EuroHPC-01-2019

DEEP-SEA

DEEP – Software for Exascale Architectures

Grant Agreement Number: 955606

D3.1
Software Specification

Final

Version: 1.0

Author(s): S. Pickartz (ParTec), M. Marazakis (FORTH), N. Eicker (FZJ)

Contributor(s): A. Mazouz (Atos), A. Roussel (CEA), A. Geiß (TUDa), A. J. Peña (BSC), C. Clauss (ParTec), C. Feld (FZJ), D. Chasapis (BSC), G. Llort (BSC), I. A. Comprés Ureña (TUM), J. Jaeger (CEA), P. Lemarinier (Atos), M. Sergent (Atos), M. Geimer (FZJ), M. Michael Ott (LRZ), M. Rauh (ParTec), M. Owais (FHG), M. Schlütter (FZJ), P. Carpenter (BSC), P. Radojkovic (BSC), P. Lesnicki (Atos), R. Machado (FHG), S. Lührs (FZJ), S. Vanecek (TUM), S. Krempel (ParTec), Th. Moschny (ParTec), T. Ridley (JSC), T. Schneider (ETHZ), V. Lopez (BSC), V. Beltran (BSC)

Date: 30.11.2021

Project and Deliverable Information Sheet

DEEP-SEA Project	**Project ref. No.:**	955606
	Project Title:	DEEP – Software for Exascale Architectures
	Project Web Site:	`https://www.deep-projects.eu/`
	Deliverable ID:	D3.1
	Deliverable Nature:	Report
	Deliverable Level: PU*	**Contractual Date of Delivery:** 30.11.2021
		Actual Date of Delivery: 30.11.2021
	EC Project Officer:	Daniel Opalka

*– 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:** Software Specification	
	ID: D3.1	
	Version: 1.0	**Status:** Final
	Available at: `https://www.deep-projects.eu/`	
	Software Tool: LaTeX	
	File(s): DEEP-SEA_D3.1_Software_Specification.pdf	
Authorship	**Written by:**	S. Pickartz (ParTec), M. Marazakis (FORTH), N. Eicker (FZJ)
	Contributors:	A. Mazouz (Atos), A. Roussel (CEA), A. Geiß (TUDa), A. J. Peña (BSC), C. Clauss (ParTec), C. Feld (FZJ), D. Chasapis (BSC), G. Llort (BSC), I. A. Comprés Ureña (TUM), J. Jaeger (CEA), P. Lemarinier (Atos), M. Sergent (Atos), M. Geimer (FZJ), M. Michael Ott (LRZ), M. Rauh (ParTec), M. Owais (FHG), M. Schlütter (FZJ), P. Carpenter (BSC), P. Radojkovic (BSC), P. Lesnicki (Atos), R. Machado (FHG), S. Lührs (FZJ), S. Vanecek (TUM), S. Krempel (ParTec), Th. Moschny (ParTec), T. Ridley (JSC), T. Schneider (ETHZ), V. Lopez (BSC), V. Beltran (BSC)
	Reviewed by:	T. Schneider (ETHZ) D. Gottwald (FZJ)
	Approved by:	BoP/PMT

Document Status Sheet

Version	Date	Status	Comments
1.0	30.11.2021	Final version	EC submission

Document Keywords

Keywords:	DEEP-SEA, HPC, Exascale, Software

Acknowledgements:

The DEEP Projects have received funding from the European Commission's FP7, H2020, and EuroHPC Programmes, under Grant Agreements n° 287530, 610476, 754304, and 955606. The EuroHPC Joint Undertaking (JU) receives support from the European Union's Horizon 2020 research and innovation programme and Germany, France, Spain, Greece, Belgium, Sweden, United Kingdom, Switzerland.

Contents

List of Figures

List of Tables

Executive Summary

The application diversity in HPC is continuously increasing and applications are combined to workflows coupling different programming models. These developments put high demands on the system software of future Exascale systems. The DEEP-SEA software stack addresses this challenge by defining a system software architecture considering both heterogeneity throughout the hardware landscape and dynamically changing resource requirements of the application workflows. The software stack is especially designed to be employed on current and future HPC systems that are built by following the design principles of the Modular Supercomputer Architecture (MSA). This way, heterogeneity, scalability, and the dynamic assignment of resources to jobs or application workflows can be guaranteed while enabling the flexible and efficient utilisation of future Exascale systems.

In this deliverable, we present the central components of the software stack that are extended and enhanced in the scope of the DEEP-SEA project while covering the following major technology areas: performance analysis, memory management, communication and programming models, resource management and scheduling, and programmer productivity. In doing so, we introduce the concept of optimisation cycles constituting typical workflows of application developers for the adaptation and the optimisation of their application codes to the target platform. These optimisation cycles identify and define the interplay of the software stack's components and by this means specify the architecture of this stack. This basic concept will be used to further drive the developments and to steer future design decisions of the project. According refinements of the optimisation cycles will be presented in future deliverables of this project.

1 Introduction

The ever-rising diversity of application workflows coupling different programming models and codes puts high demands on the system software of future Exascale systems. The DEEP-SEA software stack enables the flexible and efficient utilisation of Modular Supercomputer Architecture (MSA) systems. This goal will be achieved by defining a system software architecture taking the heterogeneity on all levels of the platforms into account while serving the dynamically varying resource requirements of complex application workflows. The convergence of HPC/HPDA/AI workloads is considered in DEEP-SEA as a principal driver in the evolution of system software infrastructure and tools. The major challenges of convergence appear to lie in combining flexibility with heterogeneity.

1.1 Purpose and Scope

This document outlines the structure and the major components required to support the vision of the DEEP-SEA project for the European Exascale HPC software stack. It provides a decomposition of the software architecture by introducing the central components that will be enhanced and extended within the scope of this project. At the same time for the sake of brevity, we will ignore all components that will be used as is even though they represent central parts of the software-stack such as the Operating System (OS) (Linux in our case), language interpreters (e. g., Python), or compilers (e. g., gcc, LLVM, etc.). Since they are common in today's HPC software-stacks, we presume them as a general prerequisite that needs no further discussion. The second part of this presentation comprises a discussion of the interplay of the listed components including the information flow among them. We introduce the concept of optimisation cycles utilising the typical workflow of an application developer to identify and describe dependencies between the components. Further decompositions of the major components themselves will be covered in follow-up deliverables including a detailed definition of the corresponding APIs.

This deliverable considers the following major technology areas covered by the DEEP-SEA software architecture:

Performance Analysis The DEEP-SEA software stack will provide a wide range of tools for the creation of both correct and expeditious programs, their execution on heterogeneous nodes and MSA systems, and the monitoring of their efficiency. These aspects contribute to PObjs 1, 2, and 3.[1,2,3]

Memory management The management and use of deep and heterogeneous memory hierarchies will be facilitated by the development of node-level memory management APIs contributing to PObj-4.[4]

[1]PObj-1: Co-design the software- and programming environment of the upcoming European Exascale systems.

[2]PObj-2: Provide tools to map complex applications and non-uniform workflows onto heterogeneous and modular computer architectures.

[3]PObj-3: Enhance the system software, programming paradigms, tools, and runtimes in order to extract the maximum performance from heterogeneous computer platforms and improve performance portability.

[4]PObj-4: Improve the use and management of new memory technologies and the placement of data in compute devices with deep and heterogeneous memory hierarchies.

Communication and programming models The DEEP-SEA software stack will address heterogeneity on all levels of an MSA system ranging from node-level memory heterogeneity and non-uniformity to a heterogeneous interconnect landscape. This has to be taken into account by both the low-level communication facilities and the programming models to support a variety of workloads running on top, e. g., the Open Multi-Processing (OpenMP) standard, the Message-Passing Interface (MPI), Partitioned Global Address Space (PGAS), ML/DL, etc.

Resource management and scheduling The existing resource management and scheduling system will be extended to support a dynamic resource utilisation on the workflow level, scalability for large node and job counts, but also malleability inside a single job step. These aspects contribute to PObj-3.

Programmer productivity A separation of the problem formulation from optimisations for parallel execution, e. g., efficient data movements, enables domain scientists to focus on the description of the actual algorithms while HPC experts may ensure their efficient execution on supercomputers. The DEEP-SEA software stack therefore incorporates the two high-level programming models Data-Centric (DaCe) parallel programming and NabLab while contributing to PObj-3. Additionally, the application deployment will be facilitated by leveraging container technologies allowing users to provide customised environments (i. e., Bring Your Own Environment) and supporting the reproducibility of results.

The integration process of all components is accomplished by applying Continuous Integration (CI) techniques. This way, we ensure interoperability right from the beginning. Therefore, all project partners will contribute to the central CI infrastructure being set up at Jülich Supercomputing Centre (JSC) by providing the necessary recipes and scripts for their components.

This deliverable starts with an overview of the proposed architectural model (cf. Chap. 2) by introducing the concept of so-called *Optimisation Cycles*. Chapter 3 identifies the main software components forming the DEEP-SEA software stack. These cover the five technology areas listed above. Finally, Chapter 4 introduces the optimisation cycles being composed of the building blocks described before.

2 Software Architecture

The DEEP-SEA software stack specifically targets the MSA as the aspired integration concept of the European Exascale System. Nevertheless, most of the concepts are not restricted to MSA but are applicable to heterogeneous HPC systems in general.

2.1 The Modular Supercomputing Architecture

An MSA system is composed of a number of distinct compute modules, each being a parallel cluster of potentially large size. The respective hardware configuration is chosen to address the needs of a specific kind of application or a certain part of an application. The modules are connected through a federated high-speed network (cf. Fig. 1) and a unified software environment enabling the distribution of individual applications across different modules. This empowers application workflows to leverage general-purpose processors, different kinds of accelerator technologies, and even innovative technologies, such as NAM, neuromorphic-processing, or quantum-processing devices at the same time. Heterogeneous applications and workflows benefit substantially from this approach since each part of a code can be executed on the most suitable platform, thus, improving both time to solution and energy efficiency.

Today, hardware heterogeneity is employed at all levels—system, node, and package—and might apply to at least compute and memory technologies. To facilitate the different node-level resources for application users, the system software has to be aware of this heterogeneity and has to manage it accordingly, e. g., data placement decisions must optimise locality in heterogeneous CPU+vector nodes, system monitoring and performance-analysis tools have to be extended to also provide information and guidelines for an efficient use of complex memory hierarchies.

As a consequence, the number of components (i. e., tools, libraries, run-time systems, etc.) composing the system software is tremendous, resulting in a potentially very complex overall software architecture due to the huge amount of possible mutual interactions. However, the number of procedure models in HPC is limited and therefore reducing the actual relevant interactions of the components significantly.

A very high-level view on the software architecture is depicted in Figure 2. More detail will be presented along the lines of the description of the various components in Chapter 3 on the one hand and use-case examples named optimisation cycles in Chapter 4 on the other hand.

2.2 Optimisation Cycles

We introduce the concept of Optimisation Cycles in order to represent the typical workflow of an application developer while implementing and optimising an HPC-code covering a specific scientific field. By this means, we can identify the software components that have to play together to optimally support the application developer and to most efficiently utilise the given resources of an MSA system. Implicitly, they also define the relevant interfaces in between the connected components. A more

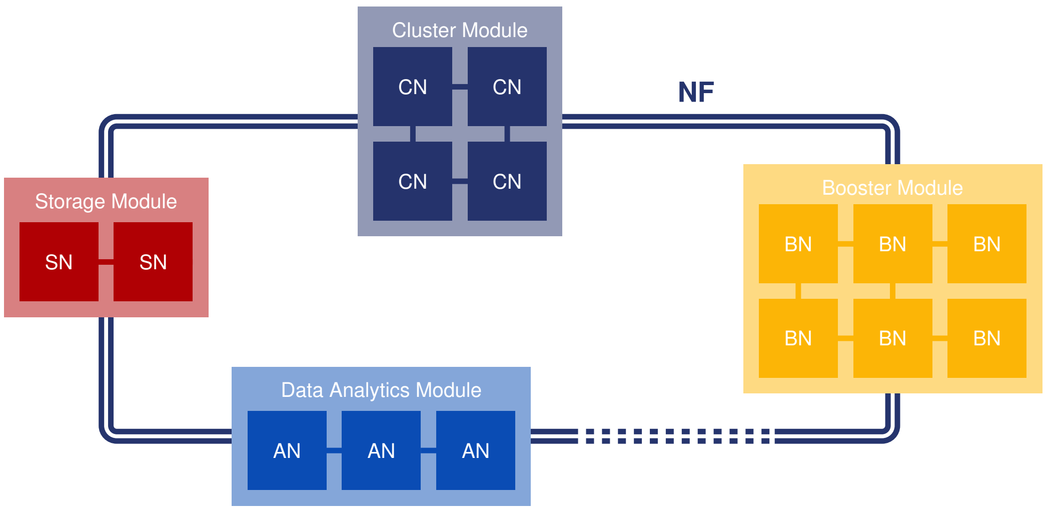


Figure 1: An MSA system consisting of CNs (Cluster Nodes), BNs (Booster Nodes), ANs (Data Analytics Nodes), and SNs (Storage Nodes) within distinct modules. The modules are connected through a Network Federation. (based on [1])

detailed definition of these interfaces will be presented in future deliverable of this project (D 2.1, D 4.1, and D 5.1). These deliverables will also present refined work-plans for the various tasks of WP 2, WP 3, WP 4, and WP 5.

An optimisation cycle is in-between a procedure model of an application developer in HPC and an automatised tool workflow. It might contain manual steps and therefore commonly the human developer is in the loop. Nevertheless, some steps might be fully automatised, and the optimisation cycle will identify the relevant interfaces. On the long run, the target will be to automatise the optimisation cycles as much as possible by introducing advanced techniques such as machine learning or AI concepts. However, most of this will be beyond the scope of this project.

Optimisation cycles might describe one time actions, but usually they describe an iterative procedure, e. g., the resulting application will be executed and benchmarked repeatedly while varying certain parameters, data distribution, utilisation of different memory types, etc. Especially for iterative procedures, a fully automatised implementation of an optimisation cycle will provide major improvement of the application developers productivity.

The description of the interfaces between the components within a cycle will be provided in different levels of detail depending on the individual optimisation cycle. It will always contain the type of information to be passed from one component to the next, but the description of the actual format might still be vague. This is due to the fact that for some components the actual requirements are not yet fully worked out. Both, the identification of the requirements and the interfaces deduced from them will be elaborated in the next months in the different tasks and documented in the next series of deliverables at M 12.

To summarise, optimisation cycles are a vehicle to identify the interplay of the various (software-)components to be utilised in the project. This way, they provide an implicit definition of the overall architecture of the DEEP-SEA software stack.

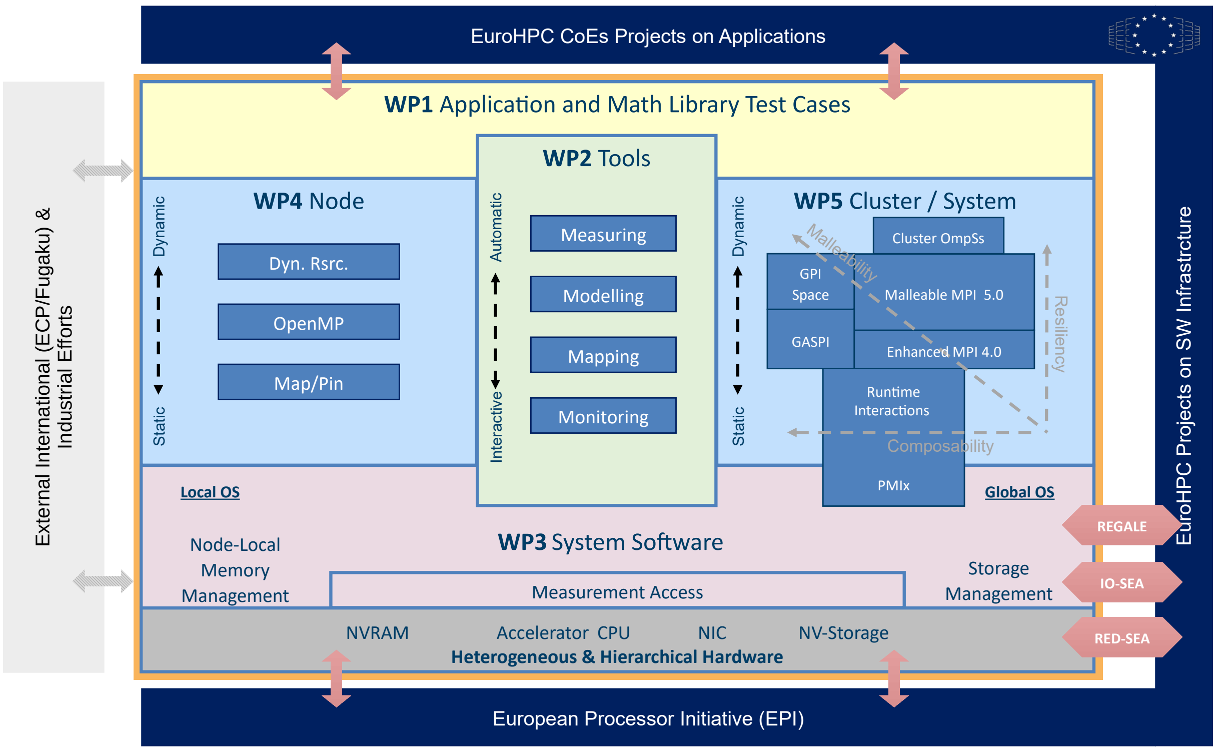


Figure 2: The high-level overview of the DEEP-SEA software stack. It shows both the main aspects covered by each work package and their relation.

3 Components

As stated previously, the DEEP-SEA software stack comprises all software components required to meet the project's vision of the European software stack for Exascale systems. This chapter introduces the central components that will be enhanced and extended within this project while covering the following major technology areas: performance analysis (Sect. 3.1), memory management (Sect. 3.2), communication and programming models (Sect. 3.3), resource management and scheduling (Sect. 3.4), and programmer productivity (Sect. 3.5). As mentioned before, all components that will be used as is (e. g., the operating system or compilers) are ignored throughout this presentation.

3.1 Performance Analysis

Performance analysis is an important part in optimising the performance of any application or system. This section describes the different components that will be developed or extended in the DEEP-SEA project to analyse performance. This comprises components that focus on the analysis of application behaviour, on the monitoring and analysis of system performance and energy efficiency, and on the mapping of applications to systems.

3.1.1 Application Analysis and Mapping

This section contains the components that are mainly concerned with optimising individual applications, including the mapping of an application to a system. It is structured into two parts, the first one describes the components which mainly analyse an application's communication and computational performance. The second part describes the components that focus on the analysis of memory behaviour, memory transfers, and processing in memory.

Analysis of Computation and Communication

A comprehensive analysis of the computational behaviour as well as the analysis of the communication patterns used by an application is crucial for achieving scalability on supercomputers. Therefore, the DEEP-SEA software stack enables the collection of performance data via Score-P, allowing various tools to analyse the collected data. These tools target different analysis aspects when running HPC codes on modular hardware and are being introduced in the following.

Score-P [2, 3] is a community-maintained instrumentation and measurement infrastructure to collect performance data from HPC applications. It is available as open-source under the 3-clause BSD license. Score-P is easy to use, highly scalable, and able to generate both summarising call-path profiles and detailed event traces. By using open data formats—CUBE4 [4] for profile data and the Open Trace Format 2 (OTF2) [5] for event traces—Score-P provides the foundation for a number of well-established performance analysis tools. In particular, Score-P's event traces can be manually examined using trace visualisers [6, 7], or automatically analysed using the Scalasca Trace Tools (see below). Likewise, the generated call-path profiles can be explored using the Cube performance report explorer [8] as well as TAU [9], including cross-experiment analyses using a performance database. In addition, they serve as input for generating empirical performance models with Extra-P (see below).

To capture details of the application execution, Score-P mainly relies on instrumentation, i. e., the insertion of “hooks” into the application code that call into the Score-P run-time libraries at important points during the execution. This instrumentation can be added to the application executables using compiler flags and/or standardised interfaces such as PMPI, the OpenMP, the Open Computing Language (OpenCL), and the OpenACC tools interfaces, as well as the CUDA Profiling Tools Interface (CUPTI) or even source-to-source translation. Score-P also provides an instrumentation API for manually annotating the source code in case the automatically added instrumentation is not adequate as well as an API instrumentation wrapper generator for third-party C/C++ libraries used by the application. At run-time, the inserted hooks trigger callbacks in the Score-P measurement libraries. The measurement system processes these events by querying the current high-resolution timestamp, collecting event-specific data such as the number of bytes transferred, and optionally retrieving hardware performance counter data (e. g., via perf or Performance Application Programming Interface (PAPI) [10]). Depending on the configured measurement mode, this data is then either summarised in a call-path profile (default) or stored in a memory buffer accumulating the event trace. Finally, the measured data is flushed to disk for further analysis with the aforementioned tools.

While Score-P can already be used to analyse the performance of many applications running on MSA systems, users often have to apply workarounds to bypass current limitations—which come along with various drawbacks. These will be addressed within DEEP-SEA to improve both the ease of use and the general applicability of Score-P. For example, we will remove the restriction that the Score-P measurement libraries currently expect a homogeneous use of parallel programming models in MPMD applications. Likewise, we will allow using different hardware performance counters and low-overhead timestamp counter register timers running at different “ticks per second” rates when using multiple MSA modules.

The Scalasca Trace Tools [11, 12] are a collection of trace-based performance analysis tools built on top of the Score-P instrumentation and measurement infrastructure introduced above. They are also available as open-source under the 3-clause BSD license and have been specifically designed for use on large-scale HPC systems. A distinctive feature of the Scalasca Trace Tools is its scalable automatic trace-analysis component, which provides the ability to identify wait states that occur, for example, as a result of unevenly distributed workloads [13]. Besides merely identifying and quantifying wait states in communication and synchronisation operations, the trace analyser is also able to pinpoint their root causes as well as their impact [14]. In addition, the analyser can identify the activities on the critical path of the target application [15], highlighting those routines which constitute the best candidates for optimisation.

To enable the analysis of huge amounts of OTF2 event trace data that can be produced at large scales, the Scalasca Trace Tools are designed as parallel programs requiring the same amount of resources (i. e., the number of processes and threads) as the target application. The analysis tools employ a parallel replay technique that re-enacts the communication and synchronisation operations performed by the target application using operations of similar type. This effectively exploits the memory and processing capabilities of the HPC system, and thus, is the key for achieving scalability. Finally, the result is written to disk as an enriched call-path profile in CUBE4 format including additional higher-level metrics, which can be examined using the same tools (i. e., Cube, TAU ParaProf/PerfExplorer, and Extra-P) than the run-time profiles produced by Score-P.

For the Scalasca Trace Tools, the objective for DEEP-SEA is to increase the MSA awareness of its analysis capabilities. In particular, we will extend the analysis to distinguish between intra- and inter-module wait states for MSA experiments, which will allow for a more in-depth understanding of the application behaviour and can provide guidelines for an improved resource distribution. To enable this, the Score-P measurement system needs to interact with the MSA-aware MPI implementation ParaStation MPI and/or the resource manager to query the module each application process is running on, and write it into the generated trace data. Finally, since MPI inter-communicators are more frequently used in MPMD and MSA settings, we plan to implement full support for inter-communicators throughout the whole toolchain, including measurement support in Score-P, extensions to the OTF2 trace file format, and extended analyses in the Scalasca Trace Tools.

Extra-P [16, 17] is an open-source tool available under the 3-clause BSD license for an automatic performance-modelling to support the user in the identification of program parts that scale worse than expected. It relies on measurements of various performance metrics at different execution configurations to create performance models.[1] All it takes to search for scalability issues, even in full-blown codes, is to run five to six small-scale performance experiments[2] per modelled parameter, launch Extra-P, and compare the asymptotic or extrapolated performance of the worst instances to the user's expectations. The easiest way to gather the measurements for this is using Score-P because Extra-P has built-in support for the call-path profiles of Score-P and the CUBE4 format. The measurements can also be taken with any other tool and manually converted in an Extra-P compatible file format, such as JSON or plain-text.[3]

Extra-P does not only generate a list of potential scalability issues but also human-readable models for all available performance metrics, e. g., floating-point operations or bytes sent by MPI calls. These models can be used to answer questions such as: How many MPI processes are approximately needed to finish a specific problem of a particular size in a fixed time frame? This can already be answered by Extra-P, however the shift to heterogeneous programming models raises new questions: Is offloading a specific computation to the GPU worth the effort or does the additionally needed data transfer harm the overall performance?

To answer these new questions, we will, as part of the DEEP-SEA project, work together with the Score-P developers to improve the integration of measurements on GPUs and add support for creating GPU performance models to Extra-P. We will also allow inputting GPU measurements using other formats so that users are not limited to Score-P for gathering measurements from GPUs.

Based on the performance modelling, we will additionally create a Mapping Toolchain (cf. Sect. 4.3) that uses Extra-P at its core to determine a suitable mapping of applications onto MSA systems. To achieve that, we will try to incorporate platform-independent or execution-device–independent metrics such as the number of floating-point operations. We will use a description of the system and

[1] A performance model is a formula expressing a performance metric of interest, e. g., execution time or energy consumption as a function of one or more execution parameters such as the size of the input problem or the number of processors.

[2] It is recommended that the measurements for these experiments are repeated at least five times, to reduce variability.

[3] More information about the file formats and the measurement requirements is available in the Extra-P repository [17].

its hardware as well as information on previous runs of the same application or applications with similar performance behaviour to assist the user/runtime in optimising the fit between the application and the underlying system.

Analysis of Memory

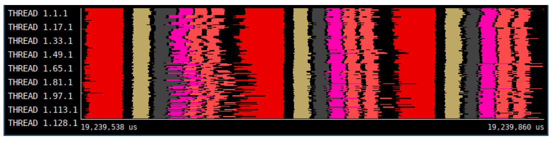
The performance of many scientific applications is limited by their memory utilisation due to bandwidth and latency constraints. Therefore, application optimisation concerning their memory access behaviour is crucial for achieving good scalability, especially with respect to the advent of heterogeneous memory and the use of distributed memory. This optimisation process is supported by the components being introduced in the following sections.

RMA-Analyzer is a new plug-in for the **PARCOACH** library [18] based on a recent initial study [19] which aims at helping developers debugging their MPI programs. MPI-RMA is a well-known distributed programming paradigm based on one-sided communications. Its properties allow for a greater asynchronicity and computation/communication overlap than traditional message passing mechanisms. Each process explicitly exposes an area of its local memory as accessible to other processes to provide asynchronous, one-sided reads, writes, and updates. While MPI-RMA is expected to greatly enhance the performance and to permit efficient implementations on multiple platforms, it also comes with several challenges concerning memory consistency. Developers need to handle complex memory consistency models and complex programming semantics.

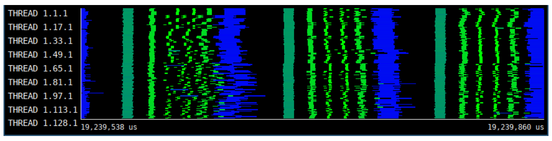
In the DEEP-SEA project, we will develop a new plug-in for the PARCOACH library called RMA-Analyzer, dedicated to the analysis of MPI-RMA programs (cf. Sect. 4.11). This plug-in enables application developers to analyse memory consistency errors (also known as data races) during MPI-RMA program executions. We will also support the notified RMA feature provided by Tk5.1.

Paraver is a very flexible data browser [20, 21] that is part of the Barcelona Supercomputing Centre (BSC)-Tools toolkit [22], available open-source under GNU LGPL v2.1 license. Paraver enables the analyst to create time-line views, profiles, and histograms (cf. Fig. 3) while displaying a huge number of metrics based on the available data. The metrics in Paraver are customisable by the user based on a filter module, a large set of time functions, and a mechanism to combine multiple timelines. The statistics can be computed over any selected region and correlate the information with up to three different time functions. To capture the expert's knowledge, any view or set of views can be saved as a Paraver configuration file that can be simply reloaded to repeat the analysis, to conduct multi-experiment comparisons, or to apply the same views to different applications.

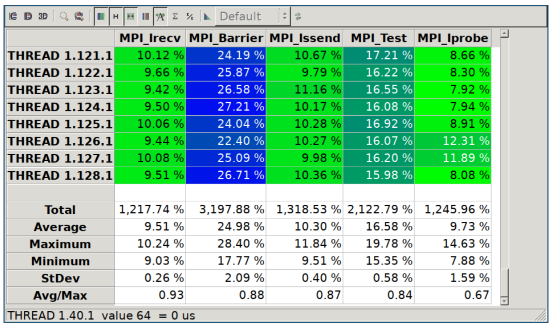
The input data format for Paraver is a timestamped trace of events, states, and communications. This is an open format that is very simple to generate by other tools, following the guidelines described in the Paraver trace-file description manual [23], to take advantage of the powerful data navigation capabilities of Paraver. For parallel applications, the traces are usually generated by Extrae, the instrumentation package included in the BSC-Tools. A common optimisation cycle with Paraver also includes Dimemas [24] (cf. Sect. 4.9), an MPI network simulator that replays a Paraver trace and predicts the behaviour with respect to different network characteristics to be compared with the original run. The key features of Paraver can be summarised as follows:



(a)



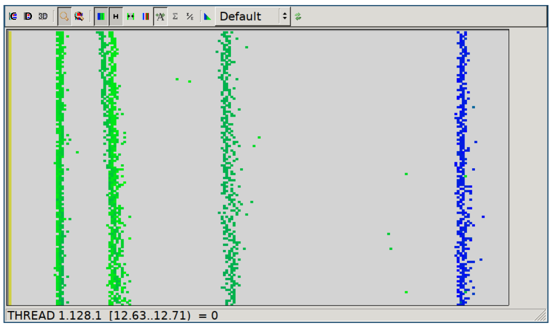
(b)



	MPI_Irecv	MPI_Barrier	MPI_Issend	MPI_Test	MPI_Iprobe
THREAD 1.121.1	10.12 %	24.19 %	10.67 %	17.21 %	8.66 %
THREAD 1.122.1	9.66 %	25.87 %	9.79 %	16.22 %	8.30 %
THREAD 1.123.1	9.42 %	26.58 %	11.16 %	16.55 %	7.92 %
THREAD 1.124.1	9.50 %	27.21 %	10.17 %	16.08 %	7.94 %
THREAD 1.125.1	10.06 %	24.04 %	10.28 %	16.92 %	8.91 %
THREAD 1.126.1	9.44 %	22.40 %	10.27 %	16.07 %	12.31 %
THREAD 1.127.1	10.08 %	25.09 %	9.98 %	16.20 %	11.89 %
THREAD 1.128.1	9.51 %	26.71 %	10.36 %	15.98 %	8.08 %
Total	1,217.74 %	3,197.88 %	1,318.53 %	2,122.79 %	1,245.96 %
Average	9.51 %	24.98 %	10.30 %	16.58 %	9.73 %
Maximum	10.24 %	28.40 %	11.84 %	19.78 %	14.63 %
Minimum	9.03 %	17.77 %	9.51 %	15.35 %	7.88 %
StDev	0.26 %	2.09 %	0.40 %	0.58 %	1.59 %
Avg/Max	0.93	0.88	0.87	0.84	0.67

THREAD 1.40.1 value 64 = 0 us

(c)



THREAD 1.128.1 [12.63..12.71) = 0

(d)

Figure 3: Different views in the Paraver data browser: (a) MPI calls timeline, (b) computation duration timeline, (c) MPI calls profile, and (d) histogram of the computation duration with average Instructions Per Cycle (IPC).

- Detailed quantitative analysis of program performance
- Concurrent comparative analysis of several traces
- Highly customisable visualisation options
- Building of derived metrics combining multiple views
- Cooperative work, save your current session to continue later or share views with others

Extrae is the instrumentation package [25] from the BSC-Tools that transparently collects performance data during the program execution and generates trace files for Paraver, and is also available open-source under GNU LGPL v2.1 license. Extrae can use several mechanisms to capture the information, ranging from static linking to dynamic binary instrumentation of unmodified executables. The most straightforward and preferred methods is the `LD_PRELOAD` interposition for an interception of production binaries at loading time.

The data collected by Extrae includes entry and exit to the programming model runtime, hardware counters through the PAPI library, call-stack references, periodic samples, system calls, user functions, and punctual user events. Focusing on the activity of the parallel runtime guarantees a minimal overhead in most scenarios, given that the application's use of the parallel runtime is not excessively fine-grained.

Extrae already supports common programming models, namely MPI, OpenMP, pthreads, OmpSs, OpenCL, and Compute Unified Device Architecture (CUDA), and keeps adding new interfaces such as OpenACC and Global Address Space Programming Interface (GASPI). It supports programs written in C, Fortran, Java, and Python, as well as combinations of different languages, hybrid and modular codes. It is available for most UNIX-based operating systems, and has been deployed in all relevant HPC architectures and platforms, including x86-64, ARM, ARM64, POWER, RISC-V, SPARC64, BlueGene, Cray, and many HPC accelerators.

PROFET (PROFiling-based EsTimation of performance and energy) [26] is an analytical model that predicts how an application's performance, power and energy consumption would change when it is executed on a new memory system. The method is based on the instrumentation of an application running on actual hardware. The application instrumentation is currently done with the LIKWID performance tools. This way, it already takes microarchitectural details of the CPU into account such as the real (and not publicly disclosed) data prefetcher and out-of-order engine.

The PROFET model is originally evaluated on two actual platforms: Sandy Bridge-EP E5-2670 with four Dynamic Random Access Memory (DRAM) configurations DDR3-800/1066/1333/1600, and Knights Landing (KNL) Xeon Phi with DDR4 and 3D-stacked MCDRAM. In the context of the DEEP-SEA project, PROFET will be enhanced to model high-end HPC servers (e. g., Intel Cascade architectures) connected to DDR4 and Optane DIMMs. The model accuracy will be evaluated by comparing to measurements on actual hardware platforms.

The MUlti-level Simulation Approach (MUSA) [27] is a simulation methodology that uses traces to enable large-scale simulations with different communication networks, numbers of cores per node, and microarchitectural parameters. MUSA can simulate both intra-node and inter-node behaviour and employs two main components:

- a tracing infrastructure that captures communication, computation, and runtime system events; and
- a simulation infrastructure that uses these traces for simulation at multiple levels (e. g., CPU, node, network, scheduling, and synchronisation).

HPC applications stress a system at multiple levels, including both the hardware and the software. Using a single simulation approach across all levels would be too rigid to adapt to the degree of detail appropriate for each level. For this reason, MUSA's simulation infrastructure is capable of changing the level of simulation detail, from cycle-accurate microarchitectural simulations to high-level analytical models. The methodology enables the combination of detailed (higher computational cost) and high-level (higher simulation speed) simulations, enabling simulation of large-scale machines with thousands of cores in a reasonable amount of computational time, while guaranteeing a high degree of accuracy. BSC's MUSA implementation integrates two different simulators; Dimemas [24] for simulating MPI/network and TaskSim [28] architectural simulator for simulating OpenMP/OmpSs events and detailed instructions.

MUSA's underlying simulator infrastructure can perform simulations either in burst or detailed mode, which allow from faster than native simulation speeds to slower but more detailed design space exploration studies respectively. MUSA also relies on analytical methods such as sampling for an identification of representative code segments/phases. At node level, MUSA needs to select which MPI ranks will be simulated in detailed mode. Periodic sampling is the default policy, which means that one out of every N ranks is chosen. At computation level, sampling is used to find iterative patterns and synchronisation points. These representative segments are to be simulated in detailed mode and their results are used to fast-forward other phases while the remainder of the application is simulated in burst mode. MUSA can help system designers to assess the usefulness of future technologies in next-generation HPC machines, at a fraction of native execution time (with an error of 10 % in the common case).

The target HPC systems of the DEEP-SEA project, such as the EPI, feature vector processors and accelerators. Irregular memory access patterns (e. g., strides) of vector load instructions, can have a significant impact on performance. Traditional memory prefetchers may be ineffective and alternatives such as cache bypassing may be preferable. Moreover, accelerators may have their own memory subsystem, creating an ecosystem of different memory technologies with heterogeneous latency and bandwidth specifications. Therefore, applications can have radically different performance on these architectures, compared to conventional scalar systems with uniform shared memories. Both the software and the hardware have very large parameter spaces that can be tuned to optimise performance (e. g., task scheduling and load balancing, memory allocation, vector length management, etc.). Exploring all these different configurations is very time and resource consuming. MUSA adds a simulation-based methodology to the DEEP-SEA software stack which allows for faster than native execution simulations, suitable for tuning large parameter spaces. In the context of the DEEP-SEA project, we plan to extend MUSA's vector engine and add support for heterogeneous memory. With these two new subsystem models implemented, MUSA users can both evaluate their application implementation on a given architecture and/or guide the architectural parameters of a system in design (cf. Sect. 4.9).

Simulation of processing in memory (PIM) will be performed with the enhanced ZSim CPU simulator [29]. Initially, ZSim was developed to simulate the Intel Westmere microarchitecture (released in 2008), and it was recently upgraded by the BSC to simulate the Skylake microarchitecture. To accurately model Intel Xeon Platinum 8160 processor, we performed significant enhancements in the ZSim pipeline, cache hierarchy, and main memory interface. For a detailed and accurate main memory simulation, we integrated ZSim with the DRAMsim3 [30] cycle accurate main memory simulator. DRAMsim3 models DDR4, LPDDR4, GDDR6, HBM, and STT-MRAM standards; its DDR4 timings are validated against manufacturer Verilog models [30].

The simulation infrastructure is evaluated against the actual hardware comprising Intel Xeon Platinum 8160 processor connected to six DDR4-2666 channels. The CPU pipeline is validated by 407 synthetic benchmarks, covering almost all instructions included in the ISA of the Intel Xeon Platinum 8160 processor. Different version of the synthetic benchmarks test in-order and out-of-order execution. Cache hierarchy and main memory latency is validated with *lmbench*. Finally, the infrastructure is evaluated by using SPEC CPU2006 and multithreaded SPLASH-2 benchmarks.

MemAxes is a tool for analysis and visualisation of memory movements within a system with the goal to provide hints to developers on a suboptimal memory access behaviour of their application. MemAxes works closely together with the Mitos project [31]; Mitos is present during the application runtime and collects the application's data regarding data movements in the system, and MemAxes uses the collected data and analyses and visualises them.

The collected data on the CPUs are memory samples. Mitos configures the CPU to (randomly) select and capture memory operations, and stores the observed values/properties, such as load latency, issuing core, virtual address of the instruction that triggered the memory operation, or the information about which cache level satisfied the operation. The address information is used to link the memory operations with the application source code, i. e., to determine which line of code and/or variable is responsible for a particular memory operation. Mitos currently only works on Intel CPUs as it relies on Intel-specific PEBS (Processor Event-Based Sampling) sampling. AMD CPUs offer so-called IBS (Instruction Based Sampling), which may enable extending Mitos's support to AMD CPUs. The support for non-Intel CPUs will be further researched.

Apart from that, we also plan to analyse possibilities of collecting complementary data to memory samples on CPUs. Other kind of information, such as aggregate data regarding overall cache and memory usage, could complement the original fine-grained memory sample mechanism. Such additional information could provide useful information to the developer regarding overall resource limits.

The MemAxes tool visualises the collected data enabling the developer to identify potential bottlenecks with respect to memory operations in their application code. The data is analysed in multiple ways resulting in different views on the raw data and the hardware presented by MemAxes. The most prominent view maps the collected samples on the hardware (cores, caches, NUMA regions). Other views show the origin of the samples in the source code or their distribution in time or across the address space, to name a few examples. The views and the dataset can be adjusted (filtered), so that the developer gets the desired set of data that stresses the suboptimal behaviour. The mapping between the memory samples and the source code assists the developer in identifying the critical source code sections that need to be adjusted.

In the scope of the DEEP-SEA project, we plan to review the current concepts and the implementation of MemAxes, and to extend its support towards modern heterogeneous hardware systems that contain multiple CPUs and GPUs. At the beginning of the project, MemAxes was still in an early stage of development, and not all of its functionality was stable or working at all. Therefore, some parts had to be refactored or reimplemented in the first months of this project, and further functionalities will be updated in the future as well. These issues include, for example, not working source code attribution or proper setting up of memory sampling procedure for different hardware architectures. The most important issues were already solved, so that MemAxes can be further extended.

MemAxes will be extended to support interactive modelling that enables changing application and/or system parameters. The performance analysis will be updated to present the expected behaviour for such a configuration. Technical details of this approach were not analysed yet, and will be a part of future documentation.

3.1.2 System Analysis and Monitoring

This section introduces the system-wide components to monitor, analyse, and optimise the performance and efficiency of the applications on the system. Furthermore, these components provide the basis to detect potential issues in both applications and the system.

LLView [32] is a set of software components being actively developed by JSC. It enables the monitoring of the clusters' utilisation that are controlled by a resource manager and a scheduler system. Furthermore, it establishes a link between performance metrics and individual jobs to provide a job reporting interface.

To avoid any job-specific instrumentation, LLView aggregates metrics of different existing sources such as information provided by the resource manager, available daemons, or other metrics which are gathered on a system-wide scale. This way, LLView neither has a measurable impact on the individual application performance nor conflicts with user-side instrumentation. At JSC, the LLView-based job reporting is automatically active for all jobs, on all larger systems, with no limitation on the job size. The metrics, such as load, memory consumption, I/O, GPU, or network utilisation are gathered once per minute.

LLView provides two different interfaces towards the end user: (1) a Perl-based, configurable client covering a global view of the current scheduler and resource situation on a specific system; and (2) a web client offering access to the gathered job metric information. Here, all metrics are pre-processed and stored for three weeks on the web server, while the visualisation is done by the web client. The platform enables a role-based access to restrict users to their own jobs while also allowing project coordinators or site supports to access job reports of their user groups. Finally, for each job a job report in PDF format covering a visual representation of all gathered metrics over time is created and available for download.

In DEEP-SEA, the capabilities of LLView are improved by connecting it to the DCDB-based monitoring infrastructure on the DEEP-SEA prototype system, providing access to new metrics in the job reports such as performance counters (e. g., cache misses) or energy information of different components. In addition, the modular system approach is taken into account by developing a mechanism for the connection and visualisation of modular jobs in the system. Currently, each individual job part utilising its own part of the modular system is listed separately by LLview, but no automatic connection is generated to link all these parts together within the web frontend or the PDF reports.

Bull Dynamic Performance Optimizer (BDPO) [33] is a lightweight system-wide tool minimising the energy consumption in HPC computing environments. BDPO is intended to be deployed on each compute node in a cluster. The idea behind it is to leverage the power/performance control knobs exposed by the hardware, and adjust them dynamically according to the application characteristics and phases. As the process of code optimisation and tuning is tedious and complex, most of scientific applications are lacking energy-optimisation awareness. Consequently, naive execution of these applications ultimately leads to poor energy efficiency. To overcome this situation, BDPO aims at providing a simple yet effective method to control energy consumption across the entire cluster.

BDPO runs in the background on each compute node involved in the execution of the considered application. It can be started either manually by system administrators or by cluster users through an interface built using Slurm SPANK Plug-ins. BDPO neither requires any source code modification or annotations nor any prior knowledge of the target applications. It is a reactive Dynamic Voltage and Frequency Scaling (DVFS) controller that minimises energy consumption while keeping performance degradation under control. The energy consumption optimisation achieved by BDPO is twofold: On the one hand, it can optimise the energy efficiency associated with the computational part of HPC codes by monitoring key CPU-centric metrics (thanks to hardware performance counters such as the number of instructions executed per cycle and the L2 to L3 cache traffic) to enforce DVFS during memory-bound phases. On the other hand, BDPO is able to optimise the energy consumption in MPI waiting phases related to collective routines.

In the context of the DEEP-SEA project, we will study the impact of DVFS reconfigurations from the performance and energy consumption savings points of view on various hardware platforms: Intel, AMD, ARM/EPI, and to some extent GPUs. On top of that, we will extend the rule-based decision engine of BDPO, notably to enhance and refine its optimisation capabilities and implement auto-evaluation algorithms to avoid inefficient reconfiguration actions.

The Data Center Data Base (DCDB) [34] is a holistic, modular, and scalable monitoring solution for HPC systems developed by LRZ. One of the design goals with DCDB has been to provide a holistic overview over the HPC operations of a data centre—not just the IT hardware, but also the supporting infrastructure, the system software, and the applications running on the system. DCDB's extensible plug-in infrastructure enables an easy integration with new system components and its modular architecture ensures horizontal scalability (i. e., in terms of storage capacity) as well as vertical scalability (i. e., in terms of system size).

DCDB is currently deployed on the DEEP prototype systems and provides access to a wide range of monitoring data: system load, memory consumption, performance counters, temperatures, power consumption, network traffic, and filesystem operations. A light-weight daemon is running on each compute node and collects in-band monitoring data. Out-of-band monitoring data is collected by the same daemon running on management servers that provide access to infrastructure data.

Access to DCDB's monitoring data is provided via a Grafana frontend that allows for an easy visualisation with a well-established tool. Data can also be queried via command line tools and a shared library that enables the integration with third-party software. The collected data can be accessed on a per-node basis as well as on a per-job basis where per-node metrics are aggregated over all nodes of a job.

In DEEP-SEA, DCDB will be integrated with LLView to make its monitoring data also available as part of the job reports. Furthermore, DCDB will be extended to provide better insights into the applications running on the system.

The Malleable Online Monitor to assist with malleability scheduling decisions will be developed alongside the malleability prototype. The malleability prototype will be designed around feedback and control mechanisms, to be defined in detail later in the project. The general scheme for interaction with the rest of the system is outlined in the Malleable Online Monitor optimisation cycle (cf. Sect. 4.4). This Malleable Online Monitor represents a core part of the feedback mechanism. Network traffic metrics will be monitored, such as network time, message counts and byte counts, in the initial prototype. The selection of metrics to capture will be made during development, based on the selected performance models and quality to overhead trade-offs. The monitor will be able to adapt to node allocation changes of malleable applications and adjust its reports accordingly.

The data will be collected at the process level, and aggregated hierarchically through the network of daemons until it reaches the scheduler at the Slurm controller daemon. The data will be collected periodically and fed by the scheduler to a performance model. The output of the model will be used by the scheduler when making malleable decisions. An initial plan for its relevant optimisation cycle is described in Section 4.4.

3.2 Memory Management

The management and use of deep and heterogeneous memory hierarchies will be facilitated by different node-level memory allocation APIs. These are divided into low-level APIs performing the actual resource acquisition on the hardware devices and high-level APIs facilitating the access to heterogeneous memory systems from the programming models.

3.2.1 Intra-node Memory Heterogeneity

BSC is providing a software ecosystem for an automatic data placement in heterogeneous memory systems comprising three main components (cf. Fig. 4). The Extrae profiler is used to extract information on how the different memory objects are accessed throughout the lifetime of the application execution. Paramedir is a post-processing tool from the Extrae package. The Heterogeneous Memory Advisor (hmem_advisor) processes the offline profile and outputs an optimised distribution for the target system. During runtime, the Flexible Memory Allocator (FLEXMALLOC) memory allocation interposer intercepts regular allocation calls and translates them to allocators specific to the respective memory subsystem. As part of the DEEP-SEA project, BSC is extending this software ecosystem by a source-to-source compiler called HMem-S2S. This will constitute an alternative to FLEXMALLOC to resolve translation of memory allocation calls during compile time. This way, runtime overhead can be alleviated in specific cases involving many allocation calls.

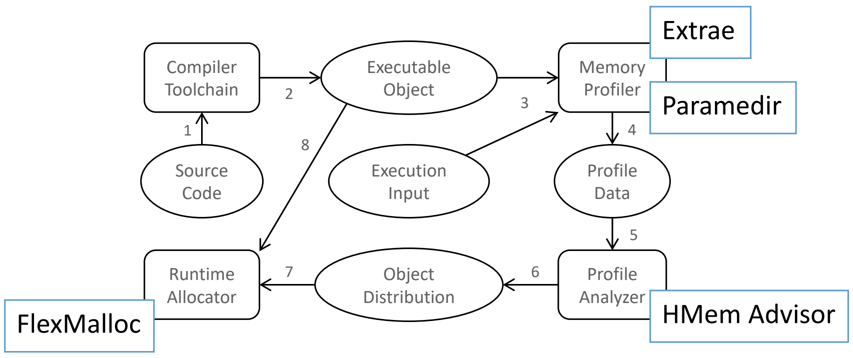


Figure 4: Automatic, programming-model agnostic data placement framework for heterogeneous memory systems.

Low-level memory allocation API To enable the programmatic discovery and utilisation of heterogeneous memory resources within an HPC node, DEEP-SEA is developing a memory allocation framework derived from Simplified Interface to Complex Memory (SICM), which is part of the software stack of the US Exascale project [35, 36]. A user-level memory allocation API will be developed as part of task Tk4.1, allowing the application or higher software layers to determine the configuration and status of the node's memory hierarchy, i. e., enumerate the available memory classes and their sizes, and then allocate and free regions of memory.

The Multi-Processor Computing (MPC) NUMA-aware allocator enables the management of DRAM memory in MPC. It is a standalone module and part of the MPC framework, that is used internally in MPC but can also be used by other software components. In DEEP-SEA, its memory allocator will be enhanced to allocate and free in heterogeneous memories such as HBM and PMEM.

High-level memory allocation API Although the low-level memory allocation API (forked from the corresponding low-level API of SICM) can be called directly from the application, it is expected to be invoked via higher-level interfaces, which support the allocation of memory subject to generalised requirements (e. g., non-volatile, high bandwidth, etc.), enabling an automatic management of memory hierarchy tiers. The OmpSs-2 programming model will be extended by a user-friendly API based on the low-level memory allocation APIs, to provide hints about memory properties and data structure placement on the memory hierarchy (e. g., NUMA, HBM, pinned memory, huge-pages, etc.). Of particular interest is the adaptation of the OpenMP 5.0 memory management directives to take advantage of heterogeneous memory resources. These directives are also of interest for the MPC Allocator module, that will handle data allocations on heterogeneous memories within OpenMP, as well as for the SICM high-level API. Finally, standard `malloc/free` will be supported by translating such calls to the low-level memory allocation APIs, with automatic data placement to different memories (via the hmem_advisor component as well as by a transparent page migration mechanism). For high-level programming environments, utilising heterogeneous memory resources can start by identifying allocation sites (e. g., in NumPy and DaCe), and then considering the potential benefit from direct calls to the respective low-level memory allocation API. Moreover, the memory allocation APIs can be utilised in the context of a memory-based optimisation cycle (cf. Sect. 4.5), by developing a sampling-based mechanism in the Linux kernel for tracking accesses in selected memory regions. This mechanism will provide a heatmap-style view of accesses, and will be utilised in the context of machines of heterogeneous memory devices as input for an online controller (to be developed within WP 4) that transparently migrates pages across memory device classes to improve overall performance (cf. migration action in Figure 17.

TensorFlow data placement BSC is working towards a heterogeneous memory-aware TensorFlow implementation. Two approaches are being explored in parallel: On the one hand, BSC is adapting its domain-agnostic software ecosystem described before (cf. Fig. 4) which, in turn, will explore the utilisation of the DEEP-SEA low-level APIs for memory allocation. Challenges include (1) just-in-time generated code; (2) the use of Python, which comprises among others the existence of large numbers of idle threads, forked processes, and deep software layers involving different programming languages; and (3) overwhelmingly long executions and large datasets. On the other hand, BSC is working on a domain-specific approach based on the existing Sentinel [37]. The aim is to compare in-depth both approaches and, if possible, obtain the best of the two of them.

3.2.2 Interfaces

DEEP-SEA will support the standard memory allocation calls (`malloc`, `realloc`, etc.), as well as the standard OpenMP 5.0 memory allocation directives, which applications may use for some or all of their memory allocations. The standard memory allocation calls will be translated via FLEXMALLOC and HMem-S2S to a variety of underlying memory allocation APIs. Currently, `memkind` and `numa_malloc` are included, and during the project the use of DEEP-SEA's own memory management APIs will be explored. At runtime, the operating system may also perform live migrations to optimise data placements in a complementary approach.

A specialised approach is taken for OmpSs-2 applications, for which additional information is available via integration with the Nanos6 runtime system and its dependency system. This information can be leveraged via the new OmpSs-2 memory allocation and distribution API.

At the lowest level is the low-level memory allocation API, which is intended to be used by the higher level components to allocate memory on the system.

3.3 Communication and Programming Models

Data exchange within a module or across modules of an MSA system requires communication among the processes of a parallel application, with the respective communication facilities commonly being part of the programming model. The DEEP-SEA software stack provides different programming models for module-level and system-level programming, which, in turn, rely on low-level communication libraries as introduced in the following section. The programming models themselves are then presented in Section 3.3.2, followed by a description of SIONlib enabling efficient file I/O for large-scale, parallel applications (cf. Sect. 3.3.3).

3.3.1 Low-level Communication

Low-level communication in the DEEP-SEA software stack relies on two main components: the MPC *lowcomm module* and the *pscom* library. The MPC lowcomm module is part of the MPC framework but can also be used as a stand-alone module for third party tools. The pscom is a library that tightly integrates into the ParaStation MPI library by implementing the ADI3 interface. They are both especially designed for HPC systems and target high-performance communication. They both already ship with a variety of plugins supporting different interconnects and interfaces relevant to the HPC domain, e. g., for pscom, InfiniBand (IB) [38], UCX, Extoll [39], and Omni-Path [40], and for MPC lowcomm module, IB and the BullSequana eXascale Interconnect (BXI) via the Portals 4 API.

The MPC lowcomm module will enable MPI communication via MPC, and also via Open MPI thanks to the implementation of a new PML MPC component. This PML MPC component will handle internal point-to-point communications in Open MPI and deliver them through the MPC lowcomm module, enabling gateway support for both point-to-point and collective communication. The pscom will enable both MPI communication via ParaStation MPI (cf. Sect. 3.5) and single address space programming via GASPI (cf. Sect. 3.3.2). The pscom itself is divided into different software layers (cf. Fig. 5): The hardware-independent layer facilitates the session management by enabling the establishment of bi-directional connections between different pscom processes. The hardware-dependent layer features a modular design that supports different communication interfaces and protocols by means of *plugins*.

The session management of the pscom behaves similar to the Berkeley Socket API and distinguishes the active *connecting* process and the passive *listening* process. Initially, both processes set up a TCP connection that is used for negotiating the actual transport by choosing the appropriate

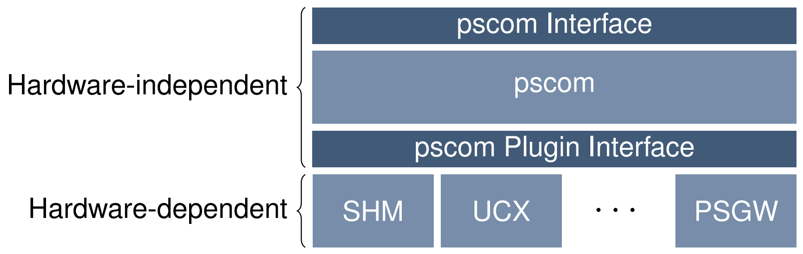


Figure 5: The layered architecture of the pscom library featuring hardware-independent and hardware-dependent layers.

communication plugin. In accordance with this session management model, each connection is established explicitly and asymmetrically by means of the `listen` and `connect` calls offered by the pscom API. This way, a fully connected graph among all processes in the initial `MPI_COMM_WORLD` group of an MPI session is created. This simple approach may result in an excessive waste of resources, i. e., $n \cdot \frac{n-1}{2}$ possible connections right upon session startup, although commonly only a fraction of these connections is required during a communication session. Therefore, the pscom provides an *on-demand* mechanism to overcome this waste of resources for huge MPI sessions comprising thousands of ranks. This is achieved by implementing a lazy connect approach: the connection setup is postponed and only triggered on occurrence of the first send request.

Scalable Session Startup

Besides considering the need for gateways between modules for the bridging of MPI payload during the applications' runtime, the hierarchical topology of an MSA system has to be likewise taken into account during the startup of a job for the sake of scalability. On the one hand, this concerns the initial establishment of the MPI payload connections. On the other hand, it also applies to the higher-level communication infrastructure for the process and resource management.

Traditional approaches, where the communication session is initialised in an all-to-all fashion, are no longer appropriate in large, hierarchical systems. Therefore, both the pscom library and the MPC lowcomm module already provide an on-demand mechanism that implements lazy connection establishment delaying the actual connection setup to the first send request posted on a particular connection. This way, resources are only allocated to connections that are actually required by the respective communication pattern of the application.

However, this on-demand connection establishment still entails a preceding negotiation of the actual payload connection via a point-to-point-based TCP/IP connection. This approach suffers from a poor scalability. Therefore, the DEEP-SEA software specification foresees at this point a mechanism that integrates this negotiation-related initial information exchange with the topology-aware communication infrastructure of the process manager. That way, the session startup benefits from an improved initialisation of the required payload connection, while taking the hierarchical nature of the MSA system into account.

Remote Memory Access

The MSA enables various usage models of the hardware such as coarse-grained co-simulations, where code parts are optimised to the particularities of certain modules, or workflows, which utilise different modules one after another. In both cases, efficient communication among the computing processes is crucial for a high application throughput in the system.

Today, basically all high-performance interconnects feature RMA technology enabling both high communication throughput and low latencies. Commonly, the RMA capabilities are used to implement efficient two-sided communication semantics on the application layer. However, as the systems become more and more complex and applications evolve, other communication models have to be offered to support the divers demands of HPC applications. Therefore, the communication interfaces on the application layer should consider the capabilities of the underlying communication hardware by offering appropriate one-sided communication semantics. On the one hand, this is already available with MPI one-sided communication. On the other hand, further emerging programming models following PGAS concepts may provide applications with even more variability regarding the inter-process communication.

To enable full flexibility in terms of resource utilisation and communication semantics, applications should not be tied to one programming model or the other. Therefore, the low-level communication layer has to enable the composability of these different concepts within the same application. The pscom library already features both message-based two-sided communication APIs and one-sided communication APIs for RMA transfers. In the context of DEEP-SEA, the communication capabilities of the pscom will be extended to meet the demands of the different high-level communication libraries running above (e. g., ParaStation MPI and GASPI). This way, efficient one-sided and two-sided inter-process communication will be possible enabling the composability of different programming models.

Furthermore, an analysis of the execution of RMA primitives may provide insights on timing variance across the MPI processes being co-located on the same node (e. g., identify the presence of *strugglers*), and then identify opportunities for dynamic optimisation by taking into account both node-level configuration characteristics (e. g., NUMA topology and availability of heterogeneous memory devices) and dynamic effects (e. g., load imbalance). For primitives involving synchronisation, the consideration of the load across nodes may prevent the assignment of a process running on a overloaded core to the role of coordinator. The availability of heterogeneous memory resources can also be considered for an improved performance (particularly tail-latency) of the data exchange and coordination protocols implementing RMA primitives within a node. For example, by using a memory allocation framework such the SICM derivative outlined in Section 3.2.1 the implementation of MPI-RMA primitives is given the option of allocating message buffers (particularly of relatively large sizes) on bandwidth-optimised memory devices (e. g., HBM).

MSA Gateways

The MSA adds a third level to the topology of a supercomputer by introducing the notion of *modules*. This concept incorporates support for different network architectures within distinct modules. To enable an efficient and flexible utilisation of MSA systems by the application developers, the runtime-system of the programming environment is in charge of transparently connecting these networks.

This way, HPC codes written for traditional supercomputers can be executed on MSA systems without further measures. Likewise, MSA-aware extensions of the runtime system enable a gradual adaptation of the application codes to fully leverage the benefits provided by this system architecture.

The pscom library already comes with support for transparent inter-module communication. Therefore, it features the so-called gateway plugin. This implements network bridging for any pair of communication transports supported by this low-level communication layer and constitutes a central building block for MSA systems, in which heterogeneity not only affects different computing modules but also the communication interconnects being employed.

Currently, the MPC lowcomm module does not support modular architectures. However, for the DEEP-SEA software stack, it will be updated to support inter-module communication via gateways where applicable. To meet the demands of the DEEP-SEA software stack, the capabilities provided by the pscom gateway plugin will be carefully reviewed with respect to the other communication-related developments. Especially, efficient inter-module RMA transfers will be a cornerstone for enabling composability of different programming models. Additionally, it will be investigated whether the interconnection of two computing modules via a third module featuring a different network technology is a viable concept for future MSA systems. This would demand for extending the pscom's gateway plugin by efficient multi-hop routing.

3.3.2 Programming Models

The programming models of the DEEP-SEA software stack are mainly based on well-known and standardised paradigms established in the HPC domain, such as MPI for system-wide parallelisation and OpenMP/OmpSs for task-based parallelisation, supplemented by programming interfaces for accelerators such as GPUs. In addition, however, DEEP-SEA also targets at alternative programming models such as PGAS, to provide the necessary flexibility to meet the different requirements of the DEEP-SEA applications. The main challenge in DEEP-SEA with respect to such different parallelisation paradigms is not only to cover the hierarchy of MSA systems at all respective levels with suitable programming models, but also to ensure MSA awareness within and between the models. This will require an embedding of the models in a holistic software ecosystem, where particular care must be taken to ensure their interoperability, their composability, and their ability for dynamic resource sharing. The various building blocks of such a hierarchical infrastructure of nested and interacting programming models, as envisioned in DEEP-SEA, are described in more detail below.

OmpSs-2 programming model

OmpSs-2 [41] is the second generation of BSC's task-based OmpSs programming model. It extends the tasking model of OmpSs to support task nesting and fine-grained dependencies across nesting levels, which enables effective parallelisation of applications using a top-down methodology. The OmpSs-2 runtime system, Nanos6, will be augmented with an online monitoring infrastructure to gather real-time information about the performance and energy consumed by application tasks. This monitoring infrastructure will support heterogeneous nodes with a plurality of compute and memory devices with different performance and energy characteristics. The information provided by the monitoring infrastructure and the memory placement API (cf. Sect. 3.2) will be used to enhance the scheduling policies to maximise application performance and minimise overall energy consumption by scheduling each task to the best suited available compute device and their corresponding data to the most appropriate memory layer.

BSC's DLB library [42, 43] is a dynamic user-transparent library that improves the load balancing of hybrid applications and manages the number of threads, and is compatible with MPI, OpenMP, and OmpSs. Since version 3.0, DLB includes three independent and complementary modules: LeWI (Lend While Idle), DROM (Dynamic Resource Ownership Management), and TALP (Tracking Application Live Performance). The integration of the OmpSs-2 runtime with DLB will be improved based on the previous information.

Malleability and MPI Sessions

Modern architectures and applications, and in particular MSA systems, will exhibit a far greater need for flexibility in terms of resources being offered and consumed. This requires applications and/or workflow frameworks to be adjusted for being able to negotiate with runtime system as well as to take adjustments based on runtime decisions. This concept, typically referred to as malleability, will be crucial for the success of modern HPC systems.

Current programming models provide little support for malleability, either at the module or the system level. For example, although the MPI standard already supports a dynamic process model since MPI-2, this has been difficult to match with the requirements of applications on the one hand, and the constraints of resource management and scheduling on the other hand. Because of these reasons, this feature of the MPI standard has been defined as optional, and most (if not all) compute centres have it disabled in both the MPI runtime system and the workload manager. The need for an updated dynamic processes API for MPI has been accepted by the MPI Forum, and is currently being addressed in the Sessions Working Group (WG).

The recently published MPI-4 standard introduces (among other things) the new concept of MPI Sessions offering a more flexible model for the initialisation, the creation, and the management of processes. However, in its current form, all MPI sessions remain static and lack concepts for mutability, but nevertheless lend themselves well to dynamic extensions.

The MPI Sessions WG is working on a proposal for malleability extensions. These efforts are in an early stage, and it is unlikely that a proposal will be ready by the M18 deadline of D5.2 (the malleability prototype). Members of DEEP-SEA are actively involved in the MPI Forum and its Sessions WG. The malleable system software prototype from DEEP-SEA will be offered as one of the available experimental platforms to demonstrate and validate malleable MPI Sessions API proposals for the forum. Support for malleability will be implemented in an adaptable and flexible way, allowing for the adjustment to different styles of APIs, that may become the outcome of the MPI Forum's standardisation activities. Malleability support requires vertical changes to the software stack, involving most system software components, such as the scheduler and process management daemons, as well as runtime systems, tools and applications.

In DEEP-SEA, the interfaces required to support malleability will be defined using the new MPI Session concept as well as the interfaces for negotiating resources during runtime. The planned prototype implementations will then demonstrate the usability of this new concept for malleability according to the DEEP-SEA software specification. For instance, ParTec will implement the extended

MPI Sessions concept in ParaStation MPI, but also realise the corresponding interface extensions to its process manager. In this respect, ParaStation MPI is particularly well suited as a prototype for these new extensions since both components (the MPI library and the process manager) are under ParTec's development responsibility. Nevertheless, corresponding malleability extensions are also planned for GPI-Space, OmpSs-2cluster, and Open MPI, always also in collaboration with the relevant standardisation bodies.

MPI Collectives and MSA Awareness

Today's high-performance computing systems are still becoming larger and more heterogeneous. Additionally, the modular design of MSA systems leads to ever deeper topology hierarchies. Nevertheless, MPI communication is expected to be possible across all corresponding hierarchy levels, although the crossing of a hierarchy level (e. g., via an MPI gateway between two modules) may constitute a bottleneck in communication.

This, in turn, poses a challenge for MPI libraries, which should take such topology-related bottlenecks into account as much as possible. This becomes especially important when performing collective communication patterns, e. g., an optimised MPI library could try to avoid redundant communication and to consolidate messages before crossing a change in the hierarchy levels by considering information about nature and extent of this possible bottleneck.

In accordance with the envisaged software specification, all three MPI libraries used in DEEP-SEA (MPC, Open MPI, and ParaStation MPI) will address this challenge in the project and provide support for topology-aware hierarchical communication for a suitable subset of the standardised MPI collectives. This way, MPI applications will easily be able to benefit from these optimisations transparently without explicit modifications.

A crucial design concern for intra-node MPI collectives is *node-level topology awareness*, a capability that enables the determination of the available data paths as part of a hierarchical graph and the orchestration of more efficiently data movements. Moreover, another relevant platform capability to leverage is XPMEM [44, 45], which enables shared address space communication between different MPI processes that are co-located on the same node. XPMEM relies on kernel-space functionality that allows a process to attach to memory pages initialised by another process, and benefits from maintaining a registration cache for memory regions that are being used for MPI message payloads. For larger messages, a single-copy data transfer scheme is used, where processes attach directly to remote ones' application buffers. For smaller messages, a copy-in-copy-out approach is more beneficial, relying on the availability of pre-allocated and pre-attached message buffers, thus keeping latency low. As part of WP 5, intra-node MPI collectives will be developed relying on the XPMEM data exchange mechanism, avoiding data copies to improve efficiency. For large core counts, the implementation of intra-node MPI should also consider the NUMA topology for buffer placement, as well as dynamic effects due to load transients that affect the timing variance across the processes participating in collective primitives.

MSA-driven extensions/improvements/tuning to MPI

Distinct heterogeneity and the rather deep hierarchies of MSA systems make improvements and especially fine-tuning a multi-dimensional optimisation problem that typically cannot be solved in a simple and general way. Consequently, optimisation cycles, as they are envisioned for the overall DEEP-SEA software specification, are also appropriate with regard to the MSA-driven extensions to MPI.

In this regard, DEEP-SEA foresees a so-called semi-automatic tuning of selected parameters for an MPI session. This enables an iterative improvement of these parameters for the respective use case involving the user. So, for instance, ParaStation MPI aims at implementing such a semi-automatic adjustment with respect to gateway configurations, in which feedback information, user hints, and runtime measurements are combined. In this way, on the one hand, a user-specified resource allocation (e. g., selected number of gateway nodes) and, on the other hand, an adapted communication handling (e. g., mapping of collective patterns) by the MPI library should become possible.

Notified RMA, an extension to MPI RMA

MPI includes an RMA interface. This API enables applications to perform RMA operations, or one-sided communications, avoiding that both the sender and the receiver process to participate in the actual data transfer. While different models of RMA operations co-exist within MPI, none enables a receiver to detect if a communication was completed without introducing a synchronisation between the two processes involved. This lack of notification of operation completion without synchronisation impairs the latency of RMA operations. Notified RMA [46] has been proposed as a possible solution to this issue enabling a completion notification without explicit synchronisation. While a first functional implementation has been proposed using existing MPI RMA operations to demonstrate this concepts, the DEEP-SEA project will propose an efficient implementation within the InfiniBand network. Furthermore, opportunities for hardware-assisted RMA operations will be explored in the context of communication middleware, particularly the request-related event notification and active messaging capabilities offered by the Unified Communication X (UCX) set of network APIs [47]. UCX functionality is offered in Open MPI as one of the supported implementations of the PMLs.

Programming environment for Processing In Memory (PIM)

To enable adoption of PIM technologies in production systems and applications, the community has to develop programming environment for PIM accelerators. Our objective in the DEEP-SEA project is to enable the programmer to interleave conventional and PIM APIs, being focused on the desired functionality and aware of the characteristics of generic PIM devices, but with the minimum exposure to device implementation details. We have already selected a tentative list of application domains and PIM functionalities that will be covered in the context of the DEEP-SEA project. We also started analysing PIM-related data structures and APIs that will be implemented in the PIM simulation infrastructure described in Section 3.1.1.

GPI-2

GPI-2 is an API for the development of scalable, asynchronous, and fault tolerant parallel applications. GPI-2 implements the GASPI specification, an API specification for asynchronous communication which leverages remote completion and one-sided RDMA-driven communication in a PGAS. GASPI is maintained by the GASPI Forum with the aim of having a specification as a reliable, scalable, and universal tool for the HPC community.

The key feature of GPI-2 is one-sided, nonblocking asynchronous communication with notifications to allow synchronisation on remote completions. GPI-2 also implements common functionality to develop parallel applications. For instance, groups (a subset of all processes) with common collective operations. Also, atomic operations are available. Another aspect is the usage of timeouts in all blocking functions. These can be used with potentially blocking procedures to provide failure tolerance.

GPI-2 also implements the concept of memory segments. This is an abstraction representing any kind of contiguous memory. Segments are globally accessible by every thread of every GASPI process. They can be used to leverage different memory models within an application and to map the heterogeneous memory hierarchy found in today's modern hardware to the software layer. In DEEP-SEA, GPI-2 will be extended to support segments for Non-Volatile Memory (NVM), adding a layer of persistence which can then be used to build more resilient applications. Moreover, this goes hand-in-hand with and contributes to the support of heterogeneous memory resources as put forward by the memory management components of DEEP-SEA.

GPI-Space

GPI-Space is a task-based workflow management system for parallel applications. GPI-Space is designed on the principle of separation between the automatic management of parallel executions and the description of the problem-specific computational tasks and their interdependencies. It allows the domain developers to build domain-specific workflows using their own parallelisation patterns, data management policies, and I/O routines, while relying on the runtime system to take care of general aspects related to scheduling, distributed memory management, and task execution. Simultaneously, GPI-Space provides support for running legacy codes and existing domain workflows as tasks within a higher-level workflow, thus reducing the turnover time of projects for the end user.

The coupling of applications with workflows using different programming models and guaranteeing their interoperability is a challenge to be addressed when considering the software of future heterogeneous systems. MPI and GASPI are important programming models. To support the execution of MPI and GASPI programs in GPI-Space is an important step towards addressing such challenge. Investigating what is currently possible, the available options and to extend GPI-Space with such support is something that the DEEP-SEA project aims to do. Another challenge is the concept of malleability. This is very relevant when considering the MSA and a core aspect in DEEP-SEA. Malleability requires a good integration between a programming system such as GPI-Space and resource management. The support for dynamic resources, access and negotiation of allocations as provided by a resource manager allows the support of malleable applications. This leads to not only to a better utilisation of resources as well as more flexible applications. In DEEP-SEA GPI-Space will contribute with requirements to a malleable resource manager such as Slurm and add prototype implementations of the defined interfaces to support dynamic resources.

OmpSs@Cluster

OmpSs is the task-based dataflow-based parallel programming model developed by BSC. OmpSs aims to provide portability and flexibility while extracting potentially asynchronous and irregular parallelism. The objective is to extend OpenMP with new directives to support asynchronous parallelism and heterogeneity (devices such as GPUs, Field-Programmable Gate Arrays (FPGAs)), as well as extending other accelerator-based APIs such as CUDA or OpenCL. The OmpSs environment is built on top of the Mercurium source-to-source compiler (with current work to adapt LLVM) and the Nanos++ and Nanos6 runtime systems.

All work on OmpSs@Cluster in DEEP-SEA uses the new implementation, OmpSs-2@Cluster. This is the distributed memory variant of OmpSs-2, which provides transparent offloading of tasks among nodes using the same directive-based programming interface as normal tasks. Multi-node scheduling and inter-node data transfers are handled by the (newer) Nanos6 runtime system, which uses MPI for communication of control and data messages. The application is compiled with Mercurium, exactly as for an SMP program, and executed by using common mechanisms for starting the processes on a cluster (e. g., `mpirun`) and with inter-node task offloading enabled in the `nanos.conf` configuration file.

In DEEP-SEA, OmpSs@Cluster is being extended to support interoperability between tasking using OmpSs@Cluster and MPI within the same program. The MPMD application starts a main process on a subset of the MPI processes with additional MPI processes able to run offloaded tasks on the same or different nodes. Large-scale decomposition is therefore done via MPI while small-scale parallelism is done with OmpSs@Cluster. The user currently specifies which MPI processes are visible to the application. Nanos6 will also be extended to enable malleability in an unmodified MPI+OmpSs@Cluster program. As shown in the Malleable Optimisation Cycle (cf. Sect. 4.4), the runtime system (Nanos6@cluster) will interface with the malleable Slurm/ParTec process manager ('Management of Malleable Jobs' on page 36) and potentially the Malleable Online Monitor (Sect. 3.1.2).

3.3.3 Buffering and caching in SIONlib

SIONlib is a library for writing and reading data from several thousands of parallel tasks into or from one or a small number of physical files. Only the open and close functions are collective, while file access can be performed independently. Programs that use standard IO in a process-local way—where each parallel process writes to a different file—can be modified to replace that IO with SIONlib. File access is then performed using SIONlib equivalents to standard C-I/O with similar semantics to their C counterparts. SIONlib bundles this data into one or a few files to avoid sequential bottlenecks in the filesystem, e. g., metadata contention. The process-per-file picture is maintained for the application, so each process has access to its logical file only. This enables in-place parallelisation for simple I/O schemes and is particularly suited for checkpoint and restart files.

In DEEP-SEA, the utilisation of SIONlib in the software xPic (Tk1.3) will be extended. Therefore, the library will be enhanced to decide where to store intermediate data in different levels of node local storage hierarchy. This provides buffering and caching capabilities for more efficient I/O resulting in a lower network contention. This is realised by using data placement hints, specified by both by the OS and the user, alongside APIs developed in this project to optimise internal data handling during I/O phases. In addition, SIONlib will be optimised for heterogeneous systems by modifying the I/O forwarding layer developed in the DEEP-EST project enabling more efficient communication between modules.

3.3.4 Interfaces

Regarding communication and programming models, the focus for the interfaces is naturally on the APIs of the respective parallelisation paradigms and their incarnations in the form of communication library and runtime environment. In terms of MSA support, these (mostly standardised) interfaces can either support topology-aware or hierarchy-aware programming to a certain degree already, or they will be extended by these capabilities during the DEEP-SEA project. Such standardised interfaces are often quite generic and have to be semantically substantiated with meaningful identifiers and information in the context of a specific system architecture in a first step.

Apart from such generic interfaces, there will be interface extensions in the project that go beyond present standards. Such new APIs and/or extensions thereof, however, can still find wide acceptance if they are successful and/or even standardised in the end.

Besides the interfaces at the application programming level, the interfaces at the system software level are also of particular relevance for accommodating the structures of an MSA system. Here, especially the resource management has a global view of these structures, so that the interfaces to and communication between the local instances of the process manager are also of significant importance. The corresponding information exchange concerns both the higher-level libraries of the programming models and their lower-level communication substrates. However, interfaces are, of course, also provided between these subcomponents and the higher layers of the programming models for conducting the high-performance payload communication of the parallel processes during runtime.

In addition to all these interfaces within the software stack, there is naturally also the matter of the interfaces to the user, for example, regarding the look and feel of command-line tools and their parameters and/or the runtime steering via environment variables. A further aspect of an indirect user interface is the employment of analysis tools. Even if the direct interfaces between the user and such tools do not necessarily belong to the context of communication and programming model, at the system level, the interfaces between the tools and the runtime environment are directly related to the components described before.

3.4 Resource Management, Scheduling, and Orchestration

New features for components of the software stack that are responsible for resource management, scheduling, and orchestration will be developed in the DEEP-SEA project. Resource management components, such as the *slurmd* and *slurmstepd*, as well as the ParaStation Management daemons, will be updated to respond and adapt to node allocation updates related to malleable jobs. The Slurm backfill scheduler will be updated with new heuristics to enable support for workflow phases with inter-dependencies. Additional implementation efforts will enable the inclusion of GPU slices as part of allocations. The batch script format will be extended to include data staging directives, so that this can be automated by the infrastructure. These and other new features will be described in detail in this section.

3.4.1 Resource Management

Resource managers for HPC systems are typically implemented as a collection of daemons that reside on compute nodes. In the DEEP-SEA project, new features will be developed for the Slurm and the ParaStation Management daemons. These will improve MSA support and enable malleability.

Management of Malleable Jobs

A job is called malleable if the applications it runs are able to adapt themselves to changing allocations. To take advantage of this capability, interactions between the resource management (Slurm Scheduler and RM), the process management (Slurm process manager or psmgmt+psslurm) and applications are required. This interaction is shown in detail in the Malleable Optimisation Cycles (cf. Sect. 4.4). In particular, the resource and process management need to be able to trigger and handle resizing of jobs at runtime. In general, there are two ways to initiate such resizing: (1) scheduler-driven or (2) job-driven. The scheduler can initiate malleability operations to optimise a global system metrics, such as the overall energy efficiency. These scheduler decisions take into account the system state, such as occupied nodes in the topology and the job queue. Malleability operations can also be initiated by an application in a job to request additional or release resources depending on its current computational efficiency. As depicted in Figure 6, these two cases only slightly differ from the resource and process management point of view. Both cases demand for an event-based messaging facility among the resource management, the process management, and the applications in the job.

The PMIx standard defines an event notification system providing an asynchronous out-of-band mechanism for communicating events between application processes and the process management system. This mechanism can be used to implement the notifications needed to trigger and coordinate allocation resizing operations for malleable applications.

ParaStation Management will be extended to fully support the PMIx notification management system for interactions with application processes. Matching PMIx functionality will be developed for the different runtime systems covered in DEEP-SEA: OmpSs, GPI-2, GPI-Space, ParaStation MPI and Open MPI. Additionally, the ParaStation manager will be extended with notification capabilities towards the scheduler and other components. The Slurm daemons will be updated similarly.

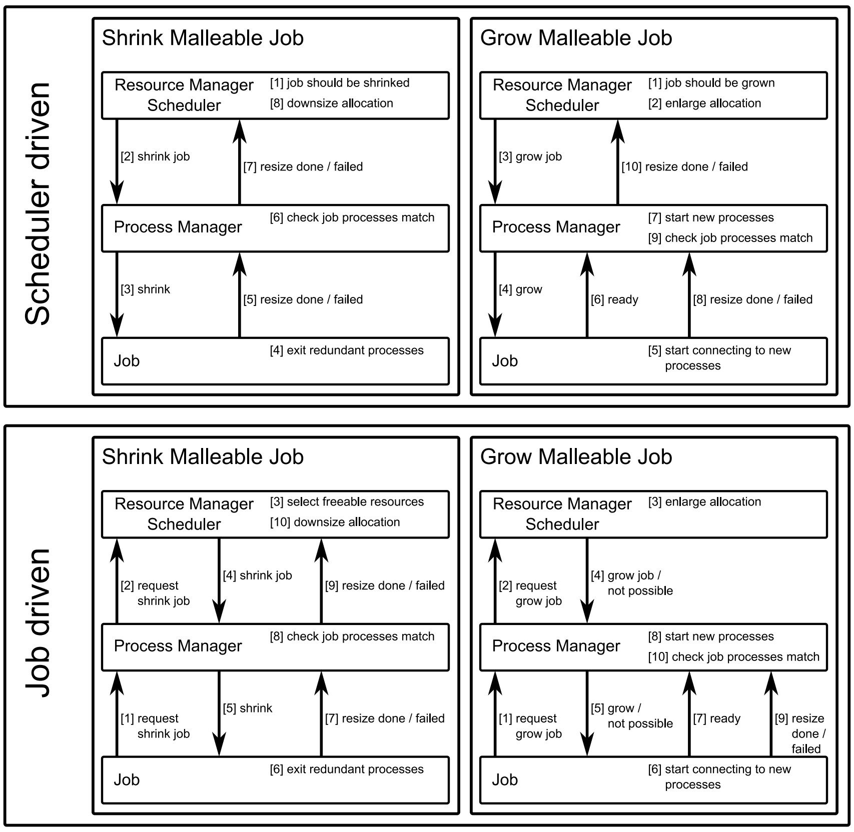


Figure 6: The communication and notification required to shrink or grow a malleable job triggered by the scheduler or by the job itself. There is only little difference between the latter two cases and that an asynchronous event system as the one specified by PMIx would be suitable for the implementation.

3.4.2 Job and Workflow Scheduling

Slurm backfill scheduler

When the term scheduler is mentioned in this document, it refers specifically to the Slurm backfill scheduler. Slurm schedulers are loaded as plugins. Currently, only the built-in and backfill schedulers are available.

DEEP-SEA scheduling-related research will be performed as new features or extensions to the Slurm backfill scheduler. This scheduler implements the heuristic most often used by supercomputing workload managers today: priority-based FIFO batch scheduling, with backfilling support (as the name implies). Updates to this existing heuristic will be necessary during research efforts related to MSA features, workflows, and malleability support.

The scheduler scans the job queue that is populated mainly by the `squeue` command. Each job entry contains the requirements of the job. The job entries are ordered based on their priority; jobs at the beginning of the queue are to be scheduled first. If resources are available, the job is started immediately. If this is not the case, a reservation is created and the job is held in the queue until it can start.

At job start time, the set of `slurmd` daemons running on the nodes of the job's allocation are notified and access controls are configured. A process to run the batch script of the job is forked at the first node of the allocation. The body of the batch scripts of multi-node jobs usually contain calls to the `srun` launcher. Each call to the `srun` launcher creates one or more job-steps. These can be made in sequence or in parallel, and can run concurrently if there are available idle resources in the allocation.

A job-step is created via interactions of the `srun` launcher and relevant `slurmd` instances. The `srun` launcher parses the requirements and instructs each `slurmd` to create the necessary `slurmstepd` instances to fulfill the job-step requirements. Each `slurmstepd` instance creates the necessary node-local processes of the job-step.

Dependencies between workflow phases

Slurm supports workflows as jobs with interdependencies that define valid start and completion orders. The wait times between these jobs are not guaranteed. This is enough to support workflows in which jobs exchange data through persistent storage. Overlapping of allocation times between these jobs can be used to avoid overheads related to serialisation and file system writes.

In the DEEP-EST project Slurm was prototypically extended by a dependency type that let a job start a specific time before the wall time of another job. This approach demands for an accurate estimation of the jobs' runtimes which is not feasible in all cases.

Therefore, the DEEP-SEA software stack will offer an event-driven mechanism to support the following use case. A preceding job will be able to notify the resource and process management system when it has reached a defined stage of its progress. This will give the scheduler information to help reduce wait times between the two jobs, while they are being setup for their data transfers. This way, the overlap of both jobs depends on the actual progress of the workflow step (cf. Fig. 7) and not on a rough estimation made by the user. In addition to this approach, malleability support will be explored as an alternative solution to this use case.

The PMIx notification system, described above, will be used to implement these notifications from the application towards the resource manager.

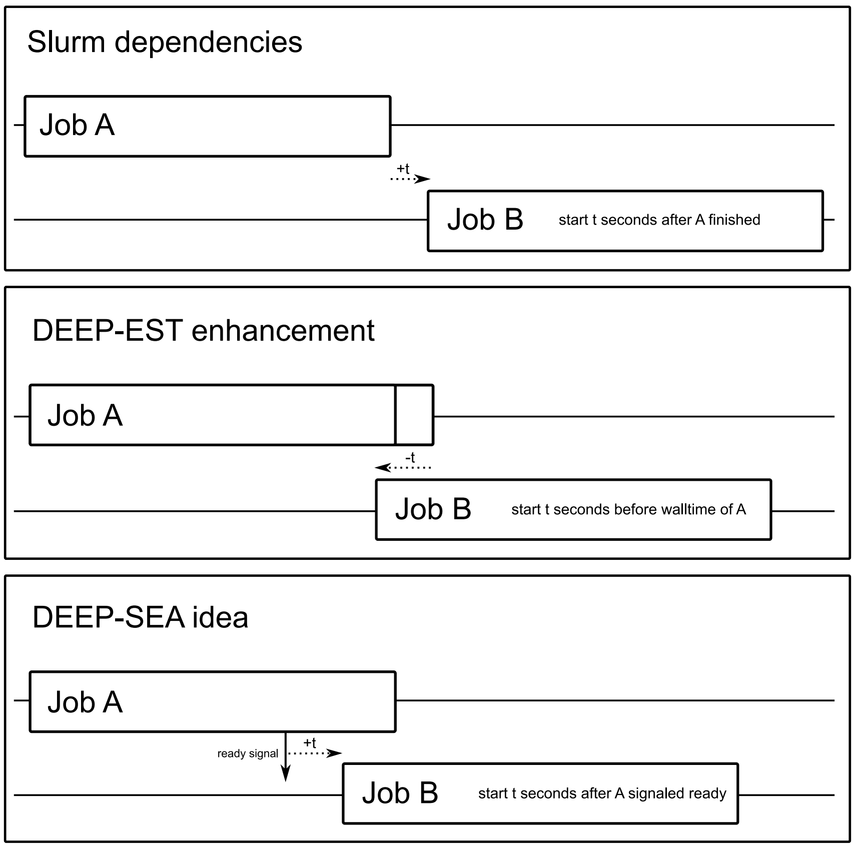

Slurm dependencies
Job A
+t
Job B
start t seconds after A finished
DEEP-EST enhancement
Job A
-t
Job B
start t seconds before walltime of A
DEEP-SEA idea
Job A
ready signal
+t
Job B
start t seconds after A signaled ready

Figure 7: The evolution of the DEEP idea of dependency management in a job workflow in comparison to the original Slurm capabilities.

A data access mechanism will be provided for data placement and its re-utilisation among stages of HPC/HPDA workflows over node-level persistent storage (e. g., NVM). These will be partly based on results from the EVOLVE large-scale testbeds project. Node-level NVM is to be leveraged as a fast buffer for workflow intermediate data instead of passing all data through the cluster-wide filesystem. Moreover, workflow stages can also be optimised to use specific memory data structures, for instance FIFO buffers or hash-maps, instead of having to reconstruct them from regular files. This memory-based data exchange mechanism is expected to reduce the time between the completion of a workflow step and the initiation of a subsequent one, with the data cwflow between workflow steps taking place over in-memory data structures rather than the underlying filesystem.

Shared accelerators

The DEEP-SEA execution environment includes heterogeneous accelerators, each of which may contain substantial hardware and memory resources (e. g., as in the case of a high-end GPUs, or specialised multi-GPU appliance such as the NVIDIA DGX system [48]). From the point of view of overall system resource utilisation, it is desirable to support fractional allocations of such accelerators, across multiple jobs that have been allocated resources on the same node. This capability is to be offered in a way that preserves the work submission interfaces of the Slurm resource manager and still supports concurrent use of accelerator resources by multiple users, rather than time-sharing of the accelerator (due to serialised exclusive access of the GPU between users) which is what currently available sharing technologies (such as NIVDIA's MPS server) are designed to offer. Moreover, access to multiple shared accelerators needs to be supported for HPDA processing pipelines and combined HPC/HPDA workflow stages. Special care is needed to preserve Slurm's resource accounting accuracy in the presence of fractional allocations and cooperation with other workload managers (e. g., Kubernetes). Work in WP3 and WP4 will provide a way to share for accelerators (particularly GPUs) across processes belonging to different jobs that have been allocated resources on the same physical host, compatible with slurm interfaces.

3.4.3 Orchestration

Effective orchestration of the components described above is needed to support the features developed in DEEP-SEA efficiently. The coordination between components will be achieved through extensions to the resource management infrastructure components.

A case study will be conducted on how to best enable a container-based cloud environment, managed by Kubernetes, to access HPC resources managed by the Slurm resource manager. Moreover, several areas of work towards enhancements to resource manager components are considered:

1. Scalability: efficient support for jobs with larger node and process counts.
2. Scheduling with improved preemption strategies: jobs starts under urgent computing scenarios [49] with saving/restoring triggers for preempted jobs.
3. Data staging: accelerate workflow execution.
4. Co-scheduling: node sharing with fractional allocation of large-scale GPU accelerators.

These enhancements will improve the combination of HPC and HPDA workloads on top of the same execution infrastructure, in the context of data-driven workloads. A case study will be conducted on how to best enable a container-based cloud environment, managed by Kubernetes, to access HPC resources managed by the Slurm resource manager.

Multiple runtime orchestration

MPI+OpenMP has become a common programming model combination (often referred to as hybrid) for HPC applications. This hybrid programming model enables the expression of parallelism at both inter-node and intra-node level, with the objective of a better resource utilisation by distributing computation to all available cores. Yet, due to their parallelism decomposition, some scientific applications exhibit their best performance when executing multiple MPI+OpenMP processes on each node.

Resource utilisation is not always optimal as some load imbalance may be observed at the OpenMP thread level. For instance, this is the case for some particle transport simulations in which the main computation load moves dynamically. It is also the case for Patmos [50], an application use case of the DEEP-SEA project. To improve the load balancing and hence accelerate the computation of the applications as transparently as possible, a finer grained orchestration of their runtime systems is required.

The DEEP-SEA software stack will provide two mechanisms realising such an orchestration. First, a system to address dynamic load shifting from processes to processes, implemented as a library relying on the OMPT interface. This will enable processes to dynamically advertise their respective loads to each other and agree on their underlying resource partitions with the goal to maximise their utilisation. Second, a mechanism to address dynamic imbalances between the threads of each process by momentarily making use of any unused cores that could take part of other application processes' load.

Dynamic load balancing for node-level malleability

The OpenMP model has been known for its fork-join model, where threads form a team of threads at each parallel region, and then all of them join again to continue with the sequential code. This model has numerous benefits for the programmer, but it lacks some malleability that other pure task-based programming models offer, e. g., the number of active threads is only set when a parallel region starts.

In the DEEP-SEA project, we will extend the OpenMP model to enable malleable applications to be developed with it. We will also develop new features for DLB (cf. Sect. 3.3.2) to improve load balancing in hybrid MPI+OpenMP applications.

The integration of the DLB and the OpenMP runtime systems will be improved by enhancing the OMPT interface. The DLB library uses the OMPT interface of the OpenMP runtime to run alongside the application as an OpenMP performance tool that can monitor and gather performance data in real-time. It can also detect computation and communication patterns at the thread level, and dynamically redistribute unused CPUs time-slices to other processes that have higher computational loads.

DLB will also be improved by adding new mechanisms to analyse the load imbalance of the application in real-time and to manage the computational resources at the node level. For example, by using the DLB interface, an application will be able to choose to release some CPUs used for OpenMP threads to the resource manage, if the measured parallel efficiency is under certain value.

In addition to the DLB integration, we will propose an implementation of free agent threads in the LLVM OpenMP runtime. These free agent threads can be enabled or disabled at any time and help to execute explicit tasks encountered in any parallel region. This feature will increase the malleability of the programming model and DLB will be able to dynamically and transparently adjust the application resources to mitigate load imbalances.

3.5 Programmer Productivity

When developing HPC software, time to solution is one of the key metrics of success. Time to solution consists of two factors: the time needed by developers to write the software and the time required to run the software to obtain the solution. In this subsection we describe two approaches (DaCe and NabLab) that aim at providing environments to achieve high programmer productivity by enabling domain scientists to write code in DSLs very close to their respective domain science. NabLab enables the writing of numerical code using $\LaTeX$ formulas while DaCe offers frontends for Python, Matlab, and ONNX. Optimisation is decoupled from the input the domain-scientists provide, in the form of a list of transformations. This allows to separate the roles of domain scientist and performance engineer. This is in line with programmer productivity research which identifies code expansion as one of the key contributors of lower productivity [51]. Allowing both roles to work on separate parts of the code keeps code expansion to a minimum.

3.5.1 High-level Programming Models

Most optimisations, including partitioning, tiling, multiple-buffering, and vectorisation, can be represented as modifications to a program's dataflow. Thus, a key component of optimisation in modern HPC is the explicit formulation of application data movement characteristics and making the dataflow accessible for optimising compilers. This is in contrast to traditional programming languages as C, where data flow analysis is hindered by aspects like aliasing. Both DaCe and NabLab make the analysability of the applications' dataflow a priority in their respective internal representations.

However, to qualify as a high-level programming model and to enable a high level of programmer productivity, it is also important to provide the right degree of abstraction to the domain scientist. In the case of DaCe, this is achieved through the provisioning of multiple frontends for different domains (currently Python+NumPy, Octave, and ONNX in addition to an API to construct the Intermediate Representation (IR) directly). In the case of NabLab, this is achieved by allowing users to specify their computations in mathematical notation, similar to what they might write in $\LaTeX$.

DaCe

DaCe [52] is a development workflow where the algorithm is independently specified from its optimisation by separating the computation from data movement. DaCe was shown to map applications to a variety of hardware architectures [52], enabling both whole-program and micro-optimisations of non-trivial scientific computing programs to state-of-the-art performance [53, 54].

At the core of DaCe lies the Stateful Dataflow multiGraph (SDFG), a nested graph-based intermediate representation of code that enforces a separation of data movements from the control flow (cf. Fig. 8). In the DaCe workflow, an input program written by *domain scientists* in various high-level languages (Python, Octave, TensorFlow, and ONNX are supported) is used to generate one or more "baseline" SDFGs. These SDFGs are then optimised separately from the original program by *performance engineers*, via changing the schedule or mapping of the dataflow using a developer-extensible set of graph rewriting transformations. The input code can be modified separately from the SDFG. As long as the same dataflow transformation chain applies to the new code, the optimised SDFG can be used as-is, promoting a full separation of concerns between development and optimisation.

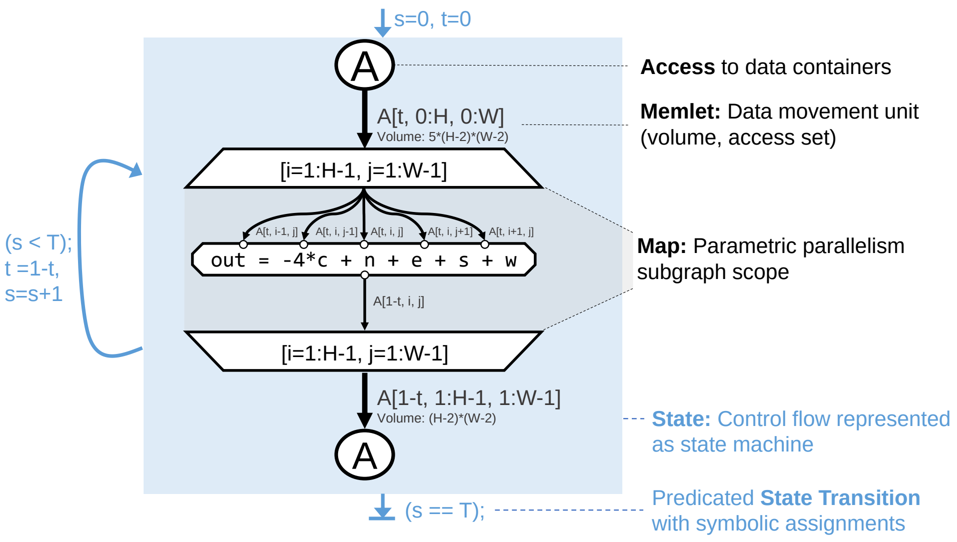

s=0, t=0
A
Access to data containers
A[t, 0:H, 0:W]
Volume: 5*(H-2)*(W-2)
Memlet: Data movement unit (volume, access set)
[i=1:H-1, j=1:W-1]
(s < T); t =1-t, s=s+1
A[t, i-1, j]
A[t, i, j-1]
A[t, i, j]
A[t, i, j+1]
A[t, i+1, j]
out = -4*c + n + e + s + w
Map: Parametric parallelism subgraph scope
A[1-t, i, j]
[i=1:H-1, j=1:W-1]
A[1-t, 1:H-1, 1:W-1]
Volume: (H-2)*(W-2)
A
State: Control flow represented as state machine
(s == T);
Predicated State Transition with symbolic assignments

Figure 8: An overview of the SDFG syntax. This example the different node and edge types as part of a stencil program.

Briefly, an SDFG is a graph of multigraphs representing a state machine of parametric dataflow graphs, where the full semantics are given in [52]. The state machine represents a program order that is not directly determined by dataflow. It consists of states (blue, rectangular nodes) and state transition edges. The latter are represented by symbolic, potentially data-dependent conditions that act as predicates for the transition; zero or more symbolic expressions represent assignments to symbol values (s and t in Figure 8).

Each state contains a dataflow multigraph comprising data movement, computation, and parametric parallelism. In particular, multi-dimensional **Data** containers registered with the SDFG are read from or written to via **Access** nodes (circular nodes), whereas computations are represented by **Tasklets** (octagonal nodes), which are deterministic and contain no-side effects. The data movement between data and computation is represented by **Memlets**, the edges in the dataflow graph.

The dataflow construction inherently represents parallelism in SDFG states. However, in many cases instantiating graphs containing all tasklets of large problems can be intractable, or impossible if the sizes are parametric. For this purpose, we define **Map** scopes (trapezoidal nodes) to express parallel tasks and parametric parallelism. The enclosed scope, defined as the subgraph dominated by a Map entry and post-dominated by a corresponding Map exit, can be scheduled in parallel.

Currently, DaCe compiles to C++ code and utilises OpenMP parallel for loops to express parallelism. In the context of DEEP-SEA we will add support for tasking. Additionally within DEEP-SEA, we added the possibility to gather and visualise runtime information (i. e., the value of cache-misses incurred by parts of the SDFG) (cf. Sect. 3.5.1).

DaCe Measurement and Debugging Interface

An SDFG can be instrumented to produce a variety of useful runtime metrics such as timer events or hardware counters. The instrumentation and reporting framework consists of three distinct components which are discussed in the following: (1) code generation, (2) report in Trace Event Format [55], and (3) visualisation.

During the code generation phase for SDFGs, a series of events is emitted whenever the code generator enters or exits specific SDFG elements or sections. A general purpose `InstrumentationProvider` captures these events and enables the injection of additional code at these specific locations. Table 1 lists the captured events.

Event	Description
`on_sdfg_begin`	Marks the beginning of SDFG code generation
`on_sdfg_end`	Marks the end of SDFG code generation
`on_state_begin`	Marks the start of SDFG state code generation
`on_state_end`	Marks the end of SDFG state code generation
`on_scope_begin`	Called at the beginning of a scope (on generating an EntryNode)
`on_scope_end`	Called at the end of a scope (on generating an ExitNode)
`on_copy_begin`	Called at the beginning of generating a copy operation
`on_copy_end`	Called at the end of generating a copy operation
`on_node_begin`	Called at the beginning of generating a node
`on_node_end`	Called at the end of generating a node

Table 1: The different events being captured by the `InstrumentationProvider`.

This general-purpose provider can be extended to generate code for capturing arbitrary metrics based on the provided events. There are currently two predefined instrumentation providers for measuring the execution time of individual SDFG elements based on the events mentioned above, one using standard C++ timers for CPU, and one using GPU events. An additional predefined instrumentation provider is available for recording performance counters using PAPI. All implementations use a report to accumulate measured times or counters as events in main memory. These events are then dumped to a report file once the program terminates.

Visualisation can be achieved either on the command-line with a text-based breakdown, using Chrome Tracing to get a timeline visualisation, or on top of SDFGs with a heatmap visualisation.

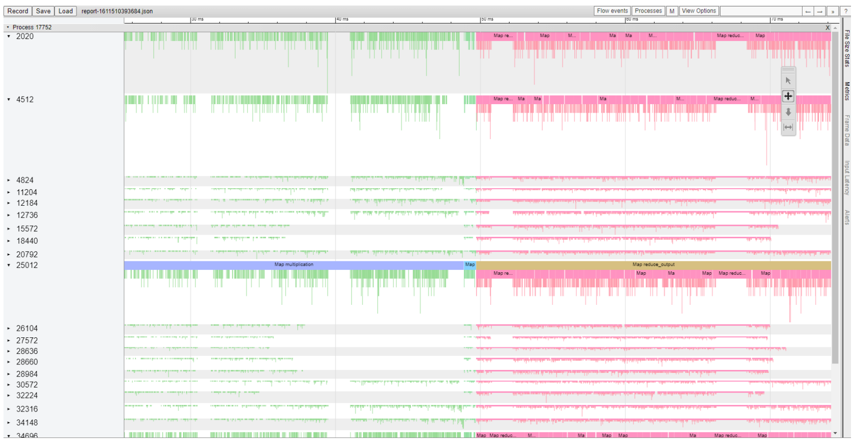
The command-line utility `sdprof` loads a given report file and prints a detailed breakdown to the terminal. The output is grouped by SDFG element and instrumented times are broken down by the minimum and maximum time recorded, as well as their mean and median values (cf. Fig. 9)

Through the use of the Trace Event Format, time measurements in runtime reports can be loaded as-is into Chrome Tracing to obtain a timeline-based view (cf. Fig. 10).

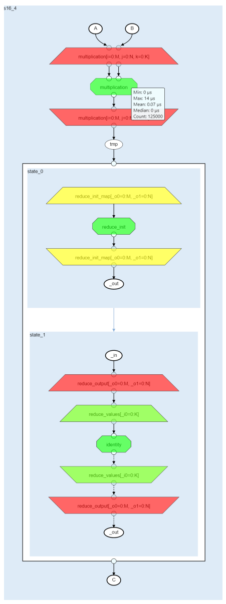
The SDFG Viewer (SDFV) additionally enables the visualisation of runtime reports directly on top of SDFGs in the form of heatmaps. The SDFV can be used as a standalone browser-based application,

```
Instrumentation report
SDFG Hash: 7ea2af8381af9d2b723dcb3f81ea044309a0c6773cbf46f050cd853c967de82d
---------------------------------------------------------------------------
Element         Runtime (ms)
                Min             Mean            Median          Max
---------------------------------------------------------------------------
SDFG (0)
| State (0)     247.335         247.335         247.335         247.335
| | Node (0)    136.121         136.121         136.121         136.121
| | Node (6)    0.0             6.7384e-05      0.0             0.014
SDFG (1)
| State (0)     2.041           2.041           2.041           2.041
| | Node (0)    2.04            2.04            2.04            2.04
| | Node (1)    0.0             4.2e-05         0.0             0.002
| State (1)     108.929         108.929         108.929         108.929
| | Node (0)    108.928         108.928         108.928         108.928
| | Node (2)    0.007           0.7726304       0.734           8.279
| | Node (4)    0.0             6.7264e-05      0.0             0.01
---------------------------------------------------------------------------
```

Figure 9: Example of a textual output of a DaCe instrumentation result.



(a)



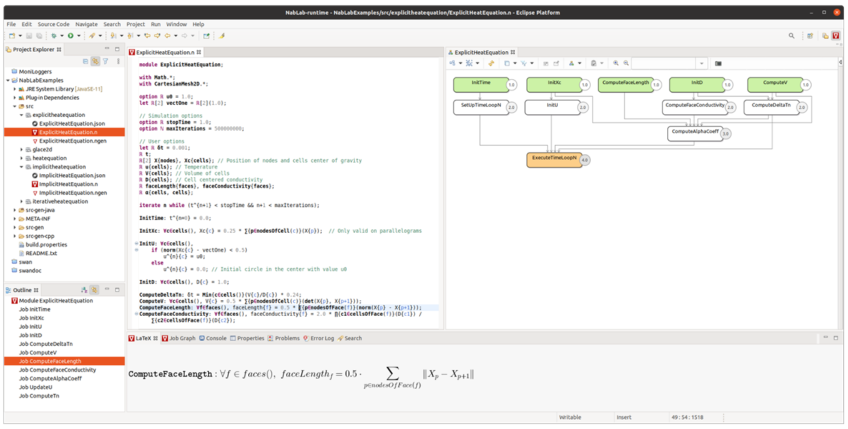
(b)

Figure 10: Chrome Tracing output for (a) the SDFG execution trace and (b) the performance data as overlay on the SDFG.

but is also found as a core part of the DaCe SDFG Editor[4] in Visual Studio Code. Upon loading a runtime report, the same aggregation of runtimes per SDFG element is performed as in `sdprof`. The resulting values are used to generate a coloured overlay on top of the SDFG to visualise areas of high and low runtime respectively. Tooltips on SDFG elements additionally provide the detailed breakdown by minima, maxima, mean, and median also found in `sdprof`, while also reporting the total number of events recorded for a given element.

NabLab

The NabLab software is an open-source research project. NabLab is both a numerical analysis DSL and an Eclipse environment. As a DSL, NabLab improves applied mathematicians productivity and enables new algorithmic developments for the construction of hierarchical and modular high-performance scientific applications. The DSL allows the conception of multi-physics applications, and is based on different concepts: no central main function, a multi-tasks based parallelism model, and a hierarchical logical time-triggered scheduling. As a full-fledged environment, NabLab enables to edit, interpret, debug specific numerical analysis sources and to generate optimised code for different runtimes and architectures called backends. Figure 11 presents the NabLab Integrated Development Environment (IDE).



```
module ExplicitHeatEquation;

with Math.*;
with CartesianMesh2D.*;

option ℝ u0 = 1.0;
let ℝ[2] vectOne = ℝ[2](1.0);

// Simulation options
option ℝ stopTime = 1.0;
option ℕ maxIterations = 500000000;

// User options
let ℝ δt = 0.001;
ℝ t;
ℝ[2] X{nodes}, Xc{cells}; // Position of nodes and cells center of gravity
ℝ u{cells}; // Temperature
ℝ V{cells}; // Volume of cells
ℝ D{cells}; // Cell centered conductivity
ℝ faceLength{faces}, faceConductivity{faces};
ℝ α{cells, cells};

iterate n while (t^{n+1} < stopTime && n+1 < maxIterations);

InitTime: t^{n=0} = 0.0;

InitXc: ∀c∈cells(), Xc{c} = 0.25 * ∑{p∈nodesOfCell(c)}(X{p});  // Only valid on parallelograms

InitU: ∀c∈cells(),
    if (norm(Xc{c} - vectOne) < 0.5)
        u^{n}{c} = u0;
    else
        u^{n}{c} = 0.0; // Initial circle in the center with value u0

InitD: ∀c∈cells(), D{c} = 1.0;

ComputeDeltaTn: δt = Min{c∈cells()}(V{c}/D{c}) * 0.24;
ComputeV: ∀c∈cells(), V{c} = 0.5 * ∑{p∈nodesOfCell(c)}(det(X{p}, X{p+1}));
ComputeFaceLength: ∀f∈faces(), faceLength{f} = 0.5 * ∑{p∈nodesOfFace(f)}(norm(X{p} - X{p+1}));
ComputeFaceConductivity: ∀f∈faces(), faceConductivity{f} = 2.0 * ∏{c1∈cellsOfFace(f)}(D{c1}) /
    ∑{c2∈cellsOfFace(f)}(D{c2});
```

$$\texttt{ComputeFaceLength} : \forall f \in faces(),\ faceLength_f = 0.5 \cdot \sum_{p \in nodesOfFace(f)} \|X_p - X_{p+1}\|$$

Figure 11: The Eclipse-based NabLab IDE for editing, interpreting, debugging, and debugging specific numerical analysis sources.

NabLab is based on the Eclipse Modeling Framework (EMF). The NabLab DSL is realised with Xtext, a framework developed on Eclipse supporting the implementation of own grammars. Xtext allows to offer a rich textual editor with syntax colouring, code completion, quick fixes, error detection, variable scoping, or also type checking. A NabLab program is divided into two parts: (1) the declarations of options, functions and variables; and (2) the definition of small unit functions called jobs. There are different steps performed on the NabLab compilation chain (cf. Fig. 12). Firstly, numerical applications are implemented in the NabLab language and in an Eclipse version offered by NabLab. Then, the NabLab software will generate an IR implemented as an Ecore metamodel, a general *model of models* in the EMF from which any model can be defined. Several transformation and optimisation passes are defined on the IR. During this step, jobs are tagged with a hierarchical logical time in order

[4]https://marketplace.visualstudio.com/items?itemName=phschaad.sdfv

to be scheduled. Jobs with a higher logical time wait for a previous one to be finished before being scheduled. Jobs with the same logical time are triggered in parallel. To highlight the execution flow of the program and to detect unintended cycles, a data flow graph is computed and displayed through a Sirius diagram. After all transformation have been realised on the IR, the NabLab software can be compiled for different backends according to the user directions. The transformation chain, based on the IR model, allows HPC engineers to introduce software engineering practices and performance optimisation before the code generation. In the scope of the DEEP-SEA project, the NabLab software will be extended to also support SDFG and SDF3 representations, which are more standardised models used for Intermediate representation. Thanks to this support, it will then be possible for NabLab to benefit from external optimisation passes developed for these representations. Currently, NabLab generates multithreaded C++ for various targets: Kokkos, OpenMP, and STL based threads. It also generates multithreaded Java code.

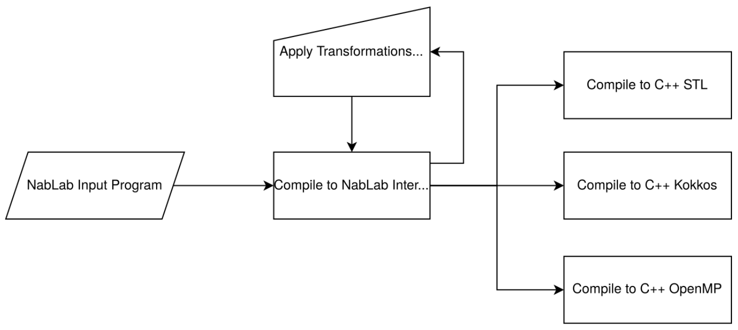


Figure 12: The different steps performed on the NabLab compilation chain.

3.5.2 Containerisation

An easy deployment and integration process is crucial for the efficient exploitation of future MSA systems in the Exascale era. This can be achieved by enabling the containerisation of applications on these systems. Container solutions allow users to provide customised software environments that can be employed to develop and to test on workstations, as well as for the later deployment on prototype platforms or production systems.

The first steps to add containerisation of HPC codes for the ParaStation process manager (psid) have already been evaluated. This was done by adding prototypical support for Nvidia Pyxis. The enroot utility is used by Pyxis to start unprivileged docker container. To support Pyxis, the `psidforwarder`, which is (among other things) responsible for setting up the process environment, had to be adapted. As a result the `psidforwarder` is calling Pyxis using various Slurm hooks to move the user processes into namespaces and therefore the container environment. Based on this work, ParTec will add support for Singularity to the ParaStation process manager.

Singularity is an open-source computer program that performs OS-level virtualisation. A Singularity container is used to encapsulate all required software and dependencies for a workflow. By using Singularity containers, developers can work in reproducible environments of their choice and design, and these complete environments can be easily copied and executed on other systems and platforms. Singularity is able to natively support high-performance interconnects such as IB and graphic accelerators. Therefore, it provides high bandwidth and low latency characteristics which is especially important for HPC applications. Additionally, it can be integrated seamlessly into MSA environments, since standalone singularity containers can be submitted to batch-systems such as Slurm without the need for modifications.

4 DEEP-SEA Optimisation Cycles

This chapter deals with the interplay of the components introduced before by presenting a set of optimisation cycles (cf. Sect. 2.2). It starts with the presentation of the monitoring infrastructure that serves as the entry point for further application optimisation. Subsequently, more specific optimisation cycles are presented covering a wide range of technology areas (cf. Chap. 1).

4.1 Monitoring Optimisation Cycle

The utilisation of the monitoring infrastructure (cf. Fig. 13) is the most basic optimisation approach and may act as an entry point to other optimisation cycles. Once the infrastructure is in place, there are no additional steps necessary to receive monitoring details. Additionally, the monitoring infrastructure supports the system administrators by evaluating the effect of individual application runs towards the overall system behaviour.

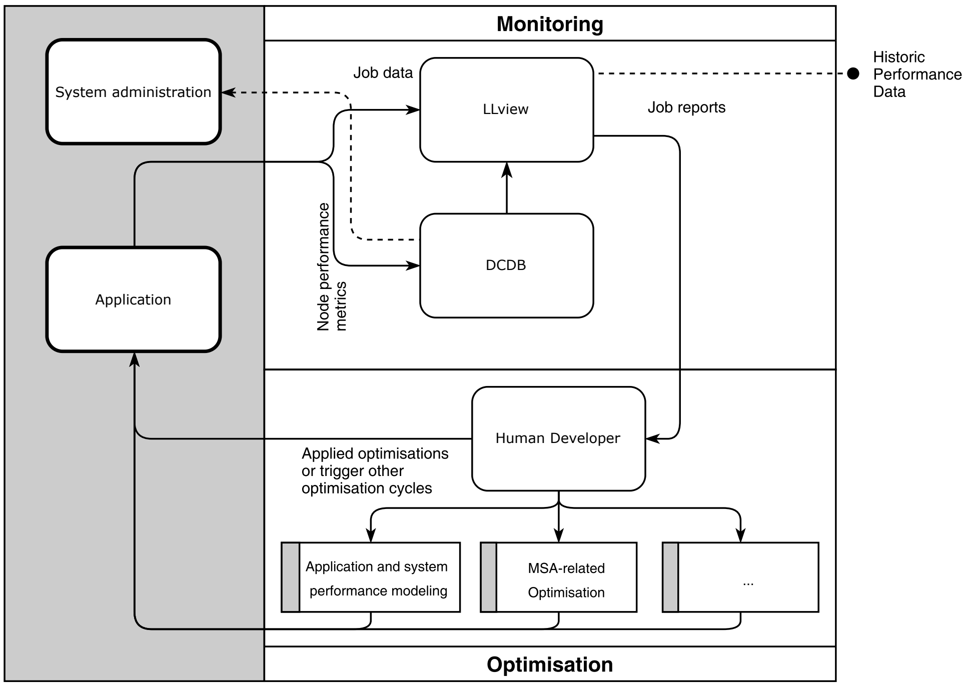


Figure 13: The monitoring optimisation cycle. This is the most basic optimisation approach acting as an entry point to the other optimisation cycles.

Node-level metrics are automatically gathered via DCDB and will be connected to individual user jobs within LLview as part of the DEEP-SEA project work. Although the cycle does not include any automatic optimisation steps, the human developer can either use the monitoring feedback to focus on manual optimisation work or to trigger one or more of the other optimisation cycles described in the following sections. Likewise, the monitoring infrastructure facilitates the verification of their outcome, i. e., it enables a high-level tracking of the individual changes made while using the other optimisation cycles.

4.2 MSA-related Optimisation

The MSA adds a further layer to the topology of traditional supercomputers. Besides intra-node and inter-node network traffic, there is additional inter-module communication that has to be taken into account when adapting applications to MSA systems. The MSA-related optimisation cycle (cf. Fig. 14) will assist application developers in identifying performance and scalability bottlenecks, and helps them to leverage the MSA awareness provided by the ParaStation MPI runtime system. While the basic cycle is also applicable to traditional, non-MSA setups, here we specifically target coupled application codes utilising multiple modules concurrently.

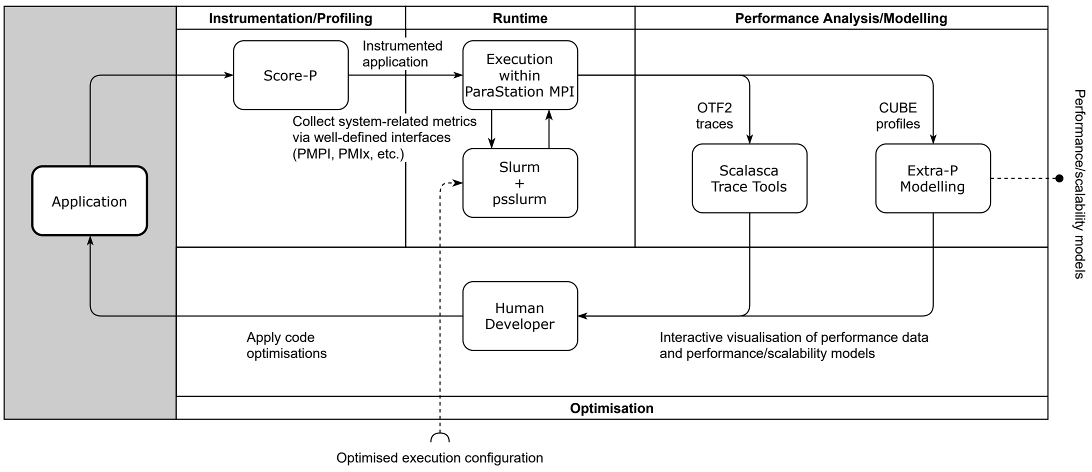

Instrumentation/Profiling
Runtime
Performance Analysis/Modelling
Score-P
Instrumented application
Execution within ParaStation MPI
Collect system-related metrics via well-defined interfaces (PMPI, PMIx, etc.)
Slurm + psslurm
OTF2 traces
CUBE profiles
Scalasca Trace Tools
Extra-P Modelling
Performance/scalability models
Application
Human Developer
Apply code optimisations
Interactive visualisation of performance data and performance/scalability models
Optimisation
Optimised execution configuration

Figure 14: The MSA-related optimisation based on profiles and traces generated by Score-P. Their analysis within the Scalasca Trace Tools and Extra-P yields hints to the human developer on how to optimise the performance and the scalability of their applications, and to better utilise the MSA awareness provided by ParaStation MPI.

The cycle is entered by taking the source code of an application, and using Score-P (cf. Sect. 3.1) to insert the instrumentation and link its measurement libraries prior to the execution on an MSA system. At runtime, the instrumented application will then collect generic as well as MSA-related metrics. The latter is achieved by querying both the MSA-aware runtime environment ParaStation MPI and the Slurm scheduler including the ParaStation resource manager. ParaStation MPI will provide communication-related metrics. This can be, for example, the actual algorithm being used for a collective communication pattern, the accurate number of bytes being sent by a process (i. e., taking communication patterns of collective algorithms into account), as well as the number of messages and/or bytes that pass through a gateway as a potential bottleneck. The scheduler and resource manager, in turn, provide system-related metrics such as the network topology defining the types of connections within and between modules. This information will be routed to the Score-P measurement system via well-defined interfaces such as PMIx, PMPI, and the MPI Tool Information

Interface (MPI_T). It remains to be decided which interface will suit the demands of the optimisation cycle best. Likewise, potential extensions and efforts for their standardisation are also part of the work realising this optimisation cycle.

In the next step, the resulting call-path profiles and event traces are fed into Extra-P and the Scalasca Trace Tools respectively. Extra-P processes a (small) series of profiles generated at different configurations (e. g., scales) to create empirical performance models, which can be analysed by the developer in a graphical viewer to pinpoint scalability bottlenecks in the application. These performance models will also serve as input for the Application Mapping Toolchain (cf. Sect. 4.3). On the other hand, the more detailed event traces are analysed by the Scalasca Trace Tools, producing an extended call-path profile enriched with higher-level metrics extracted from the trace data. This report can be explored by the application developer in an interactive report browser to locate wait states and their root causes, and to examine the critical-path profile of the application.

Finally, by leveraging the performance models from Extra-P as well as the analysis reports from the Scalasca Trace Tools, application developers are enabled to drill down and thoroughly understand the MSA-related behaviour of their application codes (e. g., whether wait states occur in communication operations within or across modules, or which code regions are more scalable than others and thus might benefit from being executed on a booster module). This facilitates the identification of opportunities for optimisation, e. g., by modifying the source code to improve algorithms and communication patterns or improving the run-time execution configuration by adjusting the mapping of processes onto the different modules of the MSA system. The latter is also supported by the Application Mapping Toolchain, which will also provide advanced mapping recommendations. Once corresponding modifications have been applied to the application code and/or the execution configuration, the cycle is reiterated to both verify that the changes led to the desired improvement and potentially focus on the next bottleneck.

4.3 Application Mapping Toolchain

When adapting applications to MSA systems, developers face the challenge of mapping their applications onto the available modules. The Application Mapping Toolchain assists the developers in finding a suitable mapping for the application and tries to automatise most of the work.

The presented cycle (cf. Fig. 15) is based on the MSA-related Optimisation Cycle and the Monitoring Optimisation Cycle (cf. Sect. 4.1 and 4.2). Besides the application itself, it requires a description of the system architecture as input. This description is created once for the specific system before the cycle starts and can be reused for all other instances or iterations of this cycle on the corresponding system. The system description entails at least information about the available modules and their hardware configuration.

The first step in this cycle is to use the application and pass it on to the MSA-related Optimisation Cycle, which will instrument and profile the application and use the gathered data to generate performance models using Extra-P (cf. Sect. 3.1.1). These performance models will then be passed back (using an internal Extra-P Format that is undefined until now) to this optimisation cycle, which continues with the Extra-P Mapping Component. Additionally, performance data is continuously captured by the Monitoring Optimisation Cycle. This data comprises the configuration options of the job, an identifier for the program and its arguments, as well as performance measurements. Using the

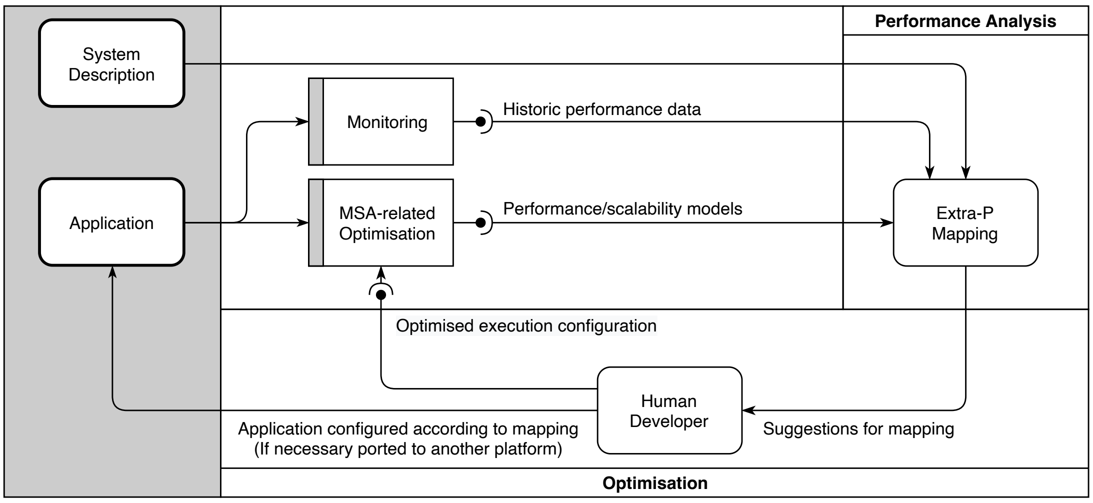

Performance Analysis
System Description
Monitoring
Historic performance data
Application
MSA-related Optimisation
Performance/scalability models
Extra-P Mapping
Optimised execution configuration
Human Developer
Application configured according to mapping (If necessary ported to another platform)
Suggestions for mapping
Optimisation

Figure 15: The Application Mapping Toolchain built around Extra-P. It relies on the outcome of both the monitoring optimisation cycle and the MSA-related optimisation cycle.

historic performance data gathered from the Monitoring Optimisation Cycle, the system description, and the performance models, the Extra-P Mapping Component will suggest suitable mappings that improve the fit between the application and the MSA system. The suggestions will be visible to the user in the application user interface in the form of human-readable performance models and matching annotations. The annotations will, for example, suggest on which type of MSA module the respective application part should be executed and how many nodes should be used. Additional recommendations may provide the number of processes that should be started on each node.

The developer can use these suggestions and apply an optimised execution configuration to the runtime system. If necessary, the developer will need to adapt the application, so that its parts can be distributed across the modules. This may even include the porting of application parts to other hardware platforms or programming models (e. g., from CPU to GPU). The suggestion mechanism of the Extra-P Mapping Component may continue to make suggestions for all non-optimal application parts. These suggestions might contradict each other, so the developer is responsible to stop the cycle once a suitable result is achieved.

4.4 Malleable Optimisation Cycles

A prototype malleable infrastructure based on Slurm will be produced by the DEEP-SEA project as an output of Tk5.2. No application or tool can be developed concretely until this prototype is produced; therefore, the optimisation cycles described in this section are early plans and are expected to change significantly.

A malleable tool is one that can adapt to allocation changes. For example, a malleable distributed memory debugger will adapt to increases and reductions of the number of processes that are part of a malleable MPI application. At this stage of the DEEP-SEA project, we will define simple monitoring tools that will serve as proof of concept. This is a conservative but realistic goal that can later be expanded upon.

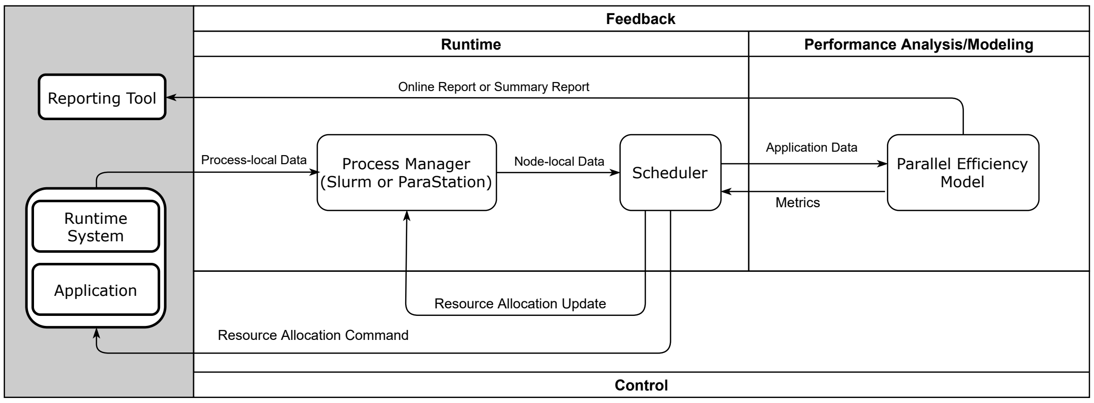

Feedback
Runtime
Performance Analysis/Modeling
Reporting Tool
Online Report or Summary Report
Process-local Data
Process Manager (Slurm or ParaStation)
Node-local Data
Scheduler
Application Data
Parallel Efficiency Model
Metrics
Runtime System
Application
Resource Allocation Update
Resource Allocation Command
Control

Figure 16: The malleable online monitor acts as performance feedback, malleability control, and reporting tool.

4.4.1 Malleable Online Monitor

A malleable online monitor that provides performance metrics of monitored applications will be developed. The monitor will react to job-step allocation changes. A job may contain static and malleable applications, represented in Slurm as job-steps, each with its own sub-allocation. A job holds an allocation that is then split into sub-allocations for each job-step. The sub-allocations of malleable job-steps will be updated with malleable operations, and the aggregated change will be applied to the job allocation. This way, both static (where allocations never change) and malleable workloads will be supported.

The relevant optimisation cycle is presented in Figure 16. Starting from the left, the job-step provides process-local monitor data to the process manager daemon at each node. Application processes include programming model specific runtime systems. Runtime systems extended in DEEP-SEA include: OmpSs, GPI-2, GPI-Space, and MPI runtime libraries, such as ParaStation MPI and Open MPI.

A reduction of the process-local data is performed to produce node-local data. The scheduler produces a final reduction of the monitoring data of the relevant job-step, from the set of node-local inputs. This is done at the Slurm controller daemon, where the scheduler plugin (cf. Sect. 3.4.2) is loaded. These reductions are collected periodically at configurable time resolutions. The application data is then fed to a parallel efficiency model.

The system will have scheduler-driven and application-driven malleability support (cf. ‘Management of Malleable Jobs’ on page 36). The difference between these use cases is based on which component initiates the negotiation. In the scheduler-driven use case, the scheduler provides malleability offers to the application, while in the application-driven use case, the application makes allocation requests. In both use cases, an agreement must be reached, and the application needs to initiate the malleability operation. The allocation metadata at the network of process manager daemons is kept synchronised with the scheduler via resource allocation updates. The resource allocation command provides the necessary allocation update metadata to the runtime system, based on the agreement.

The backfill scheduler will be extended with commands to configure the monitor as well as to request data. The collected data can be used in different ways: to prepare a summary for users, or to assist in malleable scheduling decisions. Scheduling decisions will take place in both scheduler-driven and application-driven malleable use cases. The data will be provided at the Slurm job-step level. The data of a job will be provided as the set of monitor data of the job steps where the monitor has been enabled.

A command-line interface will be provided for the users to inspect monitor data and the parallel efficiency metrics of their jobs. It is not yet defined whether this will be a new command-line interface, or an extension to an existing Slurm command, such as `sstat`. The first implementation of this user tool will provide online or summary reports of running or completed jobs and job-steps, where the monitor was enabled.

4.4.2 Malleable OmpSs@Cluster runtime

OmpSs@Cluster supports offloading of tasks to other nodes, with the scheduling of tasks to nodes, tracking of dependencies among tasks and lazy/eager data transfers controlled by the runtime system (cf. Sect. 3.3.2). In an MPI+OmpSs@Cluster application, the application's main functions run on a subset of the MPI processes, using the communicator visible to the application, whereas the runtime offloads tasks to also use the rest of the MPI processes.

The runtime system will be extended to take the role of the runtime system in Figure 16. It will make use of MPI sessions to support malleability in an unmodified MPI+OmpSs tasks program, and it will be extended to react to resource allocation commands from the job scheduler. Adding resources (i. e., nodes) can be done easily, with the data lazily migrated to the additional nodes. Removing resources requires the data to be actively migrated from the nodes that are to be released. The runtime may be able to decide which nodes can be released. It may also be extended to provide information to the scheduler, such as scalability or parallel efficiency, that can be used for malleability decisions, or to actively request additional resources. The precise nature of the interface between the job scheduler and the application/runtime is currently under discussion.

4.5 Memory Management Optimisation Pipeline

The DEEP-SEA memory management optimisation pipeline is depicted in Figure 17. Applications (e. g., TensorFlow) will be unmodified versions leveraging standard allocation calls such as `malloc` and `free`. Users may opt to perform offline memory placement optimisation at allocation granularity. This is performed via the Extrae profiler (cf. Sect. 3.2), which outputs a data-oriented profile analysis that is fed to the hmem_advisor. The advisor provides an optimised placement of the memory allocations and writes this to a file, which is parsed by FLEXMALLOC at runtime. This, in turn, decides on the final destination of the allocation leveraging different allocation calls. These memory allocations can be achieved through a high-level programming library such as the OpenMP Memory Management interface. Data is then allocated by the low-level, heterogeneous memory allocation libraries, e. g., the MPC Allocator (cf. Sect. 3.2).

During runtime, the operating system may monitor the accesses to different memory regions and perform live migrations to further optimise data placement in NUMA systems (including those featuring heterogeneous memories exposed as NUMA nodes). This procedure is complementary to the one formerly described, and is especially relevant for applications with a strong variability of access patterns and memory objects between phases, which cannot be addressed by a static approach.

A source-to-source compiler will be provided as an alternative to FLEXMALLOC, for those cases featuring many allocation calls, in which runtime allocation interposition would pose an unbearable overhead. This compiler will translate allocation calls at compile time rather than run time, which removes the interposition overhead, but also introduces some usability issues. The source code of the application and any relevant libraries have to be available, and they need to be recompiled incorporating the extra source-to-source step in the usual build procedures. Due to these extra requirements, the compile-time alternative is recommended to be used only where deemed necessary. To assess this, FlexMalloc includes the possibility of showing the number of allocation calls in the output console.

Applications leveraging the OmpSs-2 programming model will follow a specialised approach to leverage the specific knowledge with respect to data usage that the dependencies system features, in cooperation with Extrae data-oriented profiling. OmpSs-2 provides an API that allows developers to distribute data structures across different NUMA nodes using different policies. This information is not only used to allocate the data structures, but it is also leveraged by the runtime system to implement a locality-aware scheduler. In this case, the Nanos6 runtime will leverage the required allocation calls internally to optimise data placement which, coordinated with optimised task scheduling, will provide the optimised execution.

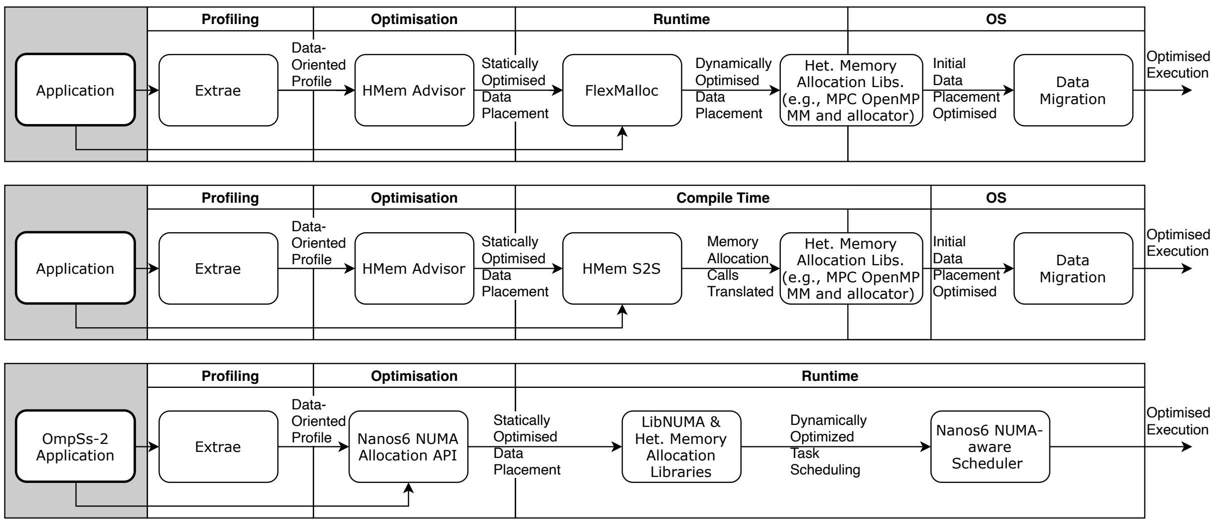

Profiling
Optimisation
Runtime
OS
Application
Extrae
Data-Oriented Profile
HMem Advisor
Statically Optimised Data Placement
FlexMalloc
Dynamically Optimised Data Placement
Het. Memory Allocation Libs. (e.g., MPC OpenMP MM and allocator)
Initial Data Placement Optimised
Data Migration
Optimised Execution
Profiling
Optimisation
Compile Time
OS
Application
Extrae
Data-Oriented Profile
HMem Advisor
Statically Optimised Data Placement
HMem S2S
Memory Allocation Calls Translated
Het. Memory Allocation Libs. (e.g., MPC OpenMP MM and allocator)
Initial Data Placement Optimised
Data Migration
Optimised Execution
Profiling
Optimisation
Runtime
OmpSs-2 Application
Extrae
Data-Oriented Profile
Nanos6 NUMA Allocation API
Statically Optimised Data Placement
LibNUMA & Het. Memory Allocation Libraries
Dynamically Optimized Task Scheduling
Nanos6 NUMA-aware Scheduler
Optimised Execution

Figure 17: The memory allocation pipeline provides means for an optimised data placement in hierarchical memory systems. This can be performed both offline and online depending on the tools being involved.

4.6 High-level Programming Interfaces Cycle

The DEEP-SEA high-level programming interfaces cycle is depicted in Figure 18. Both DaCe and NabLab (cf. Sect. 3.5.1) display the same usage pattern. First, they each offer their own syntax to write the input programs. Then, these programs are compiled into an IRs facilitating specialised transformations. As the IR, DaCe uses a SDFG model, while NabLab uses its own IR based on Ecore metamodel (cf. Sect. 3.5.1). These intermediate representations simplify the design and the application of optimisation passes. The IR is modified through the use of multiple optimisations and transformations passes, before generating the optimised code in the target language. Once the final code is generated, this will be compiled to a binary program and executed on the target hardware.

Though the transformations applied on the IR aim at improving the application performance, a transformation efficient for a class of applications may not be useful for another class of applications. Collecting runtime information while executing an application will yield useful information on the efficiency of the transformations and the code generation, in general but also in the application specific case. Through the collected metrics, it will then be possible to give feedback to the frameworks with the purpose to adapt and to improve the applied transformations. With NabLab, this collection of performance data and the analysis of efficiency of the output program (i. e., this corresponds to the efficiency of the applied optimisations) is realised by the user. The user then has to give feedback to the NabLab developers for them to enhance the optimising transformations, and the decision-making to choose which transformation should be applied.

For DaCe, the user has two options how to perform optimisations: (1) using heuristics to apply transformations that have been observed to improve performance by the DaCe developers; and (2) applying according transformations manually. These options are not mutually exclusive, i. e., a common technique is to apply aggressive optimisation heuristics, then profile an application run manually (using the tools described in Section 3.5.1) and revert transformations that show no benefit or are detrimental to performance. A common example is too fine-grained parallel regions: The DaCe optimisation heuristics try to expose as much parallelism as possible, which often leads to atomics inserted into the code in order to allow concurrent updates to variables. Through profiling, the user can identify very small (in terms of runtime) parallel sections and turn them back into sequential code with the click of a button. Automating this process further and enabling the optimisation heuristics to take runtime profiling information into account is a topic we will explore in the future, however, we note that the manual interventions by a performance engineer will always be a priority in the DaCe optimisation cycle.

4.7 Energy Optimisation

The minimisation of the energy consumption is becoming a first order optimisation criteria in HPC. In this context, application developers should be assisted with simple yet effective methods to control the energy consumption of their HPC applications. The energy optimisation cycle (cf. Fig. 19) relies on BDPO [33], a tool that transparently runs in the background of allocated compute node, and automatically optimises the energy consumption by leveraging hardware-exposed power control knobs to save energy.

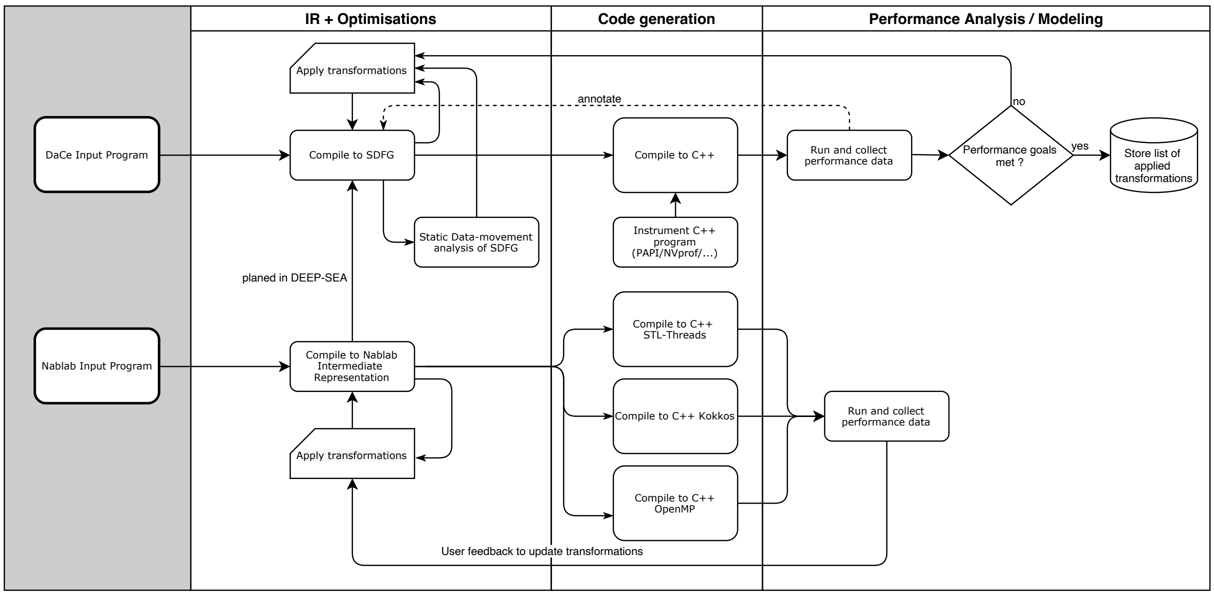

IR + Optimisations
Code generation
Performance Analysis / Modeling
DaCe Input Program
Apply transformations
Compile to SDFG
annotate
Static Data-movement analysis of SDFG
Compile to C++
Instrument C++ program (PAPI/NVprof/...)
Run and collect performance data
Performance goals met ?
no
yes
Store list of applied transformations
planed in DEEP-SEA
Nablab Input Program
Compile to Nablab Intermediate Representation
Apply transformations
Compile to C++ STL-Threads
Compile to C++ Kokkos
Compile to C++ OpenMP
Run and collect performance data
User feedback to update transformations

Figure 18: The high-level programming interface cycle visualises the common workflow for application developers using either the DaCe or the NabLab DSL.

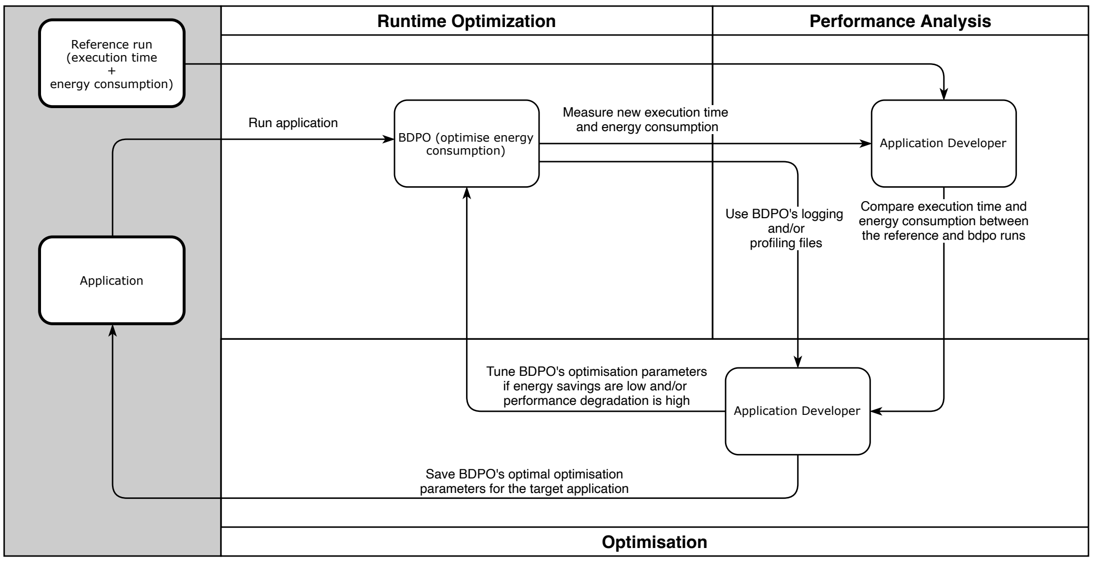

Runtime Optimization
Performance Analysis
Reference run (execution time + energy consumption)
Run application
BDPO (optimise energy consumption)
Measure new execution time and energy consumption
Application Developer
Use BDPO's logging and/or profiling files
Compare execution time and energy consumption between the reference and bdpo runs
Application
Tune BDPO's optimisation parameters if energy savings are low and/or performance degradation is high
Application Developer
Save BDPO's optimal optimisation parameters for the target application
Optimisation

Figure 19: The energy optimisation cycle relying on BDPO for a dynamic adjustment of the power and performance control knobs exposed by the hardware.

The energy optimisation cycle requires a reference run of the target application for which the execution time and energy consumption are recorded. The optimisation cycle is entered when the target application is launched alongside BDPO to dynamically optimise its energy consumption while measuring the new execution time and the consumed energy.

The effectiveness of BDPO is assessed by comparing the execution time and the energy consumption between the reference and BDPO runs. If energy savings are low and/or performance degradation is high, application developers have the option to fine tune BDPO's optimisation parameters (may rely on BDPO-produced logging and/or profiling files) and run the target application with a new BDPO configuration again.

Finally, once the obtained energy savings are satisfactory, the application developers save the best BDPO configuration (if changed from default) for future runs of the target application.

4.8 Memory System Performance Analysis

The DEEP-SEA software stack targets at highly Heterogeneous Memory Systems (HMSs) of current and forthcoming HPC platforms. With this objective in mind, we will increase the memory-related profiling, performance analysis, and prediction capabilities of state-of-the-art BSC HPC tools. Extrae and Paraver will be integrated with PROFET [26], an analytical model that quantifies the impact of the memory system on the application's performance as well as the power and energy consumption (cf. 'PROFET' on page 19 in Sect. 3.1.1).

The overall optimisation cycle is illustrated in Figure 20. The process starts with the profiling of the hardware platform and the application. The target hardware platform memory system is profiled with PROFET. The Extrae tool is used to instrument the application's execution (capturing hardware counters and MPI events) on the target platform. Based on the application and platform profiling, PROFET models the application memory behaviour (i. e., application stress/use of the memory system) and can be used to estimate application performance on different memory systems (e. g., HBM, DDR4/5, and Optane). PROFET performs the application analysis at the level of the observation interval. The observation interval is a trade-off between model precision (the smaller the intervals, the higher precision) and the application profiling overheads (the smaller intervals, the higher overheads). In the current implementation PROFET implementation, intervals of 0.1 s to 1 s are reasonable. The analysis can be also performed at the level of the CPU burst[1] comprising various observation intervals. The information of the CPU bursts estimated by PROFET is fed back to the Paraver trace file. The results are then analysed with the Paraver tool to evaluate different configurations and compare with the instrumented execution. This way, optimisations on the current HPC platforms as well as the exploration of future platforms are possible.

4.9 Multi-level Simulation Approach

Simulation is an essential step in guiding the development of complex computer hardware and systems. It reduces both financial and time costs of exploring the optimal design decisions. However,

[1]CPU burst is a region of application thread or process execution between two consecutive communication events.

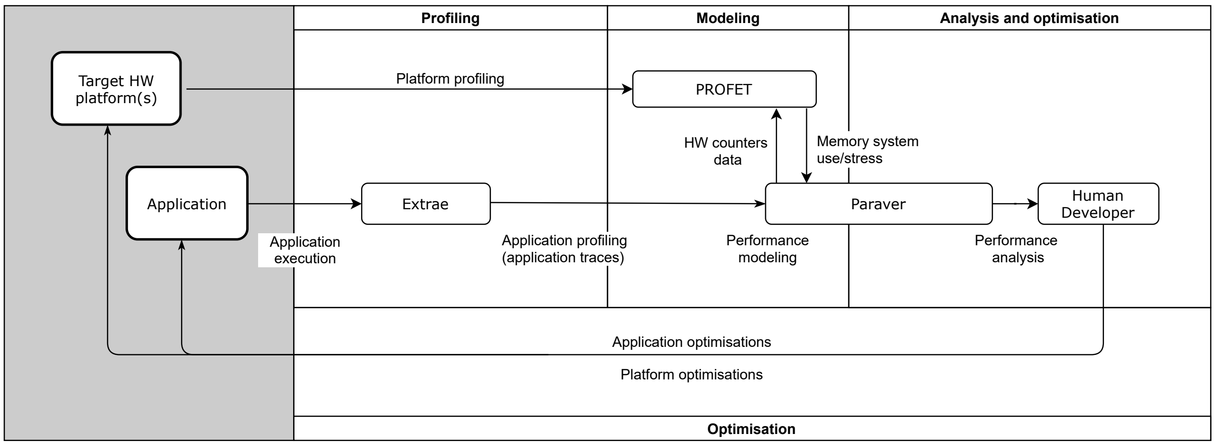


Figure 20: Performance analysis and optimisation of heterogeneous memory system using Extrae, PROFET, and Paraver.

simulation adds a significant overhead when running benchmark applications, often rendering the simulation of large computing systems impractical. MUSA is a simulation methodology developed by BSC which combines multiple levels of abstractions in order to make simulating systems of thousands of nodes and cores feasible (cf. Sect. 3.1.1).

Figure 21 shows the optimisation cycle of MUSA. The first step of the process is to trace the application used for benchmarking or that the developer wants to optimise. Tracing requires two iterations over the applications. First, using the Extrae instrumentation tool, we capture only the higher level events, such as MPI and OpenMP/OmpSs calls. In the second iteration we use DynamoRIO [56] runtime code manipulation tool to generate a detailed trace, at instruction level. Instrumenting every instruction can add significant overhead when executing an application. Therefore, MUSA targets at applications that demonstrate an iterative behaviour, a common paradigm for parallel workloads, and only traces a number of iterations of a single MPI rank. It is important that the two tracing steps are performed separately since the overhead introduced by the necessary instruction to produce the detailed trace can alter the timing of MPI and OpenMP/OmpSs events. MPI and OpenMP/OmpSs functions are not instrumented at the instruction level. Instead, they are marked with a special entry in the trace enabling the matching of events between the two traces.

The combined trace can then be used as input to the simulation infrastructure, along with a configuration file that defines the architecture details of the platform. Initially, the Dimemas MPI simulator is used to drive the simulation by following the MPI events. When a computation stage is reached, the TaskSim simulator is invoked. The computation stages can also execute in parallel before reaching the corresponding invocation point. TaskSim can either run in burst mode simulating only higher level OpenMP/OmpSs events, or it can run in detailed mode simulating each instruction (from the trace). Detailed simulations can be very slow, but higher level simulations do not consider the memory subsystem impairing the simulation precision. The MUSA methodology can either employ automatic tools (such as TaskPoint[57]) or manual input from the developer to decide which computational stages will be executed in detailed mode and use the results to fast-forward the rest. This way, the combination of two different abstraction levels of abstraction enables the fast simulation of intra-node

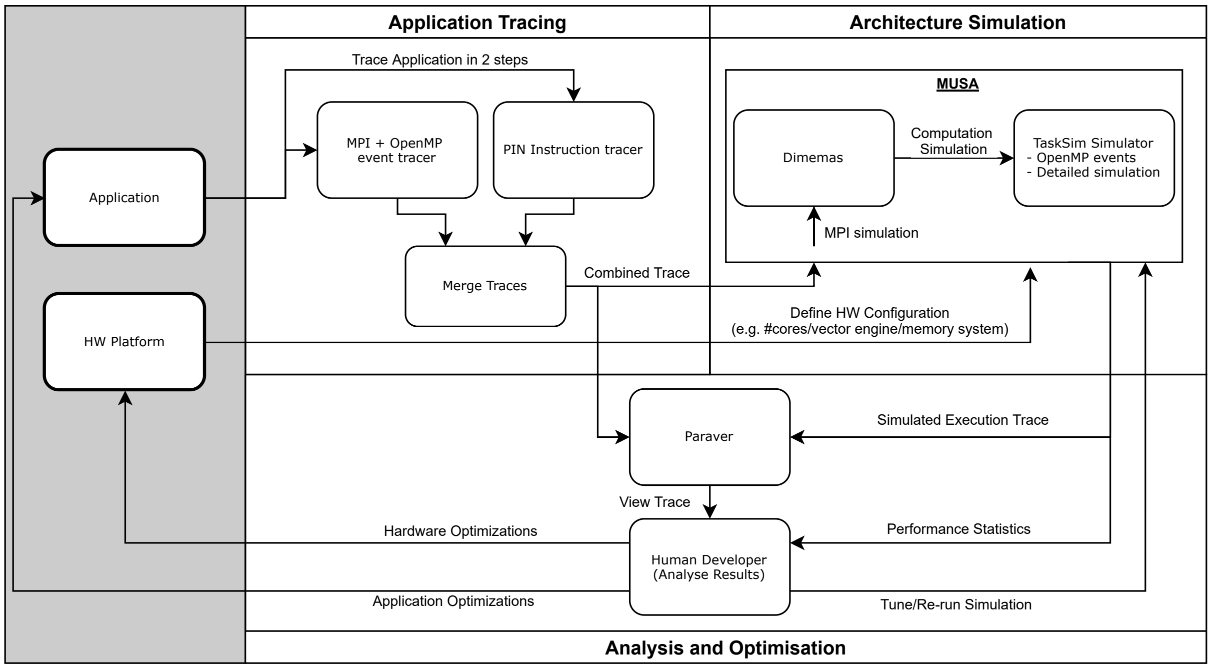


Figure 21: The MUSA optimisation cycle enables the simulation and application optimisation for systems of thousands of nodes and cores.

computational stages at a high precision. The result of the simulation stage is a generated trace with the simulated application behaviour and a file containing performance statistics.

The performance statistics can then be analysed by the human developer to drive the hardware and/or application optimisations. The trace can also be viewed by the human developer using the Paraver tool, for a more in-depth analysis of the application. Moreover, by viewing the trace, the developer can identify and define the computational stages that should be simulated in detailed mode for a more fine-grained control over the simulation process. The original trace generated by the tracing process in the first stage can also be used for the same purpose of defining the detailed simulation segments, before actually running any simulations. This way, developers can have a better control over the samples used for detailed simulation, complementing or ignoring altogether any automatic sampling techniques. The developer can also opt to tune and rerun the simulation using different hardware/application configurations before the final optimisations are committed.

4.10 Analysis of Data Transfers and Memory Behaviour

Access to data is a crucial aspect that, regardless of the level of analysis, influences the overall performance of an application. Modern systems feature great computing capabilities, which often makes the access to data the bottleneck of a running application. This is even more so if the data is located on distant unsuitable memory segments. The MemAxes tool aims at observing the data transfers within a system and analysing the collected data.

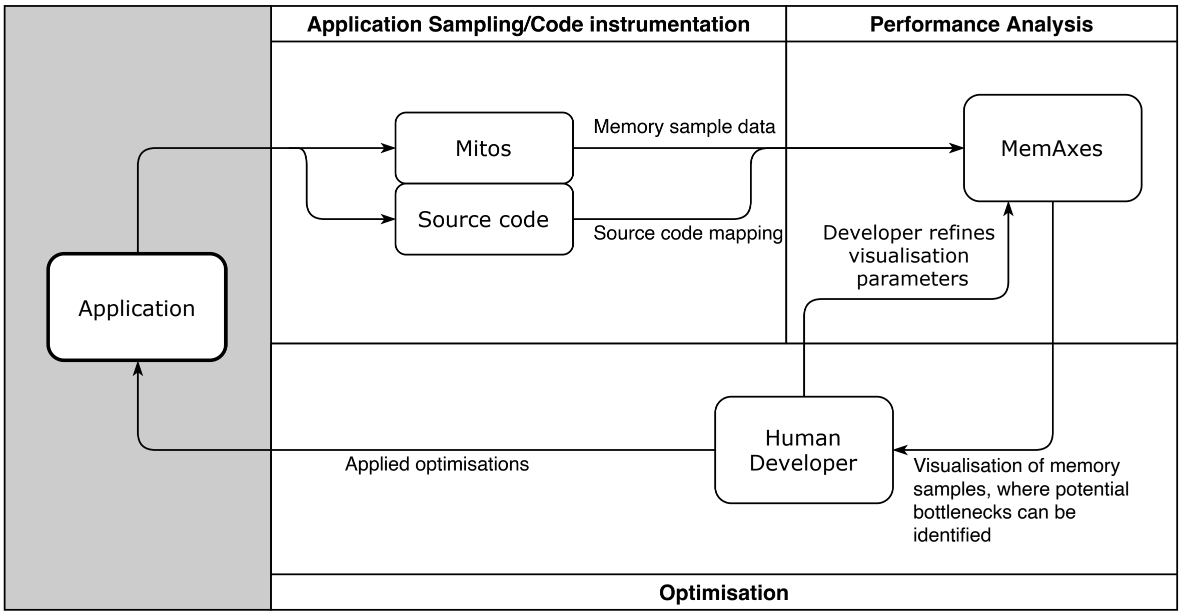

Application Sampling/Code instrumentation
Performance Analysis
Mitos
Memory sample data
MemAxes
Source code
Source code mapping
Developer refines visualisation parameters
Application
Human Developer
Applied optimisations
Visualisation of memory samples, where potential bottlenecks can be identified
Optimisation

Figure 22: The MemAxes optimisation cycle relies on Mitos for the collection of memory samples containing detailed information of single memory operations. Based on the analysis of these samples by using MemAxes, the developer may improve the application's memory access behaviour.

Figure 22 presents the MemAxes optimisation cycle. In the centre of the optimisation, there is an application that is to be analysed and potentially optimised if suboptimal behaviour is identified. The Mitos wrapper triggers the application and collects memory samples containing detailed information of single memory operations. If the developer wishes to include a mapping of the memory samples to the source code, the measured application will need to be compiled with debug information switched on. Apart from that, information regarding hardware topology is collected as well to enable mapping of each sampled memory operation to the hardware. The results of the data collection are stored in a series of output files in a predefined format.

The generated output can be opened in MemAxes (which can already run locally on the developer's computer). MemAxes analyses and visualises the collected data to present them to the developer in an easy-to-understand manner. The developer analyses the data and can interactively refine the visualisation parameters. One can, for instance, adjust the visualisation to only analysing samples from one particular CPU core to see the characteristics of the memory accesses triggered by this particular CPU core. Other example of adjusting the visualisation parameters may be restricting the data to a certain line of code to see the effects of a particular code segment. All in all, the refinement enables the developer to obtain an analysis of a fitting subset of data that magnifies the problematic behaviour, which may otherwise be hard to identify among the aggregate of all measured data.

Based on observed behaviour, the developer can design and implement optimisations to mitigate the encountered problems, if there are any. Most common use-cases include poor cache usage (data locality), loading the data to incorrect NUMA regions (first-touch policy), or load imbalance among different cores of a parallel code from the memory usage perspective. Finally, the updated application can be measured again to ensure that the proposed optimisations indeed solve the identified problematic sections, and do not cause any subsequent issues.

4.11 Analysis and Debugging of MPI-RMA Communications

When developing applications that use one-sided communications such as MPI-RMA or porting existing ones with other communication models (e. g., two-sided communication), developers may face difficulties dealing with the relaxed memory consistency nature of such communications and the differences in synchronisation handling. The RMA-Analyzer aims at helping developers during the development of such programs by giving them insights on possible memory consistency errors in their program. In the context of the DEEP-SEA project, we focus on programs that use MPI-RMA communications.

Figure 23 shows the optimisation cycle for MPI-RMA programs when using the RMA-Analyzer. Initially, the developer builds its MPI-RMA application with the RMA-Analyzer using an LLVM compilation toolchain. By doing so, the RMA-Analyzer instruments the load and store calls linked to MPI-RMA communications and the MPI-RMA calls through an LLVM pass. When running the application, the RMA-Analyzer will raise any memory consistency error it identifies during the execution, and stop the program when the first error is detected. This way a stockpiling of silent errors without knowing the root cause can be avoided. If an error is raised, the developer can identify where the error came from with the error returned by the RMA-Analyzer. After fixing the error, the cycle is started again.

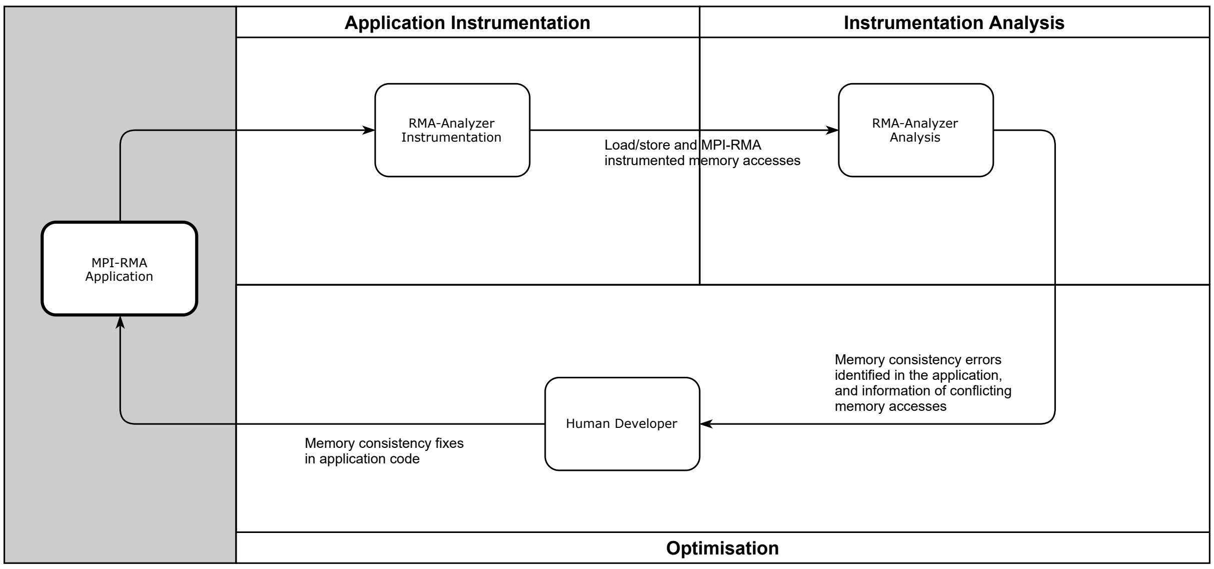

Application Instrumentation
Instrumentation Analysis
RMA-Analyzer Instrumentation
Load/store and MPI-RMA instrumented memory accesses
RMA-Analyzer Analysis
MPI-RMA Application
Memory consistency errors identified in the application, and information of conflicting memory accesses
Human Developer
Memory consistency fixes in application code
Optimisation

Figure 23: The RMA-Analyzer assists the application developer in detecting memory consistency error in their MPI-RMA programs.

5 Conclusions

This document presents the architecture of the DEEP-SEA software stack that will support application developers to port, optimise, and run their applications for an MSA system. It is the result of an intense co-design effort of all technical work-packages including WP 1.

Methodology In a first step, the partners identified all software-components that are relevant for the description of the DEEP-SEA software architecture. For this, a series of online seminars was organised in order to determine the requirements of the co-design applications organised in WP 1 on the one hand and the available tools, libraries, and runtime systems from the different partners in WP 2, WP 3, WP 4, and WP 5 on the other hand. Based on the requirements and components, the concept of so-called optimisation cycles was introduced. They represent the typical workflow of a developer of scientific applications in HPC. Their main purpose is to identify the actual relevant interactions between the components given the fact, that the large number of components results in a plethora of possible combination and interactions—most of them irrelevant in practice.

By means of these optimisation cycles the information flow in between the components is identified, and all relevant interfaces can be described.

Overview of Software Architecture In total, more than 40 components were identified and described in Chapter 3. Their interplay in the context of the project is represented by 11 optimisation cycles, each covering several components and the information flow in between them. They were presented in Chapter 4.

The components and optimisation cycles are only covering those parts of the DEEP-SEA software architecture, that will be tailored towards their use on heterogeneous (MSA-)systems in the course of the project. At the same time, components that are just used as is and have no or only minor interdependence with the described components are ignored throughout this document for the sake of brevity. Those ignored components include important software systems such as the OS, language interpreters and compilers, or unmodified runtime systems.

The relevance of the 11 optimisation cycles is metered in Table 2. After presenting them in a second series of online seminars, the application developers of WP 1 provided feedback on which optimisation cycles might be relevant for their work. The results are summarised in this table. Not too surprising the coverage differs strongly across applications as well as optimisation cycles. While very basic cycles (e. g., the Monitoring Optimisation Cycle) can be used by basically all applications, optimisation cycles representing very innovative concepts (e. g., the Malleability Optimisation Cycles) show less coverage. Nevertheless, it is worth to mention that all cycles triggered interest by at least one application partner.

Outlook and Next Steps In a next step, the detailed planning of the required interfaces will be done. This is accompanied by defining the actual features that will be implemented in the different components in order to enable their use in the identified optimisation cycles. The corresponding results will be presented in the upcoming deliverables of WP 2, WP 4 and WP 5, i. e., D 2.1, D 4.1, and D 5.1, respectively.

In parallel, a first round of deployment of all available software-components on the DEEP system will be performed. The purpose of this exercise is two-fold: On the one hand, this enables the application developers of WP 1 to become familiar with these tools and to identify current errors and shortcomings. On the other hand, each component represents a counterpart to other components in a given optimisation cycle. Thus, they are vehicles for the implementation and testing of these other components. To enable the CI strategy of Tk 3.6, the project strives for an as much as possible complete and automatised installation even at this early stage of the project.

Application	Monitoring	MSA	Mapping Toolchain	Malleability	Memory Allocation	DaCe/NabLab	Energy Optimisation	HMS	MUSA	MemAxes	RMA-Analyser
xPic	✓	✓	✓	✗	✓	✗	✓	✓	✓	✗	✗
AIDApy	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗
IFS	✓	✓	✗	✗	✓	(✓)	✗	✓	✓	✓	✗
FRTM	✓	✗	✗	✗	✗	✗	✗	✗	✗	(✓)	✗
BSIT	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗
Gromacs	✓	✗	✗	✗	✓	✓	✗	✓	✗	✗	✓
NEK5000	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✓
PATMOS	(✓)	✓	✗	✗	✓	✗	✗	(✓)	✗	✗	✗
TSMP	✓	✓	✓	✓	✗	✗	✓	✓	✗	✗	✗

Table 2: Feedback from the developers of the co-design applications to the proposed optimisation cycles. A check mark corresponds to the intention of the application developers to have a closer look at the respective optimisation cycle once this is available.

List of Acronyms and Abbreviations

AI	Artificial Intelligence
AN	Data Analytics Node
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
BDPO	Bull Dynamic Performance Optimizer, *Glossary:* BDPO
BN	Booster Node
BoP	Board of Partners for the DEEP-SEA project
BXI	BullSequana eXascale Interconnect
CA	Consortium Agreement
CI	Continuous Integration
CLI	Command-Line Interface
CM	Cluster Module, *Glossary:* CM
CN	Cluster Node
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture, *Glossary:* CUDA
CUPTI	CUDA Profiling Tools Interface
DaCe	Data-Centric parallel programming, *Glossary:* DaCe
DAM	Data Analytics Module first, *Glossary:* DAM
DCDB	Data Center Data Base, *Glossary:* DCDB
DDG	Design and Development Group, *Glossary:* DDG
DDR	Double Data Rate
DEEP	Dynamical Exascale Entry Platform (project FP7-ICT-287530)
DEEP-ER	DEEP – Extended Reach (project FP7-ICT-610476)
DEEP-EST	DEEP – Extreme Scale Technologies
DEEP-SEA	DEEP – Software for Exascale Architectures
DEEP/-ER	Term used to refer jointly to the DEEP and DEEP-ER projects
DL	Deep Learning
DLB	Dynamic Load Balancing, *Glossary:* DLB
DN	DAM Node
DoW	Description of Work
DRAM	Dynamic Random Access Memory, *Glossary:* DRAM
DSL	Domain-specific Language

DVFS	Dynamic Voltage and Frequency Scaling
EC	European Commission
EEHPC	Energy Efficient High Performance Computing
EEP	European Exascale Projects
EMF	Eclipse Modeling Framework
EPI	European Processor Initiative
ESB	Extreme Scale Booster, *Glossary:* ESB
ETP4HPC	European Technology Platform for High Performance Computing
EU	European Union
FFT	Fast Fourier Transform
FIFO	First-In-First-Out
FLEXMALLOC	Flexible Memory Allocator, *Glossary:* FLEXMALLOC
Flop/s	Floating point Operation per second
FPGA	Field-Programmable Gate Array, *Glossary:* FPGA
FTI	Fault Tolerant Interface
GASPI	Global Address Space Programming Interface, *Glossary:* GASPI
GFLOPS	Giga Floating Point Operations per Second, *Glossary:* GFlop/s
GPU	Graphics Processing Unit
GROMACS	Groningen Machine for Chemical Simulations description
GUID	Globally Unique Identifier
H2020	Horizon 2020
HBM	High Bandwidth Memory
hmem_advisor	Heterogeneous Memory Advisor, *Glossary:* hmem_advisor
HMS	Heterogeneous Memory System
HPC	High Performance Computing
HPDA	High Performance Data Analytics
HW	Hardware
I/O	Input / Output
I^{2}C	Inter-Integrated Circuit Computer Bus
IB	InfiniBand, *Glossary:* InfiniBand
IC	Innovation Council
IDE	Integrated Development Environment
IP	Intellectual Property
IPC	Instructions Per Cycle

IPMI	Intelligent Platform Management Interface
IR	Intermediate Representation
ISA	Instruction Set Architecture
ISO	International Organization for Standardization
JLESC	Joint Laboratory for Extreme Scale Computing
JUBE	Jülich Benchmarking Environment
JURECA	Jülich Research on Exascale Cluster Architectures
JUWELS	Jülich Wizard for European Leadership Science
KNL	Knights Landing, *Glossary:* KNL
LLNL	Lawrence Livermore National Laboratory
ML	Machine Learning
MoU	Memorandum of Understanding
MPC	Multi-Processor Computing, *Glossary:* MPC
MPI	Message-Passing Interface, *Glossary:* MPI
MPI-RMA	Remote Memory Access interface of MPI
MPI_T	MPI Tool Information Interface, *Glossary:* MPI_T
MPMD	Multiple Program Multiple Data
MQTT	Message Queuing Telemetry Transport, *Glossary:* MQTT
MSA	Modular Supercomputer Architecture
MUSA	MUlti-level Simulation Approach
MUSIC	Multi-Simulation Coordinator
NAM	Network Attached Memory
NF	Network Federation
NN	Neural Network
NUMA	Non-Uniform Memory Access
NV-DIMM	Non-Volatile Dual In-line Memory Module
NVM	Non-Volatile Memory
NVRAM	Non-Volatile Random-Access Memory
OA	Open Access
ODC	Other Direct Costs
OPA	Omni-Path Architecture, *Glossary:* Omni-Path
OpenCL	Open Computing Language, *Glossary:* OpenCL
OpenMP	Open Multi-Processing, *Glossary:* OpenMP

ORTE	Open MPI Runtime Environment
OS	Operating System
OTF2	Open Trace Format 2, *Glossary:* OTF2
PAPI	Performance Application Programming Interface
PARCOACH	PARallel COntrol flow Anomaly CHecker first, *Glossary:* PARCOACH
PCIe	Peripheral Component Interconnect Express
PFLOPS	Peta Floating Point Operations per Second, *Glossary:* PFlop/s
PGAS	Partitioned Global Address Space
PI	Principal Investigator
PIM	Processing In Memory
PMEM	Persistent Memory
PMI	Process Management Interface
PMIx	Process Management Interface for Exascale
PML	Point-to-point Management Layer
PMPI	MPI Profiling Interface
PRACE	Partnership for Advanced Computing in Europe
PROFET	PROFiling-based EsTimation of performance and energy, *Glossary:* PROFET
R&D	Research and Development
RAM	Random-Access Memory
RAS	Reliability, Availability, Serviceability
RDA	Research Data Alliance
RDMA	Remote Direct Memory Access
RDP	Reliable Datagram Protocol
REST	Representational State Transfer, *Glossary:* REST
RISC	Reduced Instruction Set Computing
RM	Resource Manager
RMA	Remote Memory Access
RMI	Remote Method Invocation
RML	Risk Management List
SCR	Scalable Checkpoint/Restart
SDF3	Synchronous Dataflow, *Glossary:* SDF3
SDFG	Stateful Dataflow multiGraph
SDFV	SDFG Viewer
SDV	Software Development Vehicle, *Glossary:* SDV
SICM	Simplified Interface to Complex Memory, *Glossary:* SICM
SIMD	Single Instruction Multiple Data

SME	Small and Medium Enterprises
SMP	Symmetric Multiprocessing
SN	Storage Node
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
TCP/IP	Transmission Control Protocol and the Internet Protocol
TFLOPS	Tera Floating Point Operations per Second, *Glossary:* TFlop/s
Tk	Task, *Glossary:* Tk
ToW	Team of WP Leaders
TRL	Technology Readiness Level
UCX	Unified Communication X, *Glossary:* UCX
UI	User Interface
WG	Working Group
WP	Work Package

Glossary

ARM	Family of RISC architectures for computer processors, configured for various environments. Formerly standing for Advanced RISC Machine, or Acorn RISC Machine.
ASTRON	Netherlands Institute for Radio Astronomy, Netherlands.
Aurora	Name of the subsidiary of the Eurotech Group dedicated to the HPC business. Aurora also refers to Eurotech's line of cluster systems.
BADW-LRZ	Formal, administrative acronym of the Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften. Computing Centre, Garching, Germany. Across the proposal "LRZ" is used as short version of this acronym, as it fits better in tables and headers.
BDPO	The Bull Dynamic Performance Optimizer is a job-oriented energy optimisation tool developed by Atos.
BSC	Barcelona Supercomputing Centre, Spain
BSCW	Repository used in the DEEP-SEA project to share all project documentation.
Cassandra	The Apache Cassandra key-value store.
CEA	Commissariat à l'énergie atomique et aux énergies alternatives, France
CERN	European Organisation for Nuclear Research / Organisation Européenne pour la Recherche Nucléaire, International organisation.
CM	A module of an MSA system having CN containing high-end general-purpose processors and a relatively large amount of memory per core.
CUBE4	Open call-path profile format used by Score-P, the Scalasca Trace Tools, and Extra-P.
CUDA	A parallel computing platform and programming model developed by NVIDIA for general computing on GPUs.
DaCe	A development workflow where the algorithm is independently specified from its optimisation by separating the computation from data movement.
DAM	A module of an MSA system with DNs 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	Monitoring framework for the acquisition of telemetry data developed by LRZ.
DDG	A committee of technical experts from the system SW, programming models, tools, and application domains that drives the co-design discussions and make the most important design decisions in DEEP-SEA.
Dimemas	Performance analysis tool developed by BSC.

DIMM	Dual In-line Memory Module, a series of dynamic random-access memory integrated circuits.
DLB	A library for dynamic load balancing developed by BSC.
DRAM	Typically, describes any form of high capacity volatile memory attached to a CPU.
ESB	A module with highly energy-efficient many-core processors as BNs, but a reduced amount of memory per core at high bandwidth.
ETHZ	Eidgenössische Technische Hochschule Zürich, Switzerland.
EVOLVE	EU project addressing HPC-enabled capabilities in data analytics for processing massive and demanding datasets without requiring extensive IT expertise.
Exascale	Computer systems or applications, which are able to run with a performance above 10^{18} Floating point operations per second.
Extra-P	Tool for automated performance modeling of HPC applications developed by TUDA.
Extrae	Performance analysis tool developed by BSC.
Fabri³	Interconnect technology based on EXTOLL (pron. “Fabri-Cube”).
FHG	Fraunhofer Gesellschaft zur Foerderung der Angewandten Forschungs e. V., Germany.
FLEXMALLOC	An interceptor for regular allocation calls during runtime. It performs allocations into specific memory subsystems according to the output of hmem_advisor.
FORTH	Foundation for Research and Technology – Hellas, Greece.
FP7	European Commission 7th Framework Programme.
FPGA	An integrated circuit to be configured by the customer or designer after manufacturing.
FZJ	Forschungszentrum Jülich GmbH, Jülich, Germany.
GASPI	A programming model for one-sided and asynchronous communication between computing nodes.
GFlop/s	10^9 Floating point operations per second.
GPI-2	The implementation of the GASPI specification.
GPI-Space	A task-based workflow management system for parallel applications.
GROMACS	A toolbox for molecular dynamics calculations providing a rich set of calculation types, preparation and analysis tools.
hmem_advisor	A tool developed at BSC providing an optimised distribution of memory objects to the different memory subsystems.
Hydra	The MPICH-native Process Manager.
InfiniBand	A networking communication standard for HPC clusters.

iPic3D	Programming code developed by the KULeuven to simulate space weather
JSC	Jülich Supercomputing Centre GmbH, Jülich, Germany
KNL	The second generation of Intel® Xeon Phi (TM).
Kokkos	A programming model in C++ for writing performance-portable applications targeting all major HPC platforms.
Kubernetes	A scheduler for an automated management of resources for AI workloads.
KULeuven	Katholieke Universiteit Leuven, Belgium
LIKWID	A data source for node-level performance counters.
LRZ	Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften. Computing Centre, Garching, Germany. Short name of long partner acronym “BADW-LRZ”.
MemAxes	A node-level and system-wide memory/data-traffic visualisation tool developed by TUM.
Mercurium	BSC’s source-to-source compilation infrastructure, mainly used with Nanos6 or Nanos++ to implement OpenMP and OmpSs/OmpSs-2.
Mont-Blanc	European scalable and power efficient HPC platform based on low-power embedded technology.
MPC	The MPC framework regroups an MPI, an OpenMP, and a pthread implementation in the same software, for better interoperability.
MPI	An API specification typically used in parallel programs that allows processes to communicate with one another by sending and receiving messages.
MPI_T	An interface which provides a mechanism for MPI implementers to expose variables, each of which represents a particular property, setting, or performance measurement from within the MPI implementation.
MPICH	An MPI implementation maintained by Argonne National Laboratory.
MQTT	A publisher/subscriber-based messaging protocol.
NabLab	EMF-Based language and development environment inspired from Nabla. It provides a user-friendly environment for mathematicians.
Nanos++	BSC’s runtime system used to support OmpSs-1.
Nanos6	BSC’s runtime system used to support OmpSs-2.
NVM	Used to describe a physical technology or the use of such technology in a non-block-oriented way in a computer system.
Omni-Path	Short for Omni-Path Architecture (OPA), a communication architecture owned by Intel.
OmpSs	BSC’s Superscalar (Ss) for OpenMP.

OmpSs-2	BSC's extension of OmpSs to support task nesting and fine-grained dependencies across nesting levels.
OMPT	The OpenMP tools interface.
OpenACC	OpenACC Application Programming Interface, a directive-based API for writing parallel programs that run code regions on multicore CPUs or attached accelerators.
OpenCL	A 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	Application programming interface that support multi-platform shared memory multiprocessing.
Open MPI	An MPI implementation maintained by the Open MPI Project.
OTF2	Open event trace format and reader/writer libraries.
ParaStation	Software for cluster management and control developed by ParTec.
Paraver	A Performance analysis tool developed by BSC.
PARCOACH	A debugging tool for collective operation usage in MPI and OpenMP developed by Atos.
ParTec	ParTec AG, Munich, Germany. Linked third Party of FZJ in DEEP-SEA.
perf	A data source for node-level performance counters.
PFlop/s	10^{15} Floating point operations per second.
PIC	Family of microcontrollers made by Microchip Technology Inc.
piSVM	A parallel classification algorithm.
PMT	Project Management Team of the DEEP-SEA project.
PObj	Project Objective of the DEEP-SEA project.
PROFET	Model to predict performance on given memory system developed by BSC.
pscom	The low-level communication layer of ParaStation MPI.
REST	An interface for web services.
Scalasca Trace Tools	Trace-based performance analysis toolset on top of Score-P developed by JSC.
Score-P	Community-maintained instrumentation and measurement infrastructure for parallel application performance analysis, developed by a consortium of partners including JSC.
SDF3	An open dataflow graph interchange format.
SDV	HW systems to develop software in the time frame where the DEEP-EST prototype was not available yet.
SICM	Simplified Interface to Complex Memory: APIs for managing memory allocations across heterogeneous memory tiers (originally from Los Alamos National Laboratories).

SIONlib	Parallel I/O library developed by JSC.
Slurm	Job scheduler that will be used and extended in the DEEP-SEA project.
SYCL	C++ single-source heterogeneous programming for OpenCL.
TensorFlow	An open-source software library for dataflow programming.
TFlop/s	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-SEA project.
TUDA	Technical University of Darmstadt, Germany.
TUM	Technical University of Munich, Germany.
UCX	A communication framework for modern, high-bandwidth, and low-latency networks.
x86	Family of instruction set architectures based on the Intel 8086 CPU.
XPMEM	Cross-Process Memory Mapping: mechanism for a process to access one memory owned by another.

Bibliography

[1] E. Suarez, N. Eicker, and T. Lippert. “Modular Supercomputing Architecture: from Idea to Production; 3rd”. In: *Contemporary High Performance Computing: From Petascale toward Exascale, Volume 3*. Vol. 3. 2019, pp. 223–251.

[2] A. Knüpfer et al. “Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir”. In: *Tools for High Performance Computing 2011*. Ed. by H. Brunst et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 79–91. DOI: 10.1007/978-3-642-31476-6_7.

[3] *Score-P website*. URL: https://www.score-p.org (visited on 09/23/2021).

[4] M. Geimer et al. “Further Improving the Scalability of the Scalasca Toolset”. In: *Applied Parallel and Scientific Computing*. Ed. by K. Jónasson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 463–473. DOI: 10.1007/978-3-642-28145-7_45.

[5] D. Eschweiler et al. “Open Trace Format 2 – The Next Generation of Scalable Trace Formats and Support Libraries”. In: *Advances in Parallel Computing (Proc. of the Intl. Conference on Parallel Computing, ParCo)* 22 (2012), pp. 481–490. DOI: 10.3233/978-1-61499-041-3-481.

[6] A. Knüpfer et al. “The Vampir Performance Analysis Tool-Set”. In: *Tools for High Performance Computing*. Ed. by M. Resch et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 139–155. DOI: 10.1007/978-3-540-68564-7_9.

[7] K. E. Isaacs et al. “Combing the Communication Hairball: Visualizing Parallel Execution Traces using Logical Time”. In: *IEEE Transactions on Visualization and Computer Graphics* 20.12 (Dec. 2014), pp. 2349–2358. DOI: 10.1109/TVCG.2014.2346456.

[8] P. Saviankou et al. “Cube v4: From Performance Report Explorer to Performance Analysis Tool”. In: *Procedia Computer Science* 51 (June 2015), pp. 1343–1352. DOI: 10.1016/j.procs.2015.05.320.

[9] S. S. Shende and A. D. Malony. “The Tau Parallel Performance System”. In: *The International Journal of High Performance Computing Applications* 20.2 (2006), pp. 287–311. DOI: 10.1177/1094342006064482.

[10] D. Terpstra et al. “Collecting Performance Data with PAPI-C”. In: *Tools for High Performance Computing 2009*. Ed. by M. S. Müller et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 157–173. DOI: doi.org/10.1007/978-3-642-11261-4_11.

[11] I. Zhukov et al. “Scalasca v2: Back to the Future”. In: *Tools for High Performance Computing 2014*. Ed. by C. Niethammer et al. Springer International Publishing, 2015, pp. 1–24. DOI: 10.1007/978-3-319-16012-2_1.

[12] *Scalasca website*. URL: https://www.scalasca.org (visited on 09/23/2021).

[13] M. Geimer et al. “A scalable tool architecture for diagnosing wait states in massively parallel applications”. In: *Parallel Computing* 35.7 (July 2009), pp. 375–388. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2009.02.003.

[14] D. Böhme et al. “Identifying the Root Causes of Wait States in Large-Scale Parallel Applications”. In: *ACM Trans. Parallel Comput.* 3.2 (July 2016). ISSN: 2329-4949. DOI: 10.1145/2934661.

[15] D. Böhme et al. “Scalable Critical-Path Based Performance Analysis”. In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. May 2012, pp. 1330–1340. DOI: 10.1109/IPDPS.2012.120.

[16] A. Calotoiu et al. “Using Automated Performance Modeling to Find Scalability Bugs in Complex Codes”. In: *Proc. of the ACM/IEEE Conference on Supercomputing (SC13), Denver, CO, USA*. ACM, Nov. 2013, pp. 1–12. ISBN: 978-1-4503-2378-9. DOI: 10.1145/2503210.2503277.

[17] *Extra-P repository*. URL: https://github.com/extra-p/extrap (visited on 11/08/2021).

[18] E. Saillard, P. Carribault, and D. Barthou. “PARCOACH: Combining static and dynamic validation of MPI collective communications”. In: *The International Journal of High Performance Computing Applications* 28.4 (2014), pp. 425–434. DOI: https://doi.org/10.1177/1094342014552204.

[19] C. T. Aitkaci et al. “Dynamic Data Race Detection for MPI-RMA Programs”. In: *Proceedings of the 28th European MPI Users’ Group Meeting, EuroMPI 2021, Germany, September 7th-8th, 2021*. ACM, 2021.

[20] *Paraver: A Flexible Performance Analysis Tool*. URL: https://tools.bsc.es/paraver (visited on 10/04/2021).

[21] V. Pillet et al. *PARAVER: A Tool to Visualize and Analyze Parallel Code*. Tech. rep. CEPBA-UPC. Departament d’Arquitectura de Computadors. Universitat Politècnica de Catalunya, 1995.

[22] *BSC-Tools: Performance Analysis Tools*. URL: https://tools.bsc.es (visited on 10/04/2021).

[23] *Paraver Trace Generation Manual*. URL: https://tools.bsc.es/doc/1370.pdf (visited on 10/04/2021).

[24] *Dimemas MPI Simulator*. URL: https://tools.bsc.es/dimemas (visited on 10/04/2021).

[25] *Extrae User Guide*. URL: https://tools.bsc.es/doc/html/extrae (visited on 10/04/2021).

[26] M. Radulovic et al. “PROFET: Modeling System Performance and Energy Without Simulating the CPU”. In: *Proc. ACM Meas. Anal. Comput. Syst.* 3.2 (2019). DOI: 10.1145/3341617.3326149.

[27] T. Grass et al. “MUSA: A Multi-Level Simulation Approach for next-Generation HPC Machines”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’16. Salt Lake City, Utah: IEEE Press, 2016. ISBN: 9781467388153.

[28] A. Rico et al. “On the Simulation of Large-Scale Architectures Using Multiple Application Abstraction Levels”. In: *ACM Trans. Archit. Code Optim.* 8.4 (Jan. 2012). ISSN: 1544-3566. DOI: 10.1145/2086696.2086715. URL: https://doi.org/10.1145/2086696.2086715.

[29] D. Sanchez and C. Kozyrakis. “ZSim: fast and accurate microarchitectural simulation of thousand-core systems”. In: *ISCA’ 13 Proceedings of the 40th Annual International Symposium on Computer Architecture*. 2013, pp. 475–486.

[30] S. Li et al. “DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator”. In: *IEEE Computer Architecture Letters* 19.2 (2020), pp. 106–109. DOI: 10.1109/LCA.2020.2973991.

[31] *Mitos repository*. URL: https://github.com/LLNL/Mitos (visited on 11/09/2021).

[32] *LLView website*. URL: http://www.fz-juelich.de/jsc/llview (visited on 09/23/2021).

[33] M. Stoffel and A. Mazouz. “Improving Power Efficiency Through Fine-Grain Performance Monitoring in HPC Clusters”. In: *IEEE International Conference on Cluster Computing, CLUSTER 2018, Belfast, UK, September 10-13, 2018*. IEEE Computer Society, 2018, pp. 552–561. DOI: 10.1109/CLUSTER.2018.00071.

[34] A. Netti et al. “From Facility to Application Sensor Data: Modular, Continuous and Holistic Monitoring with DCDB”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’19. Denver, Colorado: Association for Computing Machinery, 2019. ISBN: 9781450362290. DOI: 10.1145/3295500.3356191. URL: https://doi.org/10.1145/3295500.3356191.

[35] *SICM: Simplified Interface to Complex Memory*. URL: https://www.exascaleproject.org/research-project/sicm/ (visited on 11/30/2021).

[36] M. B. Olson et al. “Portable Application Guidance for Complex Memory Systems”. In: *Proceedings of the International Symposium on Memory Systems*. MEMSYS ’19. 2019, pp. 156–166. ISBN: 9781450372060. DOI: 10.1145/3357526.3357575. URL: https://doi.org/10.1145/3357526.3357575.

[37] J. Ren et al. “Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning”. In: *International Symposium on High-Performance Computer Architecture (HPCA)*. 2021.

[38] *Infiniband Architecture Specification*. Tech. rep. InfiniBand Trade Association, Nov. 2016.

[39] M. Nüssle et al. “An FPGA-based custom high performance interconnection network”. In: *2009 International Conference on Reconfigurable Computing and FPGAs*. Dec. 2009, pp. 113–118. DOI: 10.1109/ReConFig.2009.23.

[40] M. S. Birrittella et al. “Intel Omni-path architecture: Enabling scalable, high performance fabrics”. In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. Aug. 2015, pp. 1–9. DOI: 10.1109/HOTI.2015.22.

[41] J. M. Perez et al. “Improving the integration of task nesting and dependencies in OpenMP”. In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2017, pp. 809–818.

[42] M. Garcia, J. Corbalan, and J. Labarta. “LeWI: A Runtime Balancing Algorithm for Nested Parallelism”. In: *Parallel Processing, 2009. ICPP ’09. International Conference on*. Sept. 2009, pp. 526–533. DOI: 10.1109/ICPP.2009.56.

[43] *DLB: Dynamic Load Balance library*. URL: https://pm.bsc.es/dlb (visited on 10/28/2021).

[44] K. Feind and K. Mcmahoc. “An Ultrahigh Performance MPI implementation on SGI ccNUMA Altix systems”. In: *Computational Methods in Science and Technology (Special Issue)* (2006), pp. 67–70. DOI: 10.12921/cmst.2006.SI.01.67-70.

[45] *Cross-Process Memory Mapping (XPMEM)*. URL: https://code.google.com/archive/p/xpmem (visited on 11/30/2021).

[46] M. Sergent et al. “Efficient notifications for MPI one-sided applications”. In: *Proceedings of the 26th European MPI Users’ Group Meeting, EuroMPI 2019, Zürich, Switzerland, September 11-13, 2019*. Ed. by T. Hoefler and J. L. Träff. ACM, 2019, 5:1–5:10. DOI: 10.1145/3343211.3343216. URL: https://doi.org/10.1145/3343211.3343216.

[47] P. Shamis et al. “UCX: An Open Source Framework for HPC Network APIs and Beyond”. In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. 2015, pp. 40–43. DOI: 10.1109/HOTI.2015.13.

[48] *NVIDIA DGX Systems whitepaper*. URL: https://images.nvidia.com/aem-dam/Solutions/Data-Center/dgx-systems/dgx-systems-solution-brief.pdf (visited on 11/30/2021).

[49] A. Alexander V. Boukhanovsky, V. Krzhizhanovskaya, and M. Bubak. “Urgent computing for decision support in critical situations”. In: *Future Generation Computer Systems* 79 (2018), pp. 111–113. ISSN: 0167-739X. DOI: 10.1016/j.future.2017.11.003.

[50] E. Brun, S. Chauveau, and F. Malvagi. “PATMOS: A prototype Monte Carlo transport code to test high performance architectures”. In: *M&C 2017 - International Conference on Mathematics & Computational Methods Applied to Nuclear Science & Engineering, Jeju, Korea, April 16-20, 2017*. 2017.

[51] L. Hochstein et al. “Parallel programmer productivity: A case study of novice parallel programmers”. In: *SC’05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE. 2005, pp. 35–35.

[52] T. Ben-Nun et al. “Stateful Dataflow Multigraphs: A data-centric model for performance portability on heterogeneous architectures”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC’19)*. 2019.

[53] A. Ivanov et al. *Data Movement Is All You Need: A Case Study on Optimizing Transformers*. 2020. arXiv: 2007.00072 [cs.LG].

[54] A. N. Ziogas et al. “A Data-Centric Approach to Extreme-Scale Ab Initio Dissipative Quantum Transport Simulations”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC’19)*. 2019.

[55] *Trace Event Format*. URL: https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6I0nSsKchNAySU/preview (visited on 11/30/2021).

[56] *DynamoRIO*. URL: https://dynamorio.org/ (visited on 11/05/2021).

[57] T. Grass et al. “TaskPoint: Sampled simulation of task-based programs”. In: *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2016, pp. 296–306. DOI: 10.1109/ISPASS.2016.7482104.