-FETHPC-01-2016

DEEP-EST

DEEP Extreme Scale Technologies

Grant Agreement Number: 754304

D5.4

Complete System-SW Implementation

Final

Version:	1.0
Author(s):	N. Eicker (JUELICH), Th. Moschny (ParTec), C. Clauß (ParTec)
Contributor(s):	I. A. Comprès. U. (Intel), A. Jokanovic (BSC), S. Krempel (ParTec), Z. Ul Huda (JUELICH), A. Netti (BAdW-LRZ), M. Nuessle (EXTOLL), M. Ott (BAdW-LRZ), S. Pickartz (ParTec), D. Tafani (BAdW-LRZ)
Date:	31.12.2019

Project and Deliverable Information Sheet

DEEP-EST Project	**Project ref. No.:**	754304
	Project Title:	DEEP Extreme Scale Technologies
	Project Web Site:	http://www.deep-projects.eu/
	Deliverable ID:	D5.4
	Deliverable Nature:	Report
	Deliverable Level: PU*	**Contractual Date of Delivery:** 31.12.2019
		Actual Date of Delivery: 31.12.2019
	EC Project Officer:	Juan Pelegrin

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

Document Control Sheet

Document	**Title:** Complete System-SW Implementation	
	ID: D5.4	
	Version: 1.0	**Status:** Final
	Available at: http://www.deep-projects.eu/	
	Software Tool: LaTeX	
	File(s): DEEP-EST_D5.4_Complete_System-SW_Environment_Impl.pdf	
Authorship	**Written by:**	N. Eicker (JUELICH), Th. Moschny (ParTec), C. Clauß (ParTec)
	Contributors:	I. A. Comprès. U. (Intel), A. Jokanovic (BSC), S. Krempel (ParTec), Z. Ul Huda (JUELICH), A. Netti (BAdW-LRZ), M. Nuessle (EXTOLL), M. Ott (BAdW-LRZ), S. Pickartz (ParTec), D. Tafani (BAdW-LRZ)
	Reviewed by:	S. Schenk (EXTOLL) J. Kreutz (JUELICH)
	Approved by:	BoP/PMT

Document Status Sheet

Version	Date	Status	Comments
1.0	30.12.2019	Final version	

Document Keywords

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

Copyright notice:

©2017-2021 DEEP-EST Consortium Partners. All rights reserved. This document is a project document of the DEEP-EST Project. All contents are reserved by default and may not be disclosed to third parties without written consent of the DEEP-EST partners, except as mandated by the European Commission contract 754304 for reviewing and dissemination purposes.

All trademarks and other rights on third party products mentioned in this document are acknowledged as own by the respective holders.

Table of Contents

List of Figures

List of Tables

Executive Summary

This deliverable presents the complete system software implementation from WP5 comprising the full range of functionalities required for interconnect management, network bridging, resource management, job scheduling and monitoring. All these aspects support the handling of the modularity as well as the special features of the DEEP-EST system. In addition to the goal of providing a complete overview of the implemented system software architecture, this deliverable puts a special focus on the latest developments that have complemented and continued to mature the software stack recently. All these related components and functionalities have already been implemented and are either already available on the DEEP-EST system or can be tested immediately and put into operation once the corresponding hardware is available.

1 Introduction

The operation of a Modular Supercomputer requires a system software stack with new features that add to those used on today's monolithic supercomputers. Deliverable D5.1 [17] presented a detailed analysis of the corresponding requirements to set the stage for all further work in WP5. Furthermore, D5.1 presented a rough design sketch of the software stack that has been updated with more details in Deliverable D5.2 [18]. This update takes the progress on the design of the hardware modules into account that has been made in the meantime. Due to the change of the ESB design upon the review in M12, D5.2 also had to be revised partly to accommodate the new GPGPU-centric programming model for the ESB. Eventually, Deliverable D5.3 [19] gave a first comprehensive overview of the entire software architecture in WP5 and presented the prototype implementations of the related software components and products while discussing their integration status, functionality and capabilities.

This document now presents the complete software implementation in WP5 with the full range of functionality required regarding interconnect management, network bridging, resource management, job scheduling and monitoring as it currently exists and as it is ready for deployment on the DEEP-EST system. In addition to the goal of providing a complete overview of the implemented system architecture, this deliverable puts a special focus on the latest developments that have complemented and continued to mature the software stack since the presentation of the prototype implementation in D5.3. For instance on the EXTOLL side, the Fabri^{3}Manager was added as a new and important low-level software component as well as the support for GPUDirect taking the GPGPU-centric ESB architecture into account. The MPI bridging could be improved by applying different optimizations, e. g., pipelining and the implementation of a rendezvous mechanism that particularly accommodates EXTOLL's RMA semantics significantly increase the achievable throughput performance of the gateways.

Concerning scheduling and resource management, energy awareness has been introduced such that the consumed energy can be reported to the user on job granularity. Additionally, the introduction of a new scheduling policy even enables an energy-oriented adjustment of job priorities. For the scenario of workflows, the delay switch mechanism implemented for Slurm was recently extended such that not only the user but also the application itself can now influence the start time of subsequent workflow steps. In addition, a new Slurm SPANK plugin was developed supporting the management of the NAM as a capacity-limited resource for Burst Buffers. Last but not least, the Data Center Data Base (DCDB) framework and its recently developed data analytics component has matured to a scalable and modular monitoring solution for sensor data.

All these features are now available as a system software stack that has been adapted to the modularity and hardware characteristics of the DEEP-EST system, being ready for deployment.

2 Interconnect Management

The complete interconnect management software suite for Fabri³ can only be finalized when complete, working hardware is available. Since Fabri³ has been delayed, as reported in the project and the M24 review, this hardware is not yet available at the time of writing. Nevertheless, many important steps could already be done. In Deliverables D5.1 [17], D5.2 [18] and D5.3 [19] the requirements, specification and architecture of the software suite were reported. All the major features and components are implemented and can be tested and put to work, once the final hardware is available. In the meantime, with first test hardware available for bring-up and board-test, the low-level Fabri³ components could be tested on real hardware and already finished. This is in an important software component named Fabri³Manager, since it is the interface between the (higher-level) EMP software suite and the real hardware. Details of this component are described in Section 2.1. Also, GPUDirect for RDMA on the EXTOLL network was added as a new feature, when the ESB architecture was changed to incorporate GPGPUs as main compute elements. Implementation of this component has progressed, being tested on the ESP SDV, which features Tourmalet PCIe cards and NVIDIA V100 GPU cards and thus is software compatible on the kernel and user-space side to the final ESB (not on the management side though!). Details of the GPUDirect implementation are reported in Section 2.2.

2.1 Fabri³Manager

Fabri³Manager is a new software component in the EMP software suite bridging Fabri³'s hardware interfaces to the existing EMP-Framework. It is used for initialization, configuration and monitoring of Fabri³. The manager provides a RESTFUL HTTP API implementation to access Fabri³ components remotely. The EMP software suite leverages that API to access information about the Fabri³ and provide it to the user.

A Fabri³ features many components that need to be initialized and configured, e. g. the routing configuration of the EXTOLL Tourmalet chips. All those components are connected to I²C buses, which are connected to one PIC microcontroller. The microcontroller itself is connected to a management controller by USB.

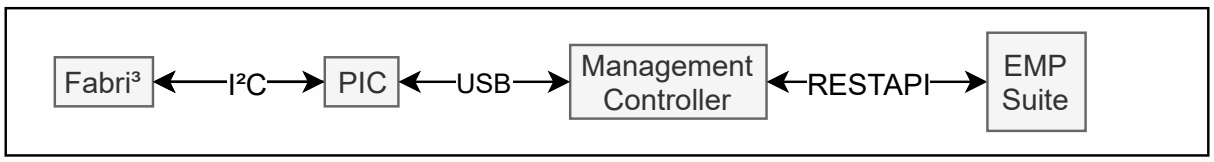


Figure 1: Fabri³ interfaces.

That gives a chain of interfaces: Fabri³ to I²C to PIC to USB and finally to the management controller as depicted in Figure 1. The Fabri³ board hosts a whole hierarchy of I²C buses connecting all active ICs on the board supporting their management and monitoring. Figure 2 shows an overview of the board with its active ICs managed via I²C.

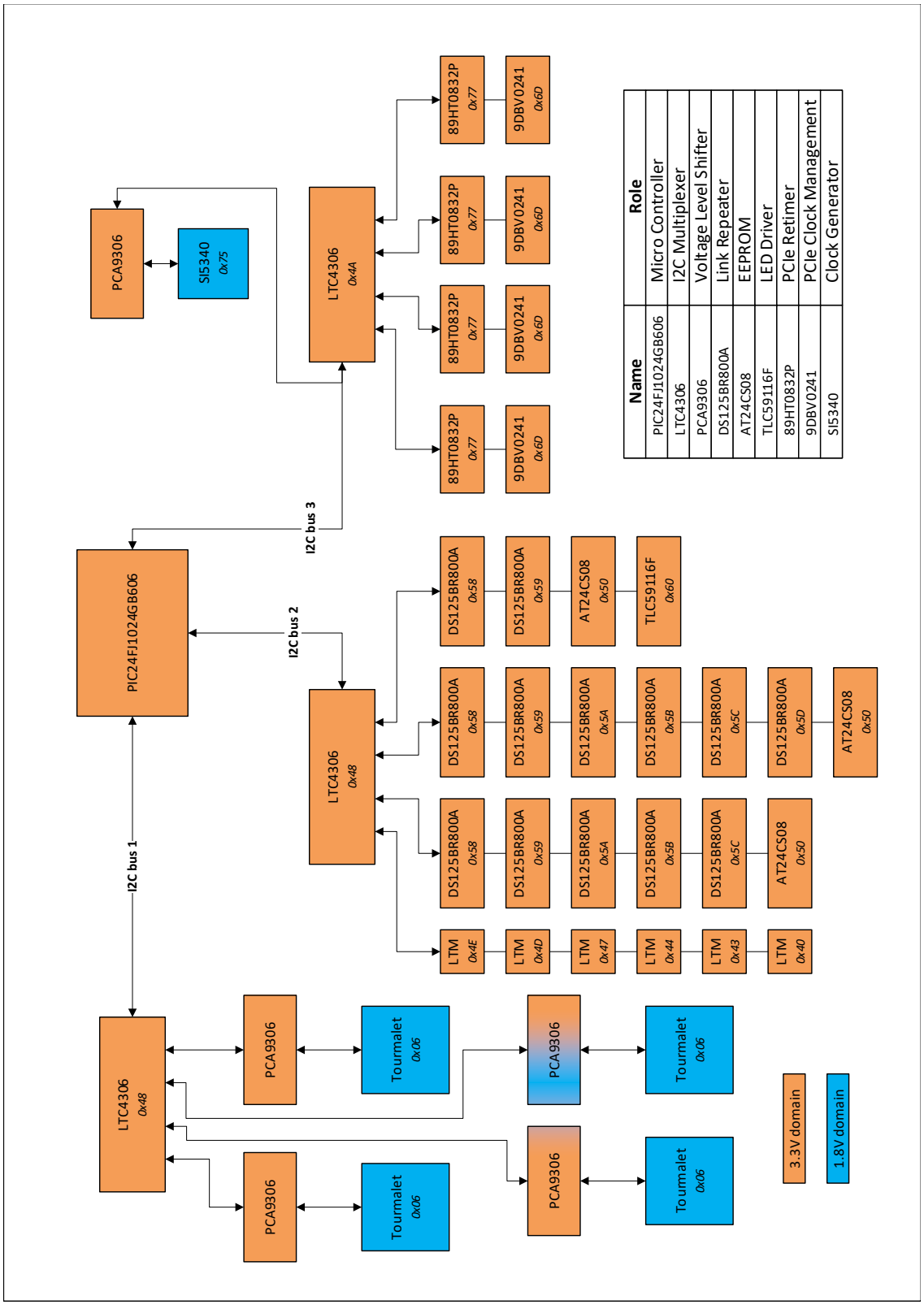


Figure 2: Fabri³ board I²C hierarchy.

Also, reset signals provided by components on the board (especially the Tourmalets) are connected to GPIO pins of the microcontroller. The microcontroller runs a newly implemented firmware, taking commands via USB, generating read/write accesses to the I²C buses and controlling the resets. The commands on the USB follow a custom protocol for the encoding of read/write accesses to the Tourmalets and other components via the SMBus protocol over I²C. The firmware on the microcontroller decodes the USB commands and issues the corresponding commands to the I²C buses, collects and returns results in case of read operations and signals the management controller if a command was successfully executed.

On the management controller runs Fabri³Manager as a server application, written in the Go language. It connects to the microcontroller firmware via libusb, wrapped to be usable in Go with the library gousb. The server takes commands from its HTTP RESTAPI and translates them to commands on USB to drive the microcontroller. Generated answers to read requests or error codes are wrapped in JSON objects and returned by the server.

The RESTAPI itself is decomposed in three main components. The first one controls read/write access to Tourmalets and the reset to the Tourmalets. This also allows to get a list of Tourmalets connected to the PIC with their position on the Fabri³ board. As Tourmalet implements a register addressing scheme incompatible to standard SMBus addressing it needed its own command for access. The second controls read/write access to all other components in the Fabri³ via SMBus. Finally, the third one manages the clock generators reset.

A problem to solve was the addressing of components in Fabri³. A natural addressing scheme would have been to use the I²C slave addresses of each component. But as Fabri³ consists of multiple boards with the same layout this is not sufficient. Also, the three I²C buses of the PIC are multiplexed, such that I²C devices of the same type that are available multiple times can use the same slave address. This leads to a more complex addressing scheme:

- Each Fabri³ board has a unique identifier, provided to the host via USB.
- The Tourmalets have a position on the Fabri³: upper and lower, left and right, which yields four possible combinations: upper left, upper right, lower left and lower right. So Tourmalets are addressed by a combination of board ID and position.
- The remaining SMBus slaves are addressed by the ID of their SMBus and a slave address. So those are addressed by the board ID, the SMBus ID, and their I²C slave address.

The resulting REST API endpoints are described in Table 1.

Some components (especially the Tourmalets) are behind an I²C multiplexer (which is an I²C device configurable via SMBus itself). To access those components, the server has to configure the multiplexer beforehand. The server needs to know the layout of the components on the PCB to configure the multiplexers before a component is accessed.

EMP will later use this RESTAPI to access Fabri³ to configure the fabric and read performance and sensor data.

When starting up Fabri³, the Fabri³Manager takes control of the initialization and configuration of the Fabri³. As soon as the USB device is accessible, the clock generator is the first component for which the reset signal is released. It is then configured to provide the correct clock to the Tourmalet ASICs.

Endpoint	HTTP Method	Function
/boards	GET	Returns a list of connected board IDs
/tourmalet	GET	Returns a list of accessible tourmalet GUIDs
/tourmalet/rf?board=<id>&pos=<pos> &address=<reg>	GET	Read value at register-address from Tourmalet with GUID
/tourmalet/rf?board=<id>&pos=<pos> &address=<reg>&value=<val>	PUT	Write value to register address of Tourmalet with GUID
/tourmalet/reset?board=<id>&pos=<pos>	GET	Returns state of reset for Tourmalet at position or with GUID
/tourmalet/reset?board=<id>&pos=<pos> &value=<val>	PUT	Set reset of Tourmalet at position or with GUID
/smbus?board=<id>&busID=<id>&slave= <slave>&reg=<reg>	GET	Read register value of slave on bus
/smbus?board=<id>&busID=<id>&slave= <slave>&reg=<reg>&value=<val>	PUT	Write register value to slave on bus
/si/reset?board=<id>	GET	Returns state of rest of SI
/si/reset?board=<id>&value=<value>	PUT	Set reset of SI to value

Table 1: Fabri³Manager REST API.

Then the Tourmalets' resets are released and their status and control register files are accessible from outside of Fabri³ via the RESTAPI.

Fabri³Manager is used for the bring-up of Fabri³. The tool TourmaletControl from the EMP software suite was extended in order to access the Tourmalet via the Fabri³Manager RESTAPI. TourmaletControl provides access to the internal status, control and debug registers of the Tourmalet chip. Due to the fact that the EMP master daemon and TourmaletControl use the same hardware abstraction layer to access Tourmalet, the changes in the EMP master daemon were very similar to the changes of TourmaletControl.

2.2 GPUDirect for RDMA on EXTOLL

The EXTOLL GPUDirect kernel module is a Linux kernel module that implements NVIDIA's GPUDirect technology in the framework of the EXTOLL Driver. This enables the user to use GPUDirect with the EXTOLL network. GPUDirect leverages features of the PCIe bus to remove the need for one copy operation when sending data from the GPU memory to the network. Without GPUDirect, the following data movements are necessary to copy data from the GPU memory to the network: copy the data from GPU memory to host memory, then copy the data from host memory to the network. And vice versa for data movement from the network to the GPU memory. With GPUDirect, the need to copy to and from host memory is removed. GPUDirect enabled memory regions can be copied directly from GPU memory to the network as shown in Figure 3.

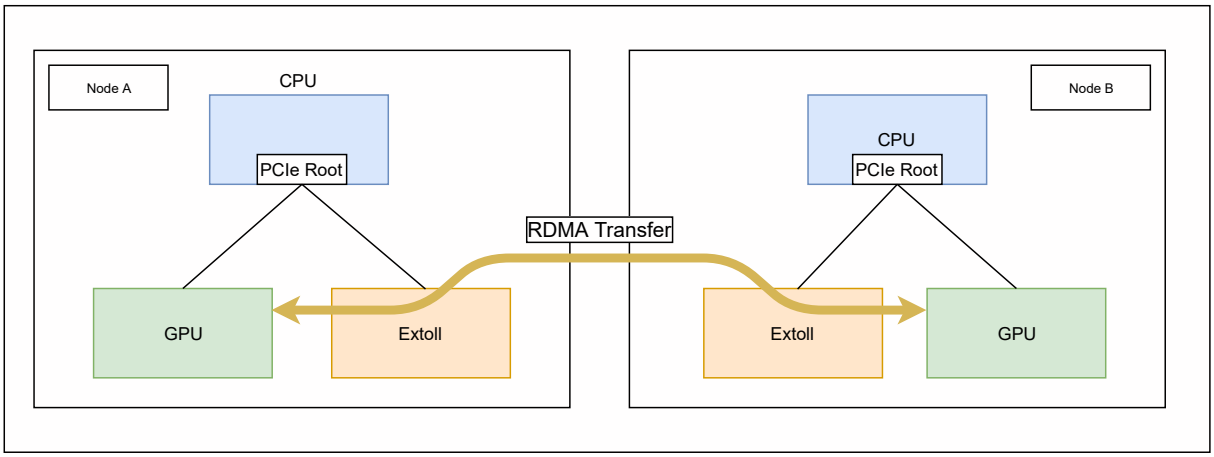


Figure 3: Direct RDMA access to GPU memory from remote GPU

As EXTOLL already uses an Address Translation Unit to not expose physical addresses on one host to other hosts, EXTOLL address translation needs to be extended to be able to get physical addresses from the GPU driver for its RDMA operations. This functionality is implemented in a new kernel module that extends the EXTOLL driver. The address mapping from physical addresses on the PCIe bus to EXTOLL Network Logical Addresses is done by one function in the new kernel module. It is called from user space, which allocated the memory on the GPU, via an IOCTL command. User space provides the logical address and the size of the memory that should be pinned. The kernel module then calls the NVIDIA driver to get a description of the pages behind that address. This description contains a field for a “physical address”, which is an I/O address on the PCIe bus. It is not the physical address that the CPU or another device can use to access the memory region. To make the memory really reachable for third party PCIe devices, it has to be mapped to be accessible via DMA. The EXTOLL GPUDirect kernel module calls again the NVIDIA driver to translate those pages to real physical addresses and expose them via DMA on the PCIe bus. Now the memory region is ready to be read and written via EXTOLL. The finest granularity supported to map are 64 Kib. That means, each address has to be aligned to 64 Kib boundaries and the pinning size has to be a multiple of 64 Kib. The kernel module holds all information needed to later unmap the memory region safely.

To bridge the kernel space code to user space code that allocates the memory for GPUDirect, we extended EXTOLL’s user space libraries to allow registration of memory regions in GPU memory. This new function in the user space library uses the IOCTL to call the kernel function described above.

To test the kernel module, we implemented two user space programs. As NVIDIA’s CUDA environment provides two APIs (the Driver API and the Runtime API), which allow to allocate memory that can be accessed via GPUDirect, those two programs test one API style each. Both programs allocate and initialise GPU memory and register that memory with the EXTOLL user space library as GPUDirect memory. Then they start an EXTOLL transfer from/to that memory using EXTOLL’s RDMA functional units. After the read/write is done, they check if the memory (copied from GPU memory to host memory with CUDA memcopy functions) contains the expected data.

In conclusion, we have shown that RDMA with EXTOLL from and to NVIDIA GPU memory via GPUDirect is possible and working correctly. Nevertheless, both user space library and kernel module are in an early development state and need more testing to mature.

3 Inter-Module Network Bridging

This section deals with the bridging of data traffic across the different modules of the prototype system. In doing so, it describes the final implementation of the bridging framework including a summary of the design decisions that have been taken in the previous deliverables [17, 18, 19]. The discussion divides into two parts: Firstly, the general system architecture is discussed with respect to the DEEP-EST prototype system. The second part discusses the implementation of the MPI gateway. This comprises the presentation of different optimizations techniques improving the cross-gateway performance significantly.

3.1 Architectural Overview

The DEEP-EST prototype exhibits three gateways, i. e., one MPI / IP gateway translating between the InfiniBand the EXTOLL network and two IP bridges realizing the connections for the I/O traffic to the SSSM. Figure 4 visualizes this architecture.

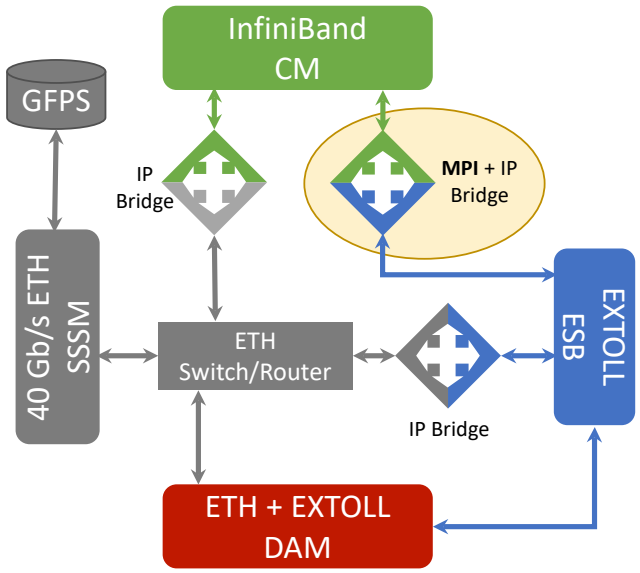


Figure 4: An overview of the bridging architecture within the DEEP-EST prototype system. The MPI gateway provides efficient communication across the three compute modules while the IP bridges support the connectivity to the SSSM.

This architecture features a Network Federation (NF) with dedicated gateway nodes directly bridging between the MSA modules, i. e., the regular compute nodes have to communicate with the accelerator nodes of the ESB and the analytics nodes of the DAM via these gateways. Thus, there is one hop, i. e., a network translation, at maximum when communication occurs within the MSA system. If this was not the case, all MPI would have to be routed through the 40 GE requiring two hops when communicating between the CM and the ESB and / or the DAM module.

3.2 Bridging of MPI Traffic

MPI provides a transparent view onto the underlying network architecture, i. e., the processes are not aware of the actual transport protocols that are used for the communication. These semantics should be preserved and were taken into account during the design of the MPI gateway solution. As a result, the lower layers of the communication stack realise the bridging between the different modules as part of the DEEP-EST MSA prototype system.

Therefore, the ParaStation pscom library has been extended by two components (cf. Figure 5): (1) the pscom gateway daemon (psgwd) running as a mediator between different decoupled networks on the gateway nodes; and (2) the gateway plugin constituting the contact point for ordinary MPI processes for inter-gateway communication. The latter enriches the pscom library by routing capabilities among different network types. These are InfiniBand and EXTOLL in the case of the DEEP-EST prototype. However, the design of the GW plugin is generic enough to translate between *any* pair of communication transports supported by the pscom.

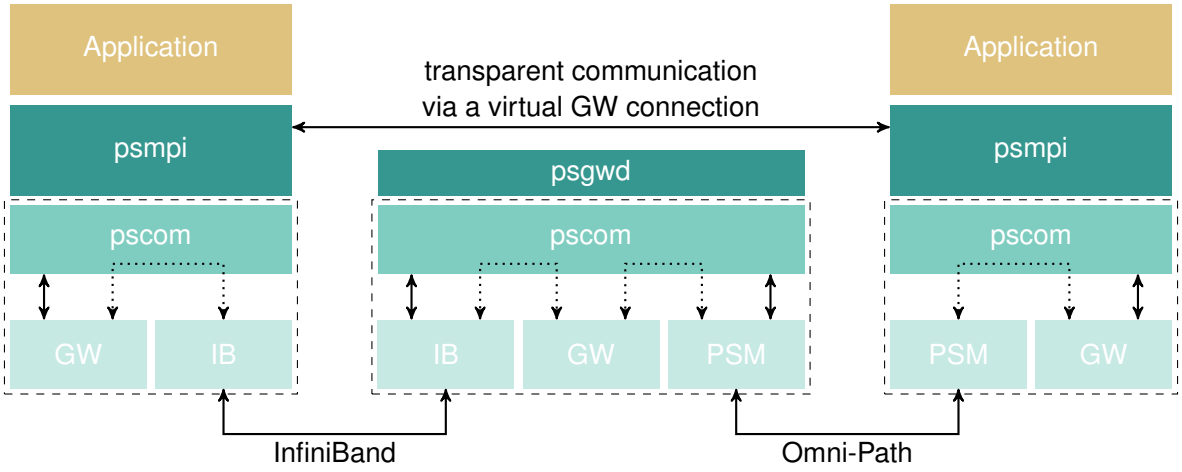


Figure 5: A schematic overview of the MPI bridging solution. This relies on the gateway concept and is implemented on top of the low-level communication interface pscom.

To pass a message from one MPI process to another residing in a different fabric, both use the gateway plugin as the communication endpoint within the pscom. This plugin basically establishes a *virtual* connection between the two processes, i. e., at this stage even the pscom processes are not aware of the network translation. The gateway plugin, in turn, encapsulates the actual MPI message within a so-called *envelope* containing the necessary information for forwarding it to the destination process, e. g., the rank ID and the network address. This envelope is then transmitted to the appropriate psgwd by using that pscom-native communication channel connecting the sending process and the psgwd. As this transmission is completely hidden within the gateway plugin, the resulting translation steps appear completely transparent to the communicating processes. The mapping of virtual gateway connections to underlying pscom-native connections has a beneficial side effect: the same physical connection may be used for multiple gateway connections at the same time, i. e., the psgwd performs an implicit (de-)multiplexing of the gateway network traffic.

The initial implementation of this gateway solution performed the network translation in a *store-and-forward* scheme, i. e., a message was completely transferred to the gateway node before being

forwarded to the destination. Therefore, the psgwd inspects the message envelope not before the complete message is stored at the gateway daemon. Subsequently, the envelope is forwarded on the appropriate connection to the destination node. Here again, the message envelope has to be stored in a temporary message buffer before it can be passed to the application layer. This is because envelope messages are always received as *unexpected* messages whether the matching receive request has already been posted or not.

Figure 6 provides a comparison of the communication performance of this solution for inter-module communication with the native communication performance on the underlying pscom connections. The above-described design disallows rendezvous communication, i. e., the direct storing of messages into the user buffer without intermediate copy, on the gateway path. Especially, this communication protocol significantly improves the communication performance on the EXTOLL link (cf. —●— and —■—). Therefore, a comparison between the gateway performance (—⋆—) and the eager communication via EXTOLL (—●—) supports the assessment of the inter-module communication performance, i. e., the EXTOLL link constitutes the bottleneck on the gateway path. Here, we roughly achieve half of the throughput that is provided by the EXTOLL link. This is expected behaviour due to the store-and-forward scheme at the gateway node.

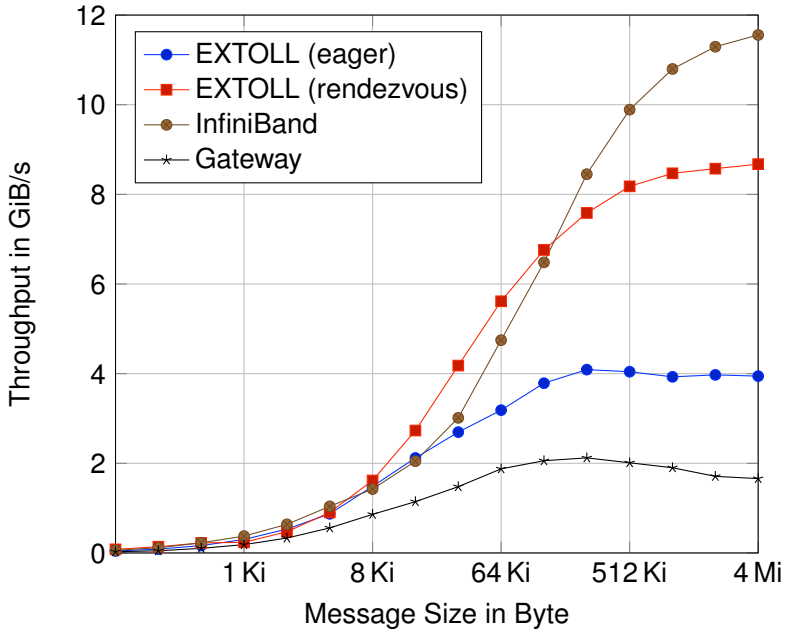


Figure 6: The performance of the first implementation of the MPI gateway solution (—⋆—) as compared with the bare-metal performance on the underlying native pscom connections. The results have been obtained by executing the PingPong benchmark from the Intel MPI benchmark Suite.

3.2.1 Optimization of the Inter-module Communication

Apparently, the gateway approach described above meets the requirement of a transparent inter-module communication within the DEEP-EST prototype system. However, the store-and-forward

scheme and the eager receipt of messages at the destination process add an overhead that can be reduced when applying different techniques improving the communication performance. There are basically two main aspects to be considered: (1) the store-and-forward scheme results in a serialisation of the inter-module traffic and (2) the receiving of message envelopes as unexpected messages adds additional memory copy operations that may be avoided. The following sections discuss according optimization techniques and present performance figures estimating their respective impact.

Pipelined Message Transfer

The serialisation of inter-module traffic can be avoided by fragmenting larger messages into smaller chunks. This approach results in a pipelining effect: while receiving message fragments on one link of the gateway connection, the psgwd may already forward fragments that have been received previously via the other communication link. Ideally, this approach results in a perfect overlapping of sending and receiving of message fragments on both links of the gateway connections. However, a decreasing fragment size increases the relative overhead due to fixed processing times that are independent of the transfer size.

The fragmentation size is a fixed quantity across all gateway connections. However, it can be influenced by setting the `PSP_GW_MTU` environment variable. The gateway plugin transmits messages in chunks matching the value of `PSP_GW_MTU` over the underlying pscom connections. This fragmentation is actually transparent to the psgwd running on the gateway node simply forwarding the chunks to the respective destination process.

Figure 7 shows the impact of the message fragmentation as described above on the PingPong performance. The results were obtained on the DEEP-EST NFGW SDV, i. e., one node as part of an InfiniBand fabric and one node as part of an EXTOLL fabric connected through a single gateway node. Other optimizations such as the rendezvous protocol have been disabled to eliminate side effects.

The message fragmentation significantly improves the communication throughput from around 2.1 GiB/s (—●—) to more than 3.1 GiB/s, i. e., an improvement of almost 50 %. The optimal fragment size for this hardware configuration is 16 KiB showing a good trade-off between the latency of mid-size message transfers and the maximum throughput. This is probably a result of the L1 cache size of 32 KiB within each of the three computing nodes. Once the fragment size reaches the cache boundary, the likelihood for cache evictions increases, i. e., we see increasing L1 cache-miss rates when comparing 16 KiB and 32 KiB fragments.

Reduction of Additional Memory Copies

As discussed at the beginning of Section 3.2, there are different places where intermediate copies occur on the gateway path. On the one hand, the psgwd has to copy the message (or the fragments) from the input buffers to the output buffers on the forward connection. On the other hand, the destination process receives envelope messages always as unexpected messages resulting in an additional copy operation.

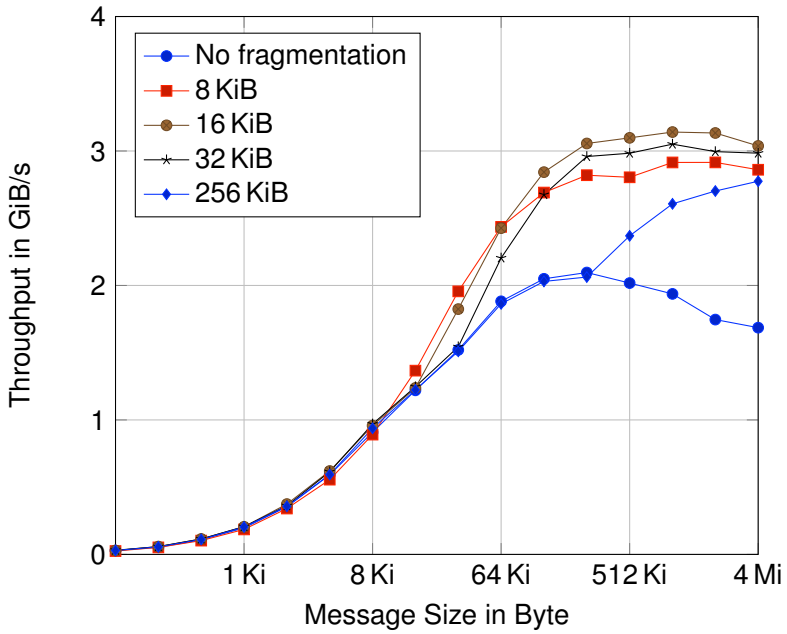


Figure 7: PingPong performance when using different fragment sizes ranging from 8 KiB to 256 KiB. Other optimizations were disabled to eliminate their impact on the results.

The first aspect is inherent to the proposed gateway solution as long as pre-allocated communication buffers are used for the message transfer. A direct re-utilisation of these buffers for message forwarding would likely result in deadlocks. This is due to the multiplexing of different virtual gateway connections onto the underlying pscom connections for an efficient utilisation of hardware resources. In contrast, the second aspect can be avoided which is actually two-fold: (1) the gateway plugin has to be capable of leveraging the pscom-internal rendezvous protocol for the transfer of message envelopes; and (2) the message envelope should not be forwarded to the destination node before the matching receive request has been posted.[1]

Therefore, the gateway protocol has been enhanced by the support for rendezvous communication on the underlying pscom connections (cf. Fig. 8). Once the message or fragment size exceed a certain threshold, the sender only informs the psgwd about the new message. This may then directly receive the data into a temporary buffer, e. g., via an efficient RMA operation. The same pattern applies to the second hop on the gateway path. In doing so, the intermediate copy operations at the gateway node can be reduced to a minimum avoiding the copying from and to the pre-allocated communication buffers. Furthermore, the rendezvous semantics support the delay of the final transfer from the psgwd to the destination process until the matching receive request has been posted.

At this point it should be noted that the delay of the message transfer *to* the psgwd is not necessary as the intermediate buffering happens in any case. Furthermore, the matching receive request will be eventually posted as required by the MPI standard. Therefore, it might be even beneficial to transmit the message to the psgwd as fast as possible to allow a re-usage of the user buffer by the application layer depending on the semantics of the respective MPI call.

[1] Of course, this is highly dependent on the size of the message envelope.

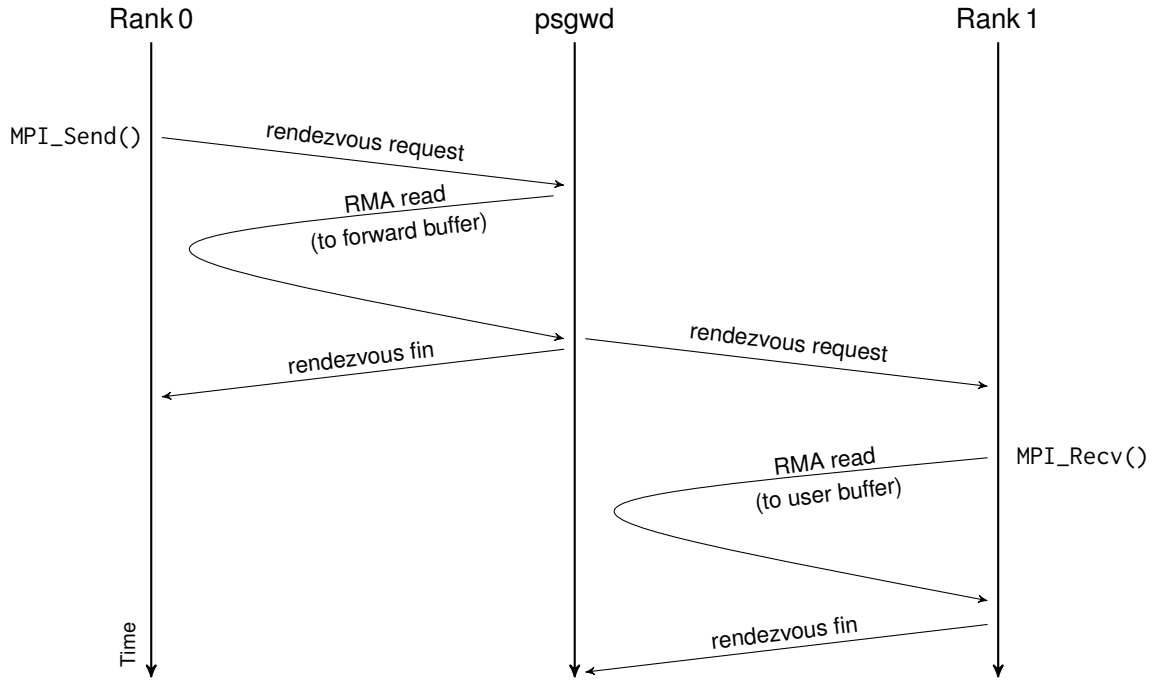


Figure 8: A time sequence diagram illustrating the message flow during a message exchange over a gateway connection using the rendezvous protocol. The final data transfer is delayed until the matching `MPI_Recv()` is posted on the application layer.

Figure 9 shows the impact of the above-described optimizations on the PingPong performance. The rendezvous semantics (—■—) significantly improves the throughput already for mid-size messages with a size of 16 KiB and above. However, the throughput drops for both eager and rendezvous semantics in case of larger messages if these are buffered at the final destination. This most probably stems from L2 cache misses, i. e., the systems exhibit 1 MiB L2 cache and once the message size exceeds half the cache size of 512 KiB, an increasing amount of cache misses is likely to occur.

If the final copy operation is omitted (cf. the dashed curves in Fig. 9), the performance drop can be avoided for both the eager and the rendezvous semantics. The latter even profits in case of small to mid-size messages from this optimization. Finally, the gateway performance including all optimizations, i. e., message fragmentation, rendezvous semantics and the reduction of intermediate copies, has been compared with the pscom-native performance on the individual links of the gateway path (cf. Fig. 10). The comparison considers both the PingPong performance and the network throughput which can be estimated using the *Uniband* benchmark of the Intel MPI Benchmark suite.

The results indicate that we achieve around 74 % and up to 91 % of the pscom-native PingPong performance and network throughput respectively. This difference can be explained by the fact that the Uniband benchmarks profits from an additional pipelining effect due to the rendezvous communication, i. e., multiple send requests from the application layer can be processed concurrently. In contrast, the PingPong communication pattern results in an implicit serialisation of the MPI messages as a further send request is not posted until the corresponding answer arrived at the MPI layer.

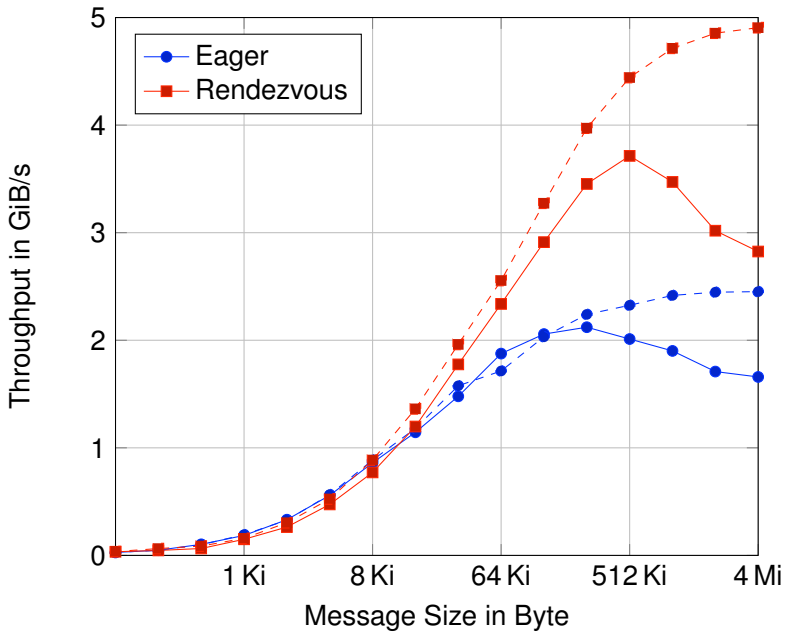


Figure 9: The impact of enabling rendezvous semantics on the gateway communication throughput. The dashed curves show the performance figures when avoiding the final copy operation at the destination process.

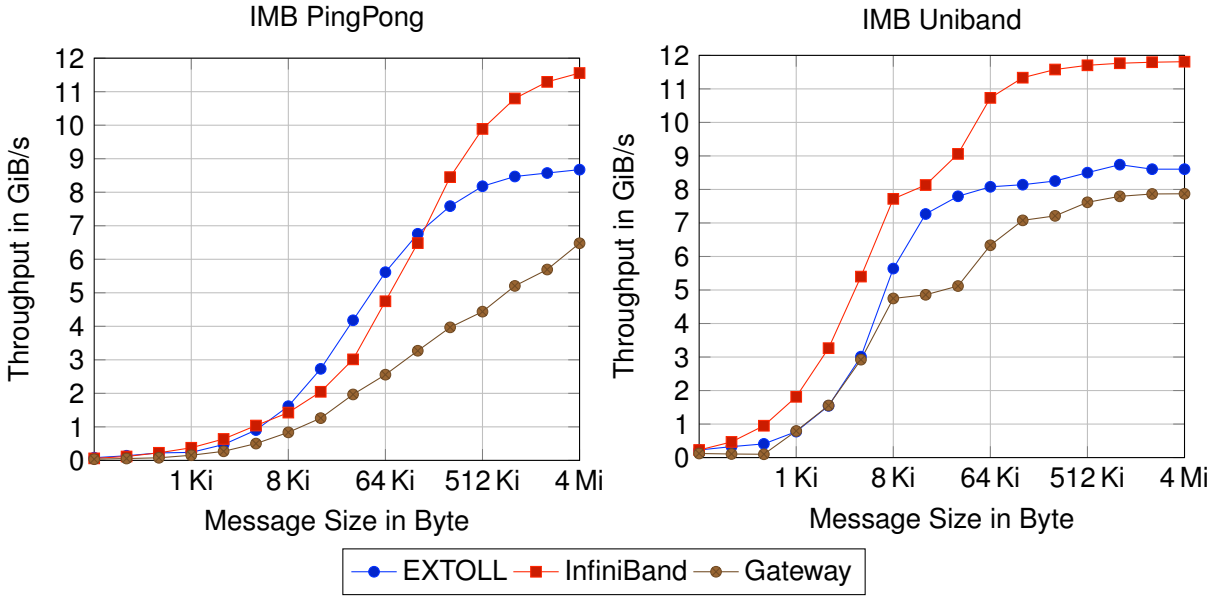


Figure 10: A final comparison of the gateway performance with enabled optimizations for PingPong communication (left) and network throughput (right).

4 Resource Management

4.1 Handling of Heterogeneous Resources

4.1.1 Summary of the Requirements

As briefly described in D5.1 [17] and revisited in D5.2 [18] most applications will go more and more into using different types of accelerators for different tasks inside their work-flow and thus become more and more heterogeneous.The Resource Management System needs to be capable of managing 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 accelerator nodes (called Booster nodes) and data analytics nodes of the DAM are connected via gateway nodes to the regular compute nodes (called Cluster nodes) (cf. Section 3.1). 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.

4.1.2 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. This feature supports the execution of jobs following the Multiple-Program Multiple-Data (MPMD) programming model. As part of this project, the ParaStation Management Daemon's plugin `psslurm` has been extended to provide full support for this Slurm feature.

Currently, Slurm does not provide any support for the allocation of dynamically determined resources such as the gateway nodes. This comprises the startup of additional daemons as the required gateway daemon on these nodes. Therefore, the resource management has to be extended by according functionality. For this purpose, we implemented a new plugin to the ParaStation Management Daemon called `psgw` utilising several existing capabilities. In the long run it would be desirable to bring at least parts of the functionality into Slurm itself.

Usage

Whenever a user wants to start a heterogeneous job using cluster nodes and booster nodes that requires inter-module communication, an additional mechanism is required for the allocation of gateway nodes. These gateway nodes are needed for the connection of processes running on the cluster nodes with processes running on the booster nodes (cf. Section 3.1). We provide according support by means of a new Slurm SPANK plugin, adding some options to the frontend commands `salloc`, `sbatch` and `srun`. The logic of this SPANK plugin is kept rather simple. It mainly sets a couple of environment variables leaving the smart part to the ParaStation Management Daemon.

The user may influence its behaviour by using several options provided by the plugin via the frontend commands:

`--gw_num=number`	Number of gateway nodes
`--gw_env=string`	Additional gateway environment variables
`--gw_file=path`	Path to the gateway routing file
`--gw_plugin=string`	Name of the route plugin
`--gw_cleanup`	Automatically cleanup the route file
`--gw_quiet`	Suppress reporting gateway startup errors in file
`--gw_psgwd_per_node=n`	Number of psgwd per gateway to start
`--gw_binary=path`	Debug psgwd

The only mandatory option is `--gw_num` to request gateway nodes for a pack job, i. e., the following command is sufficient to start a simple interactive pack job using two gateway nodes:

```
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 is then used by the MPI 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. For load balancing reasons, users can request to get more than one psgwd started per gateway node by specifying `--gw_psgwd_per_node`. The remaining options `--gw_env` and `--gw_binary` are for debugging purposes.

Background

Currently, Slurm 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. For more details, see D5.3 [19].

4.1.3 Process Manager

Once the scheduler decides to run a heterogeneous job as described in the previous section, the process manager has to setup the infrastructure by starting the required gateway daemons. Subsequently, it starts the processes on both the compute nodes and the booster nodes and provides the necessary information for inter-module communication.

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 4.1.2). 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.

Technical Details

Compute Nodes

Apart from the pelogue plugin managing the global prologue phase of a pack job, the new plugin psgw for the ParaStation Management Daemon psid is required on the compute nodes. This plugin only takes action if a node becomes the mother superior of the pack job. During the prologue phase it allocates the required number of gateway nodes, triggers the prologue script on the allocated gateway nodes and then starts the ParaStation Gateway Daemons on the respective 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. The pelogue plugin is required for the execution of a separate prologue on the gateway nodes. This is necessary as the gateway nodes are not part of the pack job from the Slurm and psslurm perspective, i. e., 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. The psexec plugin, providing support to start processes remotely, 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 the startup, a script called `psgw_error` is called on the Slurm head node. This is triggered by the `psgw` plugin on the pack job mother superior by using the `psexec` plugin. The current purpose of the `psgw_error` script is the re-queueing of batch jobs while setting the eligible time. This is necessary since the Slurm scheduler is not aware of the gateway nodes and therefore 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.

4.2 GPU Request Support

ParaStation Management now supports Slurm 19.05 which added user options for requesting GPUs via the new concept of Trackable Resources (TRES). Users can request GPUs per Job, per Node, per Socket or per Task:

`-G, --gpus=n`	Number of GPUs required for the job
`--gpu-bind=...`	Task-to-GPU binding options
`--gpu-freq=...`	Frequency and voltage of GPUs
`--gpus-per-node=n`	Number of GPUs required per allocated node
`--gpus-per-socket=n`	Number of GPUs required per allocated socket
`--gpus-per-task=n`	Number of GPUs required per spawned task
`--mem-per-gpu=n`	Real memory required per allocated GPU

All these options are only meant to select nodes for the job, not to reconfigure the nodes.

Limitation

Currently, Slurm comes with the limitation that most of these options are not working for jobs as part of a job pack. Therefore, `--gres=gpu:n` has to be used to request GPUs for jobs inside job packs. Hopefully, this will be fixed in an upcoming release.

4.3 Energy Monitoring

ParaStation Management is enabled to collect the energy consumption data provided by the Megware Hardware (Energy and Power). The data is collected by polling on each node and transferred to the mother superior node which aggregates and reports it to the Slurm Control Daemon. The user then can inspect the data with the usual Slurm tools such as `sacct`.

4.4 Global Resource Management (NAM)

As discussed in D5.2 [18] one can imagine multiple use cases for the Network Attached Storage provided by the EXTOLL network in this project. From the resource management's point of view we have to distinguish two types:

1. NAM allocations persistent over job boundaries,
2. NAM allocations valid only inside the scope of a job or job pack.

Both types can be provided utilizing the Burst Buffer mechanism supported by Slurm together with the new plugin described in detail in Section 5.4. Any necessary changes, especially in the context of the ParaStation Resource Management, including its Slurm integration psslurm, will be cared for in Task 4, if needed.

5 Job Scheduler

5.1 Efficient Job Scheduling Policy

Based on the Deliverable D1.3 [16] and a detailed survey conducted with WP1 developers we have found out the following:

- Developers tune their applications for the best performance on the specific module or a combination of modules.
- The majority of applications may run on an alternative module with suboptimal performance, i.e., longer time between start and end of the job or/and requiring additional computing resources. They may run on different modules directly, or there exists a different application's executable for preferred and alternative modules. We consider the former type of applications.

The user of a supercomputer needs shorter time-to-result, i.e., a shorter time between submission and end of the job. The flexibility of having preferred and alternative types of resources gives more options to the scheduler when deciding when and where to allocate the application. This option might help the application to start and thus to complete sooner, leading to better time-to-result than in the scenario when resources from strictly one module are requested (Figs. 11 and 12).

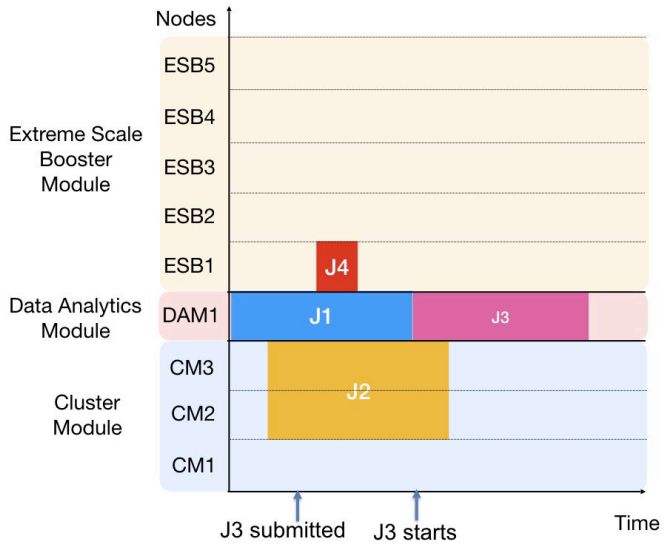


Figure 11: Illustration of load imbalance across modules. The DAM module is not available when *Job3* arrives at the system; thus, *Job3* has to wait in the queue.

We present an implementation of a job scheduling policy in which the user is allowed to specify several modules to be considered by the job scheduler for the execution of the user's application.

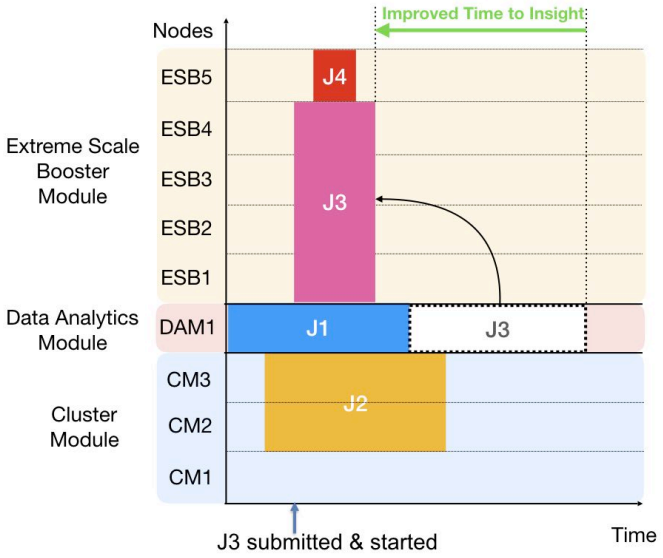


Figure 12: Illustration of an approach to balance the load across modules. Allowing *Job3* to run on a different type of resources, i.e., at ESB module, it starts execution immediately. The amount of resources and the time required has changed. Overall, response time is better than in the previous case (Fig. 11).

5.1.1 Background

Jobs that can provide optional resource requirements at submission time and let the job scheduler decide which one provides better response time are already known as *moldable* jobs [15]. Here we extend the concept of moldable jobs for heterogeneous systems.

Requesting resources from one of the multiple system modules is already supported in Slurm's current version through the mechanism of partitions [10], i.e., specifying a partition list, but it is limited to homogeneous systems. Namely, the requested amount of resources and time is the same for any of the considered partitions. This has influenced the design of Slurm in such a way that the data structures used for that purpose cannot easily be adapted for the context of heterogeneous systems.

5.1.2 Slurm implementation

Since adapting Slurm's data structures for the definition of user requests per module was too costly to enable the job flexibility in modular systems, we have extended the job scheduler as follows (Fig. 13):

- The user can specify the list of modules in the order of preference, adding a new `sbatch` option `module-list`. Then `sbatch` submits automatically one job to each of the modules specified in the module list.
- We have implemented a model that converts the time and resource requirements from one module to another. It is a rather simplistic model that takes into account CPU frequency,

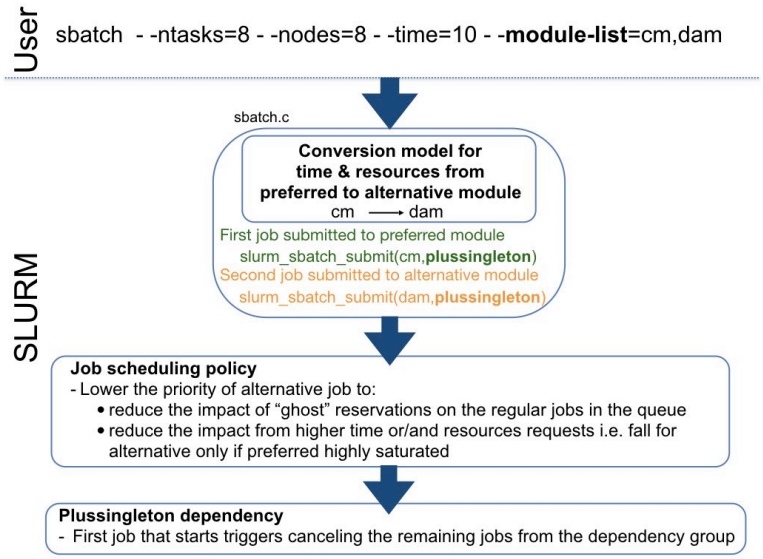


Figure 13: Slurm extensions for supporting module-list policy.

node's memory, and CPU vs. GPU performance factor. This information is used to create the description for each of the module-list triggered alternative job submissions. According to a WP1 developers' survey we can expect the following (preferred, alternative) modules combinations: (CM, DAM), (ESB, CM), (DAM, CM), (DAM, ESB).

- We consider several priority schemes for alternative jobs, as shown in Figure 14.
- Finally, we have implemented a new dependency option *plussingleton*, which will be assigned to each of these jobs. The idea is that as soon as one job from the dependency group starts, all the rest are cancelled.

5.1.3 Preliminary evaluation

Experiments are conducted in the Slurm simulator [14], configured for the DEEP-EST prototype machine size. The input trace is created using the initial workload model proposed within WP2. It consists of 5000 jobs. The amount of flexible jobs is based on the survey of WP1 applications' developers. To emulate the imbalance across modules we have created the simulator's input workload traces for each module using a different load factor. To be able to do that, we have added the load factor to the base workload model. This factor changes the inter-arrival time between jobs and thus changes the module's load.

Implementation validation

Here we evaluate whether the approach of creating additional jobs in the queue has an impact on job scheduling. We implement a dummy version of the *module-list* policy. Namely, the conversion model does not change the time and resources for alternative modules, i.e., we emulate a monolithic

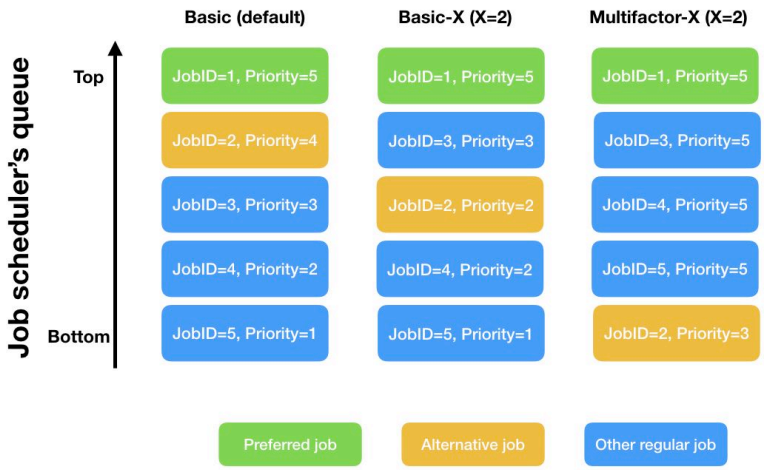


Figure 14: Priority schemes exploited by the module-list policy.

system and compare it to Slurm's *partition list* mechanism. The sizes of modules are the same as the sizes of partitions, and we use the same workload in both scenarios. We conclude that the impact of additional jobs on the system's job scheduling is negligible as the maximum difference among system metric is less than 1.80 % (cf. Fig. 15).

Policy efficiency evaluation

For the balanced system better average response time is achieved when the used priority scheme is the one that moves alternative jobs deeper in the queue. Thus, Multifactor-X gives better response time than Basic and Basic-X, as shown in Figure 16a. This happens due to less need for alternative jobs in balanced systems, thus having them high in the queue makes the scheduler consider them for execution, creating "ghost" reservations which may delay regular jobs' start time. Moving alternative jobs behind preferred and regular jobs in the queue, in the case of Multifactor-X, yields an improvement of around 10 %. On contrary, in the imbalanced systems the priority schemes that keep alternative jobs higher in the queue lead to improved average response time. In particular, the Basic scheme leads to improvement of up to 55 % in response time in the case of highly imbalanced systems. Figure 16b shows the relative average response time of jobs separated per preferred module. For example, *dam_mf* shows the improvement in average response time of jobs that asked for DAM as preferred module, but might have been executed either on DAM or an alternative module. When setting up moderate and high imbalance in the system, DAM had the highest load, followed by CM and finally ESB. Therefore, the improvement achieved by module-list policy is highest for DAM-preferred jobs in imbalanced systems. Additionally, the WP2 workload model considers both (DAM, CM) and (DAM, ESB) combinations, while CM jobs have as an alternative only DAM module, which for bigger jobs is actually not an alternative. Therefore, we see the worst average response time for CM-preferred jobs.

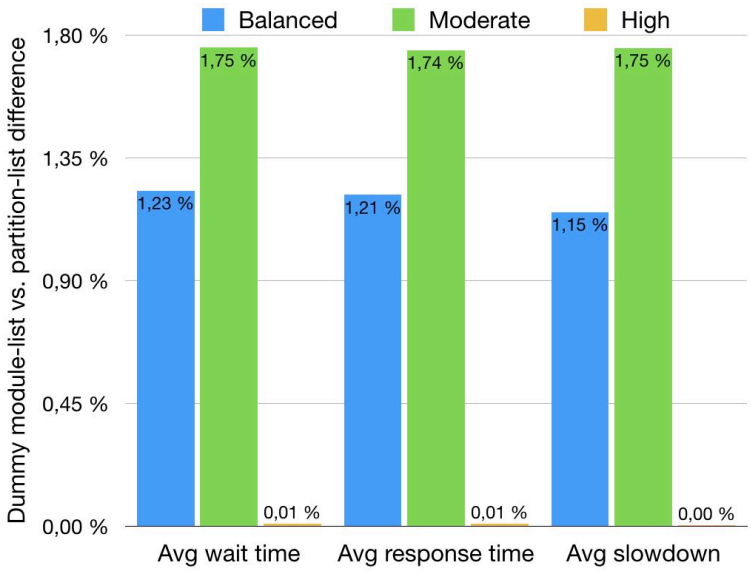


Figure 15: Module-list vs. partition list. Implementation validation.

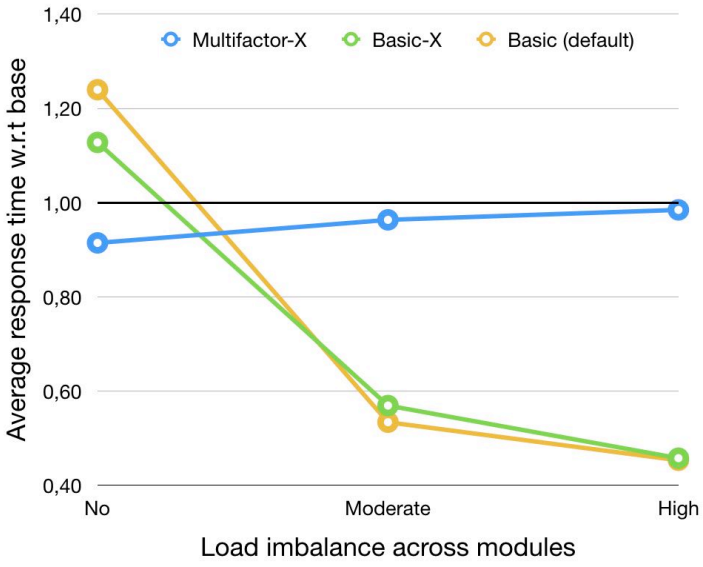
Figure 17a shows that the average slowdown, i.e., response time w.r.t. job duration, and the total completion time of the workload in imbalanced systems drops with the increasing number of flexible jobs in the workload. *WP1 flex* is the workload generated using the WP2 workload model, while *Base* is the workload where jobs request strictly one module, i.e., there are no flexible jobs. The improvement in slowdown is up to 47 %, while completion time is improved by 43 %. In the balanced system using Multifactor-X scheme for alternative jobs (Fig. 17b) the improvement in slowdown and completion time goes up to 14 % and 5.4 %, respectively.

5.1.4 Conclusions

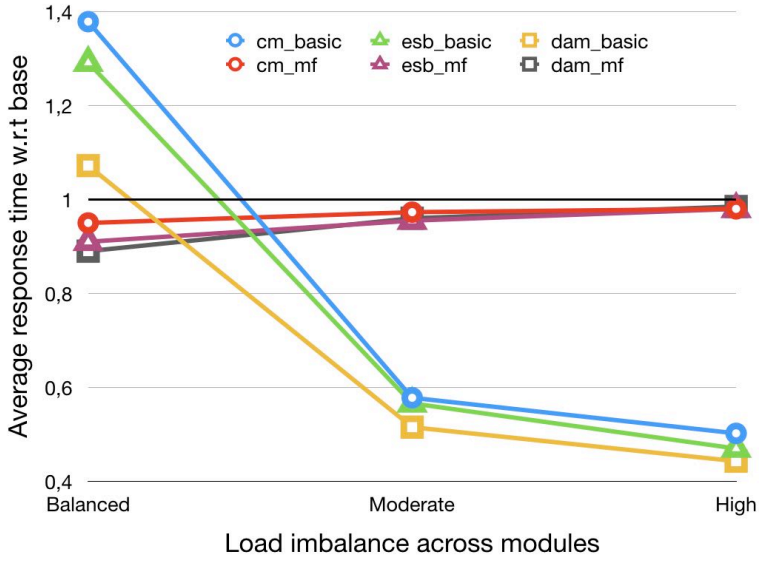
Despite more resources or/and execution time on alternative modules, computational scientists may obtain their results sooner if they allow their jobs being flexible on heterogeneous supercomputers. The relative gain in terms of response time depends on how imbalanced the load of different modules is and which priority scheme is used for alternative jobs.

5.1.5 Forthcoming Work

We plan to improve the conversion model based on the performance measurements of the applications on different modules once we get these data from WP1 developers. Another option is to allow users to specify the time and resource list together with module-list as an *sbatch* option. Further, we will explore a response-time driven policy to be implemented in the *backfill* scheduler as an alternative to priority schemes.

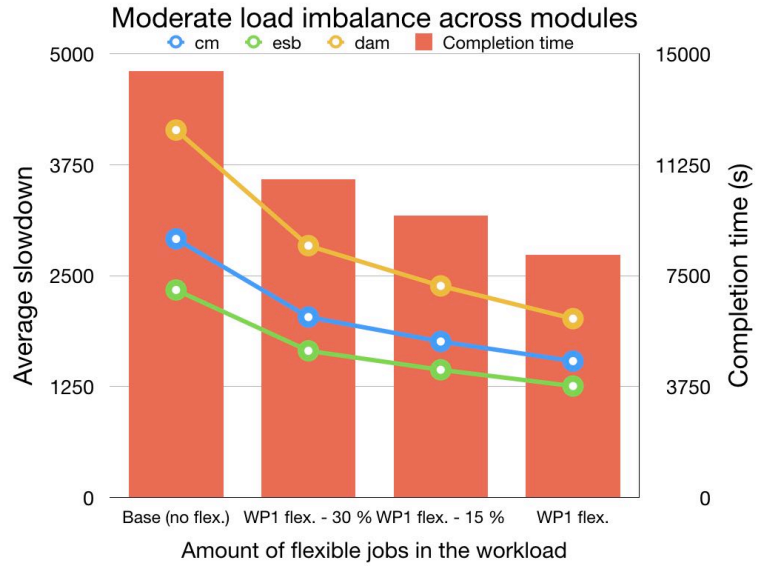


(a) System view

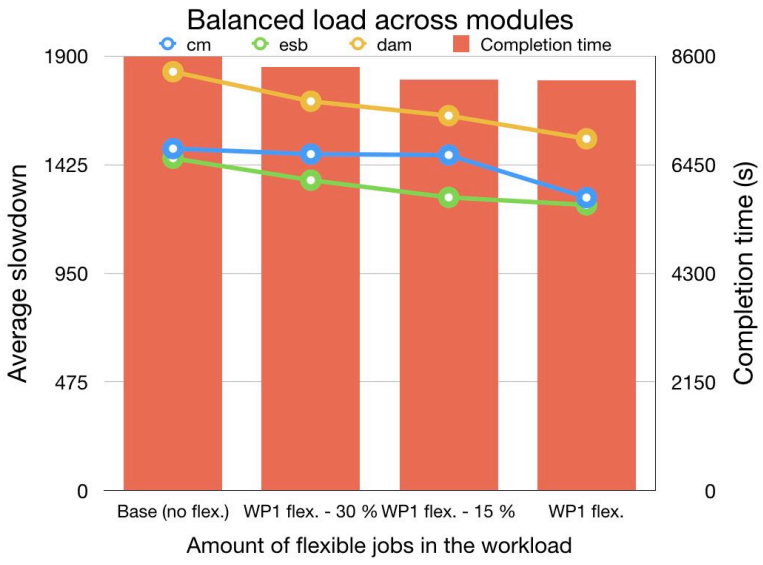


(b) Per-module view

Figure 16: Average response time for different priority schemes applied to alternative job.



(a) The load imbalance across modules is moderate. Basic priority scheme is used for alternative jobs.



(b) The load is balanced across modules. Multifactor priority scheme is used for alternative jobs.

Figure 17: Average jobs' slowdown per module and the workload completion time.

5.2 Energy-aware job scheduling policy

5.2.1 Energy models

Different energy models are proposed in the literature. Some of them are general, based on the average behaviour of applications, other are more specific, e.g. they use applications metrics as an input for the prediction of the energy consumption. Those models are generally more precise, but they need more data from applications. An energy model is trained by running a set of benchmarks on a specific hardware and applying DVFS (Dynamic Voltage and Frequency Scaling), and then extrapolating a mathematical model that is able to represent the behaviour of the benchmarks at different voltages and frequencies.

5.2.2 Energy library

We used an interface for the resource manager, developed by LRZ, to be able to calculate energy estimations for each job. The interface accepts two input files:

- Machine file: contains information about modules (partitions) and models for a machine in Protocol buffers format.
- Application file: contains information about applications.

5.2.3 Energy policy

We integrated energy models and the energy library into Slurm, to make it energy aware. By combining energy predictions with the priority plugin we are able to prioritize jobs with higher hardware efficiency, automatically apply optimal DVFS and, in case the job can run on multiple partitions, give higher priority to the optimal hardware, while keeping the alternative hardware to improve hardware utilization. We select optimal DVFS by calculating a metric obtained as a linear combination of the predicted values of energy, runtime and power. We use the same value to prioritize hardware. In the future, different metrics and policies can be evaluated on the top of this proposal, e.g. by altering DVFS at runtime, based on hardware, software or cost limits, i.e. power capping, dynamic energy policies, electricity cost, etc. Runtime is calculated on the user specified requested time, which in most cases is not representative of the actual job's runtime. Job's metrics are specified by using a tag specified by the user, which could be easily manipulated. Future work will include dynamic metrics collected at runtime as an input for the energy prediction model, to verify the veracity of the specified tag.

5.2.4 Slurm implementation

We implemented the energy policy in Slurm simulator, by mapping Slurm partitions to different clusters, with the constraint that each partition contains homogeneous hardware, and we edited the multifactor priority plugin to use the energy and runtime predictions library.

We implement this policy by editing the priority/multifactor plugin, but the same could be done with priority/basic. It works with the job_submit/all_partitions plugin, that automatically submits the job to all partitions, if the user did not specify one, or with user specified partitions by using the `--partitions` parameter with a comma-separated list of partitions. Single partition jobs are also supported, but for them only the best DVFS is calculated, while priority is not altered.

At job arrival, we use job energy, runtime and power projections to calculate optimal DVFS. In the first implementation we used one of the three policies:

- MINIMIZE_ENERGY_POLICY: sets DVFS to minimal energy per job
- MINIMIZE_POWER_POLICY: sets DVFS to minimal power per job
- MINIMIZE_RUNTIME_POLICY: sets DVFS to minimal runtime per job

The second implementation uses a combination of all predictions using the following equation, where weights are configurable by system administrators:

$$\begin{aligned} \text{energy\_metric} &= \text{energy\_prediction} \times \text{ENERGY\_WEIGHT} \\ &+ \text{time\_prediction} \times \text{TIME\_WEIGHT} \\ &+ \text{power\_prediction} \times \text{POWER\_WEIGHT} \end{aligned}$$

By calculating energy_metric for the different requested partitions, we are able to sort the hardware based on the efficiency with this specific job. We incrementally assign a value to the priority based on the position in the sorted list, from the least to the most efficient. The scheduler will try to run jobs in the order of priority, so in case of free resources in both partitions, the scheduler will prefer the most efficient one. If the preferred partition is full and the job cannot start, depending on the position of the second partition job in the job queue different scenarios could happen:

1. The less efficient jobs will be scheduled just after the more efficient one, and before other jobs in the queue that arrived later. This behavior can be obtained by using the priority/multifactor plugin configured with high enough Age Priority, or by using the priority/basic plugin with incremental priority greater than one.
2. Less efficient jobs might not be considered just after the most efficient ones, because they might be overtaken by other efficient jobs with higher priority, even if they arrived later. This behavior is obtained by configuring the priority/multifactor plugin with low enough Age Priority, or using the standard priority/basic plugin.
3. All less efficient jobs are scheduled after the most efficient ones. This is similar to having two or more queues in which the first queue, with higher priority, contains the most efficient job requests, and the other queues the less efficient ones. This behavior can be obtained by using the priority/multifactor plugin configured with Age Priority = 0, or disabling the plugin's decay thread.

We needed to modify backfill and FIFO schedulers to select the proper time limit for the job for each requested partition. The implementation was integrated in the Slurm Simulator. The simulator needed some changes to be able to run the algorithm. Later the algorithm was integrated with the module-list feature and msa-backfill, developed for the DEEP-EST project.

Conclusions and Pending work

So far we were able to implement the energy-aware policy in the Slurm simulator and test the proof of concept using the EAR model. This model describes the real hardware, or variants of it, but it is not representative of any of the DEEP-EST modules. The code has been ported to Slurm. Interaction with Slurm developers and extensive tests are needed. There were no significant changes from the Slurm Simulator to Slurm code. To be able to run the code on DEEP-EST machines, energy models for the different modules are needed from WP2, together with applications and applications' performance instrumentation to extract applications' counters used by energy models. As soon as the required parts are available extensive tests on DEEP-EST machines and in simulations can be performed.

5.3 Efficient support for coupled workflows using DEEP-EST features

Deliverable D5.3 [19] explained in detail the usefulness of workflows in the DEEP-EST project. In a workflow, different components of an application are allocated at different times. This approach delays the execution of data-consuming components until the data is produced. On the other hand, it ensures some overlap of the execution among data producer and consumer components. This helps them to communicate data directly over the network, instead of using an expensive storage system.

The heterogeneous job packs feature available in vanilla Slurm does not provide the delaying functionality. We have added a new switch named `--delay` to the heterogeneous job packs. The value for this switch is provided by the user for each job in a job pack, specifying how much the allocation of the respective job shall be delayed after the allocation of the first job. The modified scheduler of Slurm, upon receiving a job pack with delays, then creates reservations for each job in the job pack to ensure definite starting times for them. The implementation of the `--delay` switch has been explained in D5.3.

5.3.1 Improvements in the prototype implementation

The Slurm version on the prototype was 17.11. Now we have upgraded to the latest version of Slurm, which is 19.05, for this deliverable. After the evaluation of the `--delay` switch implementation in the prototype, reported in D5.3, some enhancements have been done. In the previous implementation, reservations were created according to the delays for each job in the job pack. The user was able to still see the submitted heterogeneous job as a single job pack.

This behaviour has now been changed. All the jobs in a job pack with different delay values are now separated as individual jobs. However, consecutive jobs with equal delay values are combined into a new heterogeneous job pack. This provides the additional facility of using the heterogeneous job function in the workflow scenario.

Listing 5.1 shows a script for submitting a workflow. The first job asks for an allocation of two nodes for three minutes. The second and third jobs are requesting one node each, again for three minutes. Note the delay of two minutes each. At the end, the fourth job again requests two nodes for three minutes. The delay required for the fourth job is six minutes after the allocation of the first job. Once

the reservations for all of the jobs are in place, they are broken down into separate jobs as described before.

Listing 5.2 shows the resulting list of jobs after submission of the script. Note that the first and fourth jobs are individual jobs, while both second and third job are combined into a new heterogeneous job. Only the first job is in running state and the allocation of all other jobs has been halted due to the reservations created for them have not started yet.

```
$# cat workflow.sh
#!/bin/bash
#SBATCH -N 2 -t 3
#SBATCH -J first
#SBATCH packjob
#SBATCH -N 1 -t 3 --delay 2
#SBATCH -J second
#SBATCH packjob
#SBATCH -N 1 -t 2 --delay 2
#SBATCH -J second
#SBATCH packjob
#SBATCH -N 2 -t 3 --delay 6
#SBATCH -J fourth

if [ "$SLURM_JOB_NAME" == "first" ]
then
srun app1
elif [ "$SLURM_JOB_NAME" == "second" ]
then
srun app2 : app3
elif [ "$SLURM_JOB_NAME" == "fourth" ]
then
srun app4
fi
```

Listing 5.1: An example workflow script with `--delay` values

```
$# sbatch workflow.sh
Submitted batch job 683

$# squeue
JOBID  PARTITION    NAME  USER  ST  TIME  NODES  NODELIST(REASON)
684+0      debug  second   ***  PD  0:00      1  (Reservation)
684+1      debug  second   ***  PD  0:00      1  (Reservation)
  686      debug  fourth   ***  PD  0:00      2  (Reservation)
  683      debug   first   ***   R  0:01      2  worker[01-02]
```

Listing 5.2: Resulting jobs for the submission of job script shown in Listing 5.1

5.3.2 Slurm API usage for efficiency

Since the delay values in a workflow are provided by users, they are never optimal. Users would most probably over- or under-estimate the delay values for their workflows. This drawback puts constraints

on the Slurm scheduler and bigger errors in delay estimation may result in very inefficient scheduling of jobs. To overcome this problem, we have implemented a solution which applications can use to bring forward the start times of the reservations of the other jobs in the workflow. This is done by using and extending the Slurm API.

We have implemented some changes in the Slurm API to facilitate changing the start and end times of reservations. We also provide a helper library that encapsulates API usage and provides a high level function to the applications for changing reservations for the current workflow. Algorithm 1 shows the pseudo-code of the implemented helper library. The first parameter to the library function is the current job's ID and the second parameter is the number of minutes after which the next reservation should start, based on the current time. The amount of time to move each reservation, that is `move_time`, is calculated from the starting time of the next reservation. We first try to change the start times of all reservations of the remaining jobs in the workflow to start earlier. If all the reservations' start times are updated successfully, we also update the end times of these reservations. Otherwise all the changes are reverted and a negative response is sent back to the calling application.

Once an application detects that the data for the next job in the workflow is now available or about to be available in near future, it can try to bring the start times of all remaining jobs forward. If successful, the next job will start earlier than initially anticipated, hence reducing any wait times or avoiding idle times.

5.3.3 Forthcoming Work

Although using the helper library can help to improve the scheduling algorithm of Slurm, there is still a disadvantage as initial reservations are created using user-supplied delay values. So there may still be some efficiency loss due to the reservations created initially. We propose another mechanism to overcome this problem.

In this solution, users submit their workflow jobs as individual ones. They have to provide the `--dependency=afterok:job_id` flag for all dependent jobs. Here the `job_id` should be the ID of the job that produces data for the current job to be submitted. This causes Slurm to hold the dependent jobs until the job they depend on have finished. Once the producer job detects the data needed for one or more of its depending jobs is available, it can use Slurm's API to update the dependency type of all of its dependent jobs. The new dependency type is set to `after`. This causes Slurm's scheduler to start scheduling and allocating the dependent jobs as resources become available.

The advantage of this approach is the total removal of any user-provided delay values. However, it does not guarantee the overlap of the jobs producing and consuming data, as resources may not be available for the dependent job at the moment when its dependency job has been changed. So the application running in the producer job has to adopt according to the situation. The development of this approach through a separate helper library is currently ongoing.

We also plan to evaluate the effects of workflows (both with and without delay switch) on the efficiency and performance of Slurm's scheduler. We will port the changed version of Slurm to the Slurm simulator. We will calculate typical scheduler metrics like average waiting time, average slowdown and total completion time etc.

```
Function updateWorkflowRes(int jobId, int t)
    j = loadJobInfoFromSlurm(jobId);
    wfJobsList = getWorkflowJobs(j);
    flag = true;
    jobsChangedStartList = NULL;
    time time_move = 0;
    foreach job in wfJobsList do
        if job.jobId > jobId then
            /* Update start times of all the reservations of each next job in the
               workflow.                                                          */
            res = getReservation(job.jobId);
            if time_move == 0 then
                /* calculate the time to move all reservations.                  */
                time new_start_time = TIME_NOW + t * 60;
                if new_start_time >= res.start_time then
                    return true;
                end
                time_move = res.start_time - new_start_time;
            end
            flag = updateStartTime(res, -time_move);
            if flag == false then
                /* Change not possible in all the reservations. Break the loop.  */
                break;
            end
            jobsChangedStartList.insert(job);
        end
    end
    if flag == false then
        /* No change in reservations possible. Reset any changes made and exit.  */
        foreach job in jobsChangedStartList do
            res = getReservation(job.jobId);
            updateStartTime(res, time_move);
        end
        return false;
    end
    foreach job in wfJobsList do
        if job.jobId > jobId then
            /* Update end times of all the reservations of each next job in the
               workflow.                                                          */
            res = getReservation(job.jobId);
            updateEndTime(res, -time_move);
        end
    end
    return true;
end
```

Algorithm 1: Algorithm of updating workflow reservations.

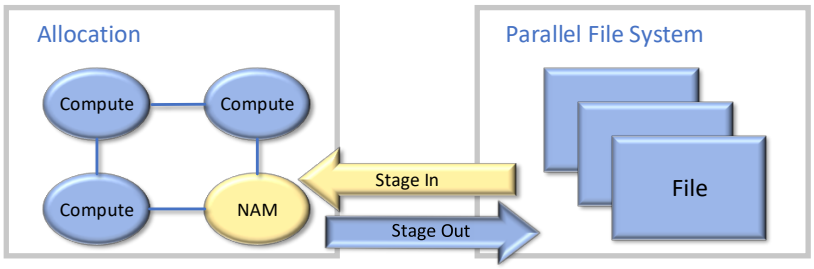


Figure 18: Stage-in and stage-out from the file system to the NAM-based Burst Buffer (BB).

5.4 Efficient Scheduling and Management of Network Attached Resources

Network Attached Memory (NAM) and the Global Collectives Engine (GCE) are the planned network-attached resources of the DEEP-EST project. Due to delays in its availability, there is currently no implementation related to the GCE in the resource manager. Necessary features will be identified and developed as we approach its delivery by EXTOLL next year. The following is a description of the current features added to the Slurm workload manager in order to support NAM resources in a feature-complete manner that is both user-friendly and efficient.

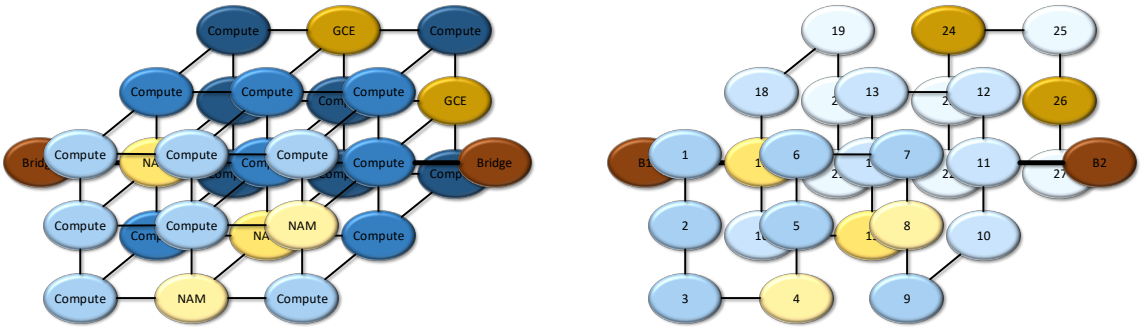
The Network Attached Memory (NAM) resource is a low-latency and high-bandwidth persistent storage device that will be attached to the EXTOLL network. It is possible to allocate NAM for jobs of all modules in the Modular Supercomputing Architecture (MSA). Because of this, its performance will vary depending on the network topology and the modules utilized by a job.

5.4.1 SPANK Burst-Buffer (BB) Plugin Implementation

There is already a family of low-latency and high-bandwidth persistent storage solutions in production supercomputing systems today: Burst Buffers (BB). These provide storage capacity that is persistent during a well-defined time window. These are not replacements for the parallel file systems that are meant as reliable and very high-capacity long-term storage.

Burst Buffers (BB) are intended to improve I/O performance on jobs that read and write a lot of data during their runtime. This is achieved by first staging-in the data from long-term storage into the fast BB storage, and then staging it back out when the job completes. A depiction of this process is presented in Figure 18. Their capacity is typically much lower than the traditional parallel file systems. These tend to rely on recent high-performance Solid State Storage (SSD) devices and can be found in several different types of configurations in current supercomputing deployments. For example, some setups implement the burst buffers as SSDs on each node, while others create dedicated BB nodes at each rack (therefore reachable over the local switch). There is no rigid set of specifications that define how a BB should be implemented today.

In our MSA the BB will be implemented as dedicated NAM nodes that will be attached to the EXTOLL network. The NAM nodes are embedded devices running their specialized firmware for



(a) Network topology

(b) Peano-based order.

Figure 19: Bridges, NAM and compute resources in the EXTOLL network (left) and an example of an order based on a space-filling curve (right).

high-performance and low-latency operation. Together with the hardware, a runtime library will be provided for both management and user access to the memory at each NAM instance. The library allows for the creation and destruction of NAM allocations and provides an API for writing and reading bytes. The API is not file I/O-based, and therefore applications need to be updated to take advantage of this type of fast memory.

Since BB capacity is a shared and limited resource, it needs to be managed. The NAM runtime library does not have support for reservations. Since reservations are needed and are not a trivial feature to implement, it has been decided to have Slurm manage the NAM; this is consistent with the setup of most compute centres with BB support in production today.

Slurm's plugin infrastructure is called SPANK and it already has a family of plugins (with its own API) dedicated to BB implementations. Currently only a generic and a Cray specific BB implementation is available. These are based on file I/O-based BB implementations, and therefore are not suitable for the NAM-based BB that is developed in this research project. As a consequence, a new BB SPANK plugin has been added to Slurm. In this plugin, the BB metadata stored in Slurm is taken as the input and the necessary NAM library calls are performed by the plugin during the lifetime of a user's NAM allocation.

5.4.2 User Interface

The BB user interface is re-used from upstream Slurm. With it, users can perform the following operations:

- Create and destroy persistent BB allocations.
- Create jobs with BB allocations within their scope.
- Specify stage-in and stage-out instructions.

The interface is available via batch scripts for the `sbatch` command and command line parameters for the `srun` command. Persistent allocations are meant to be used across multiple jobs. These are created and destroyed via batch scripts that contain only BB-specific instructions, such as the capacity and staging requirements. Alternatively, a job can be created together with BB requirements. In this case, the BB allocation is created and files are staged in before the job starts, and files are staged out and the allocation destroyed after the job completes.

5.4.3 Topology Awareness

An illustration of a 27-node 3D torus is presented in Figure 19a. Due to the MSA design with gateways and the multidimensional torus topology of the EXTOLL network, the NAM-based BB storage's latency varies based on the node allocation, especially across modules. The variability is expected to be negligible in our test system, due to its small size; however, the design was made taking into account possible future large MSA-based deployments.

Slurm already provides SPANK plugins that enable topology aware node selections. These do not need to be modified, since there is already support for torus-based networks with the linear selection plugin and orders based on space-filling curves. Figure 19b illustrates the 27-node 3D torus with a Peano-based order. Only the NAM-based BB plugin needs to be developed taking the topology into account. For this, the configuration of the NAM metadata has been designed to include neighbour information in the topology. When an allocation is made for a job, a heuristic in the NAM BB plugin will then select a set of NAM instances (based on availability) that attempts to minimize the number of hops between allocated nodes. In the case of inter-module allocations, the heuristic will attempt to select NAM instances closer to the relevant gateways first.

6 System Monitoring and RAS Plane

In this chapter we describe the development status of DCDB [4], the holistic sensor monitoring framework implemented by BAdW-LRZ, and lay out the specifications of its deployment on the DEEP-EST prototype. Section 6.1 provides details on recent changes and improvements introduced since our last report (D5.3 [19]) both in DCDB (Section 6.1.1) and in the Wintermute data analytics component (Section 6.1.2). In Section 6.2, we provide an overview of the planned deployment of DCDB on the DEEP-EST prototype, detailing the exposed data sources. Section 6.3 concludes the chapter, presenting the direction of our future efforts in the project.

6.1 Current Status

Development on DCDB mainly concentrated on the Wintermute data analytics component, but minor changes and improvements have also been carried out on the core DCDB framework.

6.1.1 DCDB Framework

The architecture of DCDB is stable and was not altered since D5.3 [19]. We have focused on testing the framework and polishing its code base, leveraging our early deployments on several HPC systems at BAdW-LRZ to expose bugs, as well as usability and scalability issues. In the following we describe the main changes done to DCDB.

Database schema. Integration of the Wintermute operational data analytics framework exposed the necessity to simplify the schema used in the Storage Backend to store sensor data, in order to allow for simpler and more flexible configurations. In the previous version of DCDB each single sensor was identified by a numerical *sensor ID*, which was used as MQTT topic and key in the associated Apache Cassandra table. The sensor ID was never exposed to the final users, and a mapping from a public sensor name to the sensor ID had to be provided alongside the configuration. This dual system proved to be difficult to use in our early deployments, and was cumbersome to deal with in the context of Grafana visualisation, as well as Wintermute. For this reason we switched to a simplified system, in which we use string sensor IDs acting as database keys, MQTT topics and public names. This removes the necessity of supplying a mapping from sensor names to sensor IDs, greatly streamlining configuration and usage of DCDB as a whole. The new naming scheme also retains the feature of DCDB to store sensor data local to the source by leveraging Cassandra's byte-order partitioner. After early testing, we could not observe any significant difference in the overall performance of the framework with the new schema.

Metadata management. It is now possible to specify metadata information for each sensor within the configuration files used for DCDB Pusher. This metadata is automatically propagated at startup to the Collect Agents and Storage Backends, and can be later leveraged when performing sensor queries. Among the supported metadata fields, we currently have the unit of measure, scaling factor, sampling interval and two flags to indicate monotonicity and integrability. Moreover, a time to live (TTL) value can be defined for each sensor, which will be used by the Collect Agent when performing inserts to the Storage Backend. This allows to define different persistence times of the data in the Storage Backend on a per-sensor basis.

GPU sampling plugin. EPCC is currently developing a DCDB Pusher plugin suitable for retrieving performance data from the NVIDIA Tesla V100 GPUs present on the DAM and ESB modules of the DEEP-EST prototype. This plugin is based on the NVIDIA NVML Library[1], which provides a series of *device query* functions to retrieve a wide range of metrics from GPU devices. Among the available metrics, the most interesting for our deployment have been identified in the GPU's clock rate, number of running processes, temperature, fan speed, power and energy, PCIe bandwidth utilisation, GPU and memory utilisation and finally ECC errors (both corrected and uncorrected).

6.1.2 Wintermute Framework

Similarly to DCDB, the architecture of the Wintermute framework has not changed significantly since D5.3. Our efforts have focused on polishing the Wintermute codebase for production use, as well as developing appropriate plugins for a variety of use cases. In particular, we did the following.

Code Improvement. The code base of Wintermute has been improved over the course of the recent months, with the objective of reaching a state suitable for production deployment. All of the features discussed in D5.3 have been implemented and thoroughly tested: analysers (now renamed *operators*) can be used in *online* and *on-demand* modes, and can perform *job-level* as well as *sensor-level* analyses. Moreover, we adapted the Wintermute architecture to allow for operator *pipelines*: several operators can now be chained together, with the outputs of each being fed as input to another operator. This simple modification expands the capabilities of the framework considerably, and also enables the use of *control* operators, which do not produce any form of sensor output, but are solely devoted to changing system knobs according to their inputs. Finally, we streamlined and improved the configuration abstractions described in D5.3, allowing in turn to define multiple hierarchical levels of output for each operator (e.g., job-level and node-level output sensors).

Development of Plugins. A series of Wintermute plugins were developed so as to evaluate its effectiveness in online scenarios. Since operators in Wintermute can be pipelined to attain complex analysis capabilities, there is usually no need to develop ad-hoc plugins, and users can simply rely on a small set of general-purpose ones. Based on this philosophy, we developed the following plugins:

[1] `https://docs.nvidia.com/deploy/nvml-api/group__nvmlDeviceQueries.html`

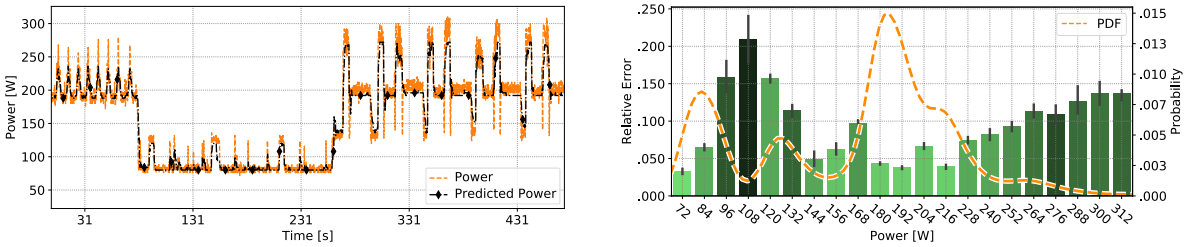


Figure 20: Power consumption prediction in a compute node with Wintermute. On the left we show an excerpt of the predicted time series against the real one; on the right we show instead the bar plot of the relative prediction error for different power states, as well as the probability density function associated to the latter.

- **Aggregator**: operators in this plugin take as input an arbitrary number of sensors, and can compute several statistical indicators by performing aggregation of their values over a certain time window. The indicators we currently support are sum, average, standard deviation, minimum, maximum, median, number of observations and percentiles.
- **Job Aggregator**: this plugin performs the same functions as the *aggregator* plugin, but on a per-job basis: for each running job, input sensors from allocated nodes are automatically selected, and aggregation is subsequently performed to obtain job-level statistics.
- **Performance Metrics**: this plugin allows to compute a series of derived performance metrics (e.g., floating point operations per second, clocks per instruction, memory bandwidth) starting from raw sensor data. This plugin proves especially useful when paired with the *job aggregator* plugin, allowing to compute and store job-level performance metrics in a simple way.
- **Regressor**: operators in this plugin enable online random forest-based regression of any monitored sensor in Wintermute. These use as input a certain number of sensors to perform regression, whose result is stored as output. Training can be performed both online and offline, by loading an appropriate model file. An example of regression task is presented in Figure 20: specifically, Wintermute is used to predict the next power consumption value in a Knights Landing-based compute node every 250 ms.

6.2 DCDB Deployment Overview

The DCDB monitoring framework will be deployed on all of the three compute modules of the DEEP-EST prototype: the Cluster Module, the Booster Module, and the Data Analytics Module. Each of the modules has a dedicated management server equipped with a 480 GB SSD that will serve as the storage backend and host a Cassandra server instance along with a Collect Agent instance. Additionally, there will be a DCDB Pusher instance on each management server to collect out-of-band monitoring data from the respective compute module and its infrastructure. A single Pusher instance will be deployed on each compute node to collect in-band monitoring data. DCDB will be configured such that sensor data collected both in-band and out-of-band from a particular module will be stored on the management server of the respective module to avoid network traffic. Figure 21 gives a schematic overview of the DCDB deployment.

In the following sub-sections we list the metrics that we will collect on the individual modules of the DEEP-EST prototype. The actual sampling rates for each metric will have to be determined once DCDB has been rolled out to find the optimal trade off between resolution and runtime overhead. However, we are aiming for 1–10 s sampling intervals for the in-band metrics and 10–30 s sampling intervals for the out-of-band metrics.

6.2.1 Cluster Module

On the cluster nodes we will deploy the perf_events, procfs, and sysfs plugins with DCDB Pusher to collect in-band sensor data. The perf_events plugin will sample the following performance counter events on each of the 24 (hyperthreading-) cores of each of the two sockets per node:

- Total cycles
- Total reference cycles
- Instructions retired
- Retired branch instructions
- Mispredicted branch instructions
- Cache accesses
- Cache misses

Additionally, we will collect the following performance counters for the six memory controllers of each socket:

- Memory read accesses
- Memory write accesses

With the procfs plugin we will sample 12 metrics from `/proc/stat` related to Linux Kernel activity on the compute nodes, 17 metrics from `/proc/mem` on memory utilisation, and 22 metrics on virtual memory statistics from `/proc/vmstat`.

The sysfs plugin will be used to collect power, energy, and temperature readings exposed by the Linux Kernel via the `/sys` filesystem:

- CPU package and DRAM energy (Intel RAPL)
- CPU temperature
- Megware SlideSX energy meter (power and energy)

Additionally to the in-band data, we will deploy a DCDB Pusher instance on the Cluster Module's management node to collect out-of-band data from the Power Delivery Units (PDUs) and cooling infrastructure of the water-cooled racks. We will use the REST-API plugin to read power draw and energy consumption from the Megware Clustsafe PDUs and SNMP to read infrastructure sensors from the Schaefer iQdata InRackcooler units in each rack, namely:

- Primary loop supply and return temperatures

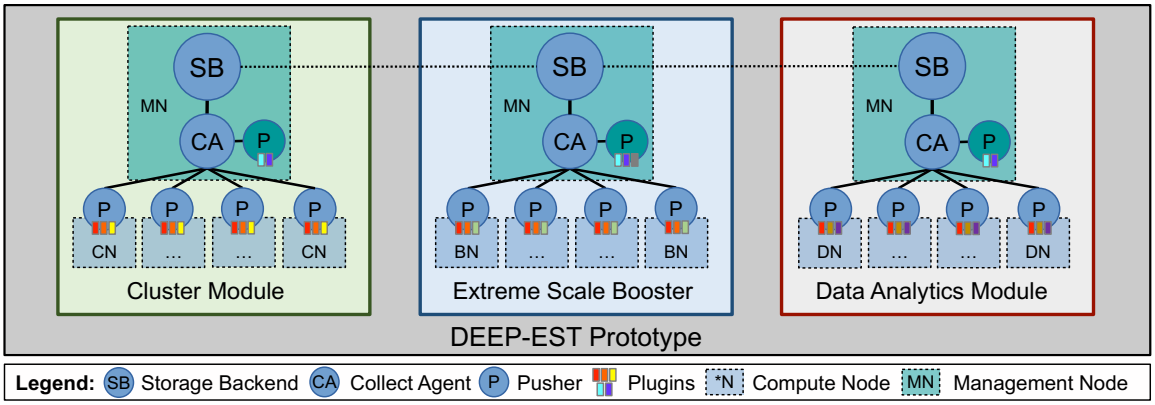


Figure 21: Overview of the DCDB deployment on the DEEP-EST prototype.

- Primary loop water flow rate
- Primary loop heat transfer
- Primary loop valve opening position
- Secondary loop supply and return temperatures
- Secondary loop supply and return pressure
- Secondary loop temperature set point

6.2.2 Booster Module

On the Booster nodes we will deploy the same plugins as on the Cluster nodes for in-band data collection and mostly collect the same metrics. The most notable difference will be in the total number of metrics collected by the perf_events plugin due to the significantly lower host CPU's core count of 8 on the Booster node. The Booster nodes will, however, feature two Megware energy meters instead of just one: one for the host system and one for the GPU. Additionally, the Booster nodes are equipped with an NVIDIA V100 GPGPU from which we will read telemetry data via the NVIDIA plugin that is currently being developed by EPCC. It will allow for the following metrics to be collected:

- GPU clock rates
- Number of running processes
- Temperature
- Instantaneous Power
- Energy
- PCIe throughput (bi-directional)
- GPU and memory utilisation
- ECC errors

In terms of out-of-band measurements, we will collect the very same data on Booster Module that we are collecting on the Cluster Module since they are essentially the same racks.

6.2.3 Data Analytics Module

On the Data Analytics nodes we will use a configuration very similar to the Booster nodes. Again, the most notable difference is in the number of perf_events metrics due to the higher core count (2 sockets, 24 cores each). Also, the SuperMicro nodes do not feature a dedicated energy meter as the Megware nodes in the Cluster and Booster do.

Since the SuperMicro nodes are air-cooled, the only available sensor data that will be collected out-of-band from the infrastructure will be power and energy readings from the Megware Clustsafe PDUs.

6.3 Future Work

Future work will concentrate on deploying the configuration described above on the DEEP-EST prototype modules and performing overhead measurements to determine the optimal sampling rates of each individual metric. Additional efforts will be devoted into developing DCDB Pusher plugins for the EXTOLL Fabri^3 and the Mellanox Infiniband fabric of the Cluster Module, as well as completing the implementation of the NVIDIA V100 GPU plugin.

7 Summary

This deliverable presents the complete system software implementation with all the functionalities required for the deployment on the DEEP-EST system. The following paragraphs summarize the most important achievements concerning the interconnect management, the network bridging, the resource management, the job scheduling and the monitoring:

Enhancement of the EMP and development of the Fabri³Manager. The EMP framework of EXTOLL has experienced some major improvements, e. g., the EMP master daemon has been extended significantly with respect to management and resiliency. In addition, the Fabri³Manager has been implemented as a new but important low-level software component that has recently been added to the EMP framework. Furthermore, GPUDirect support for EXTOLL was added as a recent feature for accommodating the new GPU-centric ESB architecture. Although the NAM and the GCE have been delayed and are at the time of writing not yet available, both have of course already been taken into account as global resources to be managed by the EMP framework and in accordance with the other parts of the system software architecture.

Implementation of an MPI bridge between InfiniBand and EXTOLL. For forwarding MPI traffic between the Cluster and the ESB module, a bridging framework has been implemented within ParaStation MPI relying on gateway daemons running on related gateway nodes. On the one hand, this required the implementation of a new plugin for the ParaStation Management framework to start the gateway daemons. On the other hand, the pscom library of ParaStation MPI had to be extended with a communication plugin for the actual forwarding. This plugin, in turn, has been optimised by pipelining and extended by rendezvous capabilities leveraging EXTOLL's RMA semantics so that it now achieves appropriate throughput performance.

Support for heterogeneous jobs by the resource manager. ParaStation Management has been extended by the integration of the Job Pack feature into the psslurm plugin as well as the adaption of the new Slurm concept of Trackable Resources. Both aspects are important developments for the support of heterogeneity and modularity. In addition, ParaStation Management has been extended by a mechanism for collecting and forwarding energy and power data as provided by the Megware hardware in order to make the software stack not only modularity but also energy aware.

Scheduling support for modularity, workflows and shared resources. Two new job scheduling policies for Slurm have been developed to further improve the support for modularity as well as for energy awareness. The policy aims at scenarios where an application can run in different modules and the scheduler should ensure that the time-to-solution for this application is as short as possible. Evaluation results for this policy have shown that the slowdown and the completion time can be actually improved. The second of these policies aims at energy-conscious job scheduling by combining energy predictions with priority adjustments for prioritizing jobs with a higher efficiency.

In addition, the new delay switch functionality has been developed for the scenario of workflows. This delays the execution of subsequent job steps within a workflow until the required results are available while ensuring a time overlap for the data exchange between the steps. However, as this feature requires a prediction of the delay value by the user, which is likely to be not always optimal, a further enhancement is an extension to the Slurm API enabling the application to adapt the start times during runtime.

Finally, for an efficient scheduling and management of network attached resources such as NAM and GCE, a new Burst Buffers SPANK plugin has been added to Slurm. This plugin will manage the dedicated NAM nodes and will perform the necessary NAM library calls during the lifetime of a user's NAM allocation. Although Burst Buffers are actually intended to improve I/O performance, it turned out that this type of Slurm plugins can also be used to elegantly manage capacity-limited resources in general, just as the NAM is one.

Implementation of a scalable and modular monitoring layer. Last but not least, a scalable middleware for system monitoring has been developed by enhancing and extending the Data Center Data Base (DCDB) to a holistic sensor monitoring framework. For a comprehensive visualisation of the gathered sensor data, a DCDB-related plugin for the Grafana visualisation tool has been developed and extended so that users can easily query sensors at specific hierarchical levels of an HPC system. Additionally, a data analytics component has been developed and integrated with DCDB in terms of the Wintermute framework enabling asynchronous data analytics of monitoring data. For Wintermute, a series of plugins have been developed and its architecture has been adapted to allow for operator pipelines so that a comprehensive framework for the analysis of sensor data is available to the DEEP-EST system, its operators and its users.

Deployment status on the DEEP-EST prototype. All these important system software components and functionalities described here are already implemented and most of them are also already deployed on the DEEP-EST system. However, as some of the related hardware components have been delayed, a few features are not available yet. However, they are ready to be tested and put into operation as soon as the corresponding hardware is available.

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

Fabri³	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
I²C	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
PIC Family of microcontrollers made by Microchip Technology Inc.
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] *Slurm documentation* [Online], Available: https://slurm.schedmd.com/documentation.html

[11] 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.

[12] 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.

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

[14] A. Jokanovic et al.: *Evaluating Slurm simulator with real-machine Slurm and vice versa*. IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems Workshop (PMBS), 2018.

[15] D. Feitelson et al.: *Toward convergence in job schedulers for parallel supercomputers*. Job Scheduling Strategies for Parallel Processing (JSSPP), 1996.

[16] P. Martinez-Ferrer et al.: *DEEP-EST Deliverable 1.3: Application distribution strategy*, June 2018

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

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

[19] N. Eicker et al.: *DEEP-EST Deliverable 5.3: Prototype Software Implementation*, March 2019

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

[21] M. Nuessle et al.: *DEEP-EST Deliverable 4.3: Network Federation, Fabri³, NAM and GCE designs*, June 2018

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

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

[24] 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

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

[26] 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

[27] 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

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

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

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

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

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

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

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

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