# Porting applications to a Modular Supercomputer

## Experiences from the DEEP-EST project

A. Kreuzer, E. Suarez, N. Eicker, Th. Lippert (Eds.)

IAS Series
Band / Volume 48
ISBN 978-3-95806-590-1

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
Forschungszentrum

Schriften des Forschungszentrums Jülich
IAS Series Band / Volume 48

Forschungszentrum Jülich GmbH
Institute for Advanced Simulation (IAS)
Jülich Supercomputing Centre (JSC)

# **Porting applications to a Modular Supercomputer**
## Experiences from the DEEP-EST project

A. Kreuzer, E. Suarez, N. Eicker, Th. Lippert (Eds.)

Schriften des Forschungszentrums Jülich
IAS Series Band / Volume 48

ISSN 1868-8489 ISBN 978-3-95806-590-1

Bibliografische Information der Deutschen Nationalbibliothek.
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der
Deutschen Nationalbibliografie; detaillierte Bibliografische Daten
sind im Internet über http://dnb.d-nb.de abrufbar.

| | |
|---|---|
| Herausgeber und Vertrieb: | Forschungszentrum Jülich GmbH<br>Zentralbibliothek, Verlag<br>52425 Jülich<br>Tel.: +49 2461 61-5368<br>Fax: +49 2461 61-6103<br>zb-publikation@fz-juelich.de<br>www.fz-juelich.de/zb |
| Umschlaggestaltung: | Grafische Medien, Forschungszentrum Jülich GmbH |
| Titelbild: | ©vegefox.com – stock.adobe.com |
| Druck: | Grafische Medien, Forschungszentrum Jülich GmbH |
| Copyright: | Forschungszentrum Jülich 2021 |

Schriften des Forschungszentrums Jülich
IAS Series, Band / Volume 48

ISSN 1868-8489
ISBN 978-3-95806-590-1

Vollständig frei verfügbar über das Publikationsportal des Forschungszentrums Jülich (JuSER)
unter www.fz-juelich.de/zb/openaccess.

## Preface

The results described in this volume have been obtained within the research and development project named DEEP-EST, which stands for Dynamical Exascale Entry Platform – Extreme Scale Technologies. DEEP-EST is the third member of the DEEP family, initiated and coordinated by the Jülich Supercomputing Centre (JSC) at Forschungszentrum Jülich and financially supported by the European Commission through the FP7 and H2020 funding frameworks. The first DEEP project started at the end of 2011, was followed in 2013 by DEEP-ER (which stands for DEEP – Extended Reach), and then in 2017 by DEEP-EST.

It was mid of 2005, when the JSC began to realize its dual supercomputing strategy, with the purpose to allow for the coordinated operation of a general-purpose cluster and a highly scalable supercomputer. The first incarnation of the dual strategy came as an Intel cluster with Mellanox networking running ParaStation as the operating system, joined by a highly scalable IBM Blue Gene/P system. This combination enjoyed position 10 and position 2 on the Top500-list in June 2009 and November 2009, respectively. Scientifically, the dual strategy was driven by the insight that the class of highly scalable problems could be computed in a particularly cost-effective and energy-efficient manner on systems optimized for this purpose, such as BlueGene machines, and that for the class of complex and data-intensive problems, the then-emerging line of cluster computers was particularly well suited. Already then, many fields and research projects in computational science relied on both technologies simultaneously for simulation and data analysis, including, in particular, access to the same external data store.

Around 2010, we at JSC and our cluster partners started to think about how we could take the Jülich dual concept with its autonomous supercomputers, so far connected only by the shared data storage, into the future, especially also because an increasing gap between general purpose clusters and highly-scalable systems was emerging. In parallel, the first heterogeneous node designs appeared, consisting of CPUs and accelerators, which were of interest as possible processors for future highly scalable systems, but as far as graphics cards (GPU) were concerned, they could only be used as coprocessors, since CPUs and GPUs are statically assigned to each other. We realized that it would be much more beneficial to take the accelerators apart from the node as an autonomous unit and flexibly assign them to the CPUs. We made these ideas the basis of the proposal for the first DEEP project, which was approved and launched end of 2011.

As part of DEEP, our first cluster-booster prototype was built with an Intel Xeon-based general-purpose cluster running on a Mellanox network, connected through a specific network interface with the EXTOLL network of a 384-node system of Intel Xeon Phi processors (KNC); the latter we referred to as the "booster." In parallel, the full software stack was developed around the ParaStation cluster middleware that eventually led to the ParaStation Modulo system we enjoy today, and quite a few initial application use-cases were adapted and tested. In the follow-up DEEP-ER project, we continued work on the DEEP pilot architecture and its environment, focusing in particular on improving input-output and resilience capabilities.

Building on these successful results, the DEEP-EST project that ran from 2017 to 2021 generalized the Cluster-Booster concept in order to address the requirements of a wider variety of applications. The resulting Modular Supercomputing Architecture (MSA) developed in DEEP-EST is designed to meet the requirements of both large-scale simulations traditionally run on HPC systems and data-intensive workloads from the field of artificial intelligence like deep learning.

Within the DEEP-EST project, a hardware prototype with three computing modules, a Cluster Module, a Booster Module, and a Data Analytics Module, were designed, built and put into operation. What is more, the MSA software stack and programming environment was enhanced to support the latest GPU and FPGA acceleration technologies and to enable a more dynamical resource allocation. These developments were driven by the codesign input from the six application-development teams participating in the project.

Indeed, the experience of the application-development teams in adapting their codes to the DEEP-EST MSA system is among the most significant results of the project. Their experience is collected and summarized in this volume, which describes in its various chapters the applications used, the code-adaptations required for the MSA, the results of benchmarking campaigns on the DEEP-EST hardware prototype, and the lessons learned by each team throughout the process. In addition, the final chapter compiles best practices considered to be useful for future users of the DEEP-EST prototype in particular, and for users of any upcoming modular supercomputer in general. In fact, we believe that much of the lessons learned in this volume are applicable to all heterogeneous supercomputers, even if they are arranged in an old-fashioned monolithic fashion.

This book is intended for computer scientists at any stage of their careers who aspire to use large modular and/or heterogeneous supercomputers. The book is laid out as a sequence of independent chapters, one per application area, which do not necessarily have to be read one after the other. However, it is highly recommended to read Chapter one first, as it serves as an introduction and describes the hardware and software

stacks developed in the DEEP-EST project, which are used by all applications in the remaining chapters of this book.

The leadership of DEEP-EST owes a great debt of gratitude to all partners and members of the project, both those who are co-authors in some chapters of this book and those who are not explicitly mentioned here but who have participated and shown great commitment to the development of the project's various hardware and software solutions.

Above all, our thanks also go to the European Commission, which has supported us so much over the past 10 years and continues to do so in the follow-up project DEEP-SEA, the fourth member of the DEEP project family, together with many member states. We are particularly thankful for the guidance and support of the Project Officer responsible for the DEEP-EST project, Juan Pelegrín, and the external reviewers Anne-Claire Mireille Fouilloux, Maria Ángeles González Navarro, and John Barr. Their suggestions and their encouragement led to the publication of this book.

Jülich, October 2021

Prof. Dr. Dr. Thomas Lippert

Director of the Jülich Supercomputing Centre

## Table of Contents

# 1 The DEEP-EST project

**Estela Suarez[(1)], Anke Kreuzer[(1)], Norbert Eicker[(1,2)], Thomas Lippert[(1,3)]**

(1) Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH, Leo Brandt Strasse, 52428 Jülich, Germany

(2) Fakultät für Mathematik und Naturwissenschaften, Bergische Universität Wuppertal, Gaußstraße 20, 42119 Wuppertal, Germany

(3) Goethe-Universität Frankfurt, Frankfurt Institute for Advanced Studies (FIAS). Ruth-Moufang-Straße 1, 60438 Frankfurt am Main, Germany

e.suarez@fz-juelich.de

## 1.1 Introduction

*DEEP-EST* (standing for *Dynamical Exascale Entry Platform – Extreme Scale Technologies*) is a research and development project funded by the European Commission through the Horizon 2020 framework program. It has run for 45 months from June 2017 until March 2021 under the coordination of Forschungszentrum Jülich (FZJ) and with the participation of 16 institutions from 9 European countries (see Figure 1.1).


| **Forschungszentrum Jülich GmbH** *(coord.)* |
|---|
| ParTec Cluster Competence Centre GmbH |
| Intel Deutschland GmbH |
| Bayerische Akademie der Wissenschaften |
| Barcelona Supercomputing Center |
| Megware Computer Vertrieb und Service GmbH |
| Ruprecht-Karls-Universität Heidelberg |
| EXTOLL GmbH |
| The University of Edinburgh |
| Fraunhofer Gesellschaft zur Förderung der Angewandten Forschungs e.V. |
| Katholieke Universiteit Leuven |
| Stichting Astron, Netherlands Institute For Radio Astronomy |
| Association National Centre For Supercomputing Applications |
| Norges Miljo-Og Biovitenskaplige Universitet |
| Haskoli Islands |
| European Organisation for Nuclear Research |

**Figure 1.1: The DEEP-EST consortium**

DEEP-EST is, after its predecessors DEEP and DEEP-ER, the third member in the DEEP project series[1] funded by the European Commission and driven by a stringent codesign spirit, in which hardware-, software-, and application developers work tightly together to develop holistic solutions to today's HPC challenges. A new breed of HPC systems is needed to support the computation and data processing requirements of both traditional high performance computing (HPC) and emerging high performance data analytics (HPDA) workloads. To do so, DEEP-EST implements the **Modular Supercomputing Architecture (MSA)**, a novel system-level design to integrate heterogeneous resources and match the requirements of a wide spectrum of application fields, ranging from computationally intensive high-scaling simulation codes to data-intensive artificial intelligence workflows.

## 1.2 Modular Supercomputing Architecture (MSA)

The Cluster-Booster concept first implemented by DEEP broke with the traditional system architecture approach (based on replicating many identical compute nodes, possibly integrating heterogeneous processing resources within each node) by integrating heterogeneous computing resources in a modular way at the system level[2]. More precisely, it connected a standard HPC cluster based on general-purpose processors with a cluster of many-core processors or accelerators (the "booster") by way of a highly efficient and high-speed network. No constraints are put on the combination of Cluster and Booster nodes that an application may select, and resources might be reserved dynamically.

The Modular Supercomputer Architecture (MSA)[3] introduced in DEEP EST takes the Cluster-Booster architecture to the next step generalizing the concept to fulfil the requirements of a wider variety of applications from HPC and HPDA domains (see Figure 1.2). In the MSA several modules – each one tuned to best match the needs of a certain class of algorithms – are connected to each other at the system level to create a single heterogeneous system. Each module is a parallel, clustered system of potentially large size. A federated network connects the module-specific interconnects,

[1] www.deep-projects.eu

[2] N. Eicker, Th. Lippert, Th. Moschny, and E. Suarez, *The DEEP Project - An alternative approach to heterogeneous cluster-computing in the many-core era*, Concurrency and computation: Practice and Experience, Vol. 28, p. 2394–-2411 (2016), doi = 10.1002/cpe.3562. http://juser.fz-juelich.de/record/203150/files/concurrency-paper.pdf

[3] E. Suarez, N. Eicker, Th. Lippert, "*Modular Supercomputing Architecture: from idea to production*", Chapter 9 in Contemporary High Performance Computing: from Petascale toward Exascale, Volume 3, pp 223-251, Ed. Jeffrey S. Vetter, CRC Press. (2019) https://juser.fz-juelich.de/record/862856

while an optimised resource manager enables assembling arbitrary combinations of these resources according to the application workload requirements. This has two important effects: Firstly, each application can run on a near-optimal combination of resources and achieve excellent performance. Secondly, all the resources can be put to good use by combining the set of applications in a complementary way, increasing throughput and efficiency of use for the whole system.

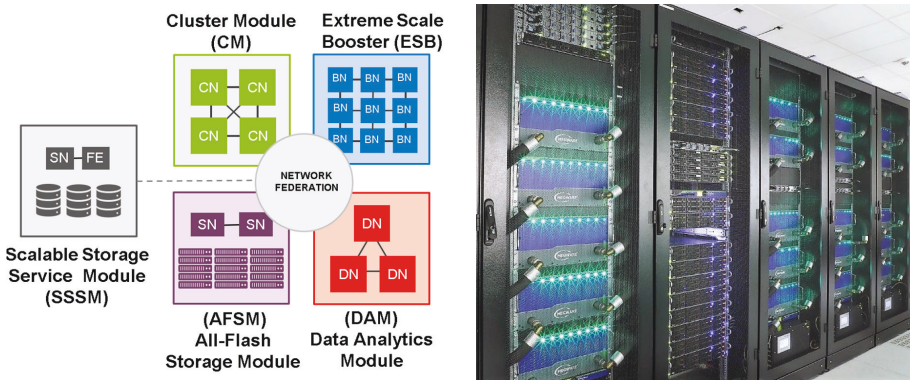


**Figure 1.2: The DEEP-EST prototype. *Left*: Architecture scheme. *Right*: picture at JSC's computer room**

## 1.3 Hardware

The DEEP-EST hardware prototype (Figure 1.2) has been defined in close co-design cooperation between applications, system software and system component architects. It includes three computing modules and two storage modules. The modules are connected through a high-speed network and, most importantly, operated with a uniform system software and programming environment. This enables applications to be distributed over several modules, running each part of its code on the best-suited hardware.

The computational core of the DEEP-EST system is given by the general purpose **Cluster Module (CM)**, the **Extreme Scale Booster (ESB)**, and the **Data Analytics Module (DAM)**. Their main characteristics are given in Table 1.1. The following subsections give an overview on the different components. More details can be found in the DEEP-EST Wiki[4].

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

| DEEP-EST system | CM | ESB | DAM |
|---|---|---|---|
| **Usage and design target** | Applications and code parts requiring high single-thread performance and a modest amount of memory, which typically show moderate scalability.<br>General purpose performance and energy efficiency are essential for the CM. | Compute intensive applications and code parts with regular control and data structures, showing high parallel scalability.<br>Energy efficiency, balanced architecture, packaging and hardware scalability are also important aspects in the ESB design. | Data-intensive analytics and machine learning applications and code parts requiring large memory capacity, data streaming, bit- or small datatype processing.<br>Flexibility, non-volatile memory and different acceleration capabilities are key features of the DAM. |
| **Node count** | 50 | 75 | 16 |
| **CPU type**<br>**CPU codename**<br>**Cores @frequency** | Intel Xeon 6146<br>Skylake<br>12 @3.2 GHz | Intel Xeon 4215<br>Cascade Lake<br>8 @2.5 GHz | Intel Xeon 8260M<br>Cascade Lake<br>24 @2.4 GHz |
| **Accelerators per node** | n.a. | 1× NVIDIA V100 GPU | 1× NVIDIA V100 GPU<br>1× Intel Stratix10 FGPA |
| **DDR4 capacity**<br>**HBM capacity**<br>**NVMe**<br>**Node max. mem BW** | 192 GB<br>n.a.<br>n.a.<br>256 GB/s | 48 GB<br>32 GB (GPU)<br>n.a.<br>900 GB/s (GPU) | 384GB+32GB(FPGA)<br>32 GB (GPU)<br>3 TB Intel Optane<br>900 GB/s (GPU) |
| **Storage** | 1x 512 GB NVMe SSD | 1x 512 GB NVMe SSD | 2x 1.5 TB NVMe SSD |
| **Network technology**<br>**Network Topology** | EDR-IB (100 Gb/s)<br>Fat-tree | EDR-IB (100 Gb/s)<br>Tree | EDR-IB (100 Gb/s)<br>Ethernet (40 Gb/s)<br>Tree |
| **Power /node**<br>**Cooling** | 500 W<br>warm-water | 500 W<br>warm-water | 1600 W<br>air |
| **Integration** | 1× Rack MEGWARE SlideSX-LC ColdCon | 3× Rack MEGWARE SlideSX-LC ColdCon | 1× Rack MEGWARE |

**Table 1.1: Main hardware features of the DEEP-EST modular prototype, in its final configuration[5].**

[5] During the project lifetime the DAM and one ESB partition featured an EXTOLL Fabri3 interconnect. For better user-experience and easier long-term maintenance in mind, after the end of the DEEP-

### 1.3.1 The Cluster Module (CM)

The CM is a general purpose HPC cluster with 50 nodes, each with two Intel® Xeon® Scalable (“Skylake” generation) Gold CPUs, 192 GB RAM and one 400 GB NVMe SSD. The nodes are interconnected by an InfiniBand EDR fabric with 100 Gbit/s bandwidth. The CPUs have relatively few cores (12 each) with a high clock frequency of 3.2 GHz. This module provides reliable performance and universal applicability with high single-thread performance, supporting highly complex and dynamic control flow HPC workloads. That means that application (parts) that do not fit well to the other more specialized modules (ESB or DAM) should be executed on the CM.

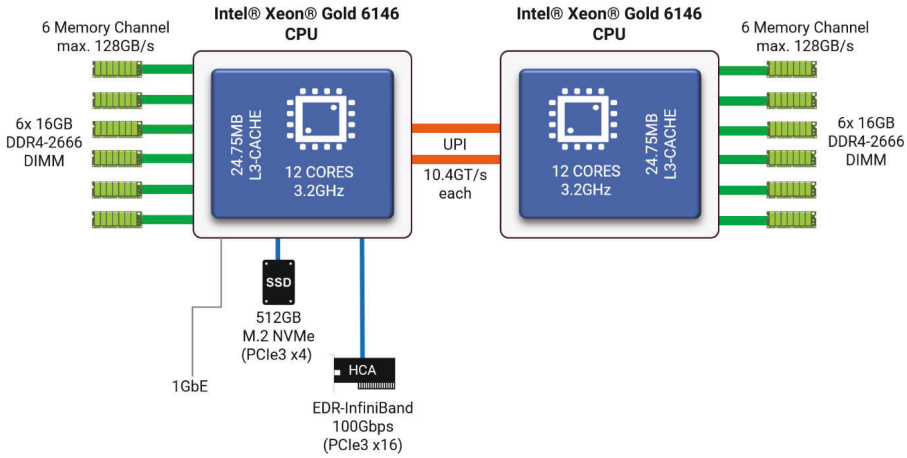


**Figure 1.3: Architecture of the CM node**

### 1.3.2 The Extreme Scale Booster (ESB)

The ESB targets the needs of highly scalable (parts of) applications and workloads and is the largest module in the DEEP-EST prototype. With its 75 nodes, each containing one Intel Xeon Scalable (“Cascade Lake” generation) Silver CPU, one NVIDIA® Tesla® V100 GPU, 48 GB of RAM, and one 512 GB SSD, the ESB is a highly scalable system, and provides very high computational throughput for applications with very wide parallelism and suitable control structures. Achieving high energy efficiency is a key objective of the system integration. The ESB nodes are thus designed for GPGPU

EST project it was decided to reconfigure the prototype with a uniform, InfiniBand-only interconnect across all modules. Since this book targets future users of this MSA platform, this final configuration is the one described in this table and taken into account across the full volume.

centric computing, with the vast majority of compute operations running on the high-end GPGPU, and the Xeon CPU controlling I/O and network communication (including MPI), as well as managing the GPGPU device attached via a 16-lane PCIe generation 3 link. It targets highly scalable (data, thread and task parallel) HPC applications (or parts thereof) and workloads. The nodes are interconnected via InfiniBand EDR providing a 100 Gbit/s bandwidth.

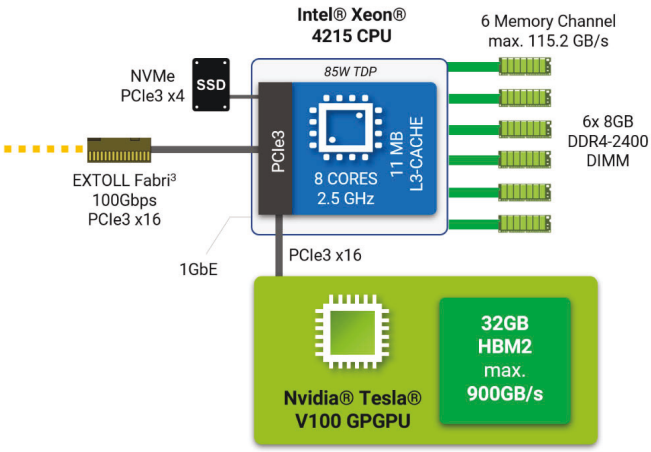


**Figure 1.4: Architecture of the ESB node**

### *1.3.3 The Data Analytics Module (DAM)*

The DAM is a specifically designed cluster for high-performance data analytics (HPDA) and artificial intelligence (AI) workloads. It is composed of 16 nodes, each with two Intel Xeon Scalable ("Cascade Lake" generation) Platinum CPUs, one NVIDIA Tesla V100 GPU, one Intel® Stratix® 10 FPGA and 384 GB RAM plus 3 TB of Intel® Optane Persistent Memory. The module uses two interconnects in parallel: 100 Gbit/s InfiniBand and 40 Gbit/s Ethernet. In contrast to the ESB, each DAM nodes provides a duo of high-end CPUs, plus one high-end GPGPU and FPGA accelerator each. It is designed for applications that share the computation load between CPUs and accelerators, require large main memory, or can profit from using an FPGA.

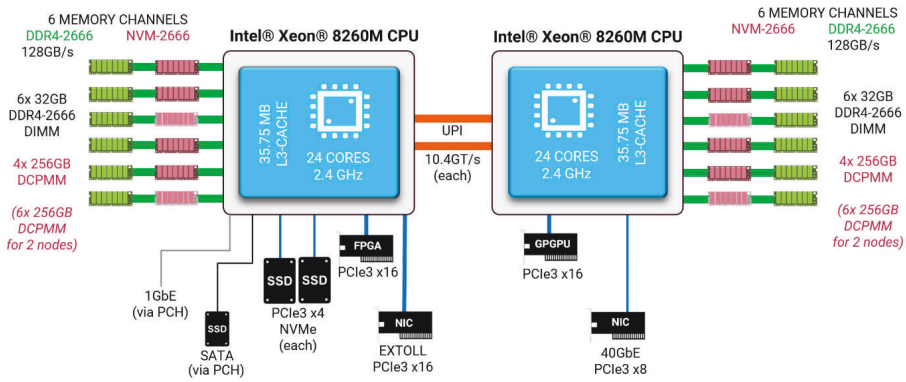


**Figure 1.5: Architecture of the DAM**

### *1.3.4 The Scalable Storage and Service Module (SSSM)*

The SSSM provides storage capacity based on conventional spinning-disks and uses a parallel file system (BeeGFS). It provides storage capacity for the workloads while they are running on the DEEP-EST prototype. It is not included in any backup scheme. The SSSM is accessible under `/work`. For more information on the storage and file system see Chapter 8, Section 8.6.

### *1.3.5 The All-Flash Storage Module (AFSM)*

The AFSM complements the SSSM and is based on modern PCIe3 NVMe SSD storage devices to provide scalable, high-performance global I/O and storage capabilities and better match the computational power of the DEEP-EST Prototype modules for data- and storage-intensive applications and workloads. On the ASFM, the BeeGFS global parallel file system is used to make 1.8 PB of data storage capacity available, supported by two Metadata servers and six volume data server systems which are interconnected by a 100 Gbps EDR-InfiniBand fabric. The AFSM is integrated into the DEEP-EST EDR fabric topology of the CM, ESB and DAM.

## 1.4 Software

The DEEP-EST software architecture (see Figure 1.6) was designed in the early stages of the project taking careful account of the requirements determined from the co-design applications. The languages, programming and parallelization paradigms, libraries, and needed tools were integrated in the stack, and platform adaptations were identified and implemented wherever needed.

The lower layers of the software stack have been adapted to provide the best support for the underlying hardware, while hiding these modifications from the end user by keeping the higher-level layers of the stack in APIs familiar to users. For example, low-level interconnect management features were developed and integrated with MPI, but without changing the MPI calls that are directly used by the application developers.

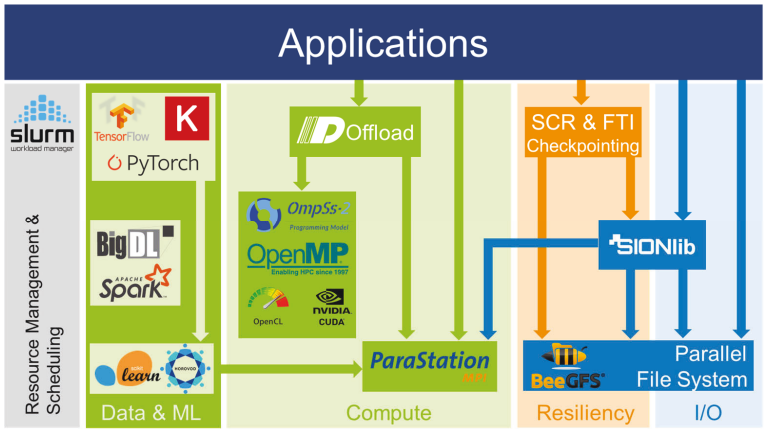


**Figure 1.6: Software stack in the DEEP-EST project**

The MSA software stack enables application developers to map the intrinsic scalability patterns of their applications and workflows onto the hardware: highly parallel code parts run on the large-scale, energy-efficient Booster, while less scalable code parts can profit from the high single-thread performance of the Cluster, or from the high memory capacity of the Data Analytics Module. Users can freely decide how many nodes to use in each module, so that the highest application efficiency and system usage can be achieved[6].

[6] A. Kreuzer, J. Amaya, N. Eicker, E. Suarez, *Application performance on a Cluster-Booster system*, 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), HCW (20th International Heterogeneity in Computing Workshop), Vancouver (2018), p: 69 - 78. [doi: 10.1109/IPDPSW.2018.00019] https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8425386

### *1.4.1 Resource management and scheduling*

Resource management and job scheduling play a pivotal role for the success of the overall MSA. Existing job schedulers guarantee efficient use of monolithic supercomputers. However, the MSA requires capabilities to manage heterogeneous resources, to enable co-scheduling of resource sets across modules, and to handle dynamically varying resource-profiles. For this reason, the resource management and scheduling software packages psslurm and Slurm[7] have been widely extended to enable an optimal utilization of the heterogeneous resources in a modular supercomputer. These extensions include the ability to dynamically allocate nodes in all compute modules, as well as global resources. Also, better support for the Multiple-Program Multiple-Data (MPMD) programming paradigm has been implemented, which is needed for heterogeneous jobs running different executables on different parts of the job allocation. Furthermore, a new switch (`--delay`) and a clause (`--module list`) have been implemented in Slurm: the former enables workflows consisting of jobs with data dependencies to overlap the executions of their different steps, so that data can be transferred directly between them without writing and then again reading them from the file system; the latter provides in order of preference the list of modules on which the job-steps can run, giving the scheduler more flexibility in the allocation of resources, depending on the demand in the given point on time.

A user may need to pre-process data before running a long simulation, then perform data-reduction, and ultimately visualize the final result. Running these codes on different modules consists simply on indicating to the scheduler on which nodes to execute each step. Data is typically transferred between the jobs of a workflow via the file-system, which means that data is written onto the external storage in one step, and then re-read in the next workflow-step. Taking into account the time and power consumed in such write-read operations, this approach is not necessarily the fastest and certainly not the most energy efficient. Because of that, the DEEP-EST project has investigated the potential implementation and benefits of directly transferring data between workflow steps via MPI.

Workflows running on a modular system architecture can benefit from dynamic scheduling support. Workload trace files including the different project features and specific metrics for workflow analysis were generated and analysed, thus comparing results from modular and homogeneous systems[8]. Special attention was put on

[7] Slurm. https://slurm.schedmd.com/documentation.html

[8] M. D'Amico and J. C. Gonzalez, *Energy hardware and workload aware job scheduling towards interconnected HPC environments*, IEEE Transactions on Parallel and Distributed Systems, doi: 10.1109/TPDS.2021.3090334. https://arxiv.org/abs/2106.12007

evaluating the scheduling features developed within DEEP-EST: module flexibility and workflow dependencies. The system configuration chosen for the scheduling modelling was based on the characteristics of the DEEP-EST prototype and the workload-mix reproduced elements of project's applications. The analysis showed that the new scheduling features provide benefits for application workflows without penalizing traditional jobs.

### *1.4.2 Programming Environment*

The DEEP-EST programming environment has been designed to provide all the functionality required by the co-design applications in the most user-friendly possible manner. The hardware complexity of the MSA is abstracted behind the interfaces and parallel programming paradigms that have become the de-facto standard in HPC: MPI and OpenMP. Specific implementations of these standards have been extended to achieve the maximum application performance on the hardware components constituting the DEEP-EST prototype. MPI and OpenMP are complemented by parallel programming tools supporting acceleration devices (CUDA, OpenACC, OpenCL) and by frameworks for machine learning and deep learning applications (TensorFlow, PyTorch, Keras, Horovod, etc.).

The MSA programming paradigm is based on MPI, in particular using the **ParaStation MPI** implementation[9,10]. In DEEP-EST, ParaStation has been made network topology aware at different levels of the software stack. Besides providing means and extensions for MPI applications to adapt their program flow by creating communicators reflecting the modular architecture, collective MPI operations have been optimised for modular systems and, in particular, those using the latest generation GPUs. For example, ParaStation MPI has been extended with CUDA awareness features to improve both productivity and performance of hybrid MPI codes.

The **OmpSs-2** programming model[11], spearhead of the OpenMP standard and developed by BSC, has been enhanced in the areas of tasking, programmability, support for hardware accelerators, and support for distributed shared memory systems using MPI. A new scheduler-design and dependency system has improved the performance and scalability of the OmpSs-2 runtime on many-core processors. In

[9] ParaStation Modulo. https://par-tec.com/software/

[10] S. Pickartz, Virtualization as an enabler for dynamic resource allocation in HPC, Dissertation, RWTH AachenUniversity, Aachen, 2019. https://doi.org/10.18154/RWTH-2019-02208.

[11] F. Sainz, J. Bellón, V. Beltran, and J. Labarta, "Collective Offload for Heterogeneous Clusters", 2015 IEEE 22nd International Conference on High Performance Computing (HiPC), p. 376-385 (2015) [doi = {10.1109/HiPC.2015.20}]. https://www.bsc.es/printpdf/research-and-development/publications/collective-offload-heterogeneous-clusters

addition, a new lightweight instrumentation plugin has been created to analyse OmpSs-2 applications, and the `taskloop` directive has been extended to support data dependencies. To ease the programming of accelerators, experimental support for OpenACC and array reductions for CUDA have been integrated into the main OmpSs-2 distribution. Also, the TAMPI library has been extended to support MPI RMA by supporting one-sided operations. Many of the new OmpSs features developed in the context of the DEEP-EST project have been already presented to the OpenMP committee for a future inclusion into the OpenMP standard.

Data analytics/machine learning components and frameworks required by the DEEP-EST applications and early access users have been installed and regularly updated on the hardware prototype. Also, the Intel oneAPI implementation has been installed and experiments were done to test its use for programming GPGPU and FPGA accelerators. It is worth mentioning that, in the same manner as it is done on the JSC production machines, the full DEEP-EST software stack has been integrated on EasyBuild, which is used for the maintenance and regular software updates on the prototype.

### *1.4.3 I/O and resiliency*

DEEP-EST has also addressed the topics of I/O and resiliency. The efficient management of data between different modules poses a great challenge for I/O systems, such as **BeeGFS** and **SIONlib**, which leverage new non-volatile memory technologies to cope with this new scenario. Moreover, traditional check-pointing libraries (e.g. FTI or SCR) have been enhanced with new features and a simpler interface to deal with new application requirements.

The European BeeGFS parallel file system, developed by FHG-ITWM, has been enhanced with new features to support storage pools and various storage hardware. BeeOND has also been re-implemented and integrated into the SLURM job manager, allowing system users to create a temporary BeeGFS file system on their allocated nodes with the options that suits their jobs best. Furthermore, the time series-based monitoring solution for BeeGFS (beegfs-mon) has been implemented, released, and integrated into the DCBD monitoring system. Users can now investigate the current and past status of the file system using Grafana panels. JUELICH's SIONlib library has been extended with a new MSA-aware algorithm for the selection of collector processes for collective I/O. The algorithm is portable and relies on platform specific plug-ins to identify processes, which run on parts of the system that are well suited for the role of I/O collector. In reaction to the GPU-based ESB concept, SIONlib's read and write functions have been made CUDA aware. They now allow the user to pass

input and output buffers that reside on a CUDA device and transparently handle the transfer of data in that case.

To improve resiliency, incremental checkpoint (iCP) and differential checkpointing (dCP) features have been implemented in the **FTI** library. A theoretical model has been created that accurately predicts the overhead reduction for dCP depending on performance characteristics of the architecture and shows a linear dependency between the reduction of overhead and the data reduction factor. This approach introduces the advantage that the checkpoint data of former checkpoints is preserved inside the checkpoint file. Furthermore, an HDF5 interface has been developed inside FTI, which enables writing with all processes into one shared file.

### *1.4.4 Benchmarking, performance analysis, modelling and monitoring tools*

A benchmark suite composed of a wide range of synthetic benchmarks and selected applications has been integrated in the **JUBE** benchmarking environment and has periodically run on the DEEP-EST prototype to measure its performance and identify potential variations. A visualization environment for the results was put in place to ease the interpretation of data, automatic emails were sent when something was not working, and automatic backups of the benchmarking results were implemented. For instance, thanks to these benchmarking activities, drawbacks with the initial BeeGFS file system configuration were identified and the sweet-spot for its configuration was found. The benchmarks were also used to assess the performance of the DEEP-EST prototype in detail.

Traces of application workloads were used to conduct efficiency analysis and project the performance of the applications at large scale. With this analysis, low parallel efficiency was identified and communicated to the application developers, which could identify and address its sources. Projection data allowed to predict the performance of the applications improved within DEEP-EST. Furthermore, the **Extrae** instrumentation software tools from BSC were extended in order to properly instrument CUDA codes.

Last, but not least, new system-monitoring capabilities have been created. The **DCDB** and **Wintermute** frameworks from BADW-LRZ have been further developed to reach a production-ready state within the project, and all components and plug-ins required for the DEEP-EST prototype have been designed and implemented. In addition to all data supplied natively by DCDB, the sensors exposed by BeeGFS covering file system activity have been integrated. The collection of all this wide variety of system-monitoring information, is accessible to users and operators through the user-friendly visualization tool Grafana. Furthermore, the monitoring tools have been integrated with the resource manager to enable energy-saving scheduling mechanisms. With the

DEEP-EST monitoring infrastructure, full control over the system utilization has been provided, opening opportunities for optimised operation policies. The power consumption of an application running on an HPC system depends on the amount of resources that it uses, and the model in which these resources are run, cooled, and operated. For this reason, an energy-model has been developed to evaluate the energy used by applications on each of the modules of the DEEP-EST prototype, assuming different operational frequencies and configurations.

## 1.5 Co-design Applications

For the DEEP-EST project several important and ambitious scientific codes from the HPC and HPDA area have been selected as the DEEP-EST co-design applications. Most of the codes combine HPC computation with advanced data processing and analytics. Thus, they do consist of multiple parts with different resource requirements and are eminently suitable to assess the potential of the MSA and the DEEP-EST prototype. The applications belong to six scientific fields:

- Neuroscience: In DEEP-EST, three applications were used to simulate functional models of brain structure. The NEST simulator investigates the dynamics of brain-scale neuronal network models. It does so at the level of resolution of neurons and synapses, where neurons are brain cells connected to each other by synapses. NEST is combined with two types of in situ analysis: computation of electrical local field potentials using the Arbor and HybridLFPy packages, and statistical analysis of spike activity using the Elephant package.
- Molecular Dynamics (MD): A MD simulation generally tracks the trajectories of many particles evolving over time. It solves differential equations of motion in time steps. GROMACS is one of the world's best MD software packages. It is a toolbox allowing users to prepare the structure that they want to simulate, run the simulation and analyse the results at the end.
- Radio Astronomy: In DEEP-EST, two parts of the imaging pipeline of the LOFAR radio telescopes were studied: the correlator and the imager. The correlator combines the data from all receivers and operates on streaming data, normally in real time. First it performs filtering, then corrections, and finally correlates the data from multiple receivers. The imager creates sky images partitioning incoming data in small blocks that are convolved and gridded onto small subgrids. These subgrids are fast Fourier transformed and then added to a large grid, which is finally inversely FFTed to a sky image.
- Space Weather: The space weather workflow consists of three applications: DLMOS, xPic and GMM. DLMOS is a Deep Learning Model of the Solar Wind

to forecast the plasma conditions at the orbit of the Earth from images of the Sun. The particle-in-cell code xPic consists of a field solver that calculates in a Cartesian grid the Maxwell equations of electromagnetism, and a particle solver calculating the motion of billions of charged particles using Newton's equations of motion. The large data volume generated by xPic's particle solver, is analysed with the machine learning model GMM.

- Data Analytics in Earth Science: In DEEP-EST, three applications are used: NextDBSCAN, NextSVM and Deep Learning (DL) frameworks. NextDBSCAN is a new parallel DBSCAN algorithm used for density-based clustering of large three-dimensional point-clouds. NextSVM is a new parallel Support Vector Machine (SVM) used for supervised learning classification tasks with labelled datasets (such as remote sensing images). The TensorFlow framework with the Keras extension is used for computer vision.
- High Energy Physics: CMSSW is the software framework for the Compact Muon Solenoid (CMS) Experiment at CERN. In DEEP-EST, two workflows were used: CMS Reconstruction and CMS Classification. The former takes the raw data coming out of the CMS detector and builds high level physics objects, which are then used for the physics analysis. The CMS Classification takes the reconstructed quantities as input and tries to identify the type of collision event. This is an analytics type of workflow that involves the use of Deep Learning for the purpose of classification.

The above mentioned applications have been analysed in detail to find out e.g. how to best map them to the modules of the DEEP-EST prototype, potentially splitting the application into separate parts (e.g. HPC computation and data analytics). Based on the codes requirements, co-design input was provided through a detailed questionnaire covering, amongst other, aspects such as: primary metric for success (e.g., throughput, accuracy, etc.); programming languages and parallelism paradigms used; rate, type, and volume of inter-process communication; computation-to-communication balance; and I/O requirements (see deliverable D1.1). Later in the project, while the DEEP-EST prototype and its SW were in development, in depth co-design discussions took place around specific design questions, e.g., preferred configuration of the DCPMM non-volatile memory; preferred network topology; kind of collective operations between MSA modules, etc.

In parallel to this co-design feedback, the application codes were adapted to the target hardware, ported to the actual DEEP-EST prototype and its software environment, and then optimised. The results of all these experiences are reported in Chapters 2 to 7 of this book.

## 1.6 Summary and outlook

The DEEP-EST project has developed the Modular Supercomputing Architecture (MSA) deploying a hardware prototype and implementing its software stack following the codesign input of six application development teams.

From the data centre operator's point of view, the MSA has several advantages. First of all, the better and more efficient system utilisation made possible by the MSA will immediately benefit the data centre operator, leading to a higher ROI (return on investment). To maximise this effect, it is secondly possible to optimise the system configuration, i.e. the number and characteristics of modules to the best match of the specific requirements of the centre and its application portfolio and mix. Additionally, maintenance of individual modules is possible without disturbing the rest of the system, reducing the overall down-times of the machine. Furthermore, the long-term sustainability is improved: new modules can be added to an existing system and old ones substituted at different points in time, keeping the rest of the system and its central resources (e.g. storage) for a much longer lifetime. At some point this might require bridging between older and newer network generations. Finally, procurement processes, which typically depend on different funding sources (e.g. regional, national, project-bound, etc.) can be split and handled individually for the independent modules in an easier way.

From the user's perspective, DEEP-EST provides a very flexible architecture that can match the requirements of very diverse classes of applications, making use of the modules according to their respective needs. The six HPC and HPDA applications in DEEP-EST have tested a variety of different scenarios. Monolithic applications may well use only one of the modules, but more complex, multi-physics or multi-scale applications distribute their code-constituents among several modules of the system and achieve better scalability and efficiency. This also offers the opportunity for more complex workflows or to conduct simulation and data analysis/visualisation concurrently, with the high-speed connection between different modules facilitating necessary data transfers.

Following the success of the DEEP project series, the Modular Supercomputing Architecture is already being applied in production systems. The JUWELS Booster[12] – as of May 2021 the fastest computer in Europe – builds up together with the JUWELS Cluster a Petascale-level modular system. Also, the recently installed EuroHPC

[12] https://apps.fz-juelich.de/jsc/hps/juwels/booster-overview.html

Petascale system MeluXina is modular[13], and more supercomputers in Europe and worldwide have been announced to follow the same philosophy[14],[15].

The implementation of MSA in large-scale production systems is not the end of its development roadmap, which continues in various upcoming EuroHPC JU projects. The software environment for MSA-platforms will be enhanced within the SEA-projects, which started in April 2021: DEEP-SEA is the direct continuation of the software efforts in DEEP-EST and aims at easing application porting to MSA systems and making their programming environment more dynamic by leveraging more malleability and composability[16], IO-SEA will improve the IO-capabilities of MSA systems with a novel data management and storage platform based on object store support, hierarchical storage management (HSM) and intelligent data placement[17]; RED-SEA finally will develop next generation European network technologies with better capabilities for intra- and inter-module communication[18]. Furthermore, at least two of the three pilot projects selected for funding within the EuroHPC-2020-01 call[19] will apply a MSA approach: the EUPEX project will build a modular pilot system integrating European technologies (including the EPI general purpose processor, ParaStation Modulo, etc.), while the HPCQS project will integrate a Quantum Module into an existing MSA system[20].

With this elaborated development roadmap, the Modular Supercomputing Architecture and the overall results of the DEEP-EST project are in the best possible position to be part of the first European Exascale platforms.

[13] https://eurohpc-ju.europa.eu/news/meluxina-live-eurohpc-ju-supercomputer-luxembourg-operational

[14] Slide 40 in https://www.r-ccs.riken.jp/R-CCS-Symposium/2019/slides/Wang.pdf

[15] https://www.hpcwire.com/2021/02/25/japan-to-debut-integrated-fujitsu-hpc-ai-supercomputer-this-spring/

[16] https://cordis.europa.eu/project/id/955606

[17] https://cordis.europa.eu/project/id/955606

[18] https://cordis.europa.eu/project/id/955776

[19] https://eurohpc-ju.europa.eu/calls/advanced-pilots-towards-european-exascale-supercomputers-pilot-quantum-simulator

[20] The EUPEX and HPCQS projects are in the GA-preparation phase at the time of writing.

## 1.7 Acknowledgements

The authors thank all the institutions and individuals involved in the DEEP series of projects and, in particular, all the members of the DEEP-EST team who have contributed to the development of the MSA architecture, its prototype hardware implementations and its software environment.

This work has been partially funded by the European Union's Seventh Framework (FP7/20017-2013) and Horizon 2020 Framework Programmes for Research and Innovation under grant agreements 287530 (DEEP), 610476 (DEEP-ER), 754304 (DEEP-EST), and 955606 (DEEP-SEA). The present publication reflects only the authors' views. The European Commission is not liable for any use that might be made of the information contained therein.

# 2 Neuroscience with NEST, Arbor and Elephant

**Hans Ekkehard Plesser, Susanne Kunkel, Håkon Mørk**
Norges miljø- og biovitenskapelige universitet, NMBU, Norway

hans.ekkehard.plesser@nmbu.no

## 2.1 Introduction

The long-term goal of the neuroscience work in DEEP-EST is to provide an optimised setup for the integrated simulation and analysis of large-scale brain activity[21]. Such in situ analysis is essential to facilitate the interactive investigations of brain dynamics, where scientists can observe network activity while a simulation is running and interact with it to ensure that dynamics stay within relevant regimes. In DEEP-EST, our focus was on simulations of functional models of brain structure using the NEST simulator[22] combined with two types of in situ analysis: computation of electrical local field potentials using the Arbor[23] and HybridLFPy packages[24] on the one side, and statistical analysis of spike activity using the Elephant package[25] on the other.

## 2.2 Application structure

### *2.2.1 NEST*

NEST is a simulation code for the investigation of the dynamics of brain-scale neuronal network models, as for example the recently published multi-area model[26]. NEST operates on the level of resolution of neurons and synapses, where neurons are brain cells connected to each other by synapses.

The simulator considers brain tissue as an abstract assembly of nodes (neurons) and connections (synapses) or, in other words, a directed graph. The neurons in these simulations are point neurons, i.e. the state of a node changes according to a set of

[21] Suarez, E. et al. (2021), „Modular Supercomputing for Neuroscience“, *Lecture Notes in Computer Science,* 2019 BrainComp Conference, Cetraro, Italy Springer International Publishing, 10.1007/978-3-030-82427-3_5

[22] http://www.nest-simulator.org/

[23] Akar, NA (2018) arXiv:1901.07454 [q-bio.NC]

[24] Hagen E et al. (2016) *Cerebral Cortex*, 26(12) pp. 4461–4496.

[25] http://elephant.readthedocs.io/

[26] Schmidt M et al. (2018) Brain Struct Funct 223: 1409.

ordinary differential equations (ODE), without taking into account the complete morphology of the cell.

The interaction between nodes is mediated by stereotyped events in the form of delayed delta pulses. These so-called action potentials (or spikes) are emitted by the nodes (neuronal activity) and propagated along the connections. The interaction strength (synaptic weight) can either be static or dynamic (synaptic plasticity) and depends on the activity of the two neurons joined by the connection.

NEST does not implement a specific network model but provides the user with a range of neuron and synapse models and efficient routines to connect them to complex networks with on the order of ten thousand incoming and outgoing connections for each neuron. Concrete network models and the corresponding simulation experiments are specified by model description scripts. These scripts are written either in NEST's built-in simulation language SLI (based on PostScript) or using the Cython-based Python interface PyNEST[27,28], with PyNEST being the default interface.

A published example of a large-scale network model is the multi-area model[26], which was relevant also in the context of the DEEP-EST project. It is the first multi-scale model of vision related brain areas and comprises approximately 4 million neurons and 6000 incoming synapses per neuron, where neurons emit on average 14.6 spikes/s. Each individual area is represented by a modified version of the Potjans-Diesmann model[29], a microcircuit model corresponding to a cortical network under a surface of 1 $mm^2$. The microcircuits representing the areas differ in neuron numbers and connection probabilities. The minimal synaptic transmission delay in the network is 0.1 ms biological time, i.e., the time simulated in the biological system. This requires frequent MPI communication of spikes (every 0.1 ms biological time). In terms of wallclock time, MPI communication occurs at approximately 10–30 ms intervals, depending on the activity level in the neuronal network. Due to long transients in the network dynamics the model needs to be simulated for 100 s biological time.

The NEST code base is open source and under continuous development in order to enable the investigation of novel models and theories in Computational Neuroscience on the one hand, and to meet the requirements of new computer hardware on the other hand. Since release 2.16, the NEST 5$^{th}$ generation simulation kernel (5G)[30] is included, which achieves excellent scaling with respect to memory usage and good scaling with respect to runtime on the largest supercomputers currently available for academic

[27] Eppler, JM et al. (2008) Front. Neuroinform. 2:12.

[28] Zaytsev YV and Morrison A (2014) Front. Neuroinform. 8:23.

[29] Potjans TC and Diesmann M (2014) Cereb. Cortex 24, 785–806.

[30] Jordan J et al. (2018) Front. Neuroinform. 12:2.

research. The key step from the previous kernel used in NEST releases 2.6.0–2.14.0 to the 5G kernel is a new connectivity representation and spike exchange scheme using directed communication based on MPI_Alltoall().

### 2.2.2 Arbor/HybridLFPy

Arbor simulates compartmental neuron models. This means that the spatial structure of each neuron is represented as a spherical cell body (soma), to which an arbitrary number of dendritic trees are attached. Each dendritic tree consists of segments, i.e. tubes or cables, of a given length and radius; in the simulation, each segment is represented by a configurable number of compartments. Each segment is either connected to one other segment at each of its ends (linear cable) or to several segments at its far end (branching point; far end: end pointing away from the soma). Electric currents flow along the cables formed by the dendritic tree. This current flow is described by ordinary differential equations, with one set of equations for each compartment, coupled to neighbouring compartments. The main task of Arbor is to solve the resulting system of ODEs; this task is highly amenable to vectorisation. In addition, Arbor also transmits spikes between neurons via synapses; this mechanism is of lesser importance for our purposes because HybridLFPy is based on simulating the dynamics of disconnected compartmental neurons based on spike input generated by NEST.

HybridLFPy computes mesoscopic electrical brain signals, called local field potentials (LFPs) based on the network dynamics simulated using NEST. Specifically, spike trains generated by neurons in a NEST simulation, using highly connected point neurons are fed into detailed models of unconnected neurons simulated using Arbor to compute the electrical currents passing through the cell membrane at different locations. From these currents, HybridLFPy then computes the LFP at different locations in a piece of brain tissue using electrostatic principles.

### 2.2.3 Elephant (ASSET)

Elephant is a pure Python library for the statistical analysis of spike activity of neurons. It can be installed using standard Python distribution tools. Elephant implements a wide and growing range of analysis methods. We focus mainly on the calculation of cross-correlations between spike trains and the detection of repeated patterns of spike activity across groups of neurons, so-called synfire chains.

Cross-correlations are detected using standard approaches, either implemented directly in Python or using NumPy convolution algorithms. Except for possible thread-parallelisation provided by the NumPy convolution implementation, cross-correlation algorithms are purely serial at present.

Detection of synfire chains uses the ASSET algorithm[31] in an optimised version[32], replacing the non-optimised version currently included in the release version of Elephant. The optimised algorithm uses MPI4Py for parallelisation.

## 2.3 Application mapping

Traditionally, NEST simulations have two distinct phases: a network construction (build) phase and a simulation phase. The key part of the build phase is the construction of network connectivity, i.e., building in largely random order a hierarchical data structure representing connections between neurons; each connection is represented only on the thread managing the connection's target neuron.

During the simulation phase, differential equations for the individual neurons are updated and spikes emitted according to a threshold criterion. Information on emitted spikes is exchanged between MPI processes and threads in steps of the minimal synaptic delay in the network, which is the maximum interval permitted by causality. Spikes are delivered to target neurons in parallel, each virtual process being responsible for delivery to the set of neurons it manages. This delivery process entails essentially random accesses to the connectivity data structure.

For the fifth generation (5G) kernel, we distinguish a third phase, called initialization phase, which comprises all necessary initialization processes at the beginning of a NEST simulation before the actual simulation takes place. In the NEST 5G kernel (NEST release 2.16), connectivity information, which is available only on the postsynaptic side after the build phase, needs to be transferred to the presynaptic side in order to enable directed communication of spikes during simulation. The transfer of connectivity data involves at least one round of MPI_Alltoall() communication, which makes the initialization phase a non-negligible component.

In the benchmarks hpc_benchmark.sli and hpc_mam_benchmark.sli, build phase and initialization phase take up a significant amount of the total runtime as the neuronal networks are simulated only for one second of biological time. In simulations of the multi-area model, build phase and initialization phase require only a small fraction of the total runtime as the network is simulated for 100 s of biological time.

To enable the interaction of NEST with Arbor/HybridLFPy (see Figure 2.1), a small fraction of the connectivity details of the multi-area network, which is available after the build phase of NEST, needs to be communicated, where HybridLFPy maps the connectivity to the detailed neuron models.

[31] Torre E et al. (2016) PLoS Comput Biol 12(7): e1004939.

[32] Canova C et al. (2017) ASSET for JULIA: executing massive parallel spike correlation analysis on a KNL cluster. Poster presented at HBP Summit 2017.

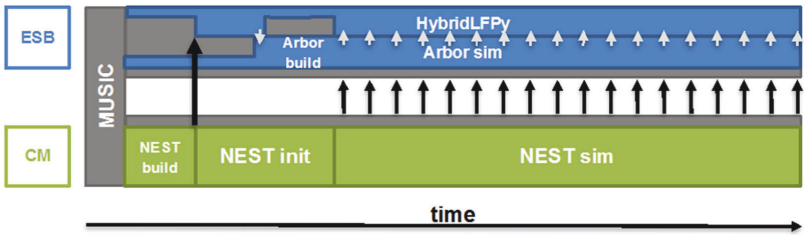


**Figure 2.1: Schematic workflow of NEST and Arbor/HybridLFPY in the MSA**

During the simulation phase NEST needs to communicate spikes from a fraction of the neurons of the multi-area model to Arbor or Elephant. Communication takes place frequently and is coordinated by the MUSIC library (see Figure 2.1 and Figure 2.2). We estimate that the total amount of data that needs to be communicated from CM to ESB or DAM in each communication round is negligible (about 1 kB if we assume communication every 0.1 ms of simulated time).

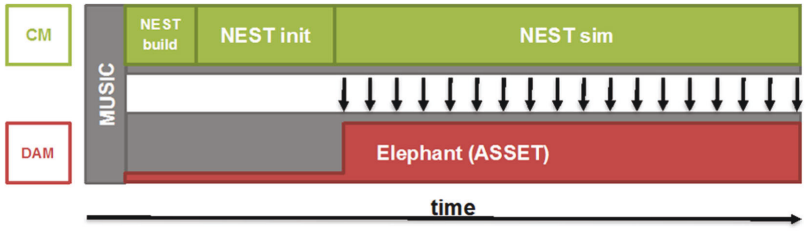


**Figure 2.2: Schematic workflow of NEST and Elephant (ASSET) in the MSA**

NEST (on CM) and Arbor/HybridLFPy (on ESB) start to run at the same time. While NEST constructs neurons and connections, Arbor instantiates neuron models. After the build phase of NEST, detailed connectivity information about the multi-area network is available. HybridLFPy requires part of this connectivity data in order to map the incoming connections of selected point-neurons simulated in NEST to their compartmental counterparts simulated in Arbor. Based on that, Arbor can build connections to the neuronal compartments.

After the communication of connectivity data from CM to ESB, NEST enters the initialization phase, which does not necessarily end at the same time as the Arbor build phase. The simulation phases of both NEST and Arbor follow, where Arbor relies on frequent spike input from NEST.

During the simultaneous simulation phases of NEST and Arbor, full network activity of the multi-area model is simulated in NEST and spikes from the previously selected fraction of the network are frequently communicated to Arbor running on the ESB using the MPI-based MUSIC library. The spatially detailed (compartmental) neuron models

simulated in Arbor consume the spikes according to the mapping created by HybridLFPy.

Locally on the ESB HybridLFPy requires frequent information about ionic currents into and out of the neuronal compartments simulated in Arbor in order to predict the LFP signals and their development over time.

Elephant is fed with spikes from selected populations of the multi-area model using the MUSIC library to coordinate MPI communication (see Figure 2.2). Therefore, NEST (on CM) and the Python script that applies the necessary Elephant functions to the incoming spike trains (on DAM) start to run at the same time but the Python script needs to wait with the analysis until NEST reaches the simulation phase and produces spikes.

We expect that in simulations of the multi-area model this initial idle time of Elephant will be irrelevant as neither build nor initialization time, but the actual simulation time, dominates the total runtime of NEST.

The simulation of the multi-area model with NEST is run on the CM using a hybrid parallelisation scheme combining MPI and OpenMP threads. CM is optimal for NEST, because NEST's irregular memory access patterns perform optimally on CPUs with large, low-latency RAM and because NEST does not benefit from vectorisation.

Selected neurons of the multi-area network are simulated in greater detail with Arbor running on the ESB, because Arbor requires considerably more compute power relative to memory, since Arbor simulation does not require full network connectivity information. Arbor benefits significantly from vectorisation using AVX2, AVX512, and GPGPUs; it uses hybrid parallelisation combining MPI and C++11 threads or Intel TBB.

Analysis of spike trains recorded from selected populations of the multi-area model is carried out by Elephant, which runs on the DAM.

## 2.4 Porting experience

Porting the code to the different DEEP-EST modules has been straightforward for all three applications (NEST to the CM, Arbor to the ESB, and Elephant to the DAM). There were, in particular, no issues with porting Arbor to the ESB as GPU support was already in place.

To use the workflows described above, we needed to implement communication back ends in Arbor and NEST. We had suggested earlier to use the MUSIC library for the communication within the NEST-Arbor coupling. More careful analysis of the interaction between NEST and MUSIC as part of this project revealed that use of MUSIC for MPI communication between NEST and Arbor would impose frequent synchronisation of threads in MPI-OpenMP hybrid NEST simulations. To avoid this, we

decided to implement NEST-Arbor coupling directly via MPI instead of using MUSIC as an intermediary. The mapping of neuron identities between NEST and Arbor, which MUSIC would have provided, was ensured through proper simulator scripting.

NEST, Arbor and Elephant could be installed and run out-of-the box using standard compiler and build tools available after we had familiarized ourselves with the software environment on the DEEP-EST system, with an effort off less than 0.5 Person Month (PM). Basic interfacing NEST and Elephant via MUSIC including minor bug fixes took also about 0.5 PM. The NEST-Arbor interface was implemented in collaboration with the Arbor development team; NMBU contributed roughly half of the effort (3 PM).

## 2.5 Scalability

Both NEST and Arbor have already been shown to scale well on modern supercomputers[33],[34] (Figure 2.3 and Figure 2.4). With the 5th generation simulation kernel, the communication scheme for the exchange of spikes between MPI processes was changed from `Allgather()` to `Alltoall()`, allowing each MPI process to send spikes only to the MPI processes that host the targets. To this end, the connection infrastructure of NEST was redesigned. Arbor has been developed considering support for GPUs and explicit vectorization from the very outset.

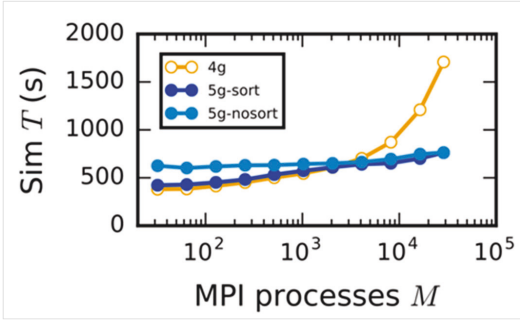


**Figure 2.3: Simulation time for NEST running the HPC benchmark[33] on JUQUEEN; shown for previous kernel (4g) and new kernel with optimizations for small-scale to medium-scale regime (5g-sort) and without the optimizations (5g-nosort). Adapted from Figure 7C in[33])**

[33] Jordan, J. et al. (2018) doi:10.3389/fninf.2018.00002

[34] Akar, N. A. et al (2019) doi: 10.1109/EMPDP.2019.8671560

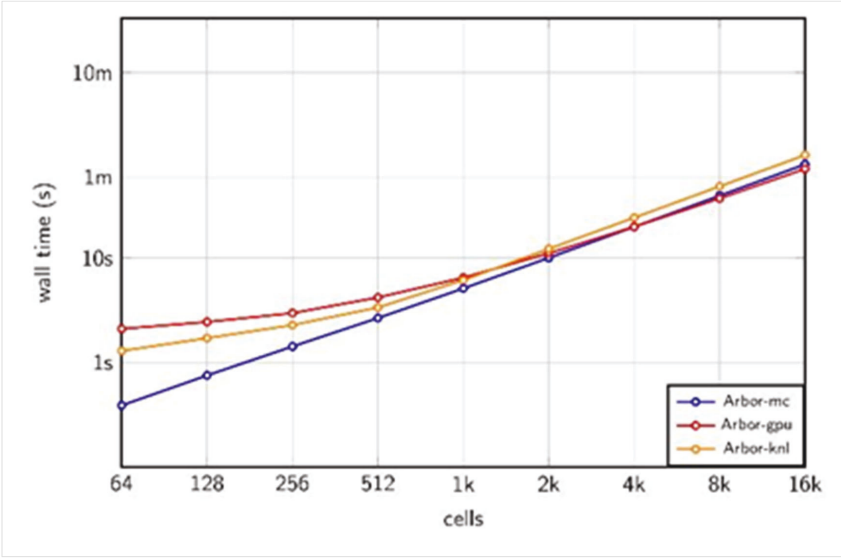


**Figure 2.4: Performance of Arbor (based on [34]): Single node wall time of Arbor running on Piz Daint multicore, GPU and Tave KNL**

The new NEST kernel shows good weak-scaling behaviour on modern supercomputers (5g-nosort, Figure 2.3, adapted from Figure 7C in Jordan et al. 2018[33]). We go from 32 MPI processes to about 32,000 MPI process, while increasing the problem size therefore in weak scaling by a factor of 1000, and keep the runtime nearly constant. For large numbers of MPI processes, the 5g kernel shows much better scaling behaviour and a decrease in runtime by more than 55% for simulations on the full JUQUEEN[35] system compared to the previous kernel (4g). The S-shaped trend of the simulation time observed for the new NEST kernel (5g-sort) can be explained as follows: For a smaller number of MPI processes, an additional reduction in memory usage is achieved by optimizations for the small-scale to medium-scale regime (compare 5g-sort: small-scale optimizations enabled to 5g-nosort: small-scale optimizations disabled). The optimizations exploit the lesser degree of distribution of each neuron's outgoing connections across processes in the regime up to few thousands of MPI processes,[33]). As gradually the optimizations get less effective with increasing numbers of MPI processes, due to an increasing degree of distribution of connections across processes, simulation times also increase. Note that this effect on scalability in the small-scale to medium-scale regime can be observed in all scaling measurements for NEST shown in this deliverable as in all cases the optimizations

[35] M. Stephan, J. Docter, JUQUEEN: IBM Blue Gene/Q Supercomputer System at Jülich Supercomputing Centre, *Journal of large-scale research facilities*, 1, A1 (2015)

were enabled (5g-sort). In the large-scale regime the outgoing connections of each neuron are fully distributed such that the optimizations for the small-scale to medium-scale regime no longer play a role. The simulation time increases slowly in this large-scale regime.

Arbor's single node performance has been analysed using a randomly connected network benchmark employing CSCS' Piz Daint multicore, GPU and KNL clusters. For more than 4000 cells the GPU is utilized enough to run the benchmark more efficiently in terms of the wall time than on multicore or KNL (Figure 2.4; based on Akar et al. 2018[34]), Table 3 and Fig 4).

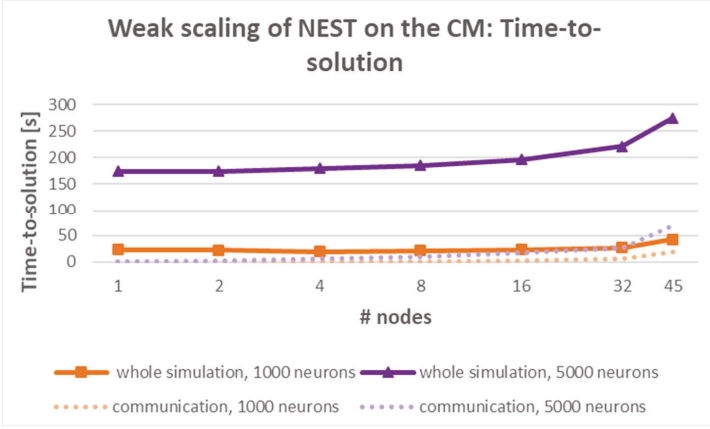


**Figure 2.5: Time-to-solution for NEST running the HPC benchmark on the CM: Simulation time and contribution of MPI communication**

Within this document we show some results obtained on the DEEP-EST system. Figure 2.5 shows a weak scaling of the HPC benchmark using NEST on the Cluster Module (CM) (1 MPI process per node and 24 threads per MPI process). Figure 2.6 shows the corresponding parallel efficiency. The benchmark network model includes plastic synapses, which need to be updated whenever they transmit a neuronal signal thereby causing workload in addition to neuronal updates. The minimum simulation time (among at least 5 repetitions) and the time spent communicating spikes across MPI processes vs. number of compute nodes is shown for a test case with 1000 and 5000 neurons per thread and 11,250 synapses per neuron[36].

[36] Benchmarks simulated with NEST@da46542 (with timers and optimization for small-scale regime)" for 1000 and 5000 neurons per thread

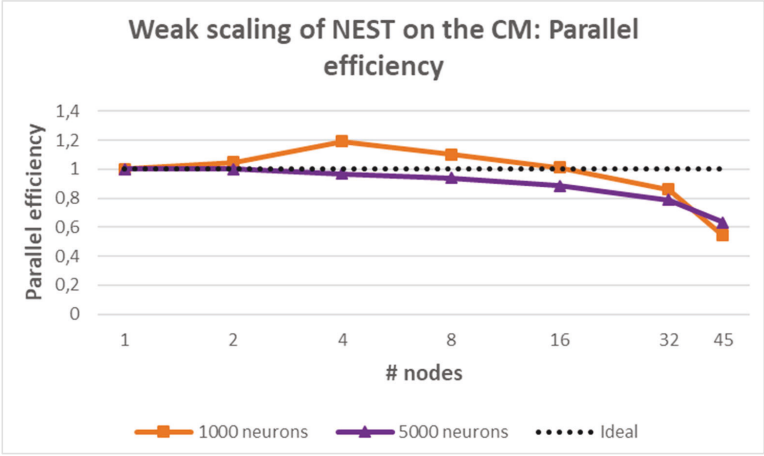


**Figure 2.6: Parallel efficiency for NEST running the HPC benchmark on the CM**

Figure 2.7 shows the mean simulation time for the 5000 neurons case but with subtracted communication time, which allows for a comparison with measurements obtained using the NEST dry-run mode. A dry-run simulation is carried out by one MPI process emulating the input from other MPI processes, which enables predictions for large-scale simulations. For all simulation time plots lower is better. As NEST optimizations for the small-scale and medium-scale regime were enabled, we observe the typical increase in simulation time described above (c.f. 5g-sort, Figure 2.3), for node counts of 64 and above.

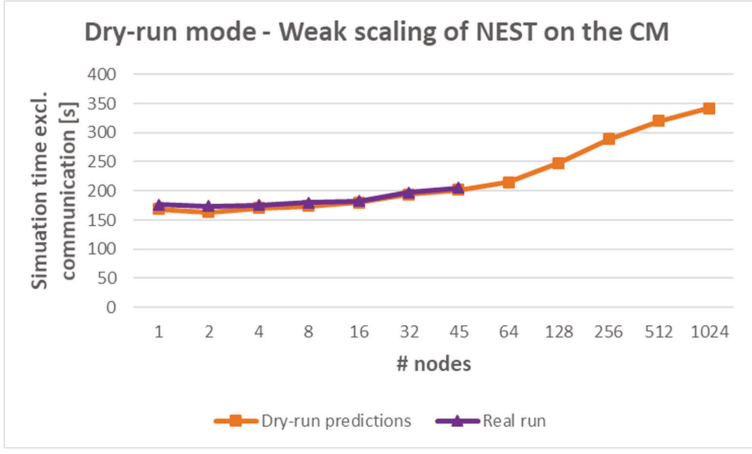


**Figure 2.7: Simulation time for NEST running the HPC benchmark on the CM: Dry-run prediction (excluding communication time)**

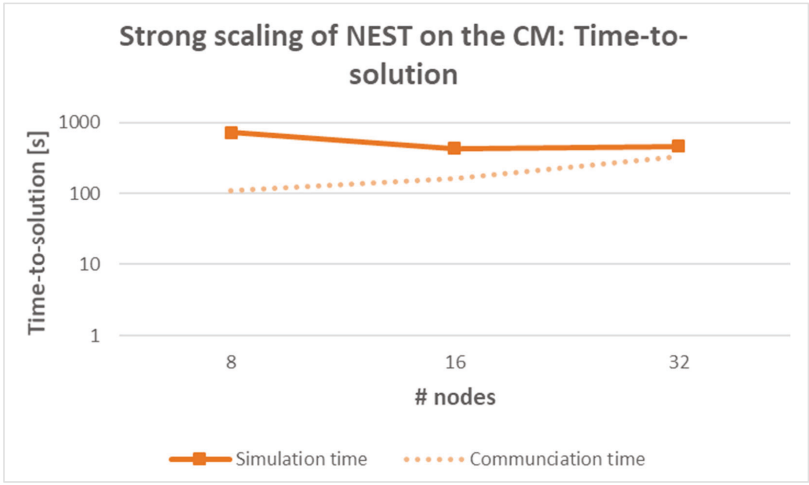


**Figure 2.8: Simulation time for NEST running the MAM benchmark on the CM**

Figure 2.8 shows a strong scaling of the multi-area model (MAM) benchmark using NEST on the CM (1 MPI process per node and 24 threads per MPI process). Figure 2.9 shows the corresponding parallel efficiency. We have developed the MAM benchmark in this project to provide a scalable benchmark network model with easily controllable parameters and stable dynamics that captures the main performance-relevant features of the multi-area model[37] such as short synaptic transmission delays requiring frequent communication. The benchmark network model consists of 4 million neurons and 5,625 synapses per neuron, where all synapses are static (no additional workload due to synaptic plasticity). The simulations scale well between 8 and 16 MPI processes, but communication time dominates the simulation time at 32 MPI processes. This is due to more frequent communication and less workload compared to the NEST HPC benchmark. The effect of the NEST optimizations for the small-scale and medium-scale regime also plays a role but cannot be distinguished from the other factors.

[37] Schmidt, M. et al (2018) doi. org/10.1371/journal.pcbi.1006359

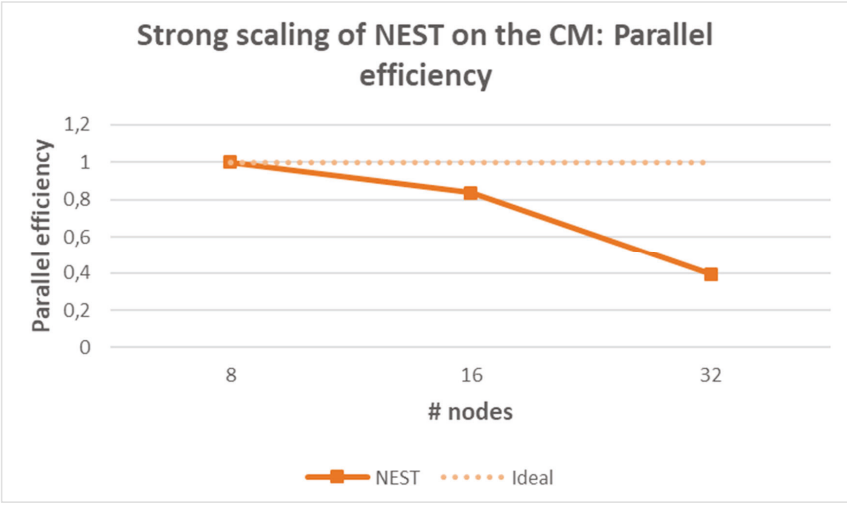


**Figure 2.9: Parallel efficiency for NEST running the MAM benchmark on the CM**

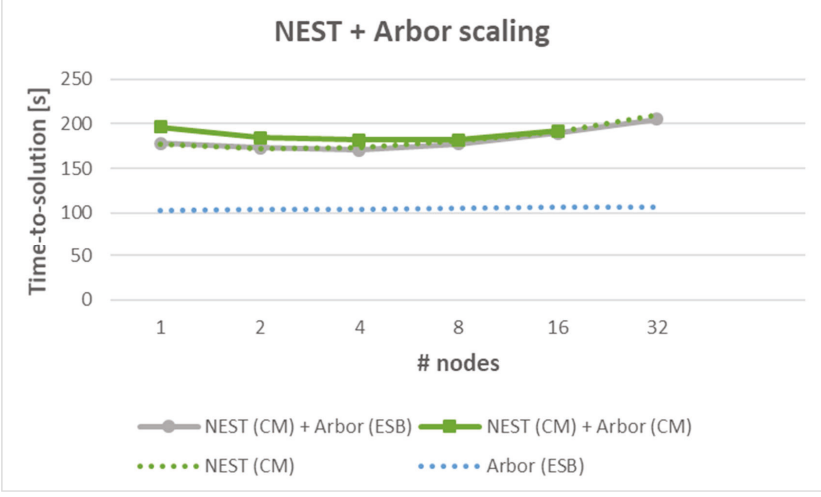


**Figure 2.10: Weak scaling time-to-solution for the combined NEST and Arbor simulations**

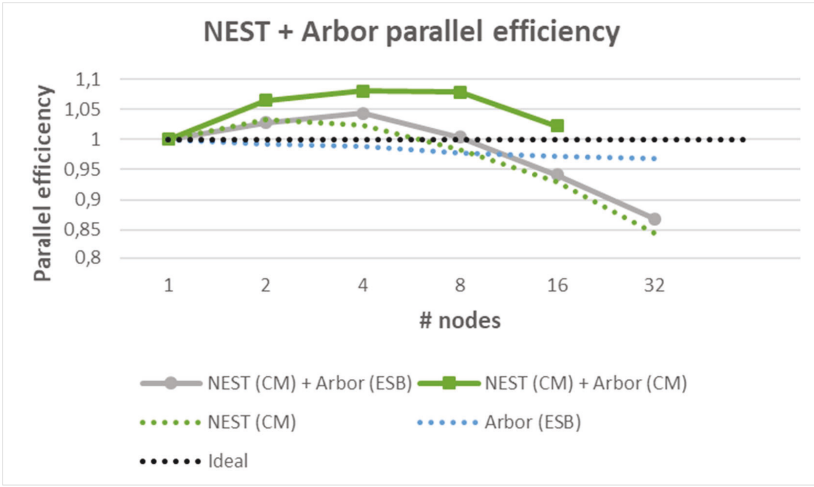


**Figure 2.11: Weak scaling parallel efficiency for the Nest - Arbor coupling**

Figure 2.10 and Figure 2.11 show the weak scaling behaviour of combined NEST-Arbor simulations[38]. In the smallest case (1 node), NEST simulates 120,000 point neurons while Arbor simulates 1% of this number, i.e., 1,200 compartmental neuron, each on a single compute node. The neuron numbers are scaled linearly with the number of compute nodes. We consider two different configurations: NEST running on the CM and Arbor on the ESB (grey) and NEST and Arbor both running on the CM (green). On the ESB, Arbor uses the GPU on each node, while on the CM Arbor runs 24 threads per node using AVX512. For comparison, we also show the simulation times for the NEST part only (dotted green) and the Arbor part only (dotted blue). Note that for ESB-only and CM-only cases experiments were limited to 16 nodes for each of the programs due to the limited number of nodes.

The underlying Arbor simulations scale perfectly on the ESB when run alone (dotted blue), while the simulation time for pure NEST simulations on the CM (dotted green) scales reasonably well. The combined NEST-Arbor simulation run on CM and ESB (grey) requires essentially the same time as the NEST simulation alone, indicating that the MSA allows us to extend the NEST simulation to a co-simulation without runtime penalty. Executing NEST and Arbor on the CM only leads to increased runtimes (green), indicating the benefit of combining CM and ESB. We also find that co-simulation on CM and ESB reduces energy consumption, see Figure 2.15.

[38] NEST@abc4e0b78

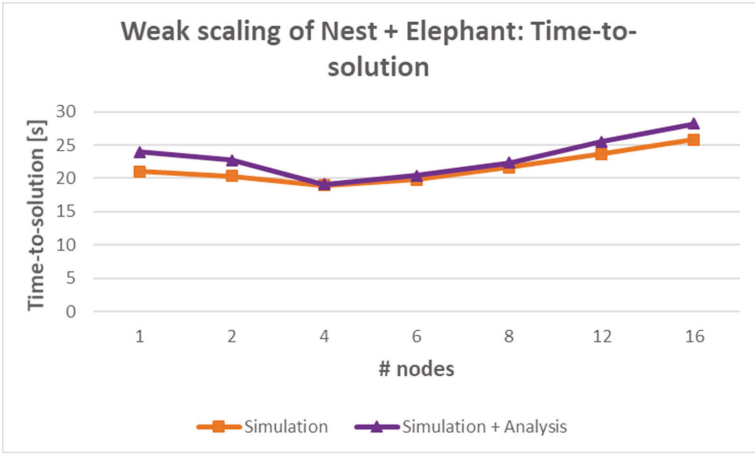


**Figure 2.12: Weak scaling of NEST simulation on CM feeding Elephant weak/ensemble-scaling: HPC Benchmark using NEST on CM and analyses with Elephant running on DAM**

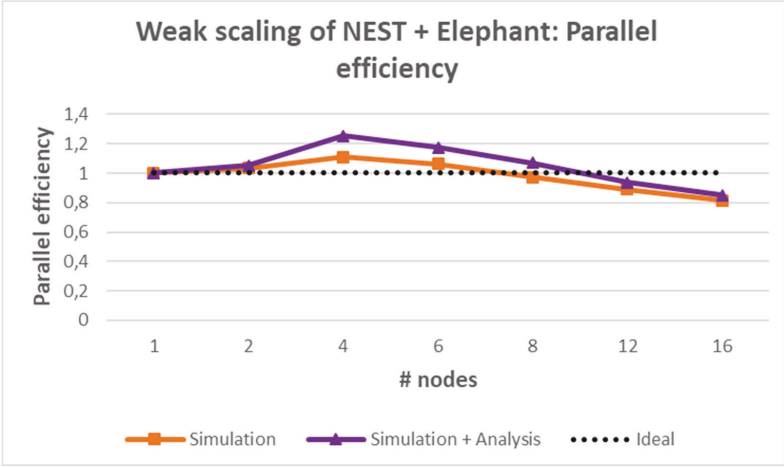


**Figure 2.13: Parallel efficiency for the weak scaling NEST + Elephant run**

Figure 2.12 shows an example of a NEST simulation running on the CM and sending data for analysis in Elephant on the DAM via MUSIC[39]. Figure 2.13 shows the parallel efficiency. Simulation time and parallel efficiency are shown as function of number of CM nodes used and network size scales linearly with the number of nodes, with 24 MPI processes running on each node (to accommodate MUSICs proper support for

[39] NEST@7616f3eb with bugfix; Elephant v 0.1.0 under Python 3.6.8; MUSIC@8c6b77a57 with path for ParaStationMPI.

threading; approximately 940 neurons per process). The analysis is performed on a single DAM node running two Python processes: one performing ASSET analysis exploiting the GPU and the other performing cross-correlation analysis. Comparison of simulation without spike transfer to the DAM (orange) and simulation with analysis on the DAM (purple) shows that the overhead for analysis is small (approximately 10%) and that, while not perfect, simulation time is roughly in agreement with a weak scaling regime.

### *2.5.1 Our path to Exascale*

Above we discussed to what extent the applications can scale at the moment. The following subsections will outline our path to Exascale

#### *2.5.1.1 What are the limitations – Can they be fixed?*

The most visible performance limitation in our work is the relatively poor weak-scaling performance of NEST on the CM for large numbers of neurons as shown in Figure 2.5, which also affects the run time of co-simulations running NEST on the CM and Arbor on the ESB as shown in Figure 2.10. In part, this weak scaling is a consequence of the optimisations for small to medium scale simulations of the NEST 5g kernel, which exploit the lesser degree of distribution of each neuron's outgoing connections across MPI processes in this regime. As the number of processes increases the exploitation potential decreases rendering the optimisations less and less effective. The optimisations reduce the total simulation times in this regime but due to the gradual decrease in effectiveness distort the observed scaling behaviour on smaller systems such as the existing CM; scaling behaviour of large-scale simulations is not affected by this. Further optimisation will focus on simulation on Exascale systems with an aim at reducing overall communication requirements by introducing support for local connectivity: in real neuronal circuits, a large fraction of the connections are local, but this locality is not yet exploited in NEST or Arbor to minimize communication.

#### *2.5.1.2 How to use future Exascale systems*

Exascale computers will be required to allow full-scale simulations of models of primate brains at the resolution of individual neurons. Only Exascale systems will provide the memory necessary to represent the connectivity in networks at the scale of entire brains, the computing power needed to advance the dynamics of neurons, and the interconnects to facilitate signal exchange between neurons. Using a network with highly simplified structure, we demonstrated the feasibility of simulating networks on the size of a cat brain on a major Petascale computer (K, JUQUEEN[40]). Since then,

[40] Kunkel, S. et al. (2014) doi: 10.3389/fninf.2014.00078

we have made important steps in resource efficient dry-run benchmarking[41,42] and directed communication[33]. Dedicated efforts as part of the DEEP-EST project have reduced spike-delivery times[43], addressing a key performance bottleneck. Parallel activities in the EC ICT Flagship *Human Brain Project*[44] focused on reducing the times required to construct networks with realistic complexity in parallel and to further optimise communication schemes for Exascale systems. This work will be pursued in collaboration with Japanese colleagues, which will allow actual experiments on the largest available pre-Exascale system, Fugaku, later in 2021.

#### *2.5.1.3 Where did the DEEP-EST project help on the way to Exascale?*

Comprehensive performance profiling allowed us to identify crucial performance bottlenecks in spiking network simulations. Network models with realistic degree (in/out-degree of $O(10^4)$ per neuron) and complexity characteristic of brain networks are represented in the simulator as large adjacency lists which are traversed in random order due to the stochastic activity in network models. This leads to unpredictable memory access patterns and thus inefficient caching. As part of our activities in the DEEP-EST project, we were able to develop new spike-delivery techniques improving caching performance and thus overall simulation performance[43]. The success of the new spike-delivery algorithm was rather unexpected as the memory bottleneck imposed by local spike routing has long been considered insuperable in neuronal network simulation technology. The techniques are not specific to the NEST simulator for which we have developed them, but are applicable to other simulators for pulse-coupled networks with high connection degrees as well. We consider this a generally useful contribution to large-scale network simulation.

Beyond this surprising success and the resulting benefit for the NEST users, our work contributes indirectly to the development of neuromorphic systems. The technology for simulations of spiking neuronal networks on conventional computer architectures informs and inspires the design of neuromorphic systems, and it constitutes an important reference benchmark for such systems regarding accuracy, energy consumption and speed. Mitigating the von Neumann bottleneck in spiking network simulations on conventional architectures by latency-hiding techniques challenges neuromorphic systems. The effective use of such techniques indicates that the memory bottleneck can likely be overcome by many-core systems as naturally each core needs to oversee a decreasing amount of memory.

---

[41] Kunkel, S., and Scheck, W. (2017) doi: 10.3389/fninf.2017.00040

[42] Kunkel, S., and Scheck, W. In preparation.

[43] Pronold, J. et al. In preparation.

[44] https://www.humanbrainproject.eu, Specific Grant Agreements 2 (2018–2020) and 3 (2020–2023).

## 2.6 Energy consumption

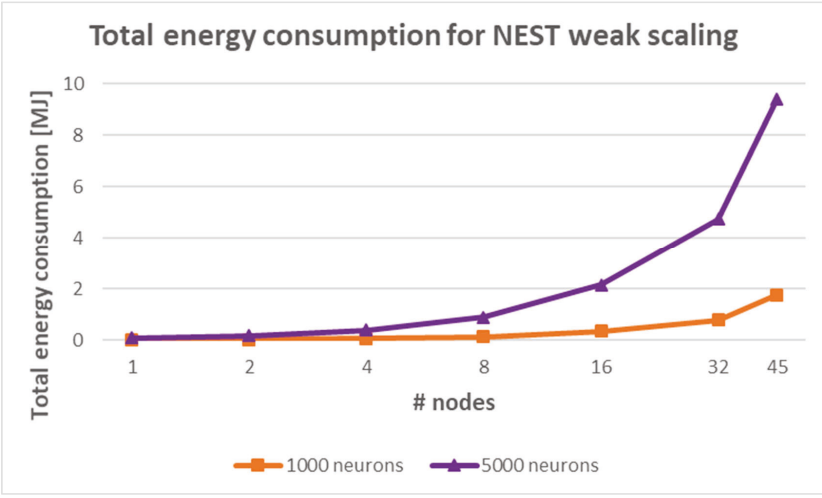


**Figure 2.14: Total energy consumption for NEST running the HPC benchmark on CM for two different network sizes**

Figure 2.14 shows the total energy consumption from the benchmark runs shown in Figure 2.5 and Figure 2.15 shows the total energy consumption for the NEST-Arbor co-simulations shown in Figure 2.10. While timings shown in Figure 2.5 and Figure 2.10 show the actual simulation time (propagation of network state), the total energy consumption includes the time required for network construction and initialization before the simulation.

For the pure NEST simulation on the CM we observe linear scaling as expected up to 32 nodes followed by a noticeably superlinear increase when simulating on 45 nodes. For the co-simulation, scaling is nearly perfectly linear when combining up to 32 nodes each on CM and ESB. Co-simulations on the CM alone were only performed up to 16+16 nodes, the limit set by the DEEP-EST system size at present, with higher energy consumption when using only the CM, especially for 16+16 nodes case. This indicates energy efficiency gains from the modular system architecture.

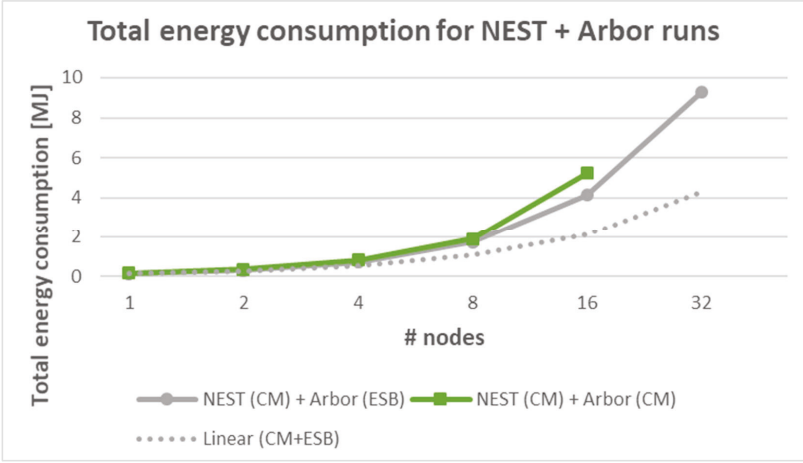


**Figure 2.15: Total energy consumption for NEST and Arbor co-simulation running on CM and ESB (grey) and on the CM alone (green). The number of nodes is per simulator, i.e., 16 nodes means NEST running on 16 nodes (always CM) and Arbor running on 16 different nodes (either ESB or CM)**

## 2.7 Performance comparison

After over three years of development, this subsection compares our current application status with their status at the start of the DEEP-EST project.

### *2.7.1 New spike-delivery algorithm*

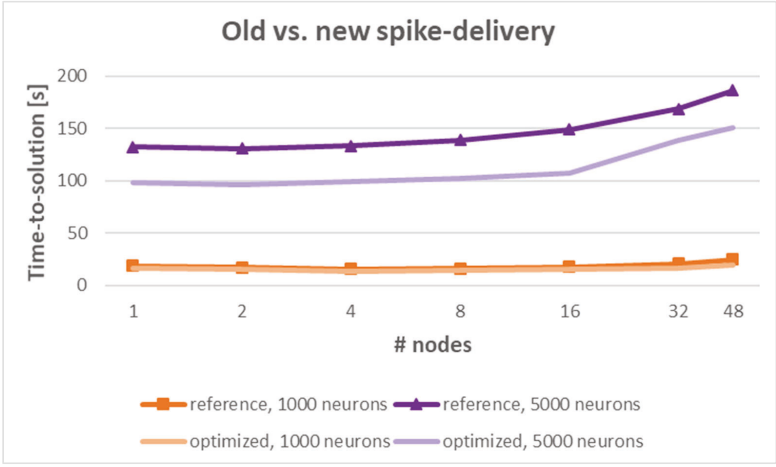


**Figure 2.16: Simulation time reduction for new spike-delivery algorithms**

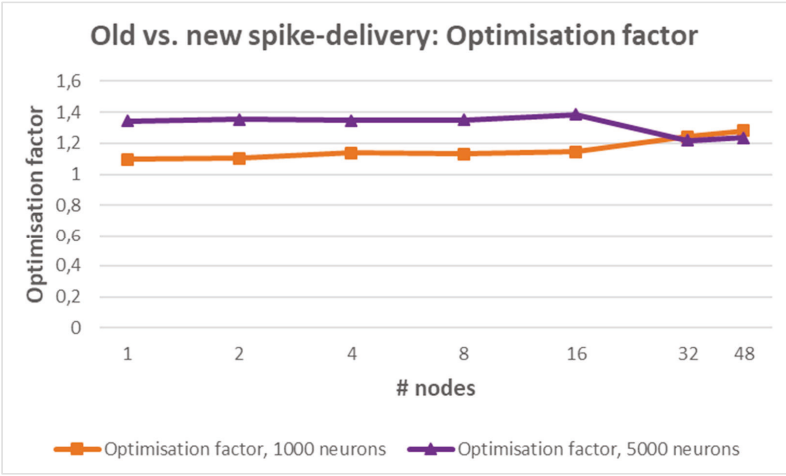


**Figure 2.17: Optimisation factor gained by the new spike delivery**

Sparse connectivity combined with irregular spiking activity leads to a practically random memory-access pattern during spike delivery. Seemingly this is a worst-case situation for the von Neumann architecture, where for any computation the content of a respective memory unit needs to be transported to the central processing unit and the result needs to be transported back. In weak-scaling spike delivery dominates the simulation time[33]. To overcome the memory bottleneck, we have rearranged the elementary algorithmic steps required to deliver the incoming, essentially random spike data to the process-local targets, such that they can be more efficiently processed by conventional computer hardware. The redesign also includes common latency-hiding techniques such as software prefetching and software pipelining. Figure 2.16 shows a significant reduction in simulation time as a result of new algorithms for spike-delivery in NEST, when comparing the HPC benchmark (2 MPI process per node with 12 threads per process and 1000 or 5000 neurons per thread) and the same case with the new spike-delivery enabled[45]. Figure 2.17 shows the optimisation factor gained by employing the new spike delivery. Note that in this version of the HPC benchmark all synapses were static (no additional workload due to synaptic plasticity), which allowed us to better expose the memory bottleneck.

[45] Optimized spike delivery: HPC-Benchmark simulated with NEST@8f1b08c (with timers and optimization for small-scale regime); reference data simulated with NEST@8897668.

## 2.8 Conclusion

Running brain-model simulations on Exascale computers to explore brain dynamics at the scale of full brains is a major challenge in computational neuroscience and in simulation technology. The focus put in the DEEP-EST project on improving application scaling and performance has allowed us to test new techniques and to understand factors affecting performance of simulators, especially NEST, even better. This has driven the development of more efficient spike-delivery techniques and the development of an advanced dry-run mode. The latter allows benchmarking of large parallel simulations on a small subset of the relevant system and presents an approach also viable for other, comparable tools.

In situ processing of spike data generated by large-scale brain simulations will be essential as networks are scaled up, since storing a raw spike during simulation and re-loading it for analysis becomes infeasible. Coupling NEST-Arbor and NEST-Elephant enables combining on the one hand simulations at different levels of description and on the other hand simulations and analysis. Our work in the DEEP-EST project in this area has shown that a hybrid approach of distributing different parts of a workflow across different modules of a MSA clearly holds potential.

# 3 Molecular Dynamics with GROMACS

**Peicho Petkov, Stoyan Markov, Valentin Pavlov**
National Centre of Supercomputing Application, NCSA, Bulgaria
peicho@phys.uni-sofia.bg

## 3.1 Introduction

GROMACS[46,47,48,49,50,51,52] is one of the fastest molecular dynamics simulators in the world. It is used mainly for soft matter molecular dynamics (MD) simulations with implementation in life sciences. GROMACS tracks the trajectories of a system of particles (atoms) that evolves in time by solving differential equations of motion at each time step. The coordinates and velocities of the particles are calculated by using their values from the previous time frame. In each time step, it calculates the forces acting on each atom, which is indeed the most time-consuming operation. From the computational point of view, the force acting on a particle is a result of a summation over:

---

[46] H. J. C. Berendsen, D. van der Spoel and R. van Drunen, "GROMACS: A message-passing parallel molecular dynamics implementation," *In Computer Physics Communications, ISSN 0010-4655, DOI: 10.1016/0010-4655(95)00042-E,* vol. 91, no. 1-3, pp. 43-56, 1995.

[47] E. Lindahl, B. Hess, D. van der Spoel and J. Mol, "GROMACS 3.0: a package for molecular simulation and trajectory analysis," *Molecular modeling annual, DOI: 10.1007/s008940100045,* vol. 7, no. 8, pp. 306-3017, 2001.

[48] D. van der Spoel, E. Lindahl, B. Hess, G. Groenhof, A. E. Mark and H. J. C. Berendsen, "GROMACS: Fast, flexible, and free," *Journal of Computational Chemistry, DOI: 10.1002/jcc.20291,* vol. 26, no. 16, p. 1701–1718, 2005.

[49] B. Hess, C. Kutzner, D. van der Spoel and E. Lindahl , "GROMACS 4: Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation," *Journal of Chemical Theory and Computation, DOI: 10.1021/ct700301q,* vol. 4, no. 3, p. 435–447, 2008.

[50] S. Pronk, S. Páll , R. Schulz, P. Larsson, P. Bjelkmar, R. Apostolov, M. R. Shirts, J. C. Smith, P. M. Kasson, D. van der Spoel, B. Hess and E. Lindahl, "GROMACS 4.5: a high-throughput and highly parallel open source molecular simulation toolkit," *Bioinformatics, https://doi.org/10.1093/bioinformatics/btt055,* vol. 29, no. 7, pp. 845-854, 2013.

[51] S. Páll , M. J. Abraham, C. Kutzner, B. Hess and E. Lindahl, "Tackling Exascale Software Challenges in Molecular Dynamics Simulations with GROMACS," *Markidis S., Laure E. (eds) Solving Software Challenges for Exascale. EASC 2014. Lecture Notes in Computer Science,* pp. pp 3-27, 2015

[52] M. J. Abraham, T. Murtola, R. Schulz , S. Páll , J. C. Smith, B. Hess and E. Lindahl, "GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers," *SoftwareX, DOI: 10.1016/j.softx.2015.06.001,* Vols. 1-2, pp. 19-25, 2015.

- pairs of particles connected by “bonds” – bonded interactions,
- pairs of particles satisfying some distance criteria – short-range non-bonded interactions,
- long-range corrections including calculations where data of all the simulated particles are needed – long-range interactions.

Usually, pairs of atoms are defined in a predefined cut-off radius calculating short-range interactions[53], while the long-range interactions are calculated using Fast Fourier Transform (FFT) based algorithms.

## 3.2 Application structure

GROMACS can run on almost every modern computing architecture[54]. The simulation program uses multi-level parallelism to utilize the computational power offered (see Figure 3.1). By means of domain decomposition, the calculations are spread along the distributed memory computational resources – compute nodes. On the domain decomposition level, MPI is used. At MPI rank level, a shared memory model is implemented with both OpenMP and GPU accelerators. Finally, SIMD registers are used to vectorise the calculations in each CPU core.

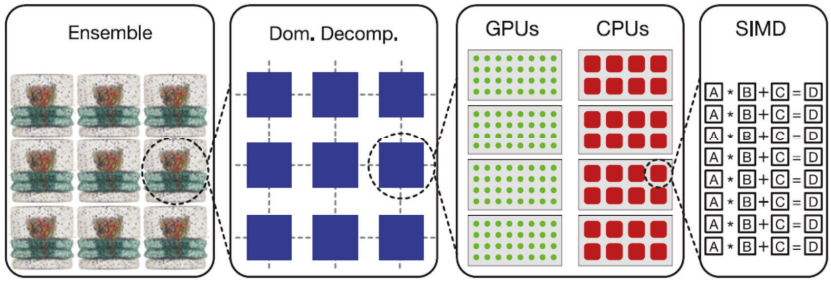


**Figure 3.1: Multi-level parallelism in GROMACS.[54]**

The Particle-mesh Ewald (PME) algorithm uses FFT to solve long-range electrostatic contributions to real-space direct Coulomb sums. The reader should consider the fact that the specific implementation involving All-to-All MPI communications causes the performance scalability to drop. The latter can be solved by overlapping real-space calculations and Fourier-space calculations. In GROMACS the MPI ranks are divided into two groups: one for real-space calculations (PP nodes) and the rest being dedicated to PME calculations (PME nodes).

[53] Computer Physics Communications 184 (2013) 2641–2650

[54] Abraham, et al. (2015) SoftwareX 1-2 19-25.

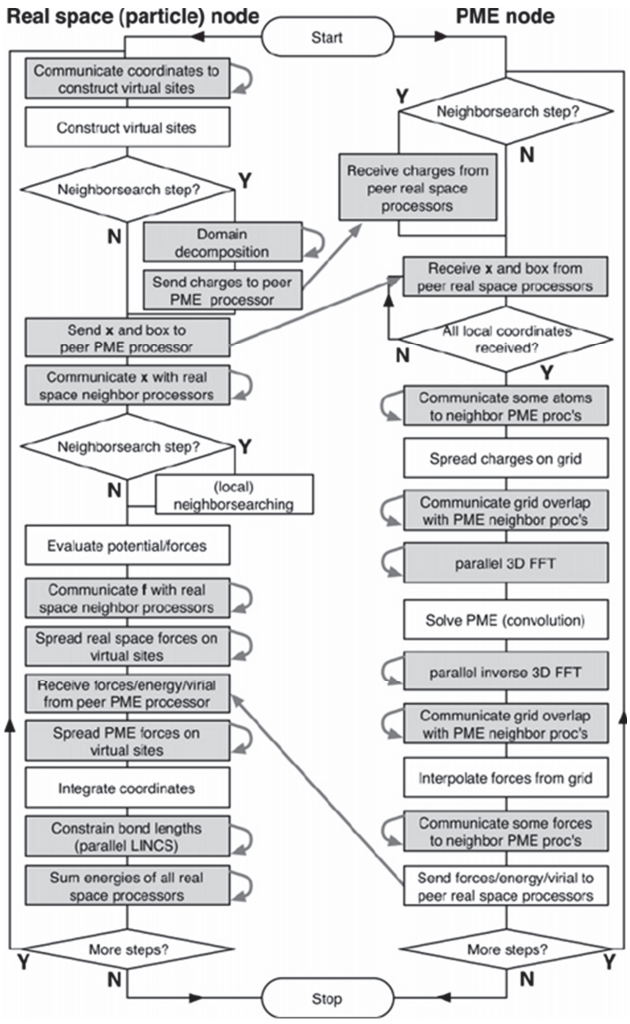


**Figure 3.2: GROMACS flowchart for a typical simulation step for both particle and PME nodes.[55]**

As shown in Figure 3.2, the resulting flowchart in an MD step can be described in the following manner. Each PP node has a corresponding PME node. At the beginning of the time step, each PP node sends coordinates and charges to its corresponding PME node and once the PME calculations are completed, each PME node sends the

[55] B. Hess, C. Kutzner, D. van der Spoel, E. Lindahl. GROMACS 4: Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation, Journal of Chemical Theory and Computation 4, 435-447 (2008)

resulting forces back to the corresponding PP node. Meanwhile, all collective communications proceed only between PME nodes as well as only between PP nodes and overlapping the FFT All-to-All communications (exchanged between PME nodes only) with real-space calculations. Consequently, one must optimise the number of PP and PME nodes in such a way that PME nodes need to send the forces that they have calculated in the exact moment when the PP nodes need Fourier-space forces, energies, etc. The GROMACS tool called *tune_pme* enables users to scan different combinations and start the simulation with an optimal PP/PME nodes ratio whilst the *mdrun* simulator further tunes the PME mesh and cut-off radius at the beginning of the MD simulation run. It should be noted that one can choose to execute bonded, non-bonded and long-range interactions on CPU or GPU devices.

## 3.3 Application mapping

How GROMACS is mapped to the Modular Supercomputing Architecture (MSA) depends on the simulation problem size and aims at optimizing the computational load. There are three computationally expensive components of the force to be calculated, namely bonded interactions, short-range non-bonded interactions, and long-range interactions (the same applies to the neighbour list construction, but this is not done every time step), and each of them can be run either on a CPU or on a GPU accelerator.

### *3.3.1 MD simulations of less than $10^4$ particles (CM)*

MD simulations of few tens of thousands of particles, e.g. many simulations of small peptide monomers in aqueous solution, should run efficiently on one node in the CM. In this case calculation time is comparable with communication time (computing node-to-computing node or host-to-device communications of small data buffers) and the performance scalability is limited. The overall flowchart of one MD integration step is show in Figure 3.3.

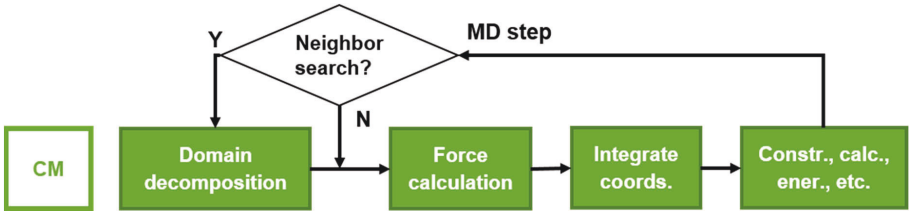


**Figure 3.3: Schematic workflow of MD simulations with less than $10^4$ particles in the MSA**

### 3.3.2 MD simulations with number of particles of the order of $10^5$ (ESB/DAM)

The ESB and DAM offer better performance for MD simulations consisting of hundreds of thousands of atoms (Figure 3.4), where particle-particle interactions are calculated on the GPU accelerators, while long-range electrostatic interactions run on the CPUs.

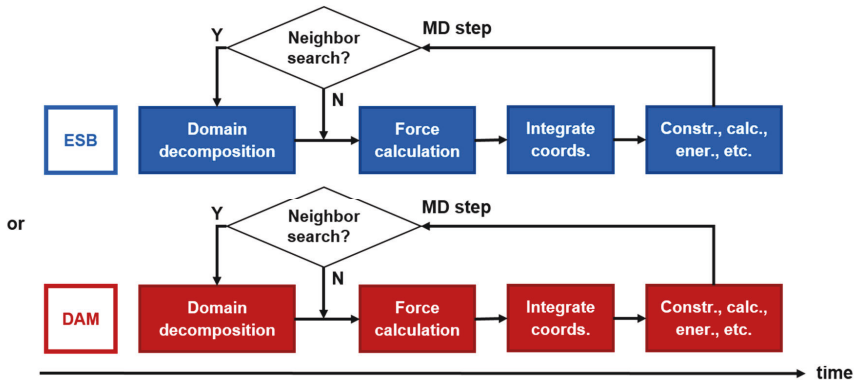


**Figure 3.4: Schematic workflow of MD simulations with $10^5$ particles in the MSA**

### 3.3.3 MD simulations of millions of particles (ESB-CM)

MD simulations of large macromolecules and their complexes at reasonable time scales[56] demand computational resources with good enough performance scalability. When simulating the time evolution of systems consisting of more than several millions of particles, one should use thousands of cores/MPI ranks in the CM or tens/hundreds of nodes with GPU accelerators (like those in DAM). In the first case, the performance scalability saturates due to the enormous number of MPI process. In the second case, pair interactions go on the GPU accelerators while PME ranks run either on the CPUs or on a single GPU (GROMACS does not support PME calculations over multiple GPUs due to need for multiple GPU-to-CPU and vice versa data transfers for 3D FFT implementation). Single GPU performance is insufficient for PME calculations to deliver long-range forces to the PP nodes at the required speed, and would introduce imbalance causing performance scalability degradation. Moreover, for larger atomic systems PME calculations should be conducted on as few nodes as possible to minimise the time spent on All-to-All MPI communications. Therefore, the PME part of the simulation should run on nodes with powerful CPUs and good inter-node network, namely the CM. GPUs are very suitable for doing PP calculations, as mentioned above,

[56] Curr Opin Struct Biol. 2015 Apr; 31: 64–74

so the MSA offers good resource utilization and management when running very large MD simulations with GROMACS in ESB-CM configuration (see Figure 3.5). Both modules will communicate through the network.

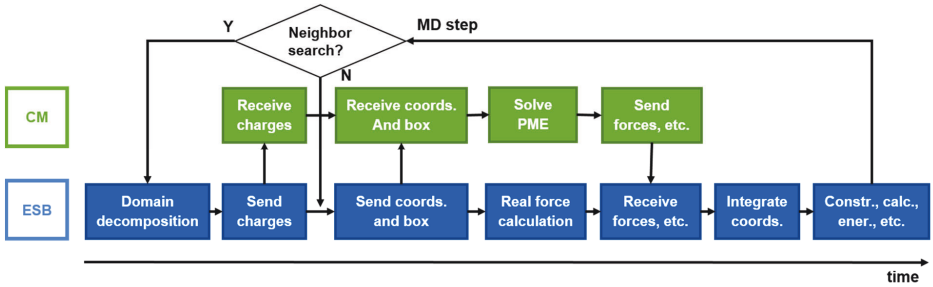


**Figure 3.5: Schematic workflow of MD simulations with millions of particles in the MSA**

### *3.3.4 MD simulations with big volumes (several million nm³)*

Two different offload modes were investigated:

Run PME on ESB and PP on CM: The option to run PME on ESB and PP on CM is not supported natively in GROMACS; it only includes a single-node GPU implementation of PME. This option was implemented and tested but the alternative FMM option (see below) delivered better performance and efficiency.

Replace PME with FMM running on ESB or CM: The primary limiting factor in the PME method is that it utilizes a uniform mesh on the problem domain and the spacing of the mesh is a function of the cut-off radius at fixed accuracy. Solving for big volumes – in the order of several million $nm^3$, where the mesh size becomes larger than e.g. 1,000x1,000x1,000 – quickly becomes inefficient or even impossible. The Fast Multipole Method (FMM)[57] is gaining significant attention in the MD community lately, namely because of its O(N) complexity compared with the O(N*logN) complexity of PME-style methods. Due to its large multiplicative constant, it usually fails to achieve the execution times of PME-style methods on CPUs. However, the GPUs utilized in ESB promise to reduce by an order of magnitude the calculation times, and thus make the FMM method competitive. Additionally, the boundary element method (BEM)

[57] Rokhlin, Vladimir (1985). "Rapid Solution of Integral Equations of Classic Potential Theory." J. Computational Physics Vol. 60, pp. 187–207.

formulation of the continuum electrostatic model[58], local alternative charge distributions treatment with minimal overhead, and λ-dynamics module[59] have been applied. The result is a GPU-accelerated fast multipole method for GROMACS[60] [61]. In DEEP-EST we also include a multiple-GPU FMM implementation that runs on the ESB, and an FMM implementation for multiple-CPUs that can run on the CM. The respective application partitioning is depicted in Figure 3.6.

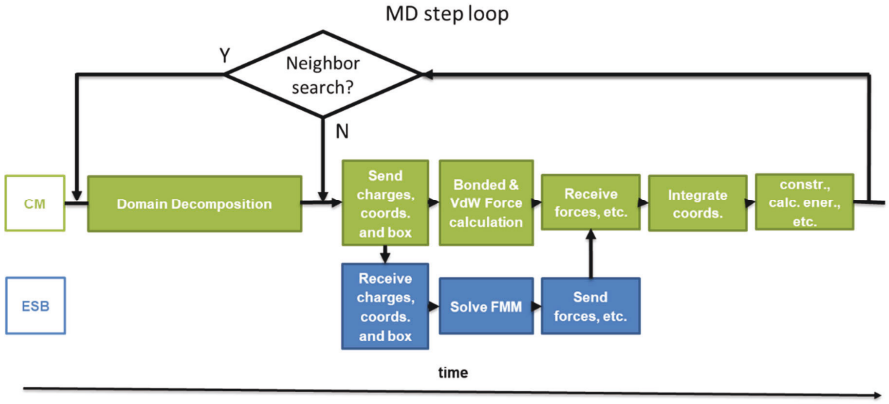


**Figure 3.6: Schematic workflow of MD simulations with very big volumes in the MSA**

## 3.4 Porting experience

The MSA can be utilized in ways not yet supported by the native GROMACS implementation, such as using the ESB to run PME calculations while CM runs PP calculation. Additionally, the computing power offered by the ESB can be harnessed for an efficient implementation of the FMM, which for certain MD simulation volumes can prove beneficial over PME. The SPPEXA project[62] already released a GROMACS

[58] Rio Yokota, Tsuyoshi Hamada, Jaydeep P. Bardhan, Matthew G. Knepley, Lorena A. Barba: Biomolecular Electrostatics Simulation by an FMM-based BEM on 512 GPUs. CoRR abs/1007.4591 (2010)

[59] Kohnke B. et al. (2020) GROMEX: A Scalable and Versatile Fast Multipole Method for Biomolecular Simulation. In: Bungartz HJ., Reiz S., Uekermann B., Neumann P., Nagel W. (eds) Software for Exascale Computing - SPPEXA 2016-2019. Lecture Notes in Computational Science and Engineering, vol 136. Springer, Cham. https://doi.org/10.1007/978-3-030-47956-5_17

[60] http://www.sppexa.de

[61] Kohnke, B., Kutzner, C., & Grubmüller, H. (2020). A GPU-accelerated fast multipole method for GROMACS: Performance and accuracy. *Journal of Chemical Theory and Computation, 16*(11), 6938-6949. doi:10.1021/acs.jctc.0c00744.

[62] http://www.sppexa.de/

version with a single-GPU implementation of FMM. In DEEP-EST we investigated the possibility of utilizing multi-GPU FMM for larger simulation volumes.

In order to study the scalability and performance of different approaches for long-range electrostatics treatment in the context of MSA, and to enable flexibility according to users' needs, we linked GROMACS with the standalone IRIS electrostatics library[63]. This non-invasive approach allows us to experiment with, and provide the user with solutions targeting MSA, while at the same time keeping the original optimized and certified GROMACS codes mostly intact. The newly developed features are available for GROMACS users by linking GROMACS to IRIS with minimal interventions.

Development of the IRIS library started in the PRACE-5IP[64] projects, with the main goal to provide MD code developers with an offloading of long-range electrostatic calculation to a dedicated group of MPI ranks in the manner that it is done in GROMACS. Such separation of short- and long-range interactions allows for better scalability of the MD application. Initially IRIS included a CPU-only version of the P3M[65] algorithm with 3D domain decomposition. It is similar to the SPME implementation in GROMACS, up to a slightly different Green function and interpolation scheme. In DEEP-EST we implemented the following changes to IRIS:

- 1D and 2D domain decomposition of the mesh, which greatly increases the overall performance and scalability compared to the already existing 3D version;
- Port to CUDA to support execution of the long-range contribution on multiple ESB nodes;
- Parallel CPU and parallel GPU versions of the FMM method, which allows running either on the CM or on the ESB.

The main challenges faced during the implementation of the aforementioned changes are related to:

- The unavoidable collective communication pattern inherent in the nature of the long-range electrostatic interaction;
- Memory transfers between the host and the device for the GPU versions of P3M and FMM.

Both these issues lead to poor scalability, since the time spent waiting for their completion cannot be reduced. In order to mitigate their impact, we used the following techniques:

[63] https://github.com/vpavlov/iris

[64] https://cordis.europa.eu/project/id/730913

[65] Roger W. Hockney; James W. Eastwood (1988). "Particle-Particle-Particle-Mesh ($P^3M$) Algorithms". *Computer simulation using particles*. CRC Press. pp. *267–304*. *ISBN* *9780852743928*.

- Overlap the collective communication with computation where possible by using CUDA asynchronous kernel execution and memory transfers;
- Utilising the CUDA-aware MPI implementation on the ESB to optimize the data transfer between GPU memory on different ESB nodes.

The implementation of the required changes consists of roughly 15,000 lines of code and together with testing and bug fixing it took about 7 PM.

## 3.5 Scalability

We measured the time to solution (T1), the time spent in MPI calls (T2), the time in GPU kernels (TG), the number of MPI messages, the volume of the exchanged data and the total duration of the MPI call of one and the same type. Based on these measurements, we estimated the load balance as:

$$\textbf{Load balance} = \frac{(1/N)\sum_{p=1}^{N}(T1_p - T2_p)}{MAX(T1_p - T2_p)} \times 100\%,$$

where N is the number of MPI ranks, $T1_p$ is the time to solution, and $T2_p$ is the time spent in MPI calls by MPI rank *p*. MAX() takes the maximum value among the all MPI ranks.

Dedicated test runs were performed to measure the performance scalability and parameters listed in the tables below for MD simulations of different size conducted on different number of codes in a single module – ESB, CM and multiple modules – ESB+CM. GROMACS performance was estimated based on 10,000 MD steps long runs, while the communications' profiling was done for 1,000 MD steps.

### *3.5.1 ESB Scalability results*

Strong scaling

The timing data and calculated values of the parameters defined in the beginning of the section are shown in Table 3.1, Table 3.2, Table 3.3 and Table 3.4 for MD simulations with 1.25M, 20M, 40M and 80M atoms, respectively. The data for single nodes are not included due to the different computation model involved: on a single node all calculations are done in the GPU. When multiple nodes are used for a parallel simulation, particle-to-particle calculations run on the GPU, while the CPUs do PME calculations, using 8 MPI ranks per node and 1 OpenMP thread per MPI rank. The performance data are plotted in Figure 3.7 with the parallel efficiency shown in Figure 3.8. For MD simulations performance is measured as the amount of simulated time (nanoseconds) that can be calculated in one day (higher is better).

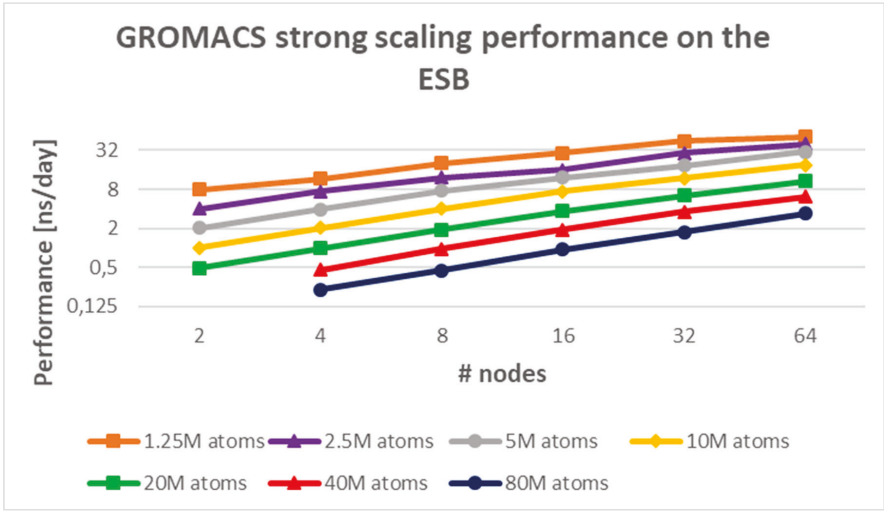


**Figure 3.7: GROMACS strong scaling performance on ESB for MD simulations of different size**

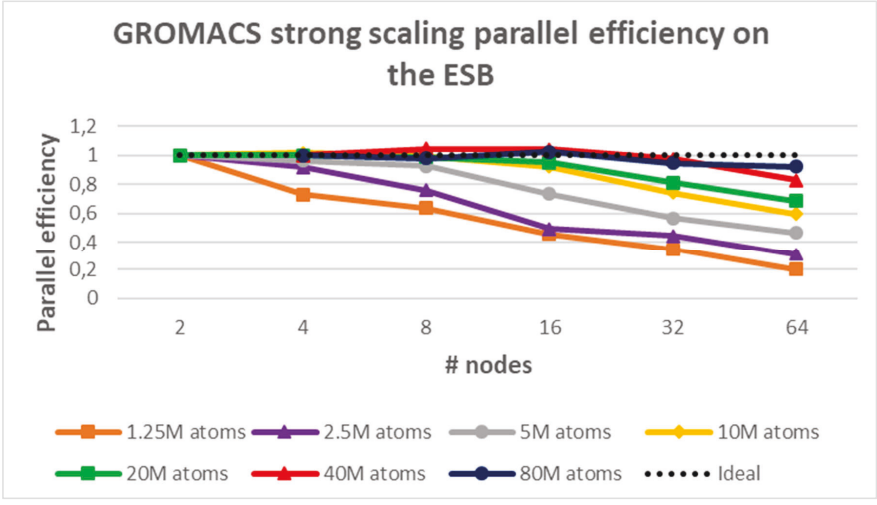


**Figure 3.8: GROMACS parallel efficiency (strong scaling) on ESB for MD simulations of different size**

| # nodes | 2 | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|
| **T1: Time to solution [s]** | 22.4 | 12.8 | 7.4 | 5.2 | 4.1 | 3.7 |
| **T2: Time spent in MPI [s]** | 4.3 | 3.3 | 2.2 | 2.0 | 2.0 | 2.2 |
| **TG: Time in GPU kernels [s]** | 22.4 | 12.8 | 7.4 | 5.2 | 4.1 | 3.7 |
| **Load balance [%]** | 93 | 94 | 93 | 86 | 85 | 77 |
| **# MPI messages [count]** | 451,779 | 897,767 | 1,796,201 | 3,837,839 | 8,309,987 | 27,067,919 |
| **MPI data volume [GByte]** | 98 | 135 | 89 | 133 | 194 | 262 |
| **Most important MPI operation** | MPI_Sendrecv | MPI_Sendrecv | MPI_Sendrecv | MPI_Sendrecv | MPI_Sendrecv | MPI Sendrecv |
| **Most important MPI operation [%]** | 76 | 82 | 80 | 77 | 73 | 69 |
| **2nd most important MPI operation** | MPI_Alltoall | MPI_Alltoall | MPI_Alltoall | MPI_Alltoall | MPI_Alltoall | MPI_Alltoall |
| **2nd most important MPI operation [%]** | 23 | 16 | 19 | 22 | 26 | 28 |
| **3rd most important MPI operation** | MPI_Recv | MPI_Recv | MPI_Recv | MPI_Recv | MPI_Recv | MPI_Recv |
| **3rd most important MPI operation [%]** | 0.7 | 1.3 | 0.4 | 0.5 | 0.5 | 1.9 |

**Table 3.1: GROMCAS strong scaling measurements for the Bombinin test case with 1.25M atoms**

As seen in the Table 3.1, the scalability of the MD simulation with 1.25M atoms saturates at 32 nodes, when the communication time becomes roughly equal to the computation time and even longer than computation time – for 64 nodes. For all other simulations, the computations takes longer than the communications and the load balance is above 93%. The most important MPI communication call is `MPI_Sendrecv`, taking more than 69% of the time spent in the MPI calls in all cases. The second most important MPI communication call is `MPI_Alltoall`, which takes almost all the rest of the MPI time, while the third most important MPI communication call takes less than 4% of the MPI time.

| # nodes | 2 | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|
| **T1: Time to solution [s]** | 352 | 177 | 91 | 47 | 27 | 16 |
| **T2: Time spent in MPI [s]** | 58 | 30 | 19 | 10 | 7 | 6 |
| **TG: Time in GPU kernels [s]** | 352 | 177 | 91 | 47 | 27 | 16 |
| **Load balance [%]** | 98 | 97 | 96 | 96 | 94 | 94 |
| **# MPI messages [count]** | 426,039 | 876,291 | 1,703,475 | 3,545,936 | 7,144,548 | 24,431,296 |
| **MPI data volume [GByte]** | 1025 | 1299 | 1622 | 2156 | 2876 | 3561 |
| **Most important MPI operation** | MPI_Sendrecv | MPI_Sendrecv | MPI_Sendrecv | MPI_Sendrecv | MPI_Sendrecv | MPI_Sendrecv |
| **Most important MPI operation [%]** | 80 | 73 | 66 | 62 | 62 | 60 |
| **2nd most important MPI operation** | MPI_Alltoall | MPI_Alltoall | MPI_Alltoall | MPI_Alltoall | MPI_Alltoall | MPI_Alltoall |
| **2nd most important MPI operation [%]** | 19 | 24 | 31 | 37 | 36 | 35 |
| **3rd most important MPI operation** | MPI_Recv | MPI_Recv | MPI_Recv | MPI_Recv | MPI_Recv | MPI_Recv |
| **3rd most important MPI operation [%]** | 0.9 | 1.9 | 2.1 | 0.4 | 0.6 | 2.4 |

**Table 3.2: GROMACS strong scaling measurements for the Bombinin test case with 20M atoms**

| # nodes | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|
| **T1: Time to solution [s]** | 262 | 178 | 92 | 49 | 28 |
| **T2: Time spent in MPI [s]** | 69 | 31 | 20 | 12 | 9 |
| **TG: Time in GPU kernels [s]** | 262 | 1782 | 922 | 492 | 282 |
| **Load balance [%]** | 96 | 98 | 93 | 95 | 94 |

| | | | | | |
|---|---|---|---|---|---|
| **# MPI messages [count]** | 824,046 | 1,723,736 | 3,323,946 | 7,198,028 | 14,259,647 |
| **MPI data volume [GByte]** | 2081 | 2599 | 3270 | 4273 | 5761 |
| **Most important MPI operation** | MPI_Sendrecv | MPI_Sendrecv | MPI_Sendrecv | MPI_Sendrecv | MPI_Sendrecv |
| **Most important MPI operation [%]** | 76 | 62 | 60 | 53 | 52 |
| **2nd most important MPI operation** | MPI_Alltoall | MPI_Alltoall | MPI_Alltoall | MPI_Alltoall | MPI_Alltoall |
| **[%]** | 21 | 35 | 38 | 46 | 46 |
| **3rd most important MPI operation** | MPI_Recv | MPI_Recv | MPI_Recv | MPI_Bcast | MPI_Recv |
| **[%]** | 1.9 | 1.5 | 1.8 | 0.5 | 0.7 |

**Table 3.3: GROMACS strong scaling measurements for the Bombinin test case with 40M atoms**

| **# nodes** | **4** | **8** | **16** | **32** | **64** |
|---|---|---|---|---|---|
| **T1: Time to solution [s]** | 752 | 376 | 188 | 95 | 51 |
| **T2: Time spent in MPI [s]** | 156 | 87 | 46 | 22 | 13 |
| **TG: Time in GPU kernels [s]** | 752 | 376 | 188 | 95 | 51 |
| **Load balance [%]** | 94 | 96 | 97 | 96 | 96 |
| **# MPI messages [count]** | 854,543 | 1,635,915 | 3,380,131 | 6,683,527 | 14,329,559 |
| **MPI data volume [GByte]** | 3564 | 4256 | 5275 | 6536 | 8552 |
| **Most important MPI operation** | MPI_Sendrecv | MPI_Sendrecv | MPI_Sendrecv | MPI_Sendrecv | MPI_Alltoall |
| **Most important MPI operation [%]** | 77 | 77 | 59 | 52 | 52 |

| **2nd most important MPI operation** | MPI_Alltoall | MPI_Alltoall | MPI_Alltoall | MPI_Alltoall | MPI_Sendrecv |
|---|---|---|---|---|---|
| **2nd most important MPI operation [%]** | 19 | 21 | 39 | 46 | 47 |
| **3rd most important MPI operation** | MPI_Recv | MPI_Recv | MPI_Recv | MPI_Recv | MPI_Bcast |
| **3rd most important MPI operation [%]** | 4.1 | 1.5 | 1.3 | 1.0 | 0.6 |

**Table 3.4: GROMACS strong scaling measurements for the Bombinin test case with 80M atoms**

The strong scalability gets better when increasing the problem size as seen in Table 3.3 and Table 3.4 for MD simulation with 40M and 80M atoms, respectively. The parallel efficiency is close to the ideal one as show in Figure 3.8. This trend holds until the duration of the PME calculations running on the CPUs of the ESB nodes would not exceed the duration of the PP calculations running on the GPUs of the ESB. Such a condition ensures overlapping communications in the PME part (on the CPUs) with the particle-to-particle calculation (on the GPUs).

Weak scaling

Figure 3.9 and Figure 3.10 show the weak scalability of the application for the evaluated simulations. In this scenario the volume of work per node is kept constant by running the 2.5M system on 2 nodes, the 5M system on 4 nodes, the 10M system on 8 nodes, the 20M system on 16 nodes, the 40M system on 32 nodes, and the 80M system on 64 nodes. As visible from Table 3.2 and Table 3.4, the `MPI_Alltoall` share of the total communication time rises from 37% on 16 nodes to 52% on 64 nodes, which limits the weak scalability when increasing the number of nodes.

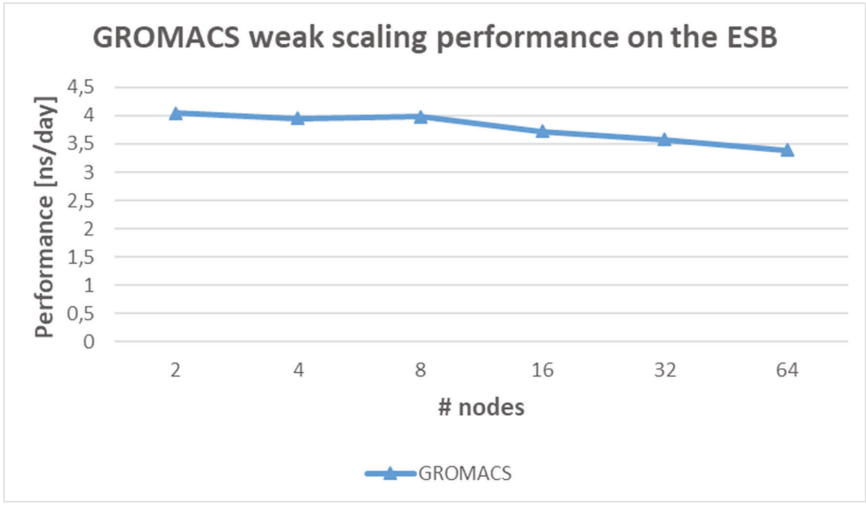


**Figure 3.9: GROMACS weak scaling performance on the ESB**

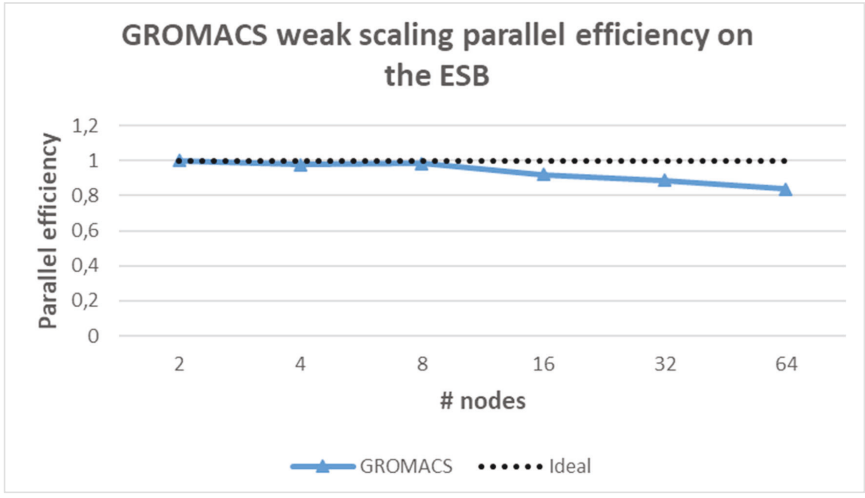


**Figure 3.10: GROMACS weak scaling parallel efficiency on the ESB**

### 3.5.2 CM Scalability results

Strong scaling

The strong scaling performance of MD simulations with 2.5M, 20M, and 80M atoms when running GROMACS on the CM is shown in Figure 3.11 and the corresponding parallel efficiency is shown in Figure 3.12. For these experiments 24 MPI ranks (18 for PP and 6 for PME calculations) per node and 2 OpenMP threads per MPI rank were used. The MD simulations with numbers of atoms between 300k and 2M show good scalability; bigger simulations presented worse strong scalability due to the limiting effect of collective communications between the PME ranks. Overall, the single-module MD simulations on both CM and ESB show good scalability for the entire range of simulation sizes up to several millions of atoms.

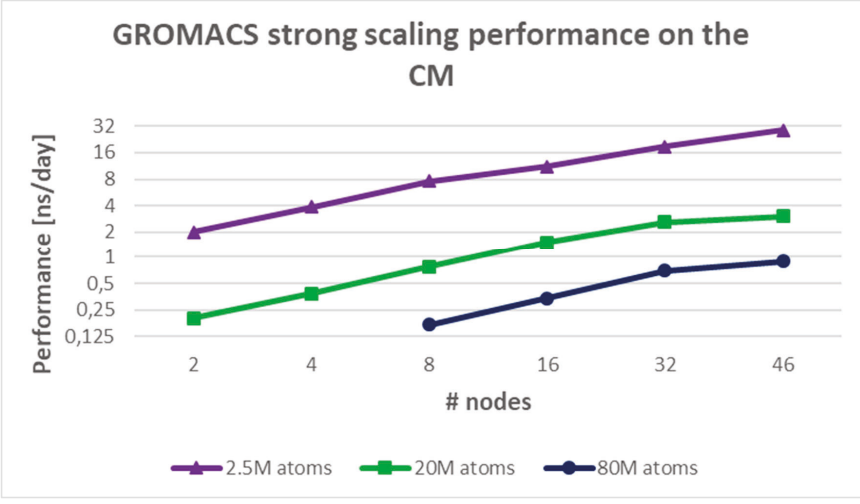


**Figure 3.11: GROMACS strong scaling performance on the CM for MD simulations of different size**

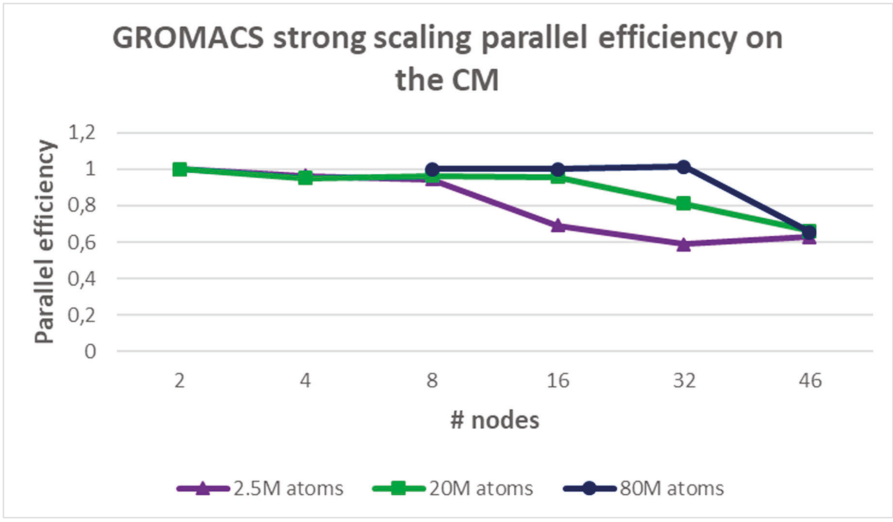


**Figure 3.12: GROMACS strong scaling parallel efficiency on the CM for MD simulations of different size**

### *3.5.3 ESB+CM Scalability results*

Strong scaling GROMACS with PME

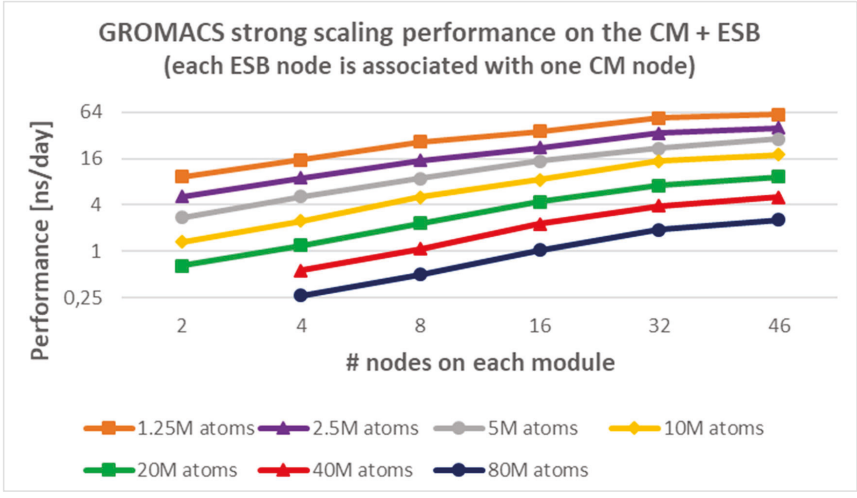


**Figure 3.13: GROMACS strong scaling performance in Cluster-Booster configuration on the ESB (PP) and the CM (PME) for MD simulations of different size**

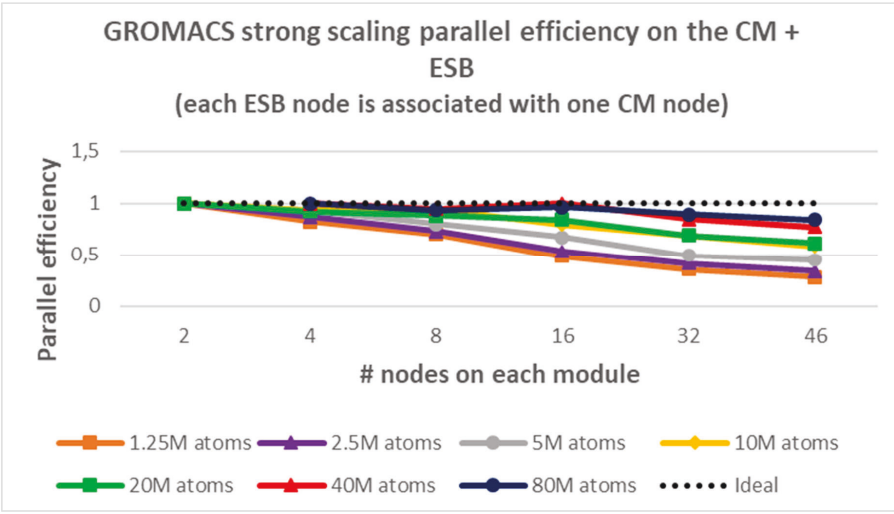


**Figure 3.14: GROMACS strong scaling parallel efficiency in Cluster-Booster configuration on the ESB (PP) and the CM (PME) for MD simulations of different size**

As discussed above, MD simulations in Cluster-Booster configuration run the particle-to-particle (PP) calculations on the ESB and long-range electrostatics calculations with PME method on the CM. The optimal performance was reached with a 1:1 ratio of ESB nodes to CM nodes. The GROMACS performance of MD simulations with 1.25M, 2.5M, 5M, 10M, 20M, 40M, 80M atoms is plotted in Figure 3.13 and the corresponding parallel efficiency in Figure 3.14. Detailed comparison of the corresponding performance of the ESB-only runs (plotted in Figure 3.7) showed performance gain of between 10% and 40% as depicted in Figure 3.15. The relative performance gain was calculated according to the following formula:

$$RPG = \frac{P_{ESB+CM} - P_{ESB}}{P_{ESB}} \times 100\%,$$

where *RPG* denotes the Relative Performance Gain, $P_{ESB+CM}$ the performance in Cluster-Booster configuration, and $P_{ESB}$ the performance on the ESB module.

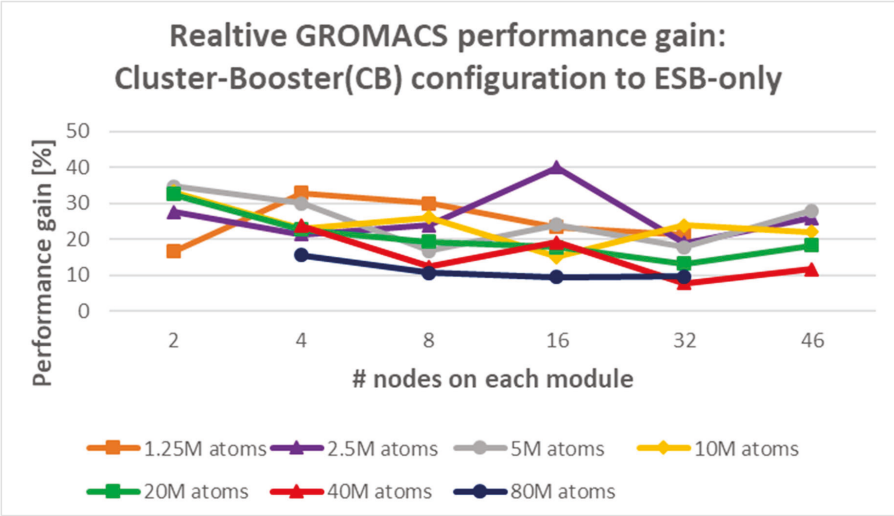


**Figure 3.15: Relative performance gain of GROMACS in Cluster-Booster configuration to single module configuration (ESB) for MD simulations with different size**

Weak scaling GROMACS with FMM

In most cases, PME outperforms FMM, while the latter starts to become beneficial for large volumes of the simulation box. However, GROMACS has an intrinsic hard limit for the input number of atoms in a system (~100 million). For dense systems such as the ones used in life sciences research this limit is reached at approximately ~1 Million $nm^3$ ($Mnm^3$) volumes. In order to perform weak scalability of FMM for much larger volumes, a sparse system needs to be taken as a test case. To evaluate the weak scalability of the newly implemented multi-GPU FMM method integrated with GROMACS, a starting aerosol problem containing 75 droplets of water[66] (217,326 atoms) in a simulation box of volume ~5 $Mnm^3$ is n-folded up to 32 times. The largest problem obtained in this way contains 6,954,432 atoms in a simulation box with a volume of ~160 $Mnm^3$. The problem is simulated for 200 MD steps on increasing numbers of ESB nodes. The number of MPI tasks on the CM is determined in order to minimize energy usage while keeping the performance balance between the ESB and CM. The amount of CM nodes needed is reduced because they do not need to calculate the pair-wise Coulomb interactions, which are already included in the FMM algorithm. The execution becomes therefore generally faster. Table 3.5 shows the obtained measurements.

[66] https://www.mpibpc.mpg.de/17532883/03_aerosol.tgz

| # nodes CM module | 1 | 1 | 1 | 2 | 3 | 3 |
|---|---|---|---|---|---|---|
| **# nodes ESB module** | 1 | 2 | 4 | 8 | 16 | 32 |
| **Full System** | | | | | | |
| **T1: Time to solution [s]** | 11 | 16 | 35 | 11 | 18 | 39 |
| **T2: Time spent in MPI [s]** | 4.3 | 8.0 | 13.4 | 3.8 | 8.7 | 16.2 |
| **Load balance [%]** | 69 | 52 | 62 | 69 | 62 | 62 |
| **# MPI messages** | 4,672 | 12,843 | 16,162 | 97,064 | 289,524 | 532,028 |
| **MPI data volume [GByte]** | 1.5 | 4.0 | 11.4 | 20.8 | 47.3 | 120.5 |
| **Module CM** | | | | | | |
| **T1: Runtime on Module CM [s]** | 11 | 16 | 35 | 11 | 18 | 39 |
| **T2: Time spent in MPI [s]** | 4.3 | 9.9 | 19.8 | 4.6 | 10.6 | 23.5 |
| **# MPI messages CM** | 1,686 | 5,144 | 5,212 | 31,436 | 55,411 | 82,498 |
| **MPI data volume [GB]** | 0.02 | 0.09 | 0.18 | 0.47 | 1.15 | 1.58 |
| **Load balance [%]** | 95 | 96 | 96 | 81 | 95 | 89 |
| **Module ESB** | | | | | | |
| **T1: Runtime on Module ESB [s]** | 11 | 16 | 35 | 11 | 18 | 39 |
| **T2: Time spent in MPI [s]** | 0.2 | 0.3 | 0.5 | 0.7 | 0.8 | 1.7 |
| **T_GPU: Time spent in GPU kernels [s]** | 8 | 14 | 31 | 8 | 15 | 34 |
| **# MPI messages ESB** | 200 | 1,000 | 1,000 | 1,000 | 1,000 | 1,000 |
| **MPI data volume [GB]** | 0.0 | 1.0 | 5.4 | 8.7 | 23.0 | 72.0 |
| **Load balance [%]** | 100.00 | 99.99 | 99.91 | 99.59 | 99.68 | 99.32 |
| **Inter-modular communication** | | | | | | |
| **MPI data transfer time [s]** | 0.2 | 0.1 | 0.3 | 0.1 | 0.3 | 0.5 |
| **# of MPI messages** | 2,786 | 6,699 | 9,950 | 64,628 | 232,942 | 448,530 |
| **MPI data volume [GB]** | 1.5 | 2.9 | 5.9 | 11.6 | 23.2 | 46.9 |
| **Most important MPI operation** | MPI_Recv | MPI_Recv | MPI_Waitall | MPI_Waitall | MPI_Waitall | MPI_Waitall |

| | | | | | | |
|---|---|---|---|---|---|---|
| **Most important MPI operation [%]** | 59 | 72 | 61 | 70 | 74 | 59 |
| **2nd most important MPI operation** | MPI_Waitall | MPI_Waitall | MPI_Recv | MPI_Recv | MPI_Recv | MPI_Recv |
| **2nd most important MPI operation [%]** | 38 | 24 | 37 | 25 | 22 | 36 |
| **3rd most important MPI operation** | MPI_Isend | MPI_Isend | MPI_Isend | MPI_Isend | MPI_Isend | MPI_Isend |
| **3rd most important MPI operation [%]** | 1.8 | 3.0 | 2.1 | 3.5 | 3.0 | 3.5 |

**Table 3.5: GROMACS + IRIS/FMM weak scaling measurements for the aerosol test case**

Figure 3.16 shows the time to solution for the different cases. A distinctive pattern is observed, which at first might lead to the conclusion that this method does not scale at all. However, it is misleading to perform a weak scaling test of the FMM method by doubling the number of processors and problem size; instead, weak scaling should be performed by multiplying the number of processors and problem size by 8 each time.

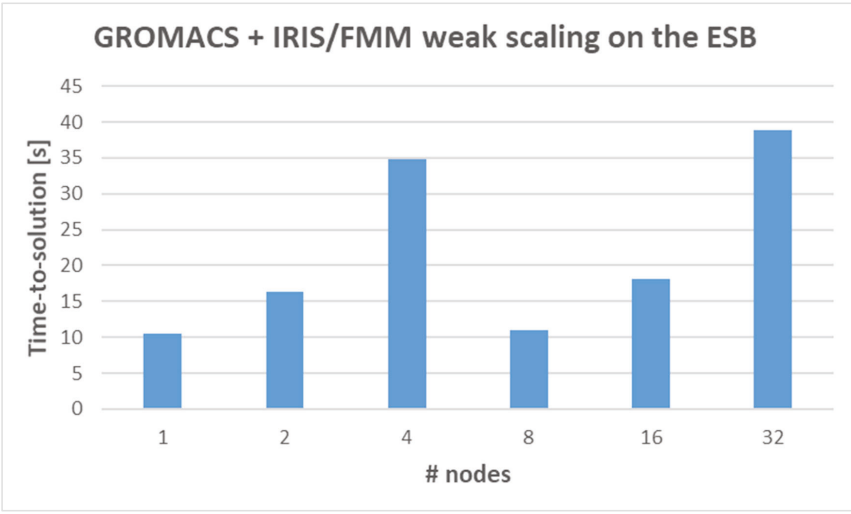


**Figure 3.16: GROMACS + IRIS/FMM weak scaling time to solution (11 MD steps). Tree depth for 1-, 2- and 4- ESB node cases is 4, while for 8-, 16- and 32- ESB node cases is increased to 5**

The FMM method relies on dividing the domain into an oct-tree up to certain configurable depth. Each cell in the tree (except for the leaf nodes) has exactly 8 children. From algorithmic point of view, it is not beneficial to increase the depth unless there are 8 times more processors. This is why the tree depth for the 1-, 2- and 4-ESB node cases is kept constant (depth 4) and for the 8-, 16- and 32-ESB node cases it is increased to 5. By doubling the simulation system size but keeping the depth constant, we end up with leaf cells containing twice the number of atoms compared with the previous case, which inevitably leads to increased simulation time. However, comparing the cases 1-node to 8-node, as well as 2-node to 16-node and 8-node to 32-node, we can see the true weak scalability, as depicted on Figure 3.17. The obvious result from these measurements is that weak scalability for larger number of nodes is preserved, if the depth is increased each time that the number of nodes grows by 8×.

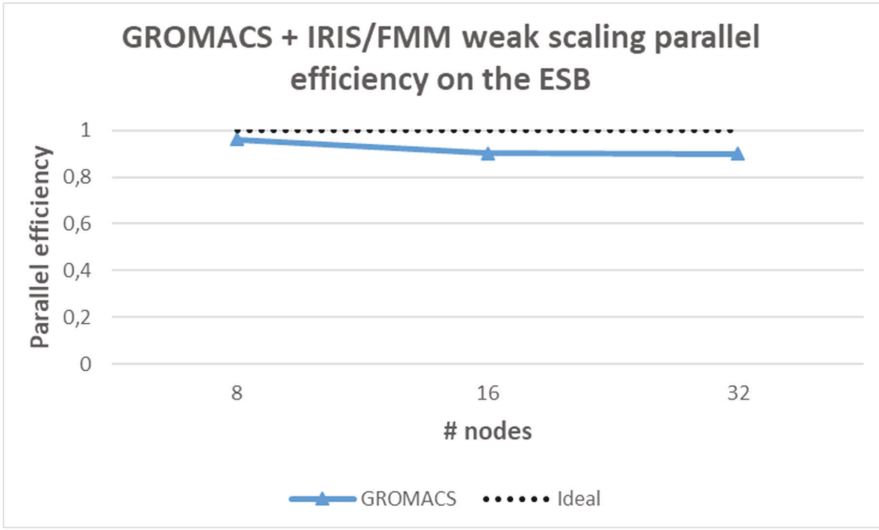


**Figure 3.17: GROMACS + IRIS/FMM weak scalability, comparing 8- to 1- ESB nodes, 16- to 2- ESB nodes and 32- to 4- ESB nodes**

Strong scaling FMM (Standalone IRIS/FMM)

The sparse aerosol system used for the weak scalability tests cannot be used to measure accurately the strong scalability of the FMM method, since there is large load imbalance due to its inhomogeneity. This load imbalance becomes even larger with increasing number of processors, which hinders the scalability. A dense problem is

more suitable for strong scalability tests. Moreover, there are previous results[67] showing strong scalability of standalone multi-GPU FMM implementation for a problem consisting of 100M particles. To this end, the test case chosen for strong scalability tests of the FMM method consists of 100M atoms worth of water molecules in 1 $Mnm^3$ simulation box. Due to the hard limit for the input size in GROMACS such a system cannot be fully simulated. Instead, only the FMM part as implemented in the IRIS library was run on the ESB nodes. The results obtained this way show the strong scaling potential of the developed FMM code itself.

The problem is simulated for 11 MD steps on increasing number of ESB nodes. The CM nodes are used only to load the input data and send it through to the ESB nodes for calculation, thus better representing the situation in an eventual full-simulation scenario. One MPI CM rank corresponds to 1 ESB node. Table 3.6 shows the results. Note that the data for the CM nodes is not representative because of the above comment.

| **# nodes CM module** | **1** | **1** | **1** | **1** | **1** | **2** |
|---|---|---|---|---|---|---|
| **# nodes ESB module** | 1 | 2 | 4 | 8 | 16 | 32 |
| **Full system** | | | | | | |
| **T1: Time to solution [s]** | 2,204 | 1,137 | 564 | 287 | 149 | 77 |
| **T2: Time spent in MPI [s]** | 1,096 | 579 | 289 | 144 | 73 | 38 |
| **Load balance [%]** | 51 | 52 | 52 | 51 | 52 | 55 |
| **# MPI messages** | 145 | 255 | 519 | 1,572 | 5,796 | 22,692 |
| **MPI data volume [GByte]** | 36 | 42 | 48 | 57 | 65 | 78 |
| **Module CM** | | | | | | |
| **T1: Runtime on Module CM [s]** | 2,204 | 1,137 | 564 | 287 | 149 | 78 |
| **T2: Time spent in MPI [s]** | 2,144 | 1,127 | 557 | 285 | 143 | 72 |
| **# MPI messages CM** | 0 | 0 | 0 | 0 | 0 | 0 |
| **MPI data volume [GB]** | 0 | 0 | 0 | 0 | 0 | 0 |
| **Load balance [%]** | 100.0 | 99.8 | 99.5 | 98.5 | 97.1 | 99.2 |
| **Module ESB** | | | | | | |

[67] Rio Yokota, Tsuyoshi Hamada, Jaydeep P. Bardhan, Matthew G. Knepley, Lorena A. Barba: Biomolecular Electrostatics Simulation by an FMM-based BEM on 512 GPUs. CoRR abs/1007.4591 (2010)

| | | | | | | |
|---|---|---|---|---|---|---|
| **T1: Runtime on Module ESB [s]** | 2,204 | 1,137 | 564 | 287 | 149 | 78 |
| **T2: Time spent in MPI [s]** | 49 | 31 | 22 | 3 | 3 | 3 |
| **T_GPU: Time spent in GPU kernels [s]** | 2,120 | 1,113 | 532 | 275 | 143 | 71 |
| **# MPI messages ESB** | 11 | 55 | 55 | 55 | 55 | 55 |
| **MPI data volume [GB]** | 4.10E-07 | 5.3 | 12 | 20 | 29 | 41 |
| **Load balance [%]** | 100 | 99.8 | 99.6 | 99.2 | 98.5 | 97.3 |
| **Inter-modular communication** | | | | | | |
| **MPI data transfer time [s]** | 6.1 | 3.6 | 1.7 | 0.7 | 0.3 | 0.2 |
| **# of MPI messages** | 134 | 200 | 464 | 1,517 | 5,741 | 22,637 |
| **MPI data volume [GB]** | 36.3 | 36.3 | 36.3 | 36.3 | 36.3 | 36.3 |
| **Most important MPI operation** | MPI_Recv | MPI_Recv | MPI_Recv | MPI_Recv | MPI_Recv | MPI_Recv |
| **Most important MPI operation [%]** | 78 | 69 | 75 | 63 | 66 | 57 |
| **2nd most important MPI operation** | MPI_Wait | MPI_Wait | MPI_Wait | MPI_Wait | MPI_Wait | MPI_Wait |
| **2nd most important MPI operation [%]** | 20 | 30 | 22 | 33 | 30 | 36 |
| **3rd most important MPI operation** | MPI_Iprobe | MPI_Isend | MPI_Isend | MPI_Isend | MPI_Isend | MPI_Isend |
| **3rd most important MPI operation [%]** | 1.9 | 0.9 | 2.0 | 3.4 | 4.7 | 6.3 |

**Table 3.6: Standalone IRIS/FMM strong scaling measurements for the 100M test case**

The presented data shows that the strong scalability of the multi-GPU FMM code has a nearly perfect parallel efficiency (see Figure 3.19). The time spent in GPU kernels also scales near ideally and dominates the execution time, as shown in Figure 3.18. The communication between ESB ranks is completely overlapped by the calculations on the GPU kernels (more specifically P2P self-interactions) and does not contribute to the total step time. Moreover, the MPI time is less than 4% of the total execution time.

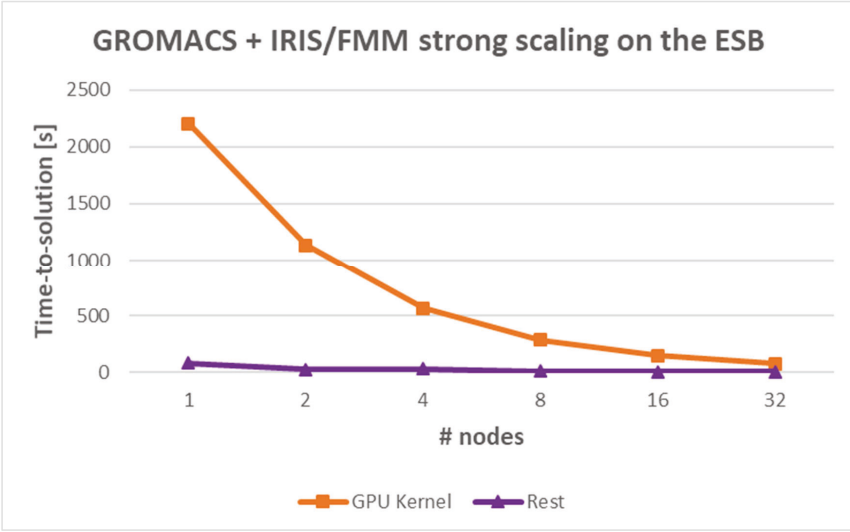


**Figure 3.18: Standalone IRIS/FMM time to solution for the GPU kernels and the complete ESB step. GPU kernels time is shown in blue, while all the rest of the activities (data preparation, CPU activities, data transfer) is shown in orange**

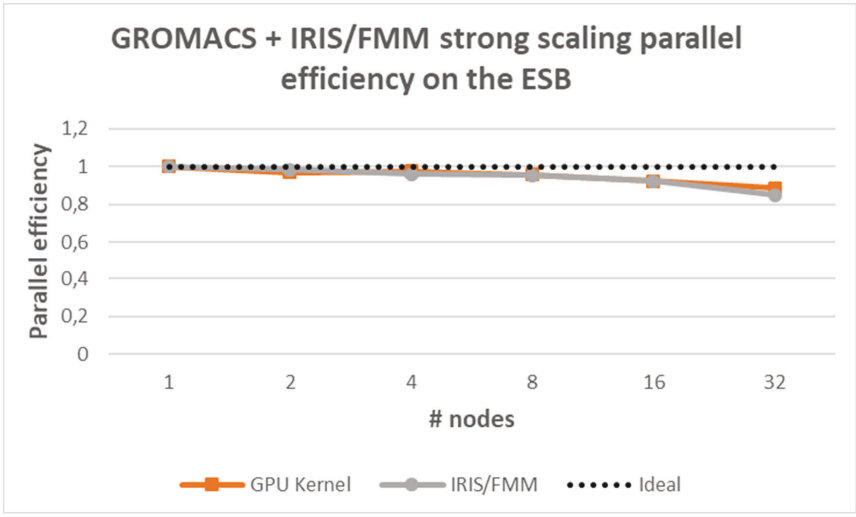


**Figure 3.19: Standalone IRIS/FMM parallel efficiency for the GPU kernels (shown in orange) and the complete ESB step (shown in grey)**

### *3.5.4 Our path to Exascale*

In order to extrapolate towards Exascale we need to look at the details of the computation load and communication patterns of a single MD step and assess their inherent scalability.

#### *3.5.4.1 P3M/PME*

The PME/P3M MD step consists of the following components:

- Receive input data;
- Particle to Mesh;
- Halo exchange;
- Forward 3D FFT, including remap;
- Calculate reciprocal space electrostatic energy;
- 3x Backward 3D FFT, including remap;
- Mesh to Particle;
- Send output results.

The ***Receive input data*** and ***Send output results*** component involves asynchronous point-to-point communication only (non-blocking send to blocking receive) and its duration depends on the amount of data transferred and the latency and throughput of the network.

The ***Particle to Mesh*** and ***Mesh to Particle*** components perform only computations. Their complexity depends on the number of atoms per rank and size of the computation mesh. Both strong and weak scaling should not be limited.

The ***Halo exchange*** involves point-to-point communication only and the amount of data to be transferred depends on the accuracy defined by the user. In the strong scaling case, the scalability is limited by the interconnecting network bandwidth.

There are two main subcomponents in the ***3D FFT***, namely 2D or 1D FFT local calculations and collective communications to remap the mesh to prepare it for the FFT in the remaining dimension(s). All ranks are involved in the collective communications and they are the main source of scalability saturation in both strong and weak scaling cases, while the FFT calculations do not influence the performance scalability. Moreover, the duration of the collective all-to-all communication heavily depends on the size of the computational mesh, the whole span of which needs to be exchanged, and this greatly influences weak scalability.

#### *3.5.4.2 FMM*

The FMM MD step consists of the following components:

- Receive input data;
- Local tree construction;
- Exchange of the Local Essential Tree;
- Dual tree traversal;
- Send output results.

The ***Receive input data*** and ***Send output results*** components involve asynchronous point-to-point communication only (non-blocking send to blocking receive) and their duration depends on the amount of data transferred and the latency and throughput of the network.

For the strong scaling case the amount of data transferred to a single MPI task is reduced as the number of ranks is increased. For the weak scaling case the amount of data transferred to a single MPI rank is constant. The total amount of data is increased, along with the number of messages. In both cases the scalability of the component is limited mainly by the latency and throughput of the network.

The ***Local tree construction*** component involves only computation and all steps are of O(N) complexity. Both strong and weak scaling should not be limited.

The ***Exchange of the Local essential tree*** component involves two all-to-all communications: one for exchanging the P2P halo atoms and one for exchanging the cells needed by other processors to perform M2L kernels. These exchanges are overlapped with the P2P in-cell interactions computed on the GPU. The solver can be optimally parametrized by the user so that this communication is completely hidden. For the CPU implementation however, this is the main bottleneck of the method.

This component also involves additional calculations for reconstructing the non-local part of the tree shared by all nodes. For both strong and weak scaling, the additional calculations in the local essential tree stays generally of the same order and does not scale, but their duration can be made relatively small if the performance of the CPU/GPU is high enough. Thus, the scalability is limited by the single node FPU performance.

The ***Dual tree traversal*** component involves only computation and all steps involved are of O(N) complexity. Both strong and weak scaling should not be limited.

#### *3.5.4.3 What are the limitations? – Can they be fixed?*

For MD simulations as a whole, the main scalability limiting factor is the number of atoms per MPI rank. When the number of atoms per node (CPU only) goes below roughly 1,000, the communication starts dominating.

The main limitation of the **PME** method is its weak scalability when the problem volume approaches millions of $nm^3$ in realistic scenarios with Fourier spacing not exceeding 2.2 Å; in this case the all-to-all MPI communications necessary for the 3D FFT become a severe limiting factor. This limitation cannot be fixed since it is an inherent characteristic of the method and these communications cannot be overlapped with meaningful computation. This is the main reason for developing the multi-GPU FMM code as part of this project. Apart from that limitation, PME shows good strong scalability and excellent performance for most MD simulations required in life sciences nowadays and can be used exceptionally well in ensemble simulations.

FMM provides a viable alternative for large problems[68] and enables simulations that are not feasible with PME nowadays. Its strong scalability is limited by the computation vs. data transfer ratio, and specifically in the GPU case by the computation vs. memory transfer ratio. For the hardware used in the ESB nodes this happens when the single step wall-clock time starts approaching ~100ms (3.5 ns/day at 4 fs MD time step).

Another important factor that limits both the strong and weak scalability in MD simulations, regardless of the method used, is load imbalance due to problem inhomogeneity. In GROMACS there is a dynamic load-balancer that aims to mitigate this problem by rescaling the local domains.

#### *3.5.4.4 How to use future Exascale systems*

According to the application's present status, the Exascale performance could be reached either by a combination of ensemble and strong scaling, or by weak scaling.

In drug design, they investigate the interaction of a particular protein with many ligands, which results in running many MD simulations to solve drug candidate discovery. The strong scaling limit of the single MD simulation determines the number of nodes per MD simulation, and the number of simultaneously running simulations depends on the available resources. The I/O operations per MD simulation, namely writing trajectories, are done once per second on average, so they are not expected to play a limiting role.

Large MD simulations are used in molecular biology, polymer science, material science, etc. Such MD simulations include increasingly larger space volumes with

[68] Tchipev N, Seckler S, Heinen M, et al. The International Journal of High Performance Computing Applications. 2019;33(5):838-854. doi:10.1177/1094342018819741

number of particles in the order of hundreds of millions or even billions[69]. In such cases, the number of particles per node is kept optimal, and the MD simulation is run on the corresponding number of nodes (increasing the number of nodes). Such large volumetric problems can be solved using the FMM and having in mind that its strong scalability starts deteriorating when the wall-clock time of the single step starts approaching ~100 ms on current hardware. We can therefore conclude that it can become possible to simulate a problem involving billions of particles on pre-Exascale and Exascale systems with at least 5 ns/day performance (at 4 fs/step).

#### *3.5.4.5 Where did the DEEP-EST project help on the way to Exascale?*

The MSA idea behind the DEEP-EST project allows MD simulation packages to run algorithmically different parts of the problem on more appropriate computing architecture to optimize the price/performance ratio of the computation. It adds versatility and allows choosing the right combination of nodes depending on the simulation size in order to achieve the best performance with as little energy as possible. This would be impossible in a homogenous system with a unique type of nodes.

The multi-GPU FMM implementation, which enables the computation of very large problems, further benefits from the MSA idea by utilizing the low-performance CPUs of the ESB nodes to do the FMM-related housekeeping tasks, like dual-tree traversal, LET construction and communications. In the meantime, the high-performance CPUs of the CM nodes are busy with bonded interactions and Van der Waals computations. Moreover, there is flexibility to tune the number of CM nodes against ESB nodes for better load balance. Keeping the particle-to-particle ranks and the FMM ranks on separate modules allows the user to bundle the particle-to-particle ranks on a smaller number of CM nodes, thus reducing the network load by keeping most of the interactions in memory.

## 3.6 Energy consumption

The energy consumption was measured for runs shown in Section 3.5. Here, only the data for MD simulations of 2.5M, 20M and 80M atoms are presented to illustrate the energy consumption for medium, big and large simulations. The data collected for 10,000 MD steps runs on the CM, ESB and Booster-Cluster configuration (ESB+CM) for different MD simulations sizes are plotted in Figure 3.20, Figure 3.21, and Figure 3.22 for 2.5M, 20M and 80M atoms, respectively. In ESB+CM configuration the number

[69] Jung, J., Nishima, W., Daniels, M., Bascom, et al. J. Comput. Chem. 2019, 40, 1919– 1930. DOI: 10.1002/jcc.25840

of ESB nodes equals the number of CM nodes and the sum of both kinds of nodes is plotted. Long-range electrostatics was calculated with PME.

These plots show that the ESB consumes the lowest amount of energy for all runs, while the CM has about 4 times greater consumption on average. The ESB-only and ESB+CM configurations show relatively good and constant behaviour in the strong scaling scenario for 20M and 80M MD simulations.

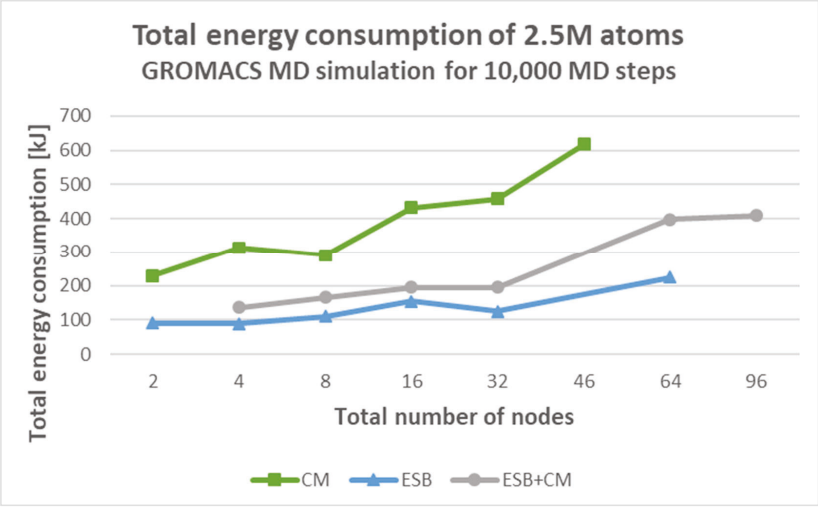


**Figure 3.20: Total energy consumption of 2.5M atoms GROMACS MD simulation for 10,000 MD steps**

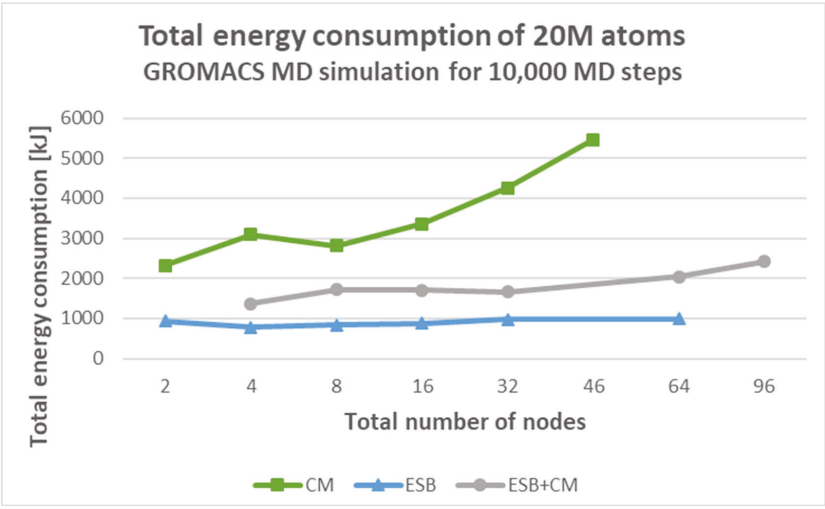


**Figure 3.21: Total energy consumption of 20M atoms GROMACS MD simulation for 10,000 MD steps**

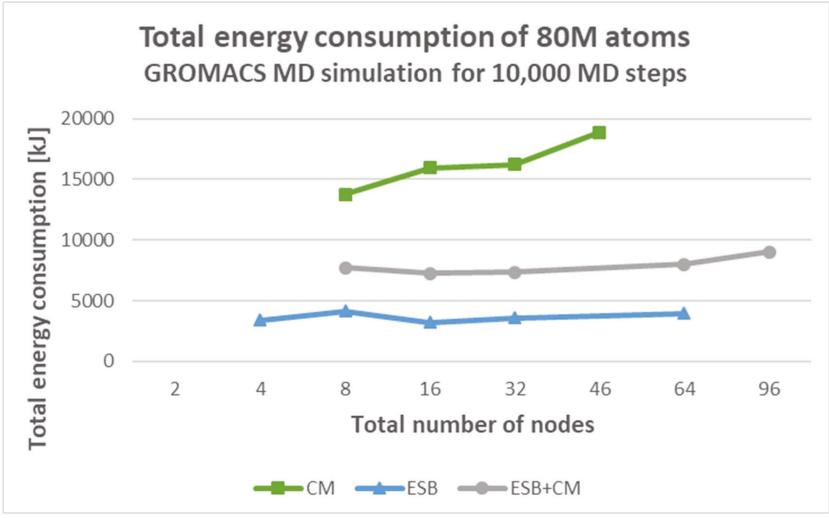


**Figure 3.22: Total energy consumption of 80M atoms GROMACS MD simulation for 10,000 MD steps**

## 3.7 Performance comparison

This sections gives an overview on the energy/performance ratio and the comparison of the old and new algorithms.

### *3.7.1 Energy/Performance ratio*

The Energy vs. Performance ratio is measured in MJ/ns (MegaJoule/nanosecond) and estimates the energy spent to simulate one nanosecond of time evolution. Energy vs. Performance of the CM, ESB and ESB+CM configuration for different MD simulation sizes are plotted in Figure 3.23, Figure 3.24, and Figure 3.25 for 2.5M, 20M and 80M atoms, respectively. In Cluster-Booster configuration the number of ESB nodes equals the number of CM nodes and the total number of nodes is plotted. Long-range electrostatics was calculated with PME.

The ESB has the best Energy vs. Performance ratio and stays constant in the strong scaling scenario for MD simulations with more than 2.5M atoms, as it does the Cluster-Booster configuration. The increase in the performance of ESB+CM configuration shown in Figure 3.15 comes at the expense of higher energy consumption.

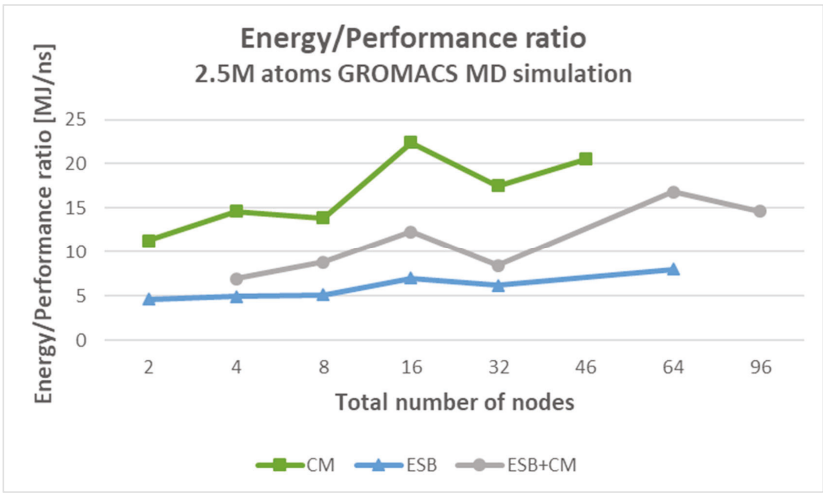


**Figure 3.23: Energy vs. Performance ratio of 2.5M atoms GROMACS MD simulation**

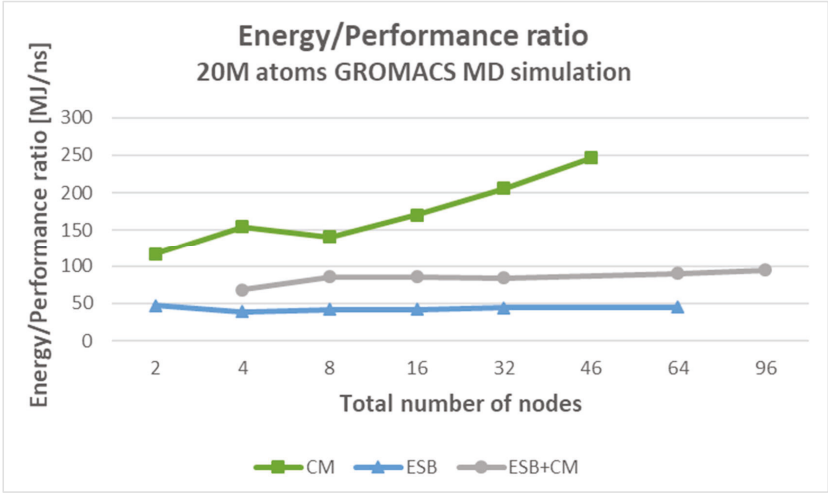


**Figure 3.24: Energy vs. Performance ratio of 20M atoms GROMACS MD simulation**

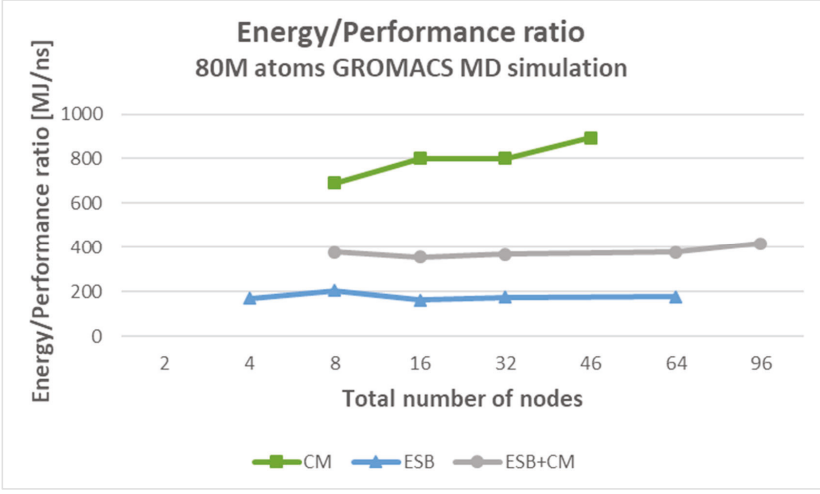


**Figure 3.25: Energy vs. Performance ratio of 80M atoms GROMACS MD simulation**

### *3.7.2 Performance of the newly implemented algorithms*

Direct performance comparison between the existing PME method and the newly implemented multi-GPU FMM is impractical since these methods have a non-overlapping domain of application. PME method outperforms FMM for all problems where it is applicable. On the other hand, FMM enables solving MD problems with very large volumes and number of particles which PME simply cannot handle within

reasonable timeframes. We expect such large problems to be solved on pre-Exascale and Exascale systems using FMM, while PME is used for ensemble simulations.

## 3.8 Conclusion

The DEEP-EST project provided means to further enhance the capabilities of MD software. In computer-aided drug design or life sciences on the MSA one can optimize the price vs. performance ratio by choosing the appropriate configuration of nodes for each particular task. For example, MD simulations of several thousand atoms should run on the CM, while the ESB is beneficial for MD simulations of millions of atoms. In certain cases, the Cluster-Booster configuration shows up to 30% better performance than using ESB nodes only, albite at higher energy cost. The applicability of such trade-off can be considered by the user when the time to solution is more important than the price of the solution itself.

The multi-GPU FMM developed as part of this project is to the best of our knowledge the first such implementation integrated with GROMACS, while a single-GPU version had been developed as part of the SPPEXA project[70]. This new functionality that allows the utilization of FMM on large number of GPUs opens new possibilities for GROMACS to perform very large simulations in fields like material science, polymer science, molecular biology, nanostructures and condensed systems. Results obtained for the MSA architecture show good promise that by using the newly implemented multi-GPU FMM such large simulations consisting of billions of particles may be run at reasonable performance. Future work for this implementation includes overcoming the hard limit on the size of the MD simulation and further optimization of the code both in terms of performance and capabilities. For biological systems in life science research the existing PME method already provides excellent performance on the MSA. The software and hardware work together to establish GROMACS as an even more versatile tool, applicable in a wide range of fields, strongly competing with non-European tools already existing in these areas.

In summary, the MSA employed in the project is suitable for a wide range of applications in the MD domain. Together with the modular hardware architecture, the additions implemented in the application provide extra flexibility to the end-users for selecting the optimal hardware and software configuration depending on their simulation needs.

[70] *J. Chem. Theory Comput.* 2020, 16, 11, 6938–6949

# 4 Radio astronomy with the GPU Correlator, the GPU Imager and the FPGA Imager

**John Romein, Bram Veenboer**
Netherlands Institute for Radio Astronomy, ASTRON, Netherlands

romein@astron.nl

## 4.1 Introduction

Within DEEP-EST, parts of the imaging pipeline of a radio telescope were studied. This is a collection of applications that transforms raw telescope data into calibrated sky images. Figure 4.1 depicts this pipeline. On the left, the signals from antennas in the field are digitised and locally combined. The data are sent over Wide-Area Network links to a central location, where the correlator application filters and combines all data in real time, and writes its output to disk. After the observation has finished, bad data (due to interference) are detected and removed, and the remaining data are calibrated to create an image. During the DEEP-EST project the focus has been on the two computationally most intensive applications in this pipeline: the correlator and the imager. The correlator's main task is to combine the data from all receivers, and the imager's main task is to create sky images.

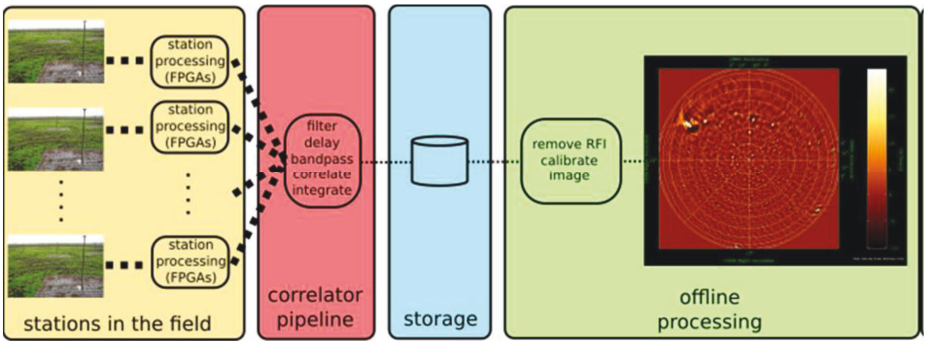


**Figure 4.1: Workflow of the imaging pipeline**

Both applications are highly optimised for GPUs and CPUs and run much faster on GPUs, for various reasons. The imager performs a large number of sine/cosine operations, for which there is efficient hardware support on GPUs but not on CPUs. The newly developed GPU correlator takes advantage of the *tensor cores* of the latest GPU generation. Tensor cores are special-purpose hardware, which achieve an

exceptionally high performance when executing matrix multiplications at mixed-precision, e.g. 71 TFLOP/s on correlations. Hence, both applications are tens of times faster on GPUs than on CPUs, and this affects the way we will use them on the DEEP-EST MSA.

The imager was also ported to FPGAs. FPGAs used to be programmed in Hardware Description Languages like VHDL and Verilog, which is difficult, time consuming and error prone, and not feasible for complex applications like the imager. New FPGA technologies (the OpenCL high-level programming language, hard Floating-Point Units, and tight integration with CPU cores) have changed this: they should reduce the programming effort of “simple” tasks like a correlator, and should allow complex applications like the imager. We explored Intel's OpenCL/FPGA technology to allow comparisons with GPUs (with respect to performance, energy efficiency, and programming effort), and to bring these technologies into radio astronomy.

## 4.2 The GPU Correlator

### *4.2.1 Application structure*

#### ***4.2.1.1 The correlator pipeline***

The correlator combines telescope data and performs signal-processing tasks. The finite impulse response (FIR) filter and FFT blocks transform a wide frequency band into narrow frequency channels. The delay compensation block applies phase corrections to the data that are necessary to follow a source on the sky. The bandpass correction applies an amplitude correction to the data, correcting errors made by another filter near the receivers. The correlate block itself multiplies the data from each pair of receivers, and integrates the products over short periods of time (typically around one second). Especially for a large number of receivers, the correlate block is computationally the most expensive block.

The focus in DEEP-EST was put mostly on improving the performance of the correlation operation. The other operations can be improved when NVIDIA's cuFFTDx library becomes available. As this library will perform FFTs directly on the GPU (unlike cuFFT, which initiates FFTs from the CPU), the first four operations can then be collapsed into one single GPU kernel instead of four kernels, so that only one pass over the data is made, reducing memory bandwidth use. Unfortunately, a pre-release version of cuFFTDx did not compute correct results, so that the integration of cuFFTDx has to be postponed to after the DEEP-EST project. Below, we only report on the correlation operation, which is, especially for large numbers of receivers, the most time-consuming operation anyway.

The correlator combines the data from all receivers by multiplying samples from each pair of receivers and integrating the products over time (typically a second). More specifically, for each frequency channel and each integration period, a matrix with observed, filtered samples S is multiplied by its own Hermitian: $V := S * S^H$. Hence, the output V is also Hermitian, and only the upper- or lower-diagonal triangle needs to be computed and stored, as the output matrix is symmetrical in the diagonal (apart from a minus sign). The BLAS function CHERK computes either side of the diagonal, but stores the output data in a rectangular matrix, wasting GPU memory and PCIe/network bandwidth. Thus, instead of using BLAS, the correlator implements the matrix multiplication itself and stores the data in a triangular data structure.

#### *4.2.1.2 The Tensor-Core Correlator*

The correlator uses new GPU tensor core technology to significantly improve performance[71]. Tensor cores are limited-precision matrix-matrix multiplication units designed to speed up deep learning (training and inference), but ASTRON uses them for signal-processing operations that can be expressed as matrix multiplication, such as a correlator or beam-former. For signal processing, the 32-bit precision provided by regular GPU cores is overkill, as Analog-to-Digital Converters near the receivers typically have an accuracy of 4 to 14 bits. Hence, the limited precision of tensor cores does not affect the correctness of the computed results.

The three major implementation challenges were the following: First, tensor cores only operate on real-valued data while the correlator works with complex-valued data. In particular, negating and swizzling real and imaginary parts, which is necessary to perform a complex multiplication, is performed by regular GPU cores because tensor cores cannot do so. Second, the output data structure is a triangular data structure, which is not supported by the tensor core API. Hardware-dependent tricks (with a portable but slower fallback solution) were necessary to write output data quickly. Third, the tensor cores compute so quickly that it is difficult to provide them fast enough with input data. Efficient caching at all levels in the memory hierarchy, as well as coalesced memory accesses, were necessary to keep the tensor cores busy.

The correlator is implemented as a library, to simplify its use in the pipelines of various radio telescopes. Major facilities like the Canadian CHIME and South-African MeerKAT (an SKA precursor) already included the tensor-core correlator library in the processing pipelines that they develop for their correlator upgrades. ASTRON will use the library for their AARTFAAC correlator upgrade (a LOFAR derivative).

[71] John W. Romein. The Tensor-Core Correlator. Astronomy & Astrophysics, 2021, to appear.

### 4.2.2 Application mapping

Due to the high data rates between the pipeline components of the correlator (in excess of PCIe bandwidth limits), it is not reasonable to separate the operations and run them on different DEEP-EST modules. All operations are performed consecutively on the GPUs of the DAM or ESB (see Figure 4.2).

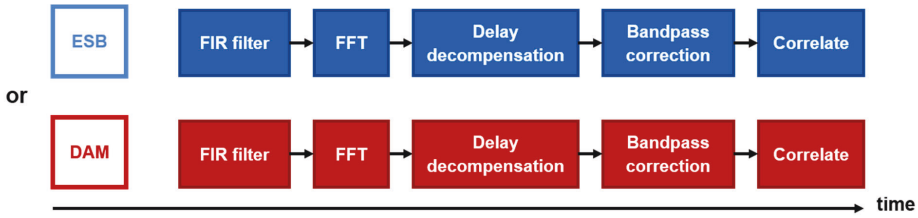


**Figure 4.2: Schematic workflow of the correlator in the MSA**

### 4.2.3 Performance comparison

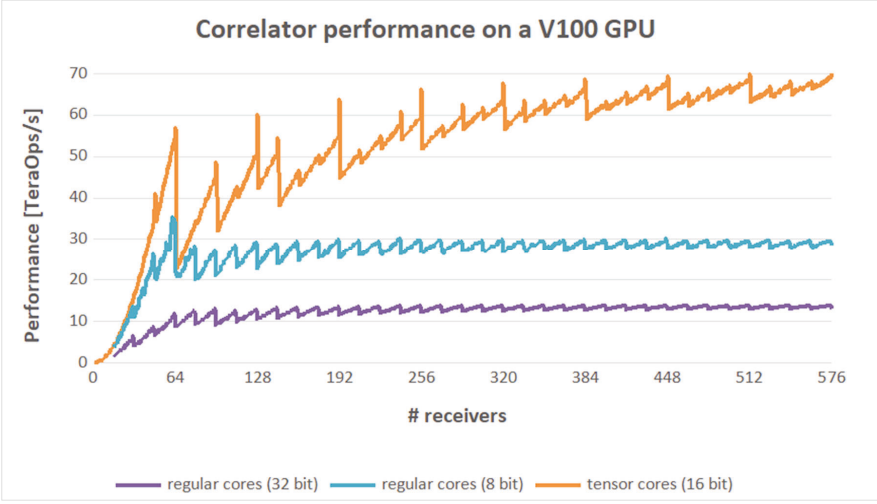


**Figure 4.3: Performance of the tensor-core correlator and a legacy GPU correlator on a V100 GPU, as function of the number of receivers, for various precisions of the input data**

Figure 4.3 shows the enormous performance benefits of using tensor cores, which depends on the required precision. If the telescope observes with 16-bit samples, the tensor-core correlator performs up to 5.4 times better than the legacy code that runs on regular GPU cores of the NVIDIA V100, at 32-bit precision. The results are obtained on an NVIDIA V100 GPU in a DAM node. The tensor-core correlator is benchmarked

against xGPU[72], commonly considered to be the most efficient correlator library for regular GPU cores. xGPU supports 8-bit and 32-bit input data. The ripples in each of the curves are due to the fact that, depending on the number of receivers, the work cannot always be distributed optimally over the GPU cores or tensor cores.

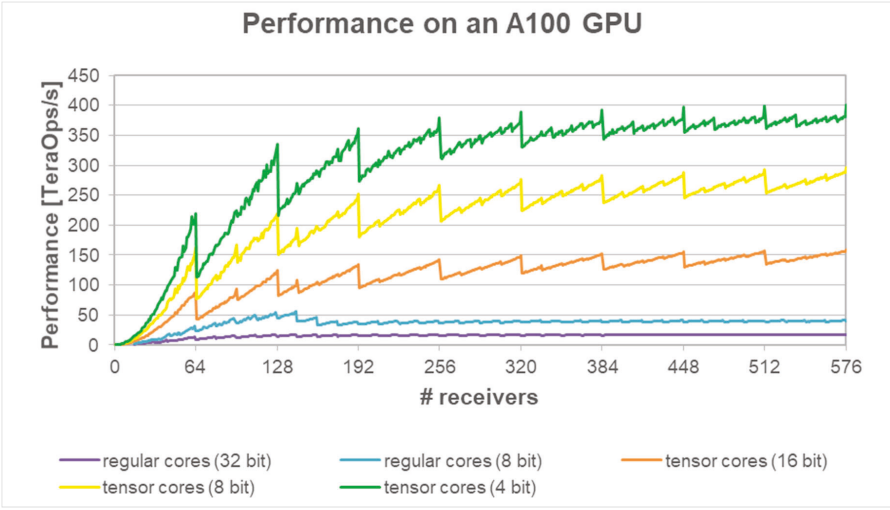


**Figure 4.4: Performance of the tensor-core correlator and a legacy GPU correlator on an A100 GPU, as function of the number of receivers, for various precisions of the input data**

Many telescopes observe with 8-bit or even 4-bit samples though, and these precisions are not naturally supported by the first-generation tensor cores of the V100. Support for 8-bit, 4-bit, and 1-bit was added later in the second-generation tensor cores of Turing GPUs and third-generation tensor cores of Ampere GPUs, which provide even higher performance. ASTRON further developed the tensor-core correlator to also support 8-bit and 4-bit correlations on the newer GPUs. Figure 4.4 shows performance results obtained on a recently introduced Ampere A100 (PCIe) GPU, the successor to the V100 GPU. The figure not only shows order-of-magnitude performance improvements for 16-bit, 8-bit, and 4-bit correlations, but it also shows that the performance gap between tensor cores and regular GPU cores has widened. Thus, the use of tensor-core technology, which ASTRON started exploring on the DEEP-EST system, will become even more important in future systems.

[72] M.A. Clark, P.C. La Plante, and L.J. Greenhill, "Accelerating Radio Astronomy Cross-Correlation with Graphics Processing units", [arXiv:1107.4264 [astro-ph]].

### 4.2.4 Energy consumption

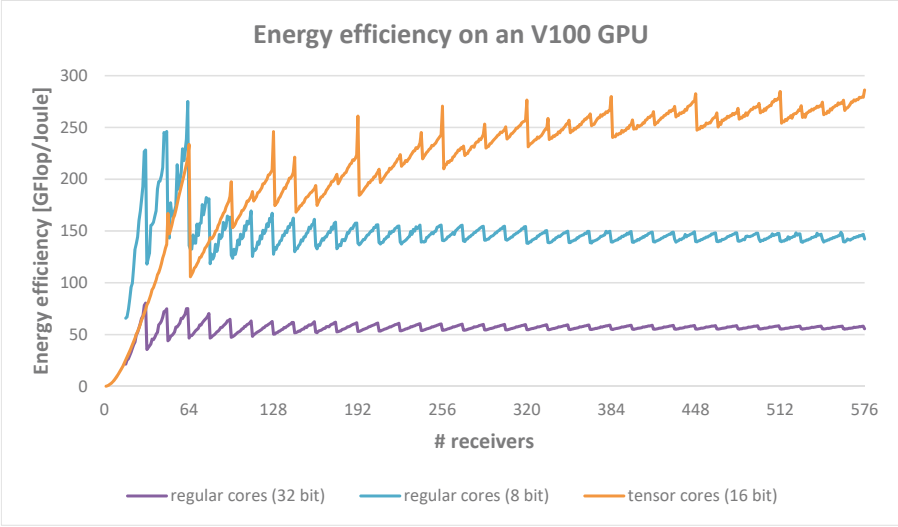


**Figure 4.5: Energy efficiency of the GPU correlator on a V100**

Figure 4.5 shows that the use of tensor cores is also beneficial to obtain high energy efficiency: in typical use cases, the tensor-core correlator is over 5 times more energy efficient than a GPU correlator that runs on regular GPU cores. The energy efficiency was measured on a V100 GPU in a DAM node. The reported energy efficiency is for the GPU only; system-level measurements are reported in Section 4.2.6. The tensor-core correlator library can accurately measure the GPU's energy use through NVIDIA's NVML library.

The tensor-core correlator is actually limited by the maximum amount of power that the GPU may consume: the GPU's clock frequency is lowered so that the energy use remains below 250W. Even though its power use is high, the energy efficiency is very good, because it performs a very high amount of computations per Joule. Since the correlator drives the GPUs in the ESB to its maximum heat production, this application was used in DEEP-EST also to test a cooling-control plugin developed within the project[73].

[73] Moschny, et al. (2021) D5.5 – Software Support Report

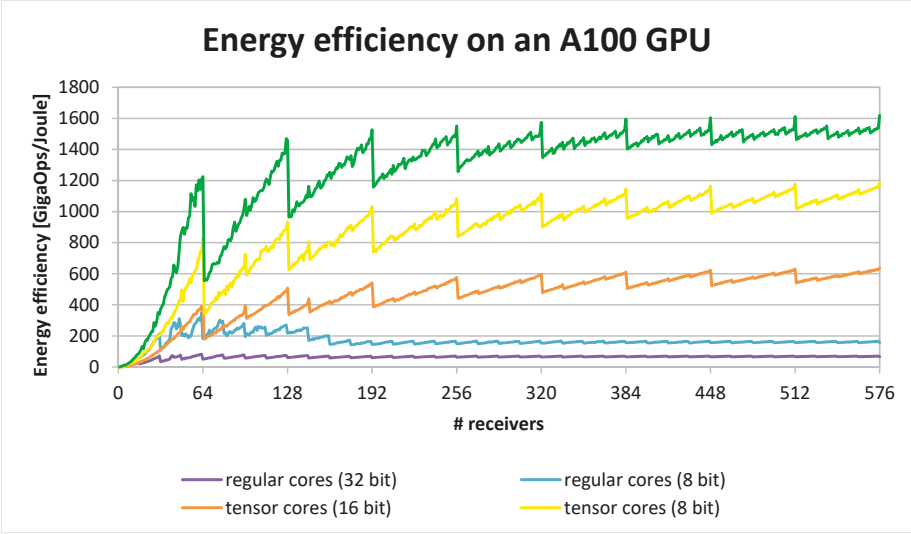


**Figure 4.6: Energy efficiency of the tensor-core correlator and a legacy GPU correlator on an A100 GPU, as function of the number of receivers, for various precisions of the input data**

Figure 4.6 shows the energy efficiency for the A100 GPU. Yet another leap in energy efficiency is made compared to the V100. On 8-bit and 4-bit input, the energy efficiency exceeds $10^{12}$ operations per Joule, a new milestone.

### *4.2.5 Porting experience*

Roughly 35% (about 12 PM) of the ASTRON effort was used to develop and optimize the tensor-core correlator. The library was developed from scratch in CUDA, so no additional effort was needed to port the software to the DEEP-EST DAM and ESB nodes. The performance-critical part is only 589 lines of CUDA code, but it is highly complex due to the use of low-level tensor-core intrinsics.

### *4.2.6 Scalability*

The correlator is parallelised over multiple nodes using independent processes, hence the correlator is trivially parallel. Older correlators (such as the LOFAR correlator) used to be MPI programs that performed (real-time) any-to-any transposes of input data across all correlator machines, but newer instruments (e.g., AARTFAAC) perform this transpose outside the correlator, on the way between the receivers and the correlator nodes on packet-switching Ethernet switches. This saves on network hardware costs and simplifies the correlator application.

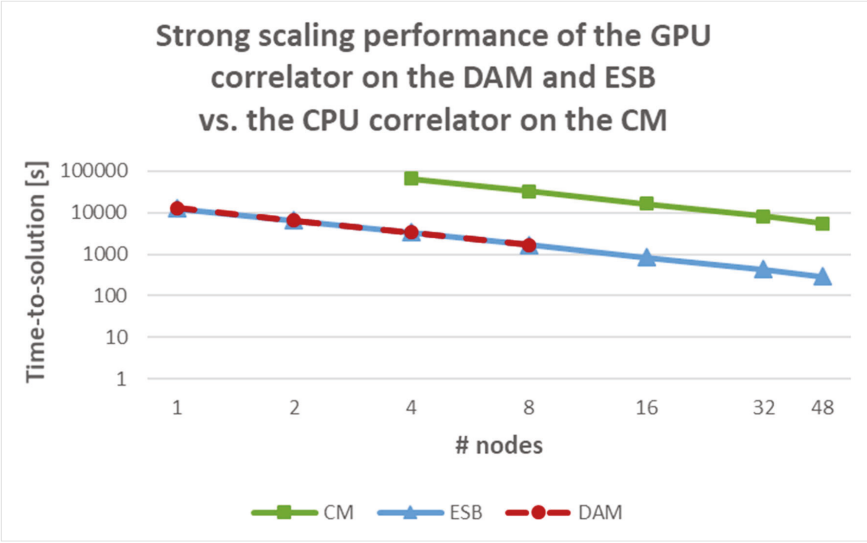


**Figure 4.7: Strong scaling results for the tensor-core correlator: runtimes for the GPU correlator on the DAM and ESB, as well as the legacy CPU correlator on the CM**

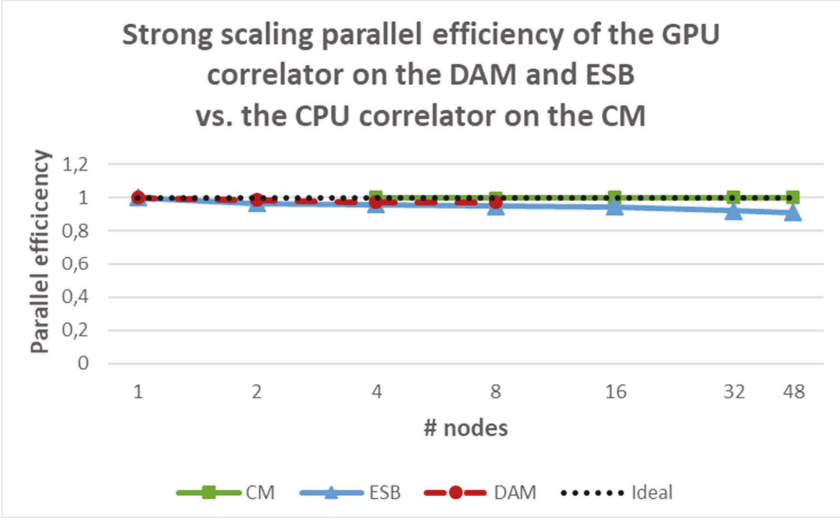


**Figure 4.8: Strong scaling results for the tensor-core correlator: parallel efficiency for the GPU correlator on the DAM and ESB, as well as the legacy CPU correlator on the CM**

Figure 4.7 shows scaling results from the tensor-core correlator on the DAM and ESB nodes. For comparison, we also included scaling results obtained when running on the CM a slightly modified version of the legacy CPU correlator that was developed for

Xeon and Xeon Phi processors in the predecessor DEEP-ER project. The legacy CPU correlator uses AVX512 intrinsics to achieve optimal CPU performance. For all performance measurements same input data (a large amount) is processed, hence the results are strong scaling results. The single and dual-node measurements of the CPU correlator are omitted, as their runtimes exceed the 20-hour time limit imposed by Slurm.

As both the tensor-core correlator and the CPU correlator processes run independently from each other, the performance scales almost perfectly, as displayed by the parallel efficiency plot in Figure 4.8. The performance on the DAM and ESB is nearly identical, because both modules use GPUs of the same type. ASTRON did notice some variation in GPU speed (up to 8% on the ESB and 3.4% on the DAM), probably caused by different thermal conditions, as all GPUs run as fast as they can within their 250W power limit; the obtained speed depends on the GPU temperature. The graph also shows that the tensor-core correlator processes the same amount of data 19 times faster than the CPU correlator. This atypically high factor is due to the use of tensor cores in the GPU correlator.

Note that in real telescope systems, the correlator is a real-time application that processes streaming data, and only needs to keep up with the incoming data streams. In practice, the correlator GPU hardware will be over-dimensioned so that it can process data faster than the data flows in, frequently stalling the GPUs as they wait for new data to arrive.

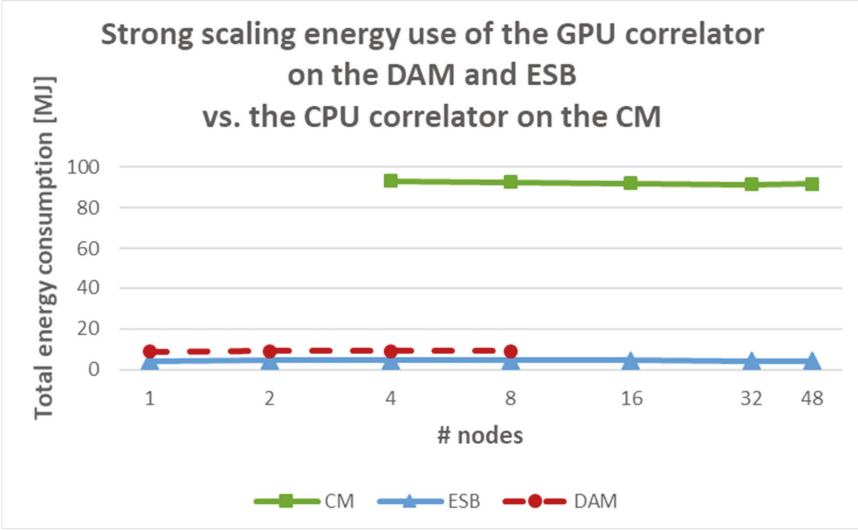


**Figure 4.9: Strong scaling results for the tensor-core correlator: energy use of the GPU correlator on the DAM and ESB, as well as the legacy CPU correlator on the CM**

Figure 4.9 shows the total energy used by the above-mentioned scaling benchmarks. The total energy consumption is independent of the number of nodes used. The DAM nodes are far less energy efficient than the ESB nodes, due to the presence of (unused) hardware (such as an FPGA, DCPMM DIMMs, powerful CPUs etc.) and of a less energy-efficient air-based cooling mechanism (the ESB is water cooled). The GPU correlator on the ESB is 22 times more energy efficient than the legacy CPU correlator on the CM. Again, this atypically high difference is due to the use of tensor cores.

#### *4.2.6.1 Our path to Exascale*

The use of tensor cores resulted in a giant leap in computational performance and energy efficiency, although an Exascale correlator would still need a MegaWatt for the computations alone. The remaining challenge is the part that we moved outside the correlator application and did not investigate within this project: the data exchange between the receivers and the correlator on packet-switching Ethernet switches. At least in theory, such switching systems can be built arbitrarily large by using multiple layers of switches.

The increasing correlator input data rates constitute another challenge. Current instruments typically use UDP/IP (unreliable datagram packets) to send data from the FPGAs near the receivers to the (central) correlator, possibly over dedicated Wide-Area Network links. As we move to the 100+ Gb/s era, the current practice to receive these packets through the kernel stack is no longer sustainable because the system call overhead becomes prohibitively high. ASTRON and its partners currently investigate how to use RoCE (RDMA over Converged Ethernet) to stream packets directly from remote FPGAs into a GPU correlator, without operating system involvement in the critical path. This is not trivial, because RoCE is a much more complex protocol than plain UDP, while the sending side of the protocol should be simple enough to be implemented on FPGAs.

### *4.2.7 Conclusion*

The use of tensor-core technology will have a disruptive impact on correlators (and beam formers) of (near-)future instruments, due to their order-of-magnitude increase in performance and significant energy efficiency compared to the use of regular GPU cores. DEEP-EST provided the means to explore this technology.

## 4.3 The GPU Imager

### *4.3.1 Application structure*

ASTRON also worked on a relatively new imaging application that creates sky images from calibrated correlations (visibilities) produced by the correlator. Unlike traditional methods that create the image almost fully in the Fourier domain, the new method performs the gridding step (explained below) in the image domain. This way, corrections for direction-dependent effects (e.g., caused by ionospheric disturbance) can be applied efficiently, which improves the image quality, especially when observing at low radio frequencies.

In the past years, the imager has become a mature application and is able to generate sky images for various radio telescopes worldwide. As shown during the project, the application runs much more efficiently on GPUs than on CPUs, because it performs many sine/cosine operations, which are expensive on CPUs and essentially free on (NVIDIA) GPUs.

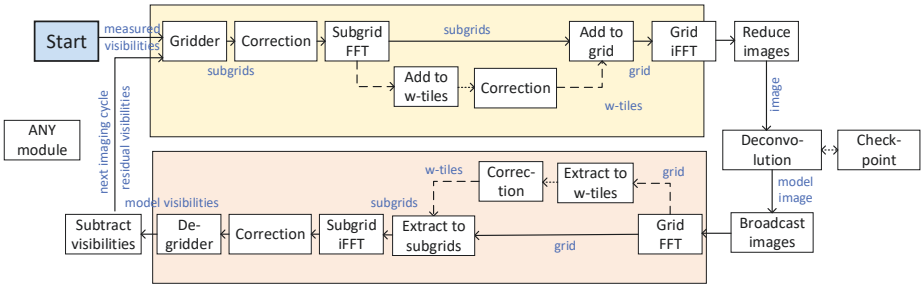


**Figure 4.10: The mockup imager**

Imaging is an iterative process, consisting of three main steps: 1) gridding, 2) deconvolution, and 3) degridding (see Figure 4.10). Gridding and degridding are the computationally most expensive steps. Hence, they are the focus of this study. In the gridding step, visibilities (measured correlations) are gridded onto a regular grid. The deconvolution step is used to detect sources in the image, and these are added to a model image. In the degridding step, visibilities are computed taking an image as input. By subtracting the model visibilities from the measured visibilities, faint sources become visible. Thus in each iteration the model of the sky is refined by adding increasingly fainter sources. This process is repeated until the sky model has converged.

The GPU imager comprises of two main components: IDG (Image-Domain Gridding) and WSClean. IDG provides gridding and degridding routines, and WSClean provides

deconvolution data handling (visibility input, image output). ASTRON created a mock-up imager around IDG gridding and degridding that emulates the full imager. The mock-up imager is an MPI application that distributes the input data (visibilities) over the nodes. Every node processes a block of data (a number of visibilities, say for an hour of observation) and creates a partial image. These images need to be combined (added together) to attain high signal-to-noise, such that deconvolution can detect faint sources in the image. The image combination step is implemented as a parallel reduction. The deconvolution step is not part of the mock-up imager. Every now and then (say every hour of processing), a checkpoint is made, see Section 4.3.4 for details. The next step is to distribute the model image to all compute nodes such that they can proceed with degridding.

### *4.3.2 Application mapping*

For two reasons, the imager should run on the DAM: the presence of accelerators and the large amount of available memory. As the gridder performs many sine/cosine operations, it should definitely run on GPUs or FPGAs.

The subgrids can be Fourier transformed efficiently on the GPU as well. Addition of the FFTed subgrids to the grid can be done either on the GPU or host CPU of the DAM; in our experience, it is somewhat more efficient to perform it on the GPU. The best place to perform the final inverse FFT of the grid is not yet determined: it seems to depend on the image size and on the efficiency of the FFT library for a particular architecture. The best place can be the GPU, FPGA, or CPU of the DAM, or even one of the other DEEP-EST modules.

Another reason to run the imager on the DAM is the presence of 3D XPoint DIMM modules, which may be used to save huge sky images that do not entirely fit in DRAM. In this case, the DRAM transparently caches the “hot” parts of the grid that are stored in the larger-capacity 3D XPoint DIMMs. On top of that, the even smaller GPU device memory will be used to transparently cache the “really hot” parts of the grid, using NVIDIA's Unified Memory technology. The imager is a particularly interesting application to demonstrate the usefulness of transparent caching (both 3D XPoint and Unified Memory) as the access pattern to the grid becomes complex (but with sufficient locality).

For low-resolution images, which require less memory, the GPU imager can also run on the ESB.

### *4.3.3 Porting experience*

The mock-up imager can run on the GPUs of the DAM or ESB, as well as on the CPUs of the CM. ASTRON expected the GPU imager to run on the DAM or ESB GPUs with

no or little additional effort, as the application was already tested on a number of other GPU-based systems, for small and medium-sized images. However, the imager initially performed sub-optimally on these small and medium-sized images, while on large images, the application simply hung. These problems were solved by making changes to the application and by using an updated cuFFT library.

Roughly 15% of the ASTRON effort (6 PMs) was spent on the GPU imager.

### *4.3.4 Checkpointing*

The input for an imaging cycle consists of two data products: measured visibilities (i.e., calibrated correlations) and model visibilities. The model visibilities are created by running degridding on the model image, after which these model visibilities are subtracted from the measured visibilities. In case of a crash, the measured visibilities can easily be recovered as these are typically read-only. To restore the model visibilities, the model image is needed. When the mock-up imager recovers from a crash, the model image is loaded from the checkpoint and distributed to the compute nodes, after which imaging proceeds as normal.

At first, ASTRON considered using the OpenCHK checkpointing library. This library is pragma-based and requires only a few additional lines of code. However, it requires a different compiler and compilation flow, which was found to be incompatible with IDG. It turned out to be easier to implement checkpointing using the FTI (Fault Tolerant Interface) library, which is internally used by OpenCHK as well.

### *4.3.5 High-resolution imaging*

There is a direct relation between the resolution of a sky image and the geographical distance between the two outermost receivers used in an observation. With the expansion of LOFAR stations all across Europe, there is a need to create increasingly higher-resolution images of tens of thousands of pixels high and wide, consuming hundreds of Gigabytes per image. Through the DEEP-EST project, ASTRON studied and handled the issues of creating such large images.

For wide-field imaging, baselines (receiver pairs) cannot be assumed to be in one plane, due to the curvature of the earth. Similarly, for wide fields of view, the sky cannot be approximated as a flat plane. Together, these effects must be corrected for and require large convolution kernels (W-kernels) during gridding and degridding. Gridding the longest baselines becomes computationally very expensive then. A technique called W-stacking alleviates this by effectively gridding onto multiple grids (one for each so-called W-layer). Short baselines are gridded onto the lowest W-layer, while the

longest baselines are gridded on the highest W-layer. This technique, however, requires a lot of memory, as the image that is being constructed consists of many layers.

The original plan was to explore the use of DCPMM memory in the DAM nodes to store the W-layers. However, recently, we started realizing that an algorithmic change would significantly reduce the memory footprint to create an image. This technique, called *W-tiling*, requires the use of only one grid (one W-layer), and a cache of so-called W-tiles. A W-tile represents a small part of a W-layer, and an algorithm maps the subgrids to W-tiles. Some additional computations are needed to 'project' a W-tile onto the one W-layer, but for large images, W-tiling strongly relaxes the need for impractically large amounts of memory, and hence we consider it a good trade-off. Therefore, there is no need to use DCPMM anymore.

The first implementation of W-tiling is a hybrid (CPU + GPU) imaging mode where gridding takes place on the GPU and W-tiling takes place on the CPU. In the second implementation of W-tiling, most of the W-tiling operations are performed on the GPU. As with gridding, where a phase shift is applied to place a visibility on a subgrid, a phase shift is applied to place a subgrid onto a W-tile and to project a W-tile to the W-layer (the "w=0 plane"). Since these phase shifts involve many sine/cosine computations, W-tiling runs much faster on the GPU compared to running it on the CPU.

Figure 4.11 shows the runtime for both W-tiling implementations. The runtime comprises two parts: the dark bar represents the time spent in gridding, the lighter part the time spent in flushing the W-tiles (constructing the image). For v1, the runtime becomes prohibitive for grids larger than about 28,000×28,000 pixels. The latest implementation (v2) scales to much larger images.

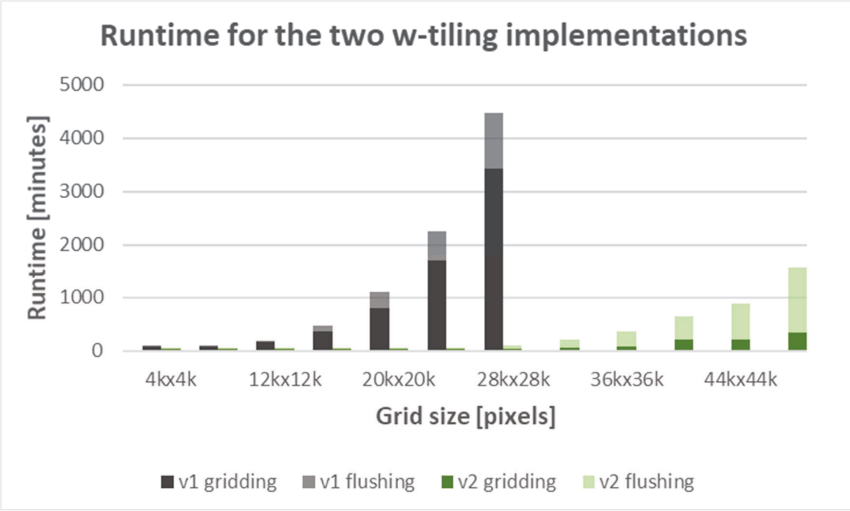


**Figure 4.11: Comparison of runtime for the two w-tiling implementations**

The method has been demonstrated to work for much larger grids (up to more than 100,000×100,000 pixels) as well, but in these cases the throughput becomes bound by the limited bandwidth of PCIe: the GPU spends a few hundred milliseconds in gridding (at over 10 TFLOP/s), after which several seconds are spent copying W-Tiles to the host (either explicitly, or by Unified Memory page migrations).

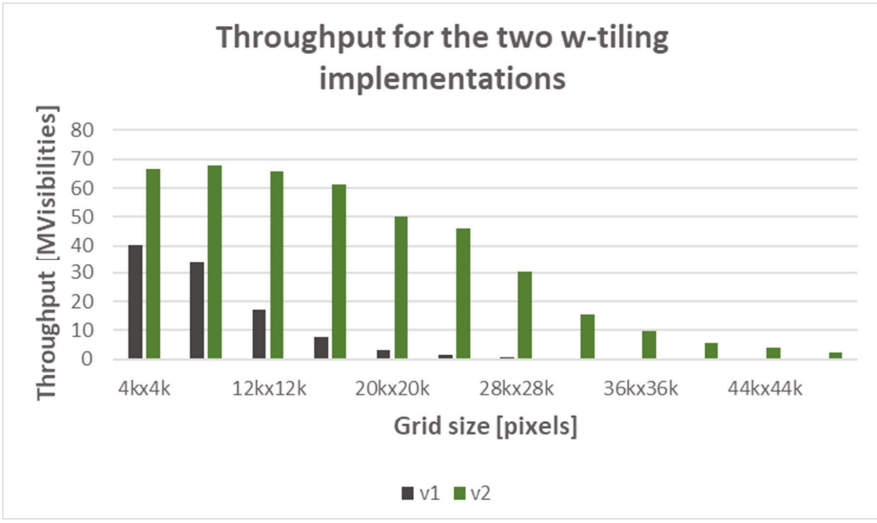


**Figure 4.12: Comparison of throughput for the two w-tiling implementations**

Figure 4.12 shows the throughput achieved by the two W-Tiling implementations. As explained above, the runtime for the CPU-only implementation (v1) becomes prohibitive for grid sizes larger than 28,000×28,000 and we therefore omit throughput results for larger grid sizes. For these cases, throughput is well below 1 million visibilities/s. The achieved throughput for the GPU- accelerated implementation (v2) also goes down as the grid size increases, but it attains much higher overall throughput.

Imaging is a lot more challenging for large images, which is reflected in the reduced throughput. Still and most importantly, DEEP-EST enabled ASTRON to speed up IDG such that large images (32,000×32,000 pixels) are now feasible.

### *4.3.6 Scalability*

ASTRON evaluated the scalability of the mock-up imager on the ESB (with GPUs, see Figure 4.13) and on the CM (CPU-only, see Figure 4.14). To this end, a simulated dataset for a 12-hour observation was used, based on all of the proposed SKA1 Low station coordinates.

On the CM, the runtime of the imager is dominated by the gridding and degridding times. The overall runtime is inversely proportional to the number of nodes used. The time spent in communication (the grid-reduce and grid-broadcast) is negligible.

As explained before, IDG runs much more efficiently on GPUs than on CPUs, which is also reflected in about a 20-fold reduction in runtime. GPUs are much faster because they compute sine/cosine operations very efficiently *in hardware*, while on CPUs, we have to resort to library functions to compute sine/cosine *in software*. Even though these libraries (e.g., Intel MKL) are highly optimized, the CPU spends 80% of the time computing sines and cosines. On GPUs, where gridding and degridding run so fast, the time spent in communication becomes noticeable, especially when more than 8 nodes are used.

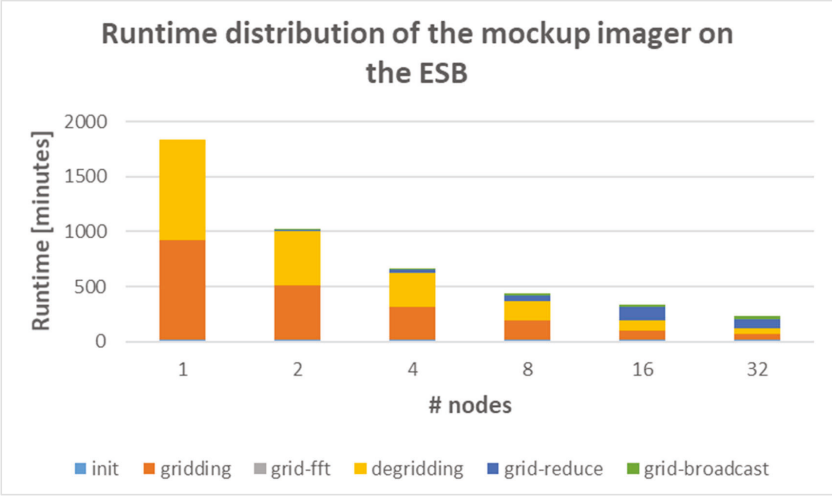


**Figure 4.13: Runtime distribution of the mockup imager on the ESB**

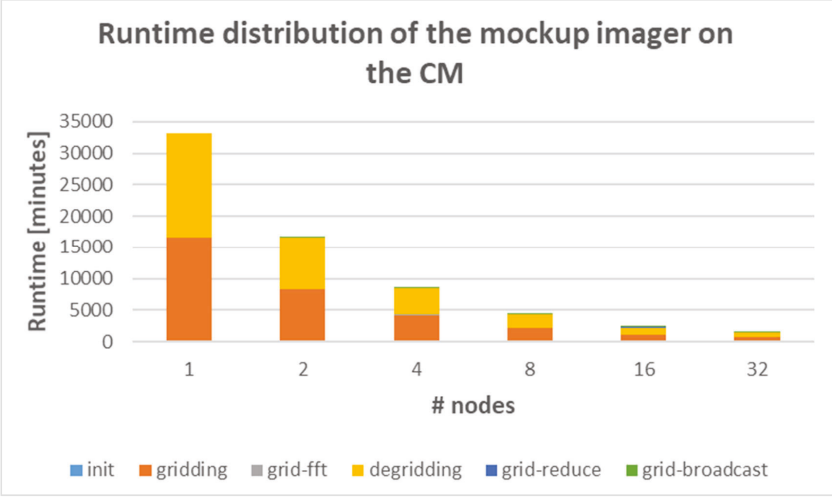


**Figure 4.14: Runtime distribution of the mockup imager on the CM**

#### *4.3.6.1 Our path to Exascale*

In its basic form, the imager is trivially parallel, as different frequency bands are processed independently by independent processes. Hence, the mock-up imager will scale perfectly. Even in the case that the work for one frequency band is distributed over multiple nodes, good scalability is achieved, as shown in Section 4.3.6.

When scaling to thousands of nodes, there is no need to run the imager as a single MPI application across all nodes: visibilities from different frequency bands can be imaged independently by multiple (groups of) imaging processes. The frequency-dependent images still need to be merged to a final image, but this is not in the critical path.

That is not to say that the whole imaging processing pipeline, which contains several other applications, is ready for Exascale yet: the imager itself, which used to be the slowest processing step (as it is the computationally most expensive step) has become so fast now that it reveals several new bottlenecks. Due to the amount of work involved, it was not possible to remove all new bottlenecks within the scope of the DEEP-EST project.

Creating very large sky images (e.g. 100,000×100,000 pixels in size) remains challenging. Unlike on POWER8/NVLink-based systems, a PCIe-based connection to the CPU provides too little bandwidth to keep the GPU busy. The new W-tiling implementation alleviates, but does not resolve this bottleneck.

Faster links between GPUs and (host) memory (e.g., CXL, NVLink) will increase the imaging throughput. Alternatively, using many small GPUs may be better than using fewer big GPUs, to take advantage of the larger total PCIe bandwidth. Moreover, small GPUs typically consume less power.

DEEP-EST provided the means to create sky images at much higher resolutions than before, which are required by future Exascale instruments like the SKA, but also current ones such as LOFAR, which is now also capable of creating sky images of 30,000×30,000 pixels in size.

### 4.3.7 Performance and energy comparison

A comparison of the runtime and energy consumption for the mock-up imager on CM and ESB is shown in Figure 4.15 and Figure 4.16, respectively. Unsurprisingly, the ESB is much more (energy) efficient.

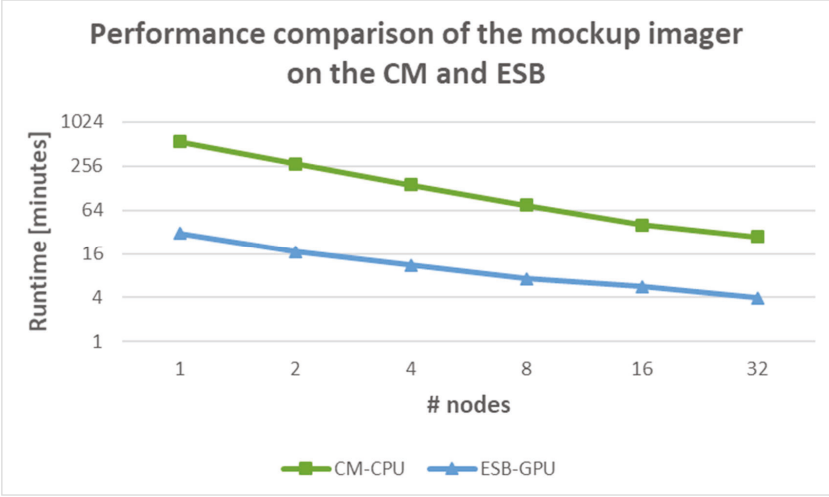


**Figure 4.15: Performance comparison of the mockup imager on the CM and ESB**

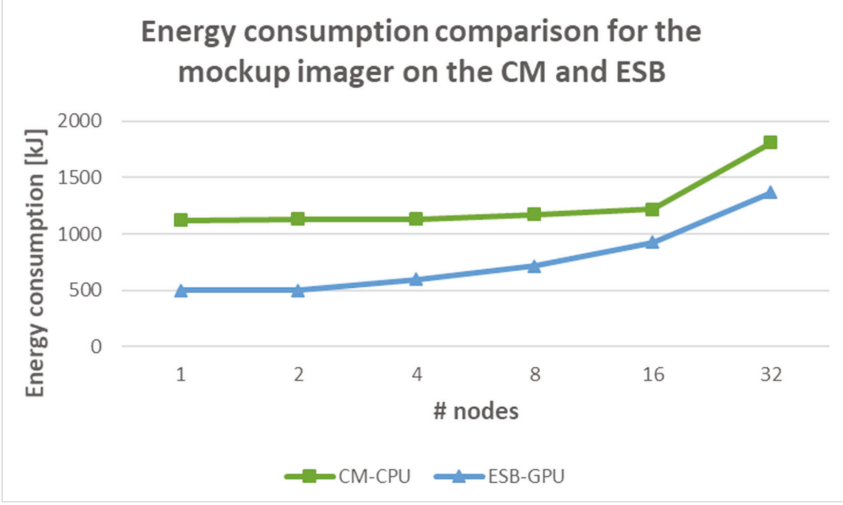


**Figure 4.16: Energy consumption comparison for the mockup imager on the CM and ESB**

### 4.3.8 Conclusion

IDG runs highly efficient on GPUs thanks to their native support for sine/cosine operations. Now that the GPU kernels (such as the gridder and degridder) run so fast, every operation in the imager that does not run on the GPU tends to become a bottleneck. ASTRON explored a new technique, called W-tiling, that significantly reduces the amount of memory used to create (very) large sky images, at the expense of a minor increase in computations. By implementing W-tiling on the GPU, image creation of up to ~30,000×30,000 pixels in size has become practical and fast, so that the painstaking effort of stitching hundreds of facets together belongs to the past. The biggest bottleneck remaining is the fairly limited bandwidth between the host DRAM and GPU memory. Similarly, the mock-up imager becomes bound by the network bandwidth between the compute nodes, especially when IDG is used with GPU acceleration. Future algorithmic improvements on deconvolution may help to reduce the amount of communication between nodes, and therefore help to improve scalability. All in all, DEEP-EST enabled us to improve the overall performance of the imager and brings us a big step closer to Exascale imaging.

## 4.4 The FPGA Imager

### 4.4.1 Application Overview

A mock-up imager able to run of FPGAs and performing only the most compute-intensive tasks of the full imaging pipeline (described in more detail in Section 4.3.1) has also been developed. Figure 4.17 shows these tasks: gridding visibilities onto 32×32 subgrids, a correction for direction-dependent and other effects, and a 2D FFT over each subgrid. The optional addition to W-tiles and associated corrections were not implemented on the FPGA, as these algorithmic improvements were explored in the GPU imager during the final months of the DEEP-EST project and there was no time left to implement and optimize them on the FPGA. The FPGA does not have sufficient on-board DDR4 memory to store the full (Fourier-transformed) image, thus the next two steps, accumulation into the full grid and the final inverse FFT over the full grid, cannot be performed on the FPGA. Instead, the Fourier-transformed subgrids are transferred back to the CPU for further processing.

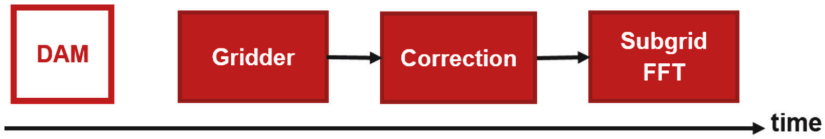


**Figure 4.17: Schematic workflow of the FPGA Imager in the MSA**

### *4.4.2 Porting experience*

The Intel OpenCL/FPGA toolkit was used to implement the FPGA imager, before oneAPI became available. Originally, the imager was developed for the (mid-range) Arria 10 FPGA, as (high-end) Stratix 10 boards were not available at the start of this project. The performance obtained on the Arria 10 was fair: much better than CPUs, but not as good as GPUs. A paper comparing CPU, GPU, and FPGA imaging with respect to architectures, programming models, optimizations, performance, energy efficiency, and programming effort received the Euro-Par'19 best paper award[74].

As soon as the Stratix 10 FPGA boards were installed in the DAM nodes, ASTRON started porting the Arria 10 imager to the Stratix 10. The Stratix 10 is not only larger and theoretically able to run at higher clock speeds, but it is also characterized by an on-chip interconnect that is architecturally different from the interconnect in an Arria 10. As a result, Intel advises to not use blocking channels (FIFOs, an OpenCL extension), because this reduces the maximum clock speed at which an FPGA application runs on a Stratix 10. And indeed, the original imager was initially very slow, both because of excessive resource usage and because of a low clock at which the application ran.

ASTRON already foresaw that the port to the DAM FPGA would be a major effort. Essentially, the OpenCL code was fully re-implemented. The original code consists of hundreds of small (partly replicated) OpenCL kernels connected by blocking channels (FIFOs); the new imager consists of one single, highly complex OpenCL kernel that performs all operations and essentially fills the whole FPGA. Both the single-kernel and many-kernel imagers were optimized further, in order to find out which one would be the better approach. Many Stratix 10 specific optimizations were necessary, such as avoiding the use of double-pumped memory (i.e., memory that transfers two words per cycle per port), limiting the use of memory to only one read location and one write location. In many cases this made the code more complex. Another important Stratix 10 specific optimization is to write code in such a way that it allows the compiler to enable hyper-optimized handshaking; this increases the clock speed at which the design will run. Furthermore, many optimizations were implemented that reduced the resource usage other than multipliers (one does want to use as many multipliers as possible: the more multipliers are used, the more simultaneous computations can be performed). Reducing resources makes it easier to fit the design onto the FPGA.

[74] Bram Veenboer and John W. Romein. Radio-Astronomical Imaging: FPGAs vs GPUs. Euro-Par'19, Göttingen, Germany, August 2019

The sine/cosine operations are implemented using a lookup table. A lookup table is less accurate than the compiler-built-in sine and cosine operations, but sufficiently accurate for this application. The lookup tables do not require the use of multipliers, which can be used to perform more gridding (and other) operations instead. However, the table uses a large amount of internal memory (16% of the total available memory blocks), because the compiler (automatically) replicates the table 320 times to provide enough memory bandwidth for 320 simultaneous sine and cosine operations per cycle. The table is compressed, meaning that for each table entry (a pair of single-precision floating point numbers), 13 of the 64 bits are always the same and are not stored in memory.

Although both FPGA imagers were eventually successfully implemented and optimized, there were many workarounds for compiler issues necessary. Some issues were obvious compiler bugs (e.g., a crash when compiling legal program code), but most cases were on code constructs that were not well handled or optimized by the compiler (e.g., a sharp increase in memory use when shrinking the width of a table from 60 to 51 bits). Yet in other cases, the compiler could not be blamed; some constructs cannot be handled efficiently in any way by an FPGA. Finding solutions for all issues turned out to be a very time-consuming effort, especially when it requires the synthesis of a full FPGA design, which can take a full day, even on a fast computer. Several compiler bugs were reported to Intel and were fixed or will be fixed in subsequent compiler releases. Over the years, the tools improved significantly, but it takes a long time for them to reach full matureness.

Roughly half of ASTRON's effort (about 18 PM) in DEEP-EST was spent on implementing the FPGA imager. The many-kernel imager consists of 835 lines of performance-critical OpenCL code, the single-kernel imager about 560 lines, not counting the generated sine/cosine lookup tables. Thousands of small modifications to these programs were made to minimize resource usage, identify and reduce idle times, optimize clock frequency, and work around compiler issues.

As we implemented and optimized Image-Domain Gridding for CPUs, FPGA, and GPUs, we found differences and similarities with respect to architecture, programming model, necessary optimizations, performance, energy efficiency, and implementation effort. We discuss them in the sections below.

### 4.4.3 Performance

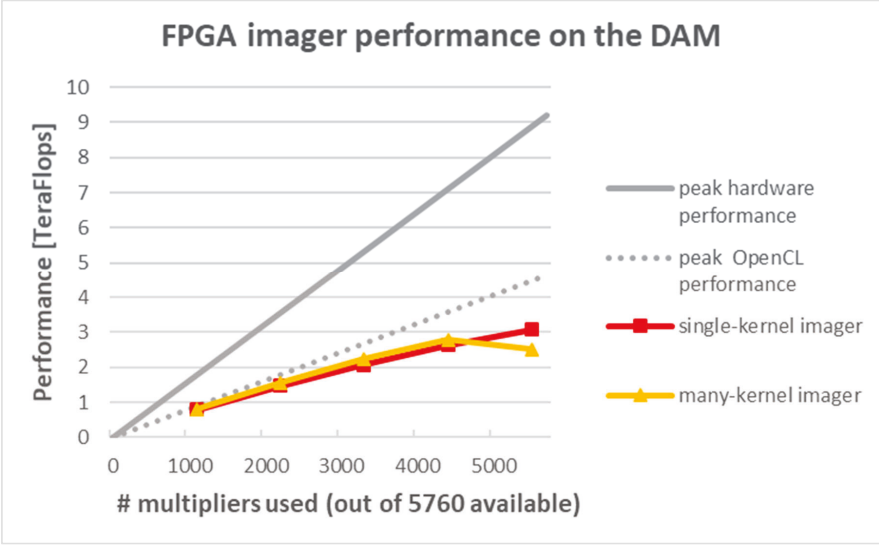


**Figure 4.18: Performance of the FPGA imager on the Stratix 10**

Figure 4.18 shows the performance of the FPGA imager on the Stratix 10 FPGA board in a DAM node, as a function of the number of multipliers used. The FPGA hardware has a peak clock frequency of 800 MHz, which translates to the “peak hardware performance” line in the figure. However, the OpenCL board support package does not run at more than 401 MHz, limiting any OpenCL program to at most 50% of the theoretical performance, as reflected by the “peak OpenCL performance” line in the figure. Two other curves show the performance of the multi-kernel imager and the single-kernel imager, respectively. The performance is reported in terms of floating-point operations per second, where only all multiplications, additions, and subtractions are counted, not the sine/cosine lookups. Each measured value in the figure is the result of a series of compilations over a large number (50–200) of random seeds, and the performance of the design that runs at the highest clock speed is reported in the figure.

Figure 4.18 tells that the performance is a factor of three from the advertised hardware peak performance. Eventually, the single-kernel imager is slightly faster than the many-kernel imager. On a full design where almost all multipliers are used, the logic use of the many-kernel imager is so high that the design is difficult to place and route; if the placement succeeds at all, the frequency at which the design runs is no more than 236 MHz. The single-kernel imager runs at 289 MHz for a full design.

### 4.4.4 Energy consumption

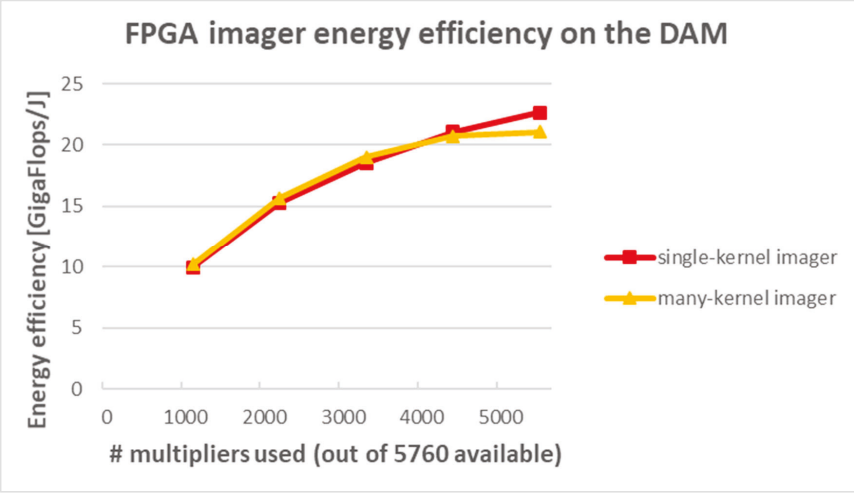


**Figure 4.19: Energy efficiency of the FPGA imager on the Stratix 10 FPGA**

The Stratix 10 FPGA board can measure its own power use. Figure 4.19 shows the energy efficiency for the multi-kernel and single-kernel imagers. Unsurprisingly, a full design is more energy efficient than a partially-used FPGA. The maximum energy efficiency is 22.7 GigaFlops/Joule. The section below discusses how this compares to other devices.

### 4.4.5 A comparison between CPU, GPU, and FPGA imaging

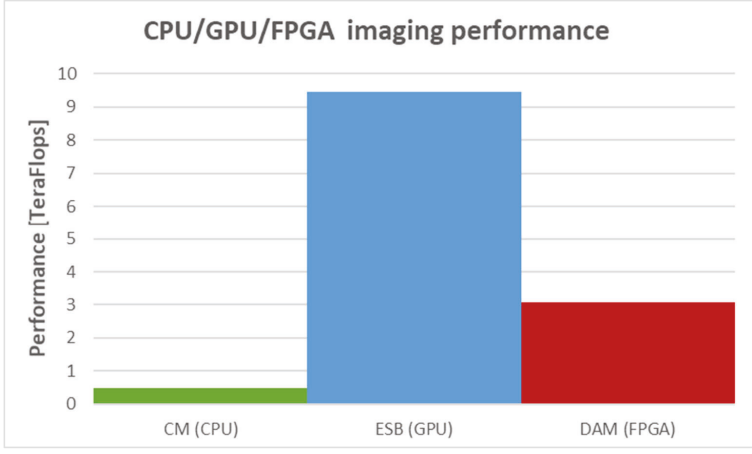


**Figure 4.20: A comparison of imaging performance on a CPU in a CM node, a GPU in an ESB node, and an FPGA in a DAM node**

As ASTRON has optimized CPU, GPU, and FPGA implementations for the computationally most intensive parts of the full imaging application, the performance and energy efficiency of these processors could be compared. Figure 4.20 shows the performance in TeraFlops (so higher is better) counting only the multiplications, additions, and subtractions, not the sine and cosine computations or lookups. Section 4.3 already showed a huge performance advantage for GPUs over CPUs for sky-image creation. Figure 4.20 shows that the FPGA is somewhere in between. The FPGA computes much faster than the CPU, mostly because the FPGA performs the sine/cosine lookups much more efficiently than the CPU performs these operations. ASTRON also tried the lookup-table approach on a CPU, but this did not improve performance. The FPGA does not perform as well as the GPU. This was expected, as the FPGA has a lower peak performance (9.2 vs. 14 TFLOPS), but the GPU imager runs at 68% of the GPU peak performance, while the FPGA imager runs at 33% of the FPGA peak performance. For the FPGA, the maximum clock rate at which the imager runs, is too low to compete with GPUs.

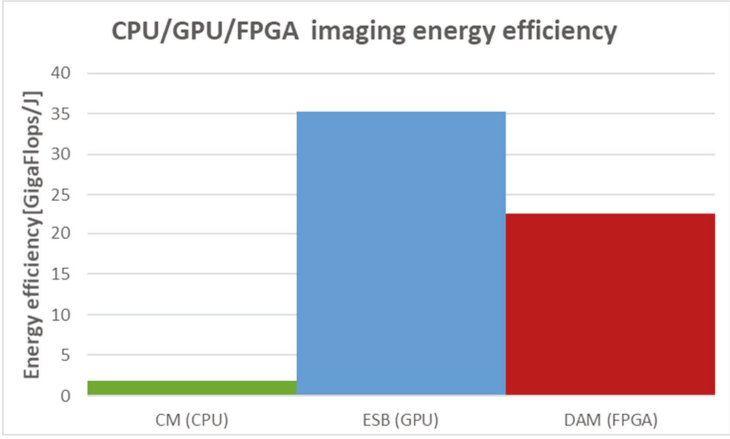


**Figure 4.21: A comparison of imaging energy efficiency on a CPU in a CM node, a GPU in an ESB node, and an FPGA in a DAM node**

Another (and arguably fairer) way to compare these processors is to analyze their energy efficiency. Figure 4.21 shows the energy efficiency in terms of visibilities per second (GigaFlops/Joule, higher is better). For a fair comparison, only the GPU or FPGA device power is measured, not the total system power, as the DAM contains many unused components that draw power and are not used for this comparison. For the CPU measurements, the processor and DRAM power are measured. The figure shows that the energy efficiency of the FPGA is an order of magnitude higher than the

energy efficiency of the CPU, but still somewhat lower than the energy efficiency of the GPU.

### *4.4.6 FPGA vs GPU: Lessons learned*

The GPU imager was implemented in both CUDA and OpenCL and the FPGA imager only in OpenCL. Even though the GPU and FPGA imagers share a common programming language, hardly any code reuse was possible. This is mostly due to the different programming models: with FPGAs, one builds a dataflow pipeline, while GPU code executes instructions. On the FPGA, the programmer has to think about how to divide the resources (multipliers, memory blocks, logic, etc.) over the pipeline components, so that every cycle all multipliers perform a useful computation. At the same time, the use of other resources (such as logic and memory blocks) should be reduced, while bottlenecks, underutilization, stalls, and constructs that lead to a low clock speed should be avoided. For the imager, this pipeline is complex, and consists of many subpipelines. The speed at which data flows through each of the subpipelines had to be managed carefully. Data should not flow too slowly through any of the subpipelines, or that subpipeline becomes an overall bottleneck. Nor is there any point in making data flow too quickly, because that wastes resources.

There are more differences. On an FPGA, the programmer constructs local memory in some optimal way, while local memory on a GPU has a fixed, banked configuration that the programmer uses. On FPGAs, non-performance-critical operations, such as initialization routines, can consume many resources, while on GPUs, performance-insensitive operations are not an issue. On FPGAs, it is also much more important to think about timing (e.g., to avoid pipeline stalls), but being forced to think about it leads to high efficiency: 96.3% of all multipliers/adders perform a useful operation 96% of the time.

FPGAs have typically less memory bandwidth than GPUs, but we found that with the FPGA dataflow model, where all kernels are concurrently active, it is less tempting to store intermediate results off-chip than with GPUs, where kernels are executed one after another. In fact, our FPGA designs use DDR4 memory only for input and output data; we would not have used the DDR4 memory at all if the OpenCL Board-Support Package would have implemented the PCIe I/O channel extension. In contrast, the cuFFT GPU library even requires data to be in off-chip memory.

Both FPGAs and GPUs obtain parallelism through kernel replication and vectorization; FPGAs also by pipelining and loop unrolling. This is another reason why FPGA and GPU programs look different.

Surprisingly, many optimizations for FPGAs and GPUs are similar, at least at a high level. Maximizing FPU utilization, data reuse through caching, memory coalescing, memory latency hiding, and FPU latency hiding are necessary optimizations on both architectures. For example, an optimization that we implemented to reduce local memory bandwidth usage on the FPGA also turned out to improve performance on the GPU, but somehow, we did not think about this GPU optimization before we implemented the FPGA variant. However, optimizations such as latency hiding are much more explicit in FPGA code than in GPU code, as the GPU model implicitly hides latencies by having many simultaneously instructions in flight. On top of that, architecture-specific optimizations are possible (e.g., the sine/cosine lookup table).

Overall, we found it more difficult to implement and optimize for an FPGA than for a GPU, mostly because it is difficult to efficiently distribute the FPGA resources over the kernels in a complex dataflow pipeline. Even so, we consider the availability of a high-level programming language and hard FPUs on FPGAs an enormous step forward. The OpenCL FPGA tools have considerably improved during the past few years, but have not yet reached the maturity level of the GPU tools, which is quite natural, as the GPU tools have had much more time to mature.

### *4.4.7 Conclusion*

Despite the amount of programming effort that is put into the FPGA imager, the performance is still not as good as we hoped for, but the energy efficiency is fair. The attempt to avoid blocking memory channels, which are known to limit performance on the Stratix 10, resulted in a new program code in which the whole application is implemented as a single, complex OpenCL kernel that fills the entire FPGA. Eventually, the single-kernel imager performs slightly better, but it took a lot of time to implement and optimize the program code. Successive compiler releases managed to recognize some of the complex structures better and better, but it remains difficult for a compiler to efficiently translate such a highly complex kernel.

The FPGA imager is much more (energy) efficient than the CPU imager, mostly because of the CPU's poor support for vectorised sine/cosine computations. However, there is not a single aspect (performance, energy efficiency, programming effort, flexibility, compilation time) where the FPGA surpasses GPUs. Typical FPGA advantages over GPUs, such as strict real-time behaviour, low latency, integrated 100 Gigabit/s Ethernet interfaces, and the ability to interface with other electronics like Analog-to-Digital converters, are not advantages from which the imaging application could profit.

On the other hand, the experience that was obtained with the OpenCL/FPGA toolkit has been very useful. ASTRON now uses this experience for other applications where

FPGAs are indispensable. Eventually, the use of a high-level programming language will significantly reduce programming effort compared to traditional hardware description languages like VHDL.

# 5 Space weather with DLMOS, xPic and GMM

**Jorge Amaya**
Katholieke Universiteit Leuven, KU Leuven, Belgium
jorge.amaya@kuleuven.be

## 5.1 Introduction

The applications implemented by KU Leuven during the DEEP-EST project are used to study the Space Weather system connecting the Sun to the magnetosphere of our planet. There are three main applications in the system:

- **DLMOS** (Deep Learning Modelling of the Solar wind): used to forecast the solar wind conditions in front of the Earth from images of the Sun.
- **xPic** (extended Particle-in-cell): a first-principles plasma physics code used to study the plasma environment in the solar wind and the magnetosphere.
- **GMM** (Gaussian Mixture Model): a machine learning algorithm that analyses velocity distributions functions extracted from particle information generated in the xPic code.

## 5.2 Application structure

### *5.2.1 DLMOS*

The DLMOS model is coded in Python. Multiple frameworks are available to code Machine Learning (ML) models. The efforts will concentrate on using PyTorch. As with another ML algorithm, DLMOS needs to be deployed in two modes: training and scoring (also called inference). The first mode takes large amounts of input data and trains the model. The second mode uses the trained model to predict a singular input sequence. While scoring a Deep Learning (DL) model can be generally performed quickly in regular CPU architectures, training the model requires important resources in disk access, memory size and computing power.

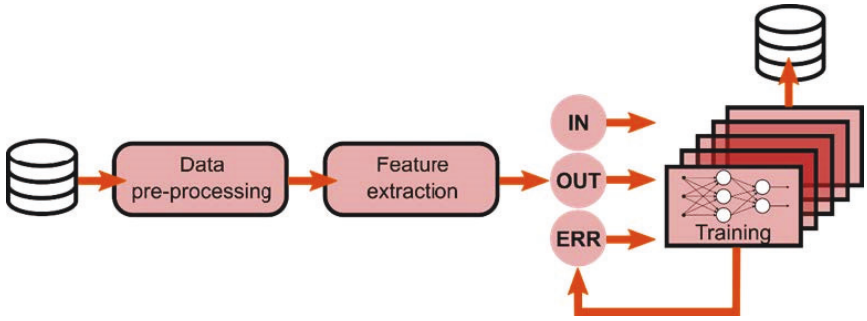


**Figure 5.1: Training mode of the DLMOS model**

Figure 5.1 shows the general structure of a ML algorithm. It consists of two main parts: 1) a data pre-processing pipeline, and 2) a Neural Network (NN). For high performance of the ML model, the input data of the NN has to be as clean and homogeneous as possible. So during the development of a ML project, most of the time is spent testing different methods to clean the raw input data. The pre-processing pipeline is the final result of this process.

### *5.2.2 xPic*

This Particle-in-Cell code has been partitioned into two solvers:

- Field solver: a numerical algorithm that solves Maxwell's equations for electromagnetism in a 3D Cartesian grid. The solver uses an iterative Krylov subspace method to solve a large system of linear equations. This method requires the computation of a residual every Krylov iteration. Global communications are required to keep track of such residual, and neighbor communications are required to compute the derivatives of the source terms of the linear system. Parallelism is obtained by domain decomposition.
- Particle solver: uses Newton's equations of motion to compute the movement of billions of charged particles, which integrate the system. Collisions and other particle-particle interactions are not computed (their interaction is mediated through the electromagnetic fields). All particles are independent of each other. The particle solver is also parallelised by domain decomposition, so communications are required to move particles from one domain to the neighbouring domain when they cross boundaries. In addition to the movement of particles, the particle solver also performs the calculation of particle statistics called moment gathering. In this last step particle properties are projected on the 3D grid of the code and transmitted to the field solver.

The field solver and the particle solver are inter-dependent and require constant exchange of information. However, they show different numerical strategies that can be mapped to different hardware architectures.

### *5.2.3 GMM*

The GMM analysis runs “on the fly”. The execution frequency depends on the particular simulation tested, but it is not performed at every xPic iteration. In a normal simulation, field and particle I/O is performed every few hundred iterations. GMM analysis will be executed at every few I/O calls (once every few thousand iterations). The Space Weather application allows to test the execution of consecutive jobs in the DEEP-EST system, the execution of concurrent jobs in different modules, and the new deferred job launch from SLURM.

## 5.3 Application mapping

### *5.3.1 DLMOS*

The DLMOS application from KU Leuven is characterized by a heavy and continuous movement of data from hard disk to processor memory. Therefore, it requires good data movement management between the different levels of computer memory.

This application also uses large amounts of data to train a deep neural network. This process is based on the constant use of tensor operations (matrix and vector multiplications) that can benefit from relatively weak computing units with large number of threads and vectorisation. These operations are also relatively fast, so the memory bandwidth needs to be high. A more quantitative description of the requirements is not yet available.

These are the reasons why the DLMOS application would benefit from the large number of cores and higher memory bandwidth proposed for the DAM.

#### *5.3.1.1 DLMOS-DPP*

The data pre-processing procedure is not based on highly parallel and multi-threaded, vectorisable, tensor operations. It requires high performance per core and high memory capacity. It also requires the full I/O infrastructure to move the data from disk to processor. Data pre-processing can be implemented on accelerator cards and can also take advantage of the host processors of accelerator nodes (like the DAM nodes).

It would be possible to deploy the DLMOS-DPP sub-package to the CM where both memory and sequential performance is higher. The final decision on the correct mapping of the DLMOS-DPP (DAM or CM) will need to be done on the prototype modules.

As it is today, it is more convenient to maintain the multiple elements of the application as close as possible in hardware, using the same module for the DLMOS-DPP and the DLMOS-Training sub-packages.

DLMOS-DPP will take advantage of the different levels of data storage of the DEEP-EST system, moving data from the Internet to the SSSM, local disk, memory, and the processor. This package will generate new enhanced data that need to move in the opposite direction, up to the SSSM.

DLMOS-DPP can run continuously, processing new data, for multiple applications of multiple data-sets. It does not depend on the results of any other components of the DLMOS package. Parallelism can be achieved by processing multiple inputs at the same time. No data communications are required in this package.

The pre-processing also involves downloading of data to a local storage space, detecting anomalies, normalizing the data, and selecting and building the training data-sets for the DLMOS-Training sub-package.

#### *5.3.1.2 DLMOS-Training*

Training is based on highly parallel multi-threaded vectorisable tensor operations that can be deployed on accelerator cards. These operations also require a constant stream of data, thus taking advantage of high bandwidth memory.

For these reasons the DLMOS-Training sub-package will be mapped to the DAM (although it could be potentially translated to the GPUs in the ESB).

This python code takes the processed data from the DLMOS-DPP, stored in the SSSM, and performs the training process of the DLMOS Neural Network Models (DLMOS-NNMs). The architecture of the DLMOS-NNMs is not yet defined and will be tested during this project.

Parallelism of the DLMOS-Training will be achieved in three different ways:

- Model parallelism: Different accelerators will train different NNMs for the same training data, selecting and cross-breeding the best performing models and achieving good convergence to a satisfactory result.
- Data parallelism: A single data-set can be divided in multiple smaller data-sets that can be used to train a model in independent accelerators.
- Graph parallelism: A single ML model is divided in multiple sub-tasks that can be mapped to different accelerators, requiring continuous data exchanges.

The DLMOS-Training sub-package implements python algorithms for methods 1 and 2, but method 3 is implemented in an ML framework (e.g. TensorFlow, Keras, PyTorch). For methods 1 and 2, network latency and bandwidth are not a constraint. The amount of data transferred between nodes is very low in all methods. While

method 1 does not require frequent MPI communication, method 2 can demand frequent ALL_REDUCE operations.

Method 3 is very restrictive and requires constant data movement of small amounts of data. Parallel efficiency is achieved by the framework's particular communication algorithm, which is not based on MPI. TensorFlow has shown in recent tests low parallel efficiency using multiple GPU node systems. We studied the use of other frameworks such as PyTorch and MXNet as potential replacements of TensorFlow and chose a combination of PyTorch and mpi4py.

#### *5.3.1.3 DLMOS-Inference*

This procedure is computationally similar to the DLMOS-Training sub-package but requires only one input (instead of hundreds of thousands) and produces only one output. The full procedure is extremely fast and does not require special hardware components. The sub-package is mapped to the ESB. The code takes inputs from two different sources: the SSSM, where recent input data and NNMs have been stored by the DLMOS-DPP.

The DLMOS-Inference code is mapped to the ESB, because it is also the current module used for I/O in the xPic code. The inference process is very fast and requires minimum resources, so it is not necessary to use the DAM for this procedure.

DLMOS-Inference will generate the output that will be stored in local disk and used by the xPic initialization tool to create the initial and boundary conditions of the code.

### *5.3.2 xPic*

The code xPic will be run in Cluster-Booster configuration, i.e. using the CM and the ESB. All I/O is performed from the ESB.

#### *5.3.2.1 xPic initialisation*

Data from the DLMOS-Inference is read and interpreted by a python script that belongs to the xPic code. The script creates the initialization files for the code. It writes one field file (up to 1 GB of data for the largest cases) and, if it is possible, it creates a particle file (up to half a TB for the largest cases). The files are written on the SSSM and later read in parallel from all the allocated ESB nodes at the beginning of the xPic run.

#### *5.3.2.2 xPic particle solver*

The particle solver of xPic performs very fast calculations on a very large number of independent particles. The data of each particle is stored in aligned vectors in memory. Such vectors contain information about the particle location, velocity and charge.

Memory aligned temporal vectors are used to store the projected values of the electric and magnetic fields on the particles.

All vectors are fitted in the accelerator memory, aligned and ready for SIMD vector operations. The main operations performed are multiplications and additions. All these conditions demand a system which is highly parallel and independent, and benefit from a large number of cores with access to very high memory bandwidth. Following our past developments in the DEEP and DEEP-ER projects we decided to map the particle solver to the ESB.

The SLURM batch script calls the executable of xPic, pinning it to the allocated processors of the ESB and the CM. The executable splits the main communicator into two parts, each corresponding to the field (CM) and particle (ESB) solvers. Following the ESB architecture, each node runs a single MPI process connected to the accelerator. Each MPI process in the particle solver is connected to an MPI process of the field solver in the CM.

#### *5.3.2.3 xPic field solver*

The field solver requires the resolution of an iterative linear system. It also performs finite-element differential operations on a Cartesian grid. The differential operations communicate data between neighbouring processors and the iterative method does the global gathering of a residual value (the difference of the result between two iterations). These procedures are complex and require high performance in a single thread. Memory access is not necessarily cache optimised. The two communication patterns stress the system in different ways, but the amount of data transferred between processors in relatively low. The field solver of xPic is mapped to the CM to take advantage of the higher per-thread performance.

The field solver runs with its own global communicator on the CM. The field solver does not perform any type of I/O. Communication between the field and the particle solver is performed using a point-to-point MPI intercommunication. This intercommunication is less frequent than the communication inside each one of the solvers. The message size is about ten times the size of the Cartesian grid in each MPI process. For a typical run of 10×10×10 cells per MPI process, the message size between the CM and ESB modules is around 80 KB.

### *5.3.3 GMM*

A second ML model, the GMM, is used to discover new information hidden in hundreds of GB of particle data. It allows to discover the real velocity distribution of particles in the plasma, as a combination of a few Gaussian distributions, instead of considering a single Maxwellian distribution. The difference between these two representations allow

to uncover critical zones in the plasma and to correct real energy miscalculations. The analysis of particle data from the xPic particle solver with GMM runs on the DAM.

The use of GMM is a gold mine of information for a plasma scientist. Each particle output from the xPic code can carry up to a few Terabytes of information, so storing a few time steps for later analysis is impossible. On-the-fly analysis of particle information using GMM is a major advantage in the discovery of new plasma physics.

### *5.3.4 Space Weather workflow*

First, in the DLMOS workflow, solar images and solar wind information is downloaded and pre-processed using a Data Pre-Processing (DPP) tool. These large datasets are used to train a Neural Network that is capable of predicting the solar wind properties from images of the Sun. The *training phase of DLMOS* is performed in the DAM. A second independent phase of the application, the Inference, uses the trained model to infer a single solar wind condition from a single solar image. This *inference* can be performed in the ESB before the initialization of an xPic job. Following this inference the initial conditions required by the xPic code are generated. The code xPic is then executed concurrently in the ESB and the CM modules. Every few hundred iterations, detailed particle data is transferred to the GMM for analysis. The GMM is executed in the DAM while the xPic code runs. Figure 5.2 shows this workflow.

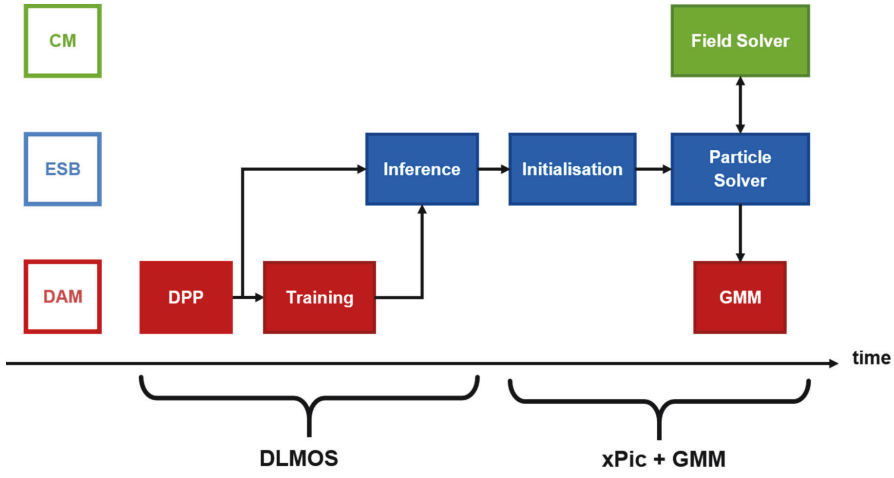


**Figure 5.2: Schematic workflow of DLMOS + xPic + GMM in the MSA**

## 5.4 Porting experience

### *5.4.1 Porting of DLMOS*

The code DLMOS has been developed during the DEEP-EST project. It has been created and tested on the local laptops and workstations of KU Leuven, and porting them to the DEEP-EST system only required the use of the appropriate python software packages. We rely on the competence of the Jülich Supercomputing Centre (JSC) to compile the compute critical packages of the system, including PyTorch and mpi4py, but we require additional packages that cannot be fully compiled from source in each cluster. The DLMOS code uses packages like SunPy, AstroPy, scikit-learn, Pandas, among others. These, and other dependent packages, are installed using “conda install” or “pip install”. It would be extremely cumbersome if each one of them had to be installed by the system administrators.

An alternative is to use containers. With Singularity we can package and deploy our software without worrying about the installation and compilation of all our required packages. We keep outside the container the python packages that are critical for the execution in the MSA, including PyTorch, mpi4py and TensorFlow.

This approach limits the complexity of porting the code to the MSA. The DLMOS application is independent from the other two applications and does not require special data transfers.

Porting the code to the DEEP-EST system only required the intervention of the JSC team in the installation of the critical Python packages. Secondary packages were installed in the user home directory as part of their virtual environment. We estimate that the porting effort took about one working day.

### *5.4.2 Porting of GMM*

The second ML application of KU Leuven is GMM. This software has already been used to study the complexity of plasma flows in the magnetosphere of our planet. The results have already been published in an international peer-reviewed journal[75]. The model uses mainly a customized version of the scikit-learn package. This customized version includes new functions that add features to scikit-learn. These python scripts do not need to be compiled, and run on top of the existing installed packages.

[75] Dupuis et al (2020): https://doi.org/10.3847/1538-4357/ab5524

The GMM application performs the characterization of particle velocity distributions in different regions of space. The code is embarrassingly parallel. It subdivides the physical space in multiple sub-domains and each one of them is processed in parallel by a different process running on the DAM.

We have ported the GMM algorithm to use PyTorch for the computations, following the developments of external authors. Our plan was to use the GPU offloading capabilities of PyTorch to maximally utilize the DAM. However, the advanced features that we included in the scikit-learn version could not be added in time to the PyTorch version. These functionalities are critical for our application and include a full correlation matrix and a point weighting for each plasma particle. For this report we will be using the GMM version developed with scikit-learn.

Porting the code required the intervention of the JSC team. To allow a connection with the code xPic, the mpi4py python script must be installed using the same software stack used by xPic, including the same version of ParaStation MPI. The estimated porting time was one working day.

### *5.4.3 Porting of xPic*

#### *5.4.3.1 Initial XeonPhi version*

The code xPic was already ported to the Cluster-Booster system in the DEEP-ER project. Three different levels of parallelization were used: 1) MPI parallelism for inter-node communications, 2) OpenMP multithreading for intra-node computation and memory sharing, and 3) SIMD vectorization to take advantage of Intel vector registers in the Xeon processor line.

In addition, to make a good use of the cache hierarchy we implemented tiling of the particle solver deployed in the Booster module. The size of the tiles has an important effect on the cache accesses, in particular for the moment gathering in the particle solver. This three levels of parallelism and four levels of memory management were specially designed to work on Intel Xeon Phi processors.

Memory management is a critical component of Intel processors. In particular the allocation of aligned vectors and registers for the vectorization of operations. These fine-grained implementation details are less relevant today as compilers have implemented more and more optimization procedures. However, the code sections dedicated to memory handling become irrelevant when new architectures are used to run the codes. In our case the use of GPUs required KU Leuven to restructure large parts of the code.

#### *5.4.3.2 GPU version*

For the ESB of the DEEP-EST MSA we needed to re-orient the parallel structure and memory strategy in order to take advantage of the GPU accelerators. At that point, we made an strategic decision: we cannot depend any longer on a single technology that can disappear in the future. The discontinuation of the Xeon Phi processor line was a strong reminder that other proprietary technologies create dependencies that can have a very strong effect on our productivity. There is no perfect scaling that will recover the months of lost simulations due to a forced change in the basic structure of the code provoked by a change in the hardware architecture. This is why at KU Leuven we strongly feel that the path towards Exascale must include decoupling the hardware and the software development. A clear example of the need for vendor-agnostic programming approaches is the recently announced European LUMI supercomputer. This EuroHPC JU project is based on AMD technology, including GPU and Data Analytics partitions based on AMD GPUs.

For these reasons we decided not to use CUDA when porting the particle solver to GPUs. We opted for the most recent possible versions of OpenMP to offload computations to the GPU of the ESB (or the DAM if needed). This required a close collaboration with the JSC support team to compile a full stack based on GCC 10.2.0, compatible with parts of the OpenMP 5.0 standard. This was a delicate procedure as the full stack had to be updated, including the compilation of cross compilers with support to NVPTX, the use of updated glibc and binutils, and the recompilation of all the libraries in the stack. The full procedure was not straightforward and might require updates when new versions of the GCC compiler are available.

Among all the compilers GCC presented the closest compatibility with the OpenMP 5.0 standard and with the stack libraries. However, we are aware that the performances of offloaded code to the GPUs with OpenMP 5.0 and GCC is not the best in published tests. Right now we are focusing our efforts on the deployment of a code that can be easily transported to a different system without major headaches. Performance optimizations will be carried on in future projects (e.g. the DEEP-SEA project).

The method used to port the code to the GPU architecture is detailed in the Best Practice Guide, section 8.4.3.4 of this volume (OpenMP 5.0). The main changes performed are the following:

- Replace the existing OpenMP SIMD pragmas with a generic macro-defined pragma. This macro definition changes depending on the type of offloading device used. For GPUs the pre-compiler imposes a target section, while for CPUs it imposes a SIMD for loop section.

- Memory transfers from/to host and device are performed using the OpenMP map-directives of the Cartesian fields used by the particle solver. These include the electric and magnetic fields (to), and the particle moments (from).
- Particle initialization and transport is performed only in the device. Information about the position, charge, and velocity of the particles is maintained only in the device memory.
- Particle memory allocation is performed by a wrapper routine that selects the memory allocator between `malloc` (CPU) and `omp_target_alloc` (GPU).
- All pointers to particle information are carefully identified as `is_device_ptr` in the target directive sections.
- The sorting algorithm used to arrange and select particles for communications had to be re-written from scratch to compensate the limitations of the OpenMP offloading system. The sorting algorithm has been decomposed and reduced to basic for loops with auxiliary vectors. It currently still requires the use of a serialized section that hinders its scalability. This serial section will need to be re-worked in a future version of the code. We believe that sorting algorithms that use the OpenMP offloading to the GPU will be a major requirement in future developments for KU Leuven and for other application developers.
- The moment gathering algorithm of the code also needed a major restructure. The previous version of the code included sections that where tailored for the vector registers of the Intel processors. These algorithms could not work under the GPU architecture. Major changes, simplifications and testing under CPU and GPU architectures implied a large number of bugs and memory leaks that required repairs. This is the single largest change in the code.
- The previously existing parallelism layers are maintained, but in the case of GPU offloading we limit the number of OpenMP threads to 1 per each MPI process in the particle solver, in order to use only 1 GPU per MPI process. We also limit the number of blocks to 1. This means that each MPI process will count only 1 OpenMP thread, with only one block. The CPU version of the code can make use of more complex combinations to take advantage of memory cache in Xeon processors. In the future we will work on the use of multiple threads per MPI process to deploy on more than one GPU per MPI.
- GPUDirect communications are used to move particles between subdomains located in different GPUs. We have added a new communication buffer in the particle solver to define specific memory locations in the GPU devices, required by the CUDA-aware MPI layer.

Random number generators and sorting could be wrapped in functions that call standard C or CUDA routines. This shows interoperability between OpenMP and CUDA runtime libraries. However, for the current version of the code we have used the standard C library implementations of all mathematical functions. For scientific codes the correct generation of random numbers is a major issue that requires more careful developments in the future. In particular for MSA systems, where CPUs, GPUs, multiple nodes, and multiple modules are involved, the generation of random numbers will require much more attention.

We estimate the total personnel effort of the changes and adaptation of the code to roughly 4 PM, where the typical ratio of development time to bug correction time is in the order of 3:1, i.e. one day of developments lead to 3 days of corrections, testing and optimizations.

## 5.5 Scalability

### *5.5.1 Scalability of GMM*

The code has been executed in the DAM of the DEEP-EST system, using the results of a previously executed simulation. A list of 384 particle files were available for processing, each one corresponding to a subdomain of the simulated 2D box. The files were written independently by each process of a parallel execution that ran on a NASA supercomputer using 16 nodes with 24 cores each, which by coincidence has the same core distribution per node as the CM of the DEEP-EST system.

We performed a weak scaling test, where a single processor reads and processes a single file. We use 48 threads per node and scale the execution from 1 to 8 nodes, for a total of 384 threads in the largest execution, which processes the full dataset.

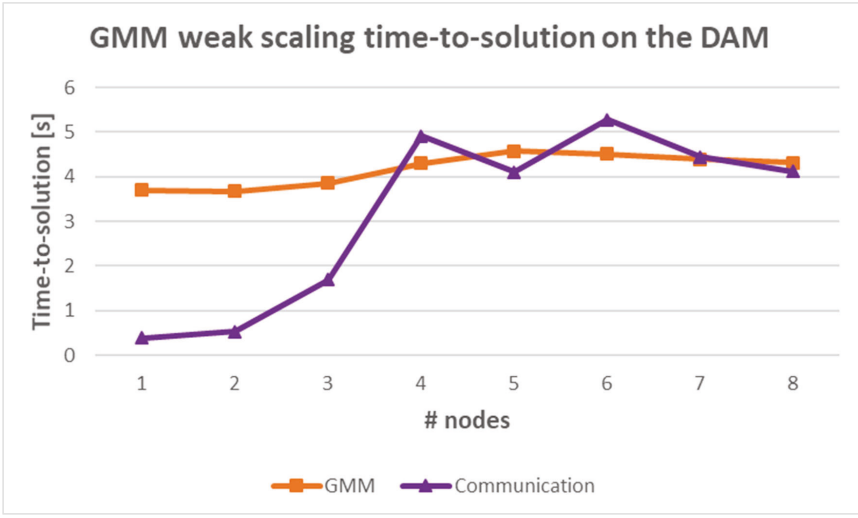


**Figure 5.3: GMM weak scaling performance on the DAM**

Figure 5.3 shows the runtime of the GMM in the DAM. We have extracted three timers from the script: 1) I/O (not shown in the figure), 2) GMM execution, and 3) MPI communications. The code uses MPI at the end of the execution to collect in a single node all the results and produce the final output figures. The I/O time was negligible for all the runs, in the order of milliseconds. The points in the figure correspond to the average value extracted from all the runs.

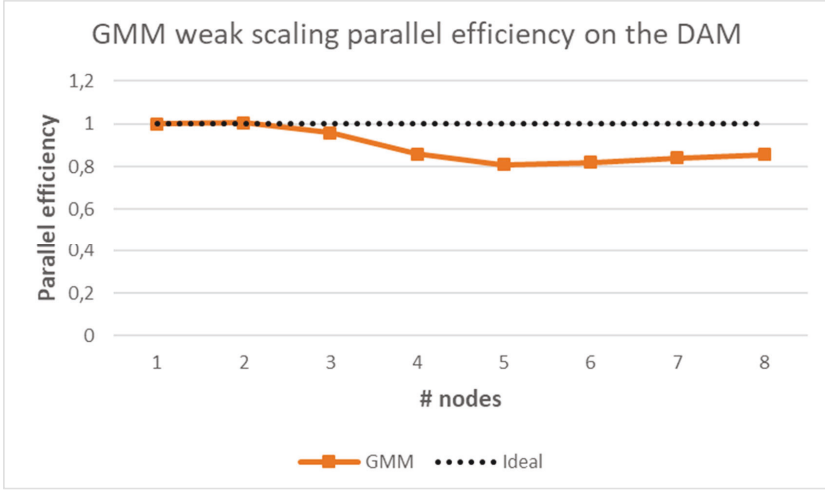


**Figure 5.4: GMM weak scaling parallel efficiency on the DAM**

It is clear from Figure 5.3 and Figure 5.4 that the GMM execution presents an almost perfect weak scaling efficiency with very small variability (i.e. good load-balancing). However, the very rudimentary MPI communications implemented present a major bottleneck in the code. When the number of MPI processes exceeds 144, the communication time becomes equal to the processing time.

We are satisfied with the performances of the machine learning algorithm, but we will work on the parallelization of the final I/O to avoid this critical bottleneck. Such work will take place after the end of DEEP-EST in the frame of the DEEP-SEA project.

### *5.5.2 Scalability of the particle mover of xPic*

In this section we will focus specifically on the scalability of the particle mover on the ESB. To isolate the particle mover and test the basic functionalities of the newly improved code, we turn off the moment gathering in the particle solver and the field solver entirely. We execute the code on the ESB exclusively. With this setup the particles will still move following the Newton equations of motion in a constant electromagnetic field that does not change.

The tests show the scalability of the particle mover and the performances of the GPU under low and high memory loads. Figure 5.5 and Figure 5.6 present the parallel efficiency of the xPic code for a strong scaling case. The memory occupancy of the strong scaling tests changes from 31 GB per GPU (1 node) to 0.5 GB per GPU (32 ESB nodes). Such a decrease in the use of GPU resources (memory and computing) has an impact on the efficiency of the code. Moving from 8 to 16 ESB nodes we can see a strong drop in Figure 5.6. This is a trend that continues when using 32 nodes.

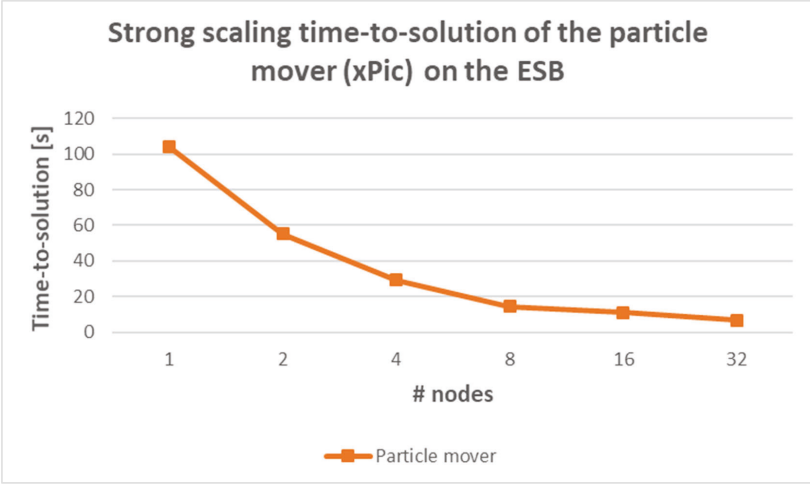


**Figure 5.5: Strong scaling time-to-solution of the xPic particle mover on the ESB**

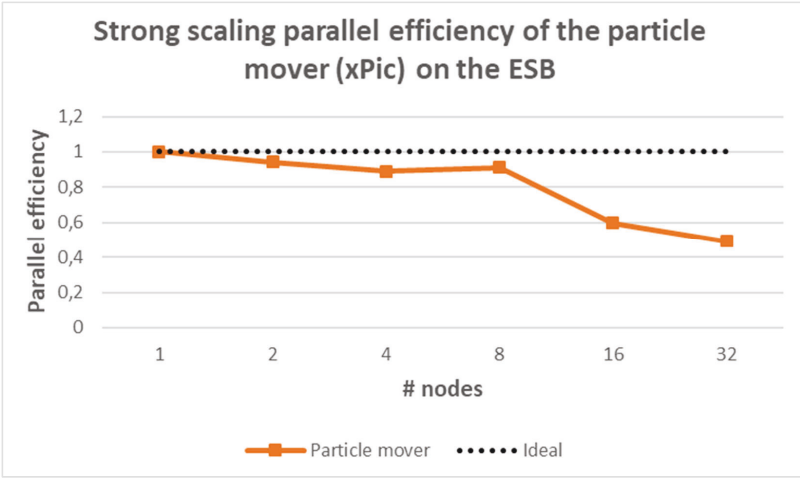


**Figure 5.6: Strong scaling parallel efficiency of the xPic particle mover on the ESB**

This shows that it is extremely important to occupy as much memory as possible in the GPU to take advantage of the fast calculations on the simple operations used in xPic, and to minimize memory management overheads. The NVIDIA V100 GPUs used in the ESB have a capacity of 32 GB that we can fill with half a billion particles.

One of the priorities for KU Leuven is to demonstrate a good weak scaling of the code. Figure 5.8 and Figure 5.8 present the results of a weak scaling test performed from 1 to 32 nodes on the ESB. The figures show that the particle mover keeps a constant, nearly perfect scalability, but has a small performance-drop moving from one to multiple nodes. We think that GPUDirect communications between GPU nodes can account for this initial loss in performance.

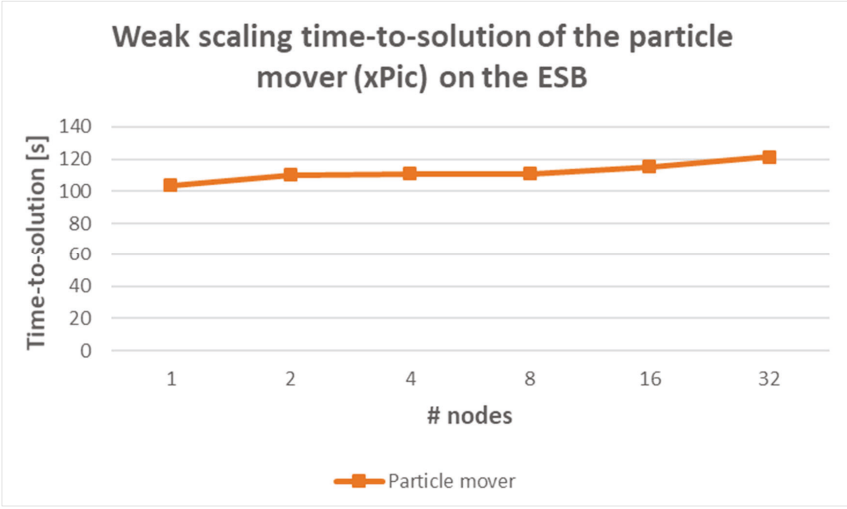


**Figure 5.7: Weak scaling performance of the xPic particle mover on the ESB**

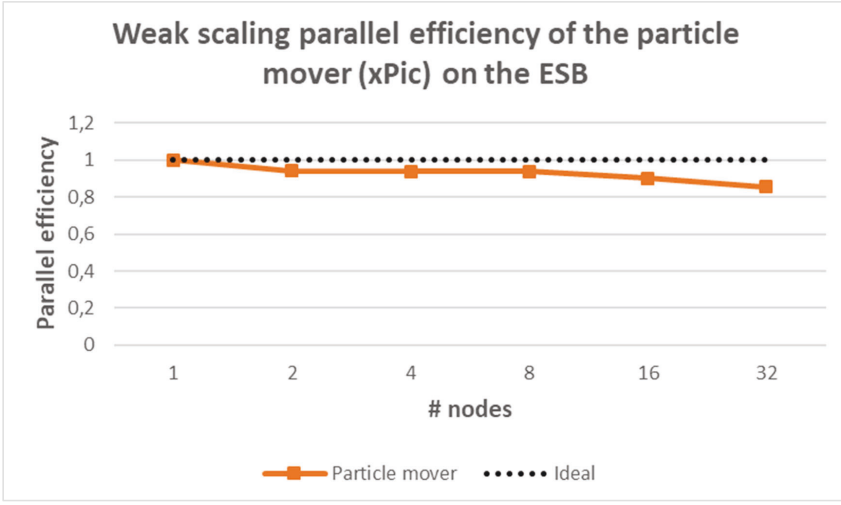


**Figure 5.8: Weak scaling parallel efficiency of the xPic particle mover on the ESB**

### *5.5.3 Testing of the number of tasks per node in the ESB*

For the results in this section we have activated all the phases of the xPic code: the field solver, the particle mover, and the moment gathering. We have executed the code in what we call the “MONO” mode (from the word monolithic). In this mode the code is

executed using a classical monolithic architecture where all the phases of the code run in the same node.

The MONO mode can be run in any of the modules of the DEEP-EST system. When executed in the CM, the code runs each one of the compute phases in the CPU, including the two phases of the particle solver. When the code is run with the MONO mode in the ESB or the DAM, the particle solver offloads its compute intensive parts to the GPU in the corresponding node. A single ESB or DAM node contains a single GPU. However, when multiple MPI processes are launched in the same node, each one makes use of the same GPU. This action is possible thanks to the Multi-Process Service (MPS) of the NVIDIA V100 cards: the GPU can hold multiple concurrent tasks. It is then possible to launch an 8 MPI process job in a single ESB node with one MPI process per core, while these 8 processes share the single GPU available in the node. This means that we either: a) use a single core per node attached to a single GPU to maximize its performance, or b) use all the CPU cores of the node, each one sharing the GPU and using 1/8th of its memory and computing capacity.

To test the MPS of the NVIDIA V100 cards we tested the number of concurrent tasks that can be executed in a GPU. We performed a weak scaling test with a fixed number of 16 ESB nodes, this time changing the number of MPI processes per node, from 1 to 16. For this weak scaling test we used a number of particles that could be fitted in the GPU memory.

In this test we perform a basic neutral plasma simulation that solves the transport of particles immersed in an electromagnetic field. The simulation is 2D and the total number of cells used for each case is shown in Table 5.1. The total number of particles per task in each node is equal to 1,572,864. The largest of the simulations listed in the table contains a total of 402,653,184 particles, corresponding to 64 particles per cell for each one of the two particle species used, multiplied by the total number of cells listed in the last row of the table.

| # nodes | # task/node | # cellx | # celly | # total cells | Total (sec) |
|---|---|---|---|---|---|
| **16** | 1 | 384 | 512 | 196,608 | 137,442 |
| **16** | 2 | 768 | 512 | 393,216 | 143,21 |
| **16** | 4 | 768 | 1024 | 786,432 | 157,467 |
| **16** | 8 | 1536 | 1024 | 1,572,864 | 194,166 |
| **16** | 16 | 1536 | 2048 | 3,145,728 | 433,753 |

**Table 5.1: Experiment setup for testing the number of tasks per node**

Figure 5.9 and Figure 5.10 show the weak scaling efficiency of the field solver in the CPU when testing the number of tasks per node and Figure 5.11 and Figure 5.12 show the strong scale efficiency of the particle solver when testing the fraction of GPUs used per task. There is a reason for the appearance of these plots: a single ESB node is composed of an 8-core CPU and a single V100 GPU. Going from 1 task to 2 tasks per node means that the field solver will be using double amount of CPU cores, while the particle solver will be using only 0.5 GPUs per task. For this particular reason the efficiency of the two solvers has to be computed using a different metric.

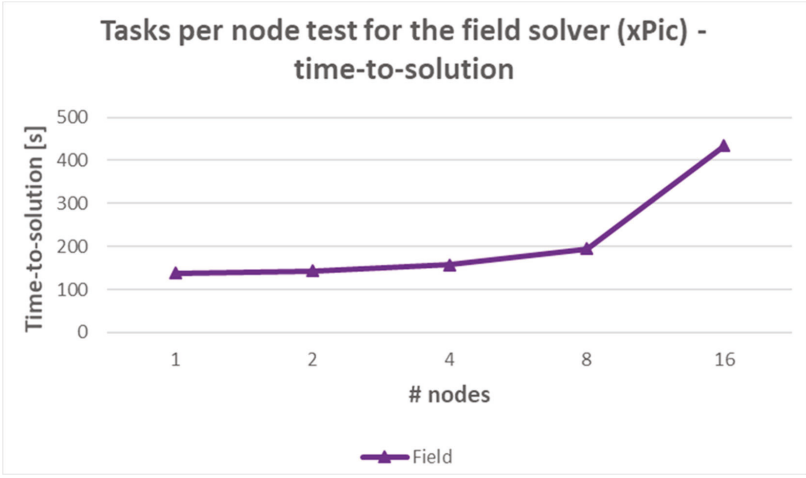


**Figure 5.9: Tasks per node test for the xPic field solver – time-to-solution**

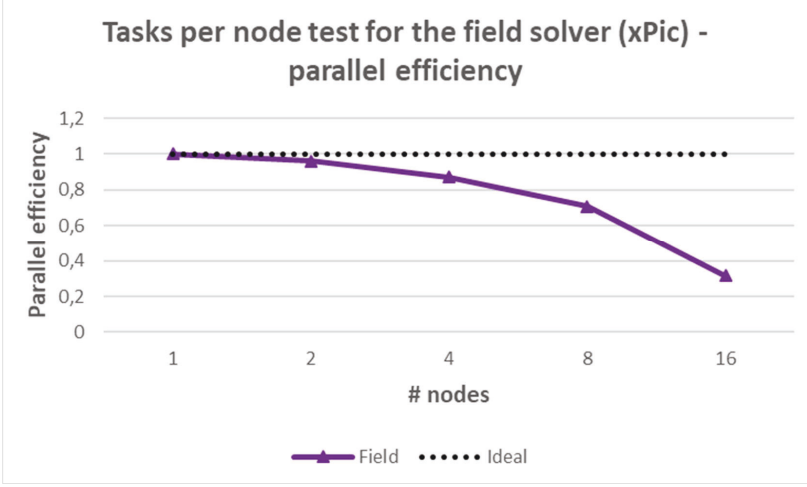


**Figure 5.10: Tasks per node test for the xPic field solver - parallel efficiency**

In this case the field solver shows a clear degradation of its scalability. We are still investigating the reason for such poor performances in the CPU side. This behaviour has been observed in other runs. The field solver relies on the parallel algorithms of the library PETSc. It is possible that the amount of cells used per task (12,000 cells) is too large for the solver to handle it efficiently. In different runs with a smaller number of cells per task we have noticed that PETSc switches to a different numerical solver that converges faster.

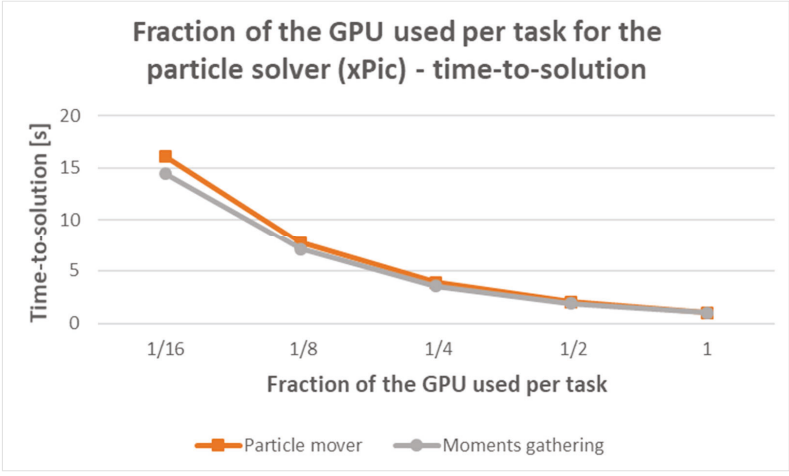


**Figure 5.11: Fraction of the GPU used per task for the xPic particle solver - time-to-solution**

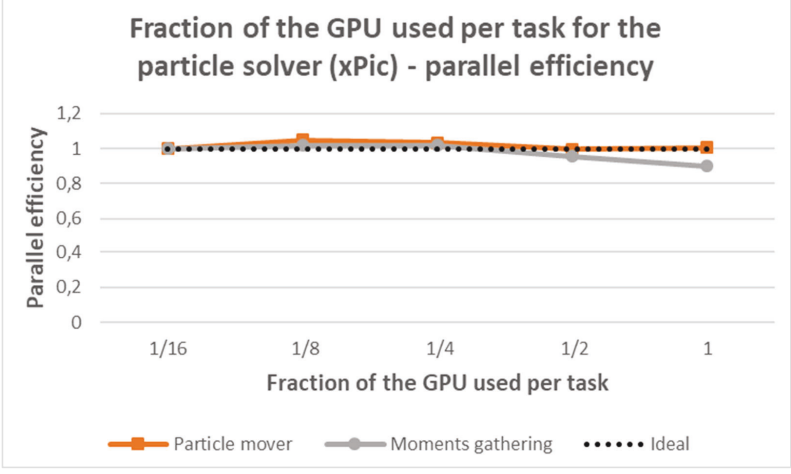


**Figure 5.12: Fraction of the GPU used per task for the xPic particle solver - parallel efficiency**

On the other hand, the particle solver presents a nearly ideal parallel efficiency. This shows that the MPS of the Nvidia GPU has no problem handling multiple simultaneous tasks. Every time the number of tasks doubled, the runtime of the particle solver halved.

### *5.5.4 Testing the number of particles per cell*

One of the main features of the xPic application is using accelerator nodes to execute the particle solver. The advantage of our application is that we can fill the accelerator with operations by increasing the number of particles used. Figure 5.13 shows a run of the xPic code using a 2D setup with 1,536×1,024 = 1,572,864 cells and two particle species. The number of particles per cell and per species was increased from 64 to 128 and finally 256 (×2 due to the number of species). Inspired from the results of the previous section we use a total of 8 tasks per ESB node. This means that each task working on the particle solver has a maximum memory capacity of 4 GB (8 tasks × 4 GB = 32 GB of memory onboard the GPU).

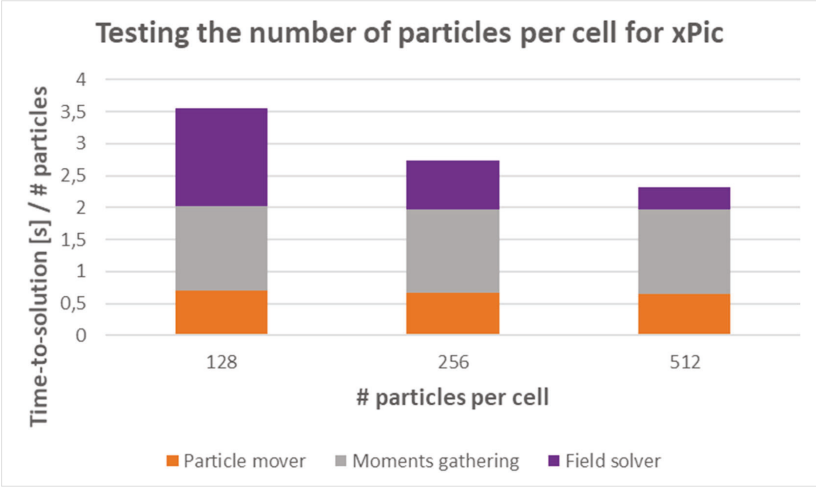


**Figure 5.13: Increasing number of particles per cell**

This figure shows that the particle solver scales almost perfectly with the number of particles. This also means that the GPU and its MPS makes very good use of the available resources when the GPU memory is full, as well as when only a fraction of the accelerator is used. Because the execution of the field solver does not depend on the number of particles, it shows a constant runtime for each one of the jobs above, i.e. the total runtime decreases when it is normalized by the number of particles.

In the next sections we will perform similar runs in the “MULTI” module mode, executing xPic across Cluster and Booster. We had to make a compromise between the performances of the code and the available number of resources. Each MPI process in the field solver can be executed in a single CPU core. This means that there is a total of 1,200 available cores for the particle solver, while the number of maximum GPUs available for the particle solver in a single MLA module is 50 (ESB nodes in the IB network). Using a distribution of 8 tasks per ESB node we can extend the available resources, dividing each GPU in eight tasks. We will be able to couple 400 CM cores with 400 ESB tasks, for a maximum run of 8 CM nodes coupled with 50 ESB nodes.

### *5.5.5 xPic weak scaling on the CM and on the ESB*

We have performed a weak scaling test of the full code in the MONO mode using only the ESB nodes (Figure 5.14). Figure 5.15 shows the parallel efficiency of the code from 1 to 32 ESB nodes, where each one of the phases has been normalized to one. The setup of this test is the same as the one presented in the previous section. For these jobs each node has been divided in 8 tasks (MPI processes) and the number of particles per cell is set to 256.

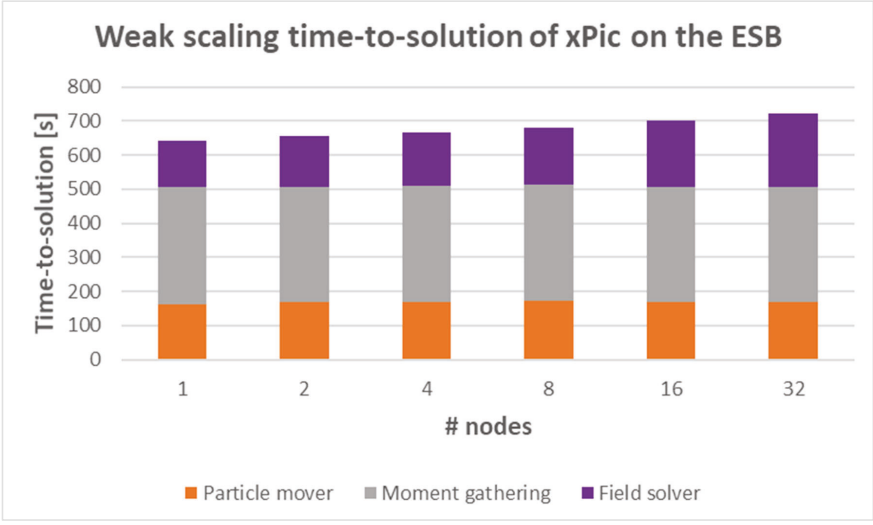


**Figure 5.14: Weak scaling time-to-solution of xPic on the ESB**

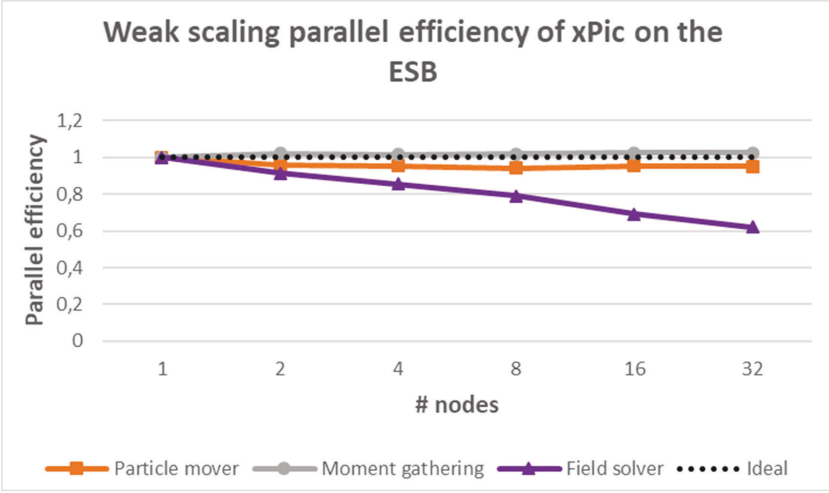


**Figure 5.15: Weak scaling parallel efficiency of xPic on the ESB**

These runs show again an almost perfect scalability of the particle solver. Both the particle mover and the moment gathering have an almost perfect parallel efficiency. Only the field solver struggles to maintain its scalability using the CPUs of the ESB nodes, as previously shown in the tests in Figure 5.9. The field solver efficiency drops to 62% at 32 ESB nodes.

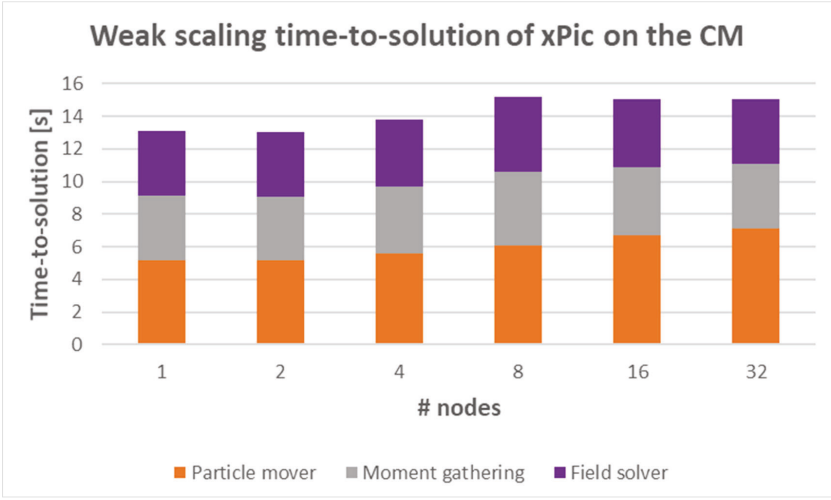


**Figure 5.16: Weak scaling time-to-solution of xPic on the CM**

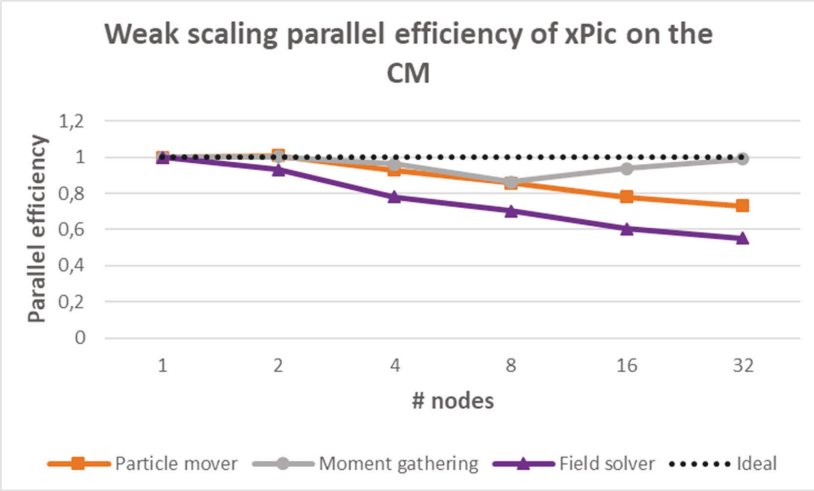


**Figure 5.17: Weak scaling parallel efficiency of xPic on the CM**

Figure 5.16 and Figure 5.17 show the same test, this time executed in the CM. In this test the field solver also shows a degradation in parallel efficiency, down to 55% at 32 nodes. At the same time the particle mover shows a reduction to 73% at 32 nodes, but the moment gathering execution fluctuates with a minimal efficiency of 86% at 8 nodes and a maximum of 99% at 32 nodes. These two figures show that an execution of the particle solver on the ESB accelerators allows to reach an almost perfect code-scalability, compared with the results obtained in the CPU runs. It also shows that the field solver requires a more careful optimization.

### *5.5.6 xPic weak scaling on CM+ESB*

We have ported the code xPic to the Cluster-Booster (CN+ESB) mode, which we also call the „MODULAR“ mode. Both the MONO and the MODULAR versions of the code cohabitate in the same sources. They share the most important segments of the code, but have a critical difference: the transfer of information between the field solver and the particle solver is done using MPI communications, and each one of the two solvers is executed as an independent application.

The selection of the MODULAR version of xPic is performed during compile time. A CMake option turns ON/OFF the declaration of a compile-time variable that selects the pieces of code that need to be compiled. The particle solver can then be compiled with or without GPU offloading. This means that we can perform tests of the performances of the MODULAR version in a CN-CN arrangement, using the CN both for the field and the particle solver.

In this section we report the performances of the MODULAR xPic using the CN-ESB arrangement, with all the nodes using the same InfiniBand network.

For this test we have used a 2D plasma simulation with 256 particles per cell. Every time we double the number of nodes in our tests we try to increase the number of cells in the simulation by two. However, scaling-up the problem in this manner is not straightforward. First, to avoid doubling the size of communications in only one dimension, we enlarge the simulation domain every time in both dimensions. Second, the number of cells in each dimension must be divisible by the number of processors automatically assigned by the MPI cartesian communicators. Finally, we need to maintain a total number of cells per process as close as possible for each one of the runs.

Following these requirements we performed a first run, where the total number of cells per MPI process is approximatively 16,330. In this first run we tried to use the largest possible number of nodes in the CM. The runs use 1 to 16 CM nodes, with a total of 24 MPI process per node (1 per core). On the particle solver side, in order to reciprocate the very large number of MPI processes, we launch from 3 to 48 nodes, each one with 8 MPI processes per node. This run creates a very intense load in the ESB, and stresses the MPS of the NVIDIA V100 cards.

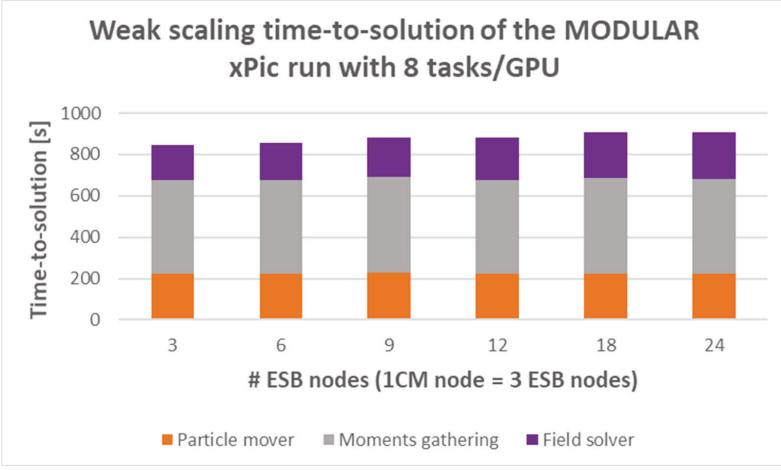


**Figure 5.18: Weak scaling time-to-solution of the MODULAR xPic run with 8 tasks per GPU**

Figure 5.18 shows the total runtime of the code. It shows that most of the execution time is spent in the moment gathering phase of the code. The particle solver, which includes the mover and the moment gathering, shows an almost ideal weak scale efficiency (Figure 5.19). The field solver on the other hand shows an increase of 34% in the execution time, moving from 1 to 8 CM nodes.

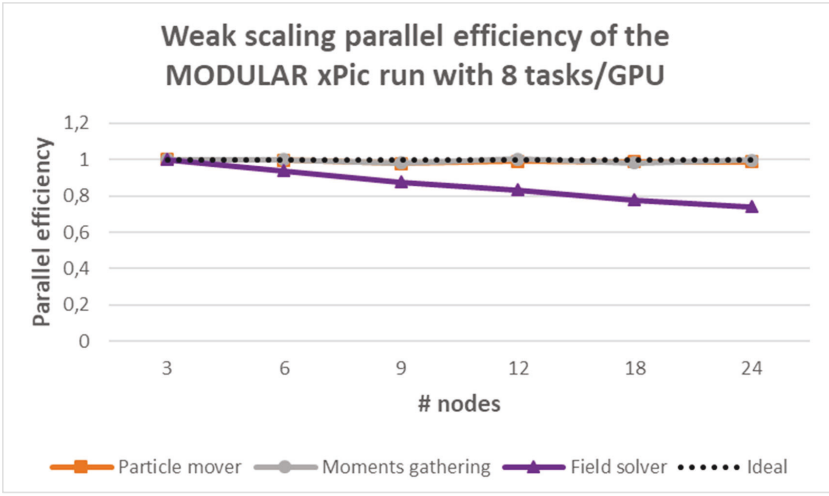


**Figure 5.19: Weak scaling parallel efficiency of the MODULAR xPic run with 8 tasks per GPU**

We performed a second test to verify the performances of the code when only one MPI process is assigned to each ESB node, i.e. we do not make use of the MPS of the GPU card. In this case the maximum number of MPI processes executed is 48 (the maximum allocation of ESB nodes was 48). We maintain the total number of cells per task in the order of 16,330, as in the previous test. Figure 5.20 shows the runtime of the code as a function of the number of ESB nodes.

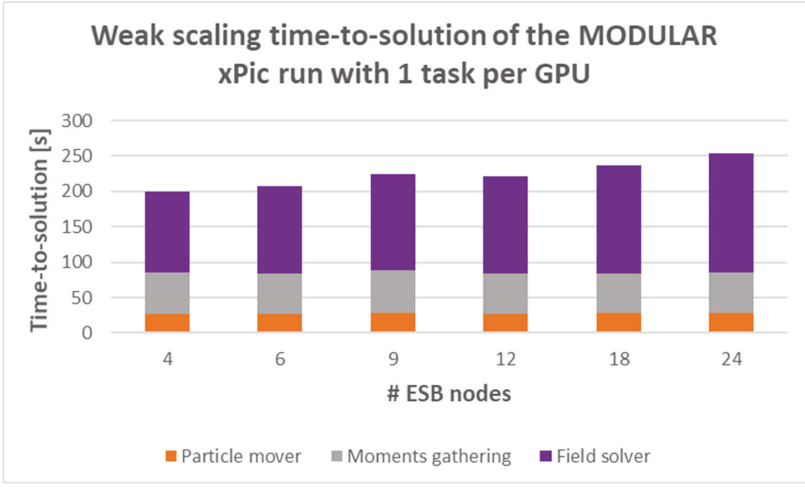


**Figure 5.20: Weak scaling time-to-solution of the MODULAR xPic run with 1 task per GPU**

As we expected, the particle solver is clearly accelerated by the use of the full GPU for each MPI task, instead of using only 1/8th. The particle solver presents again an almost

ideal weak scaling efficiency, while the field solver once again shows difficulties to sustain an efficient scalability (Figure 5.21). The execution time of the field solver increases by 45% from the smaller to the largest simulation. Notice, however, that only one CM node was used for this test, employing a range of 1 to 24 cores of the CM node.

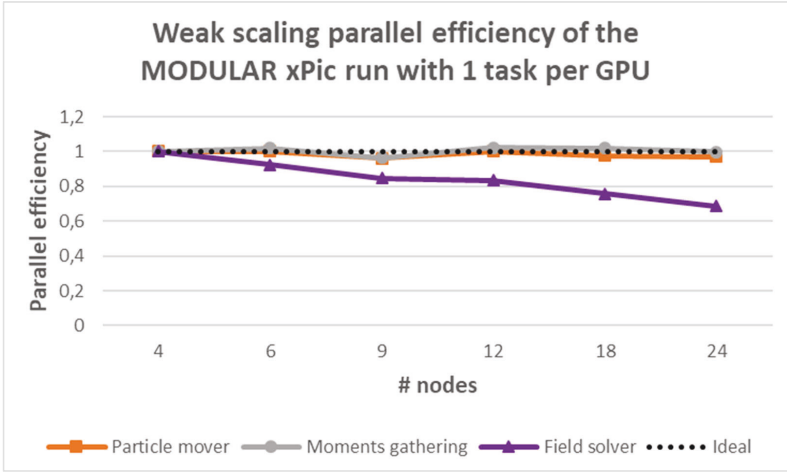


**Figure 5.21: Weak scaling parallel efficiency of the MODULAR xPic run with 1 task per GPU**

We decided to stress the system slightly further. We use exactly the same number of nodes and processes as in the previous test, but we dramatically increase the number of cells in each process. This allows us to fill the memory of the GPU cards and to increase the computation-to-communication ratio of the field solver. In this test we are still using only one CM node and up to 24 ESB nodes as in the previous case, but we charge each MPI process with a mean of 132,700 cells, a problem 8 times larger than before.

Figure 5.22 and Figure 5.23 show that the particle solver presents a clear parallel efficiency that is not matched by the field solver. The larger computing load also shows that the field solver does not scale as well as the particle solver and a larger portion of time is dedicated to this phase, in comparison to the previous test with lighter computing loads.

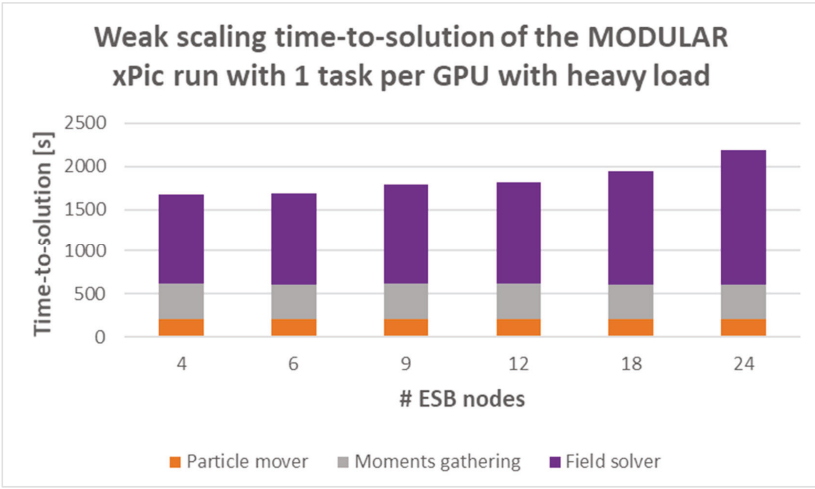


**Figure 5.22: Weak scaling time-to-solution of the MODULAR xPic run (heavy load) with 1 task per GPU**

We are convinced that the particle solver, the section of the code that required most changes during the DEEP-EST project, presents a parallelization strategy that is optimal in its current state. We have made last-minute corrections to some of the code-sections of the particle mover that included serial code, but we are aware that there are still some sections that can be further improved. The field solver on the other hand, requires a re-evaluation of our scaling strategy. We will evaluate the reasons why the PETSc algorithms do not scale as expected. Such evaluation is part of our future work.

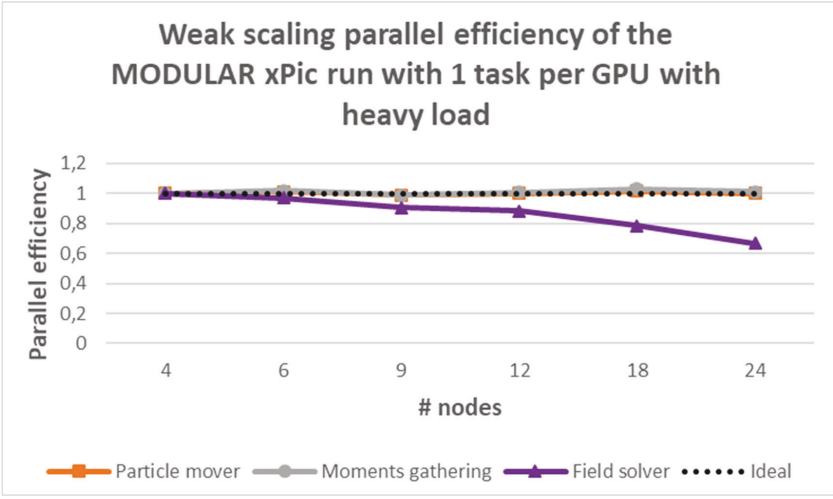


**Figure 5.23: Weak scaling parallel efficiency of the MODULAR xPic run (heavy load) with 1 task per GPU**

### 5.5.7 Our path to Exascale

#### 5.5.7.1 What are the limitations? – Can they be fixed?

There are three factors that are currently limiting our ability to obtain better performances and reach scalability towards Exascale:

- First, the field solver presents deficiencies in its scalability. We are currently using the code at an inflection point of the library: smaller matrices could allow to switch the code into a faster and simpler model, at the expense of usability (large simulation runs require large matrices). On the other hand PETSc shows better scalability performances when the matrices solved are larger and the code spends more time performing computations and less time performing communications. This means that the subdomains have to be larger, containing many more cells, in order to gain in efficiency,
- This brings the second issue: we are currently coupling one MPI process in the CN with one MPI process in the ESB. The interface between the particle solver and the field solver uses MPI communications and a vector copy that maps 1-to-1 each side of the code. We believe that the next important step in the development of our code is to map one MPI process in the CN with multiple MPI processes (or a single process connected to multiple GPUs) in the ESB. With the good efficiency presented by the particle solver and a better ratio between CPUs and GPUs we expect to gain in scalability.
- Finally, the particle solver still contains segments that perform poorly when offloaded to the GPU. In particular the particle sorting and the moment gathering contain serialized segments that require more attention. We also need to explore the possibility of using in parallel the CPU and the GPU in the ESB nodes simultaneously. Some authors have described good performance by overlapping these two processors at the expense of data transfers between the Host (CPU in the ESB node) and the Device (GPU in the ESB node).

A more detailed analysis of all the different code phases is under investigation. We will continue the optimization and development of the application codes from KU Leuven in the future.

#### 5.5.7.2 How to use future Exascale systems

There is a clear difference on the computational needs of the field solver of xPic and the particle solver. We believe that the Cluster-Booster division of work for this particular code is extremely important. Our goal is to perform simulations that minimize the noise. This is accomplished by increasing the number of particles in the simulation. Our goal is to reach up to 10,000 particles per cell in the future, while we currently use

256 particles in the tests performed above. This means that the stress of the system will be put in the accelerated section of the code.

We are planning to maintain the development of the Cluster-Booster model of xPic, but we think that our future code will make use of very large number of ESB nodes and a very small number of CN nodes. We will connect 1-to-many nodes between the CN and the ESB in future systems in order to increase the number of particles per cell.

#### *5.5.7.3 Where did the DEEP-EST project help on the way to Exascale?*

Until the beginning of the DEEP-EST project we were reluctant to use the GPU architectures with the CUDA language. We have been careful of not blocking our developments and become dependent on only one architecture. This is the reason behind our support for Intel Xeon Phi architecture that promised acceleration under the same software stack and hardware architecture as ordinary CPUs. It is obvious that such approach has changed since the beginning of this project, and the discontinuation of the Xeon Phi accelerators played a big role in our decision making.

However we were still reluctant to be forced to use a closed and proprietary language as CUDA. The advent of OpenMP 4.5/5.0 was a perfect opportunity for us. The knowledge we gathered in previous projects on the use of OpenMP has been very valuable here. Thanks to the help of the consortium we were able to port the code xPic to the GPUs of the ESB and the DAM, without making use of CUDA. This has been a very arduous process because compilers are still in the process of implementation of the newest OpenMP standards. We are persuaded that in the near future, compiler developers will be able to integrate OpenMP interfaces that can compete with native CUDA codes.

Without the help of the DEEP-EST consortium it would have been impossible to set up, deploy, and test this new software development approach. We are also looking forward to test our new code in systems that use AMD GPUs instead of NVIDIA GPUs. Computer centres financed by EuroHPC have invested in the purchase of large-scale AMD GPU clusters. OpenMP is the main method of porting proposed by these new computing systems, and the DEEP-EST project has enabled us to prepare port our code to this offload environment.

We have also relied on the competence of our partners to deploy libraries for parallel computing, I/O and machine learning. This has allowed us to port our ML codes and be ready for future large scale HPDA calls in European centres.

## 5.6 Energy consumption

We recovered the energy consumption for each one of the runs performed in the previous section from the DEEP-EST energy measurement database (DCDB[76]). We present in this section the energy consumption per node, calculating the total energy as a sum of multiple measurements of the instantaneous consumption every 10 seconds. Jobs that run during a short period produce data with larger error bars. All energies reported in the figures bellow are given in Megajoules (MJ).

Figure 5.24 shows the energy measurements obtained for the strong scaling tests in the ESB. As the work per node decreases, the time to solution is shorter and as a consequence the energy consumption per node is reduced. In this figure we plot the total energy consumed by all the nodes in each job. Under ideal conditions the energy consumed should remain equal throughout all the executions. We notice that for 8 nodes and above the energy consumption increases rapidly. As mentioned in the previous section, for more than 8 nodes the problem analysed here is too small and the GPU is launched with a very small load of work. This figure shows that the GPU is not used efficiently in this particular case for jobs with more than 8 nodes.

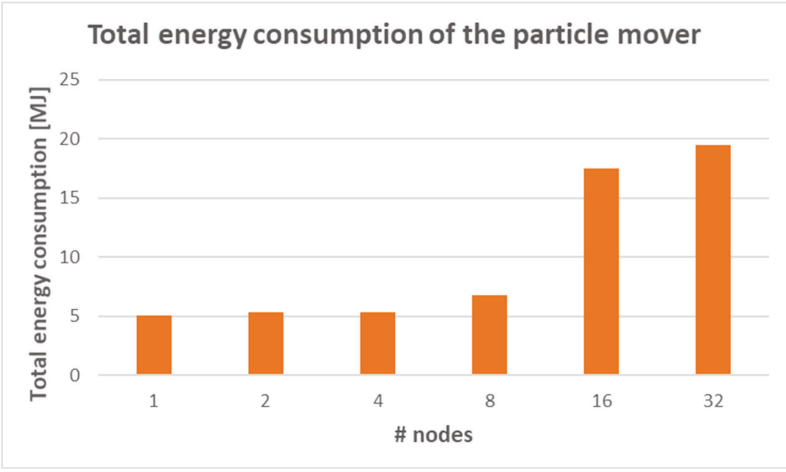


**Figure 5.24: Total energy consumed for strong scaling test case**

[76] https://deeptrac.zam.kfa-juelich.de:8443/trac/wiki/Public/Energy#UsingDCDB

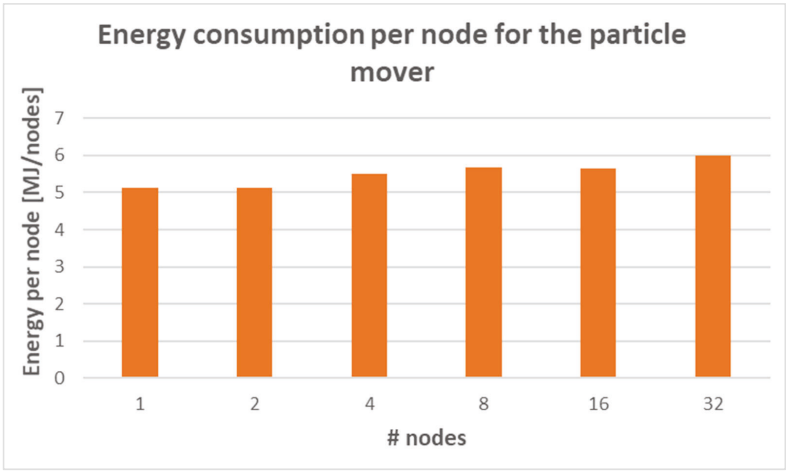


**Figure 5.25: Energy consumption per node for the weak scaling test**

Figure 5.25 shows the energy consumption per node for the weak scaling tests. In this case we would expect to have a constant energy consumption per node. However, this figure shows an increase when the problem is larger. We are recurrently investigating the causes and possible correctives for the strong increase of 20% from 1 to 32 nodes.

We have also gathered the energy consumption for a MODULAR run of the code xPic. The energy measurements presented here correspond to the simulation described in Figure 5.18 in Section 5.5.6. This test corresponds also to a weak scaling test. Figure 5.26 shows the total energy consumption in MJ for the CM and the ESB nodes. The measurements have not been scaled or normalized.

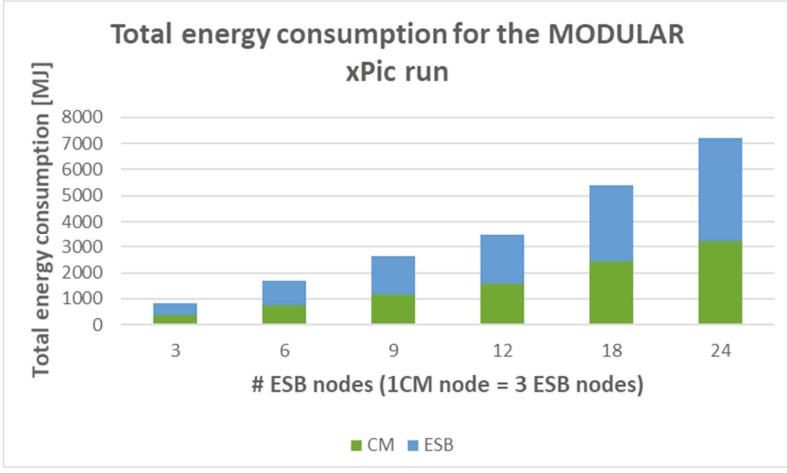


**Figure 5.26: Total energy consumption for the MODULAR xPic run**

Let us remember that this run was performed using one node in the CM for every 3 ESB nodes, and each ESB node was launched with 8 tasks per GPU. On average, the total consumption of energy was 24% higher in all the ESB nodes as it was in all the CN nodes. This is a very balanced use of resources.

When we perform a normalization by the number of nodes for each one of the runs (Figure 5.27), it is possible to observe the weak scaling efficiency of the energy consumption. It is also clear that each one of the ESB node consumes only 40% of the energy consumed by the CN nodes. The ratio of 1:3 nodes between the CN and the ESB is energetically balanced.

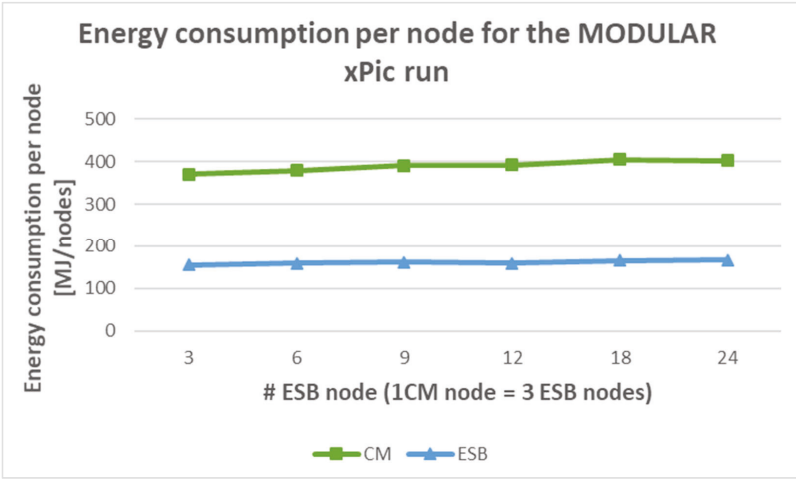


**Figure 5.27: Energy consumption per node for the MODULAR xPic run**

## 5.7 Performance comparison

### *5.7.1 Performance comparisons for the particle mover of xPic*

Figure 5.28 shows a comparison between the runtimes for the particle mover deployed in the CPUs of the CM and on the GPUs of the ESB. The figure shows the strong scaling tests described in the previous sections. While there is a clear under-use of the ESB for runs with more than 16 nodes, the ESB performs almost at all times one order of magnitude better than the CPUs of the CM.

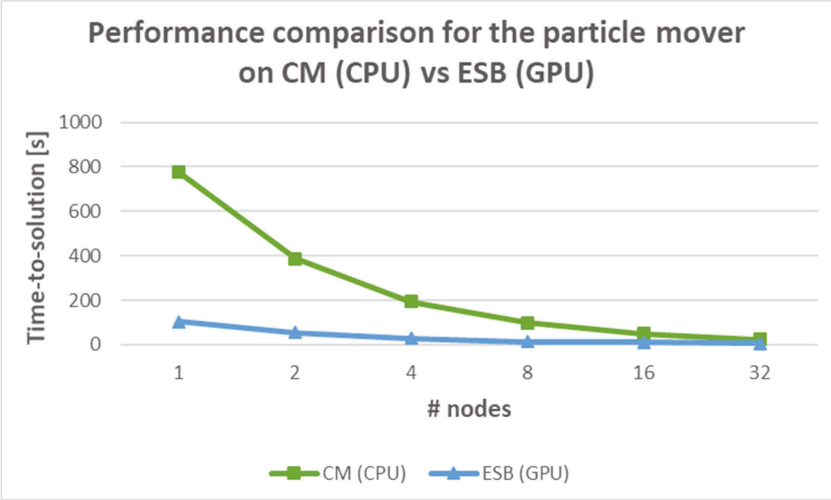


**Figure 5.28: Runtime comparison (strong scaling) of the particle mover on CPUs (CM) and GPUs (ESB)**

This performance comparison is more obvious in the weak scaling test (Section 5.7.2). The weak scaling bellow shows that the execution in the ESB and the CM presents good scalability, with runs on the GPU performing almost 10 times better than in the CPUs of the CM.

### *5.7.2 Weak scaling comparison between CM and ESB*

It is difficult to perform a fair comparison between CPUs and GPUs. The main concern is the selection of what to compare: should we compare CPU core vs GPU SMD? Or is the unit of comparison the node? Here we compare the runtimes obtained during the weak scaling tests performed in Section 5.5.5. The code xPic was executed in the MONO mode, with all the phases of the code residing in the same module. In the figures bellow, the x axis represents the number of nodes used in the CN and in the ESB. The y axis represents the runtime per CPU core and the runtime per GPU. Our goal here is not to infer that one architecture is better that the other. We are simply collecting information on the current performances of the code under the basic computing units available to the users: the CPU core and the GPU card.

In Figure 5.29 we plot the total runtime in minutes for the jobs performed in the ESB (GPU) and we compare them to the execution time on the CM (CPUs). As discussed in previous sections each GPU is shared between 8 tasks, i.e. each task has only access to 1/8th of the GPU. In a similar way on CPU core has access to only 1/24th of the computing power of the full CN node. These runtimes take into account the

execution of the field solver in the corresponding CPU of each node. The field solver has shown bad scalability, contrary to the particle solver that features almost a perfect scalability. The field solver accounts for around 80% of the execution time on the CM, and for 30% of the time on the ESB.

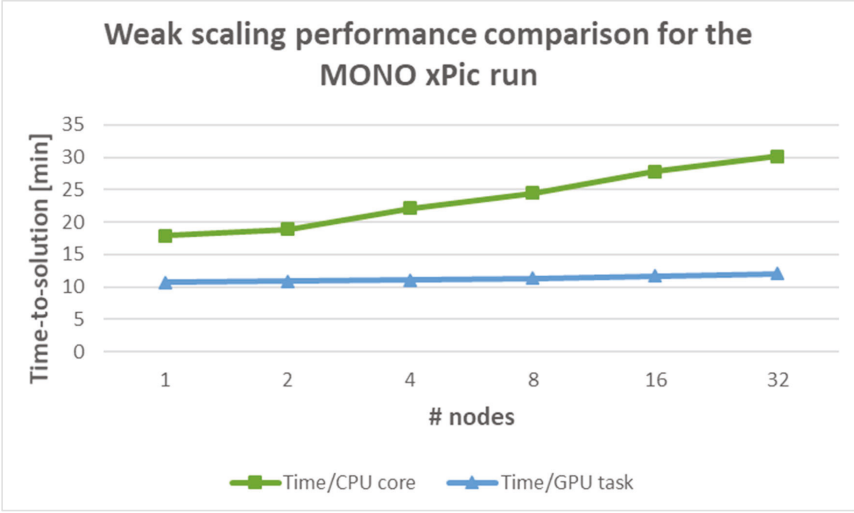


**Figure 5.29: xPic runtime comparison on CM and ESB**

In this figure each CPU core runs two to three times slower than the GPU. However, each CPU chip outperforms the execution time of a single GPU node. Notice that the CPU that manages the GPU in the ESB nodes is less powerful than the CPU processor in the CM nodes. The field solver is executed in these less powerful processors and accounts for at least 20% of the total execution time. While a single node of the CM advances the field solver in 35 seconds, the same procedure requires 130 seconds in the CPUs of a single ESB node.

## 5.8 Conclusion

As application developers for scientific software, we are constantly at the edge of new algorithm implementations. For the past few years this meant that we had to adapt our codes to different architectures: classical x86 processors, many-core architectures, and multi-core architectures. Each one of these hardware requires the developer to learn about the architecture itself and about the libraries and programming languages specific to each one of them. Now a new generation of processors (AMD, ARM, …) and a new generation of accelerators (AMD GPUs, NEC vector cards, NVIDIA GPUs, …) suggests that we need to adapt, once again all our codes to remain competitive.

We have been reluctant to use CUDA to offload computing to the GPUs. We do not want to become dependent on a single, non-European, company, which requires the use of proprietary compilers and libraries. The emergence of ROCm from AMD demonstrates that the community is eager to move forward with open source driven tools. For this reason, we favour the support and the use of OpenMP as an alternative for computing offloading to any accelerator. The EuroHPC JU program has also seen that companies like AMD are producing very competitive products, and it has invested a significant budget in the installation of pre-exascale centers based on AMD technology. This is an additional reason for us to stay away from pure CUDA implementations of our codes.

We showed in the previous sections the performances of the codes of our Space Weather application. For the past year and a half we focused our attention on the use of OpenMP to port the massively scalable particle solver of the xPic code. It consists of two phases: particle mover and moment gathering. We showed that the particle solver has an almost ideal performance when it is executed in the CM alone, in the ESB alone, and as part of the Cluster-Booster MODULAR mode, in which the field solver is executed in the CM and the particle solver is executed in the ESB.

We showed that the Multi-Process Service (MPS) of the NVIDIA cards can be safely used by application developers without compromising performances. In the MPS multiple tasks (kernels) can be executed concurrently in the same GPU accelerator. For our largest test we have used 24 ESB nodes with 8 MPI processes per GPU, while simultaneously launching 16 nodes with 24 MPI process per node in the CM. We are satisfied with the scalability efficiency of the particle solver, even as we know that some serial zones remain to be improved.

This good particle solver scalability was possible to detect and to improve thanks to the work done in the DEEP-EST project. KU Leuven has been able to track and improve segments of code which were previously identified as problematic. However, during our tests we have noticed that the performance issues have shifted towards the field solver running in the CM nodes. We will continue to perform improvements in the future.

We have used the resources of the DAM to test the scalability of the machine learning code GMM. We showed that the algorithm itself presents an almost ideal weak scaling, but the code uses MPI communications to gather the final results in a single process. Adding more and more nodes renders the MPI communications prevalent and hinders the scalability of the code. We are working on the parallelisation of the I/O in order to avoid such costly communications. Our current tests show that we can perform the machine learning analysis of the xPic particles in only a few seconds per subdomain. We are planning to launch very complex simulations with on-the-fly analysis of the

particles with the GMM algorithm at a cadence equal to the I/O of the code xPic itself. This is an extremely useful development: we do not need to save the very large particle files at every output iteration, we just perform the analysis on-the-fly and retain only the high level data analysis results from our machine learning models.

In our road towards Exascale, we believe in the continuous development of the code xPic and coupling its execution with multiple on-the-fly machine learning analysis tools. We have already applied for a pilot program with the LUMI supercomputer centre where the Cluster-Booster architecture will be deployed using AMD CPUs and GPUs. We are also very happy that the energy consumption of our CPU and GPU computing loads is equal across the modules, i.e. our code presents a good energy balance.

# 6 Earth Science with NextDBSCAN, NextSVM and Deep Learning

**Ernir Erlingsson, Helmut Neukirchen, Morris Riedel**
University of Iceland, UoI, Iceland

ere29@hi.is

## 6.1 Introduction

The University of Iceland (UoI) explored the possibilities of combining machine learning methods with the MSA offered by the DEEP-EST system.this aim, UoI selected three machine learning methods and tailored their implementations for the DEEP-EST system. These three applications are:

- NextDBSCAN, a new parallel DBSCAN algorithm used for non-approximate and approximate density-based clustering of arbitrary datasets, such as large three-dimensional point-clouds generated via LiDAR scans. Note that NextDBSCAN supersedes HPDBSCAN, which was used at the start of the project, as HPDBSCAN proved unsuitable for GPU platforms and Exascale systems due to critical inherent scaling issues. Therefore, we started from scratch and developed NextDBSCAN, a DBSCAN algorithm which exhibits good scaling properties irrespective of the input dataset and parameters. We believe that our implementation of NextDBSCAN can be employed by future Exascale HPC systems, especially if it is optimized further for such systems. We substantiate our claim in the following subsections. We have also released the application as a Free- and Open-Source Software (FOSS) to the general public via a Github repository[77].
- NextSVM, a new parallel Support Vector Machine (SVM) for supervised learning classification tasks using labelled datasets (such as remote sensing images with ground-truth). Similar to NextDBSCAN, NextSVM supersedes PiSVM, which was used initially but proved unusable as its performance scaling plateaus after only a few nodes. NextSVM, however, scales much better and supports the usage of GPU accelerators. Through the DEEP-EST project, we have managed to improve especially the model training performance and scalability towards Exascale, which we illustrate and discuss in the following subsections. We also

[77] https://github.com/ernire/nextdbscan-exa

provide NextSVM as a FOSS repository for the public and machine learning communities[78].

- Deep Learning, via TensorFlow and the Keras extension, for computer vision (including remote sensing images), using Convolutional Neural Networks (CNNs). For scalability across multiple nodes, the Horovod framework is used to run TensorFlow/Keras in a distributed fashion.

## 6.2 Application structure

### *6.2.1 NextBSCAN*

For clustering, the new parallel NextDBSCAN algorithm is used. The core partitions are those related to the pre-processing of the input data, the actual clustering algorithm (i.e. local clustering and global merge) and storing its results. The data selection partition, used for further data analysis and processing, is optional and is only used if there is further data study. The individual partitions of NextDBSCAN are described in the following sub-sections.

#### *6.2.1.1 Data processing*

In this phase, the point-cloud dataset is spatially divided using a hyper-grid overlay of different size and/or offsets. The point-cloud dataset is then divided equally among the processes and each point is sorted into its respective cell. Afterwards, the sorted list is stored, and a heuristic is applied to attempt to load-balance the data-grid by dividing it into chunks that fit in RAM, i.e. the total number of executions (and the cell span of each) is determined so that the whole grid can be processed without overwhelming the available hardware resources. The heuristic attempts in particular to divide the dataset into equally sized execution tasks with respect to the number of point comparisons.

After the data has been divided, each chunk is processed further, resulting in smaller chunks, which can then be finally processed by a local DBSCAN implementation.

#### *6.2.1.2 Data chunk pre-processing*

Upon execution, the chunk which is being processed is divided into further smaller chunks equal to the number of MPI processes, where a similar load-balancing heuristic to the one described in the section above, i.e. the hyper-grid cells are divided among processes, tries to keep the number of point comparisons for each process as close as possible.

[78] https://github.com/ernire/next-svm

#### *6.2.1.3 Local parallel DBSCAN*

Each MPI process performs a local DBSCAN clustering on its assigned cells, using OpenMP for shared memory parallelism. Hence, clusters that span different MPI processes are not yet detected in this step and, as a consequence, a merging approach is performed in the next step.

#### *6.2.1.4 Merge clusters*

After clustering, the locally obtained cluster labels are exchanged among the MPI processes to make sure that clusters spanning over multiple cells receive the same unique global cluster label. This is done with selected rules in the algorithm.

#### *6.2.1.5 Results and resiliency*

The old HPDBSCAN was not particularly robust as it did not include any measures to increase the application's resiliency. This was not really necessary because the limit on the size of the point-cloud datasets also limited the execution time to such a degree that a system failure would never be very cost intensive. For larger datasets such as those expected in the Exascale era, this must be improved.

We therefore apply a simple but effective measure to add resiliency to the NextDBSCAN application by storing calculated cluster labels to the persistent memory, taking advantage of the inherent compartmentalization of the computation offered by the dataset hyper-grid overlay. This allows the execution to restart using the most recently stored data. In effect, we are adding checkpointing to the application.

#### *6.2.1.6 Data selection*

When applying Level of Detail (LoD) or continuous Level of Importance (cLoI) studies, it is possible to modify the point-cloud, e.g. zoom in/out, and perform clustering on sub-set selections of the original dataset, possibly using a new hyper-grid overlay. This in turn may result in various clusters in memory on different datasets that may be often re-read depending on zoom levels. Therefore, it makes sense to store clustered datasets of sub-sets into persistent memory for further iterations of LoD/cLoI studies.

### *6.2.2 NextSVM*

This second machine learning application performs classification of data using the parallel SVM implementation called NextSVM. It can be divided into four partitions described in the following subsections.

#### *6.2.2.1 I/O*

For training, the feature engineered labelled HDF5 input dataset is read in parallel by numerous processes. The input dataset has been feature engineered to increase the

likelihood that the model converges, and to reduce the overall computation time by skipping features that have otherwise little or no effect on the training process. Therefore, feature engineering is usually performed with the goal of increasing the accuracy. But as different feature engineering techniques are usually applied, the input datasets may in fact change from time to time even though the raw data is the same.

#### *6.2.2.2 Training*

NextSVM training is performed iteratively on the non-linear input data, processing one sample at a time using sequential minimal optimisation (SMO), but using the so-called "kernel-trick" to linearly separate the data in a higher dimension space. The aim is to construct a model which can be used for classification with a high accuracy. This phase is computationally expensive, generally requiring very many samples to be able to achieve good classification accuracy.

#### *6.2.2.3 Validation*

Validation is a process in machine learning for model selection that in turn is not only related to the right model (e.g. SVM, neural network, Random Forest, etc.) but also their parameters. We are using a non-linear SVM with RBF (Radial Basis Function) kernels (having a kernel parameter *gamma*) and soft margins (i.e. allowed cost of *error* parameter), therefore an exhaustive search must be made, e.g. using a 10-fold cross-validation, to determine those input training parameters that give the best training results. This is typically performed via a grid search over the parameters and is a process that is embarrassingly parallel, i.e. parallelises nicely: depending on the number of parameters, the overall computing time could be quite significant, but the different runs do not require interaction between them.

#### *6.2.2.4 Inference*

Model inferencing in NextSVM is an embarrassingly parallel operation that performs predictions using an otherwise unseen labelled dataset, which can be used to determine a model's accuracy. Furthermore, when a model exhibits good accuracy, it can then finally be used for making classifications on new unseen datasets.

### *6.2.3 Deep learning*

The third machine learning application does classification using deep learning. It uses partly the same dataset as the SVM application for supervised learning to allow for a comparative study of the different classification approaches. For unsupervised learning, however, different multi-spectral datasets are explored. The application uses state-of-the-art deep learning for image pattern recognition, namely Convolutional

Neural Networks (CNNs), that are known for detecting spatial properties in data. The partitions of the deep learning chain are described in the following sub-sections.

#### *6.2.3.1 I/O*

As mentioned above, the application can, in principal, process and classify the same input datasets as the SVM application. However, it uses the raw, non-feature-engineered datasets whilst SVM uses a processed, feature-engineered version of it. The reason is that 'feature learning' is an intrinsic part of deep neural networks in general and CNN in particular. In the future, other datasets will be used, e.g. to support Sentinel satellite data provided by the European Copernicus remote sensing programme. This dataset offers enormous volume, with over 23 Terabytes of new data per day, and requires Exascale computing when performing land cover analysis at large scale over time.

#### *6.2.3.2 Training*

Training is performed using CNNs since we are mostly handling remote sensing image input data for which CNNs perform best. Additionally, Stochastic Gradient Descent (SDG) and back-propagation are used as standard techniques employed during the training phase. Due to the multi-spectral nature of the input datasets, a 3D CNN is used, which is a special form of regular CNNs that can better take advantage of the multiple input data dimensions (2D spatial data with multiple spectra).

#### *6.2.3.3 Inference*

The trained models acquired in the previous partition are evaluated by measuring their prediction accuracy on previously unseen, but labelled input data and thus inferring their suitability for further training. Finally, the trained models can be used for making classifications on new (and even unlabelled) datasets.

#### *6.2.3.4 Transfer learning*

After a neural network model has been trained and tested, that model can be re-used, even in parallel by multiple users, as a foundation for additional training on other datasets using transfer learning techniques. The simplest technique involves making an incision in the neural network next to the output layer and adding more layers in-between, prior to fresh training. There are also known existing pre-trained neural networks (e.g. OverFeat) that make sense to have available in the persistent memory for different application use cases. Each network mostly consists of a matrix of weights in multiple dimensions (i.e. for each layer), but memory-footprint can nonetheless be significant when deep learning architectures are used.

## 6.3 Application mapping

For each of the three applications, we selected multiple mappings to the MSA of the DEEP-EST system, as it was not clear in advance which path would offer each application the greatest benefits.

While the initial assumption was that NextDBSCAN benefits from a hybrid usage of CPU and GPU and therefore would use either CM together with ESB or CM together with DAM, the NextDBSCAN algorithm, data structure, and implementation has since then been improved so that NextDBSCAN runs fastest when using solely CPUs or solely GPUs. Measurements (see Sections 6.5 and 6.6) have shown that it depends on the dataset and the DBSCAN clustering parameters whether using CPU or GPU is preferable (a hybrid approach suffers from a data transmission overhead), although the GPU provides a clear benefit for the vast majority of examined cases. However, NextDBSCAN can run on the CPUs of CM or DAM (benefiting from the huge RAM in the DAM) or on the GPUs of ESB or DAM. Depending on the actual use case, only one MSA module might therefore be used, but to give an idea of a more complex workflow and mapping to the MSA, Figure 6.1 shows an example where a grid parameter search is performed in parallel on all MSA modules, i.e., doing DBSCAN clustering with different values for its two parameters to find out which parameter combination yields best clustering results. By analysing the dataset (e.g. point density), it is first determined which parameter combination is best executed on which MSA module and after that, all available CPUs and GPUs of the MSA modules can be used for the embarrassingly parallel computation using the different parameters on the same dataset (“ensemble scaling”). Optionally (dashed lines in Figure 6.1), it is possible to re-run the clustering with a narrowed down dataset selection or a different parameter range selection.

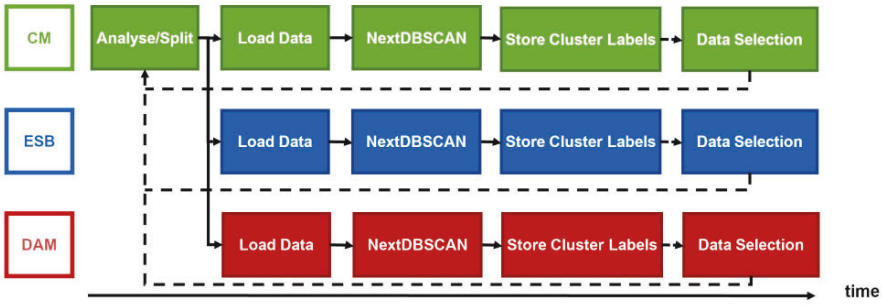


**Figure 6.1: Schematic workflow of a grid-search using NextDBSCAN in the MSA**

In addition, the MSA usage of NextSVM has been adjusted while implementing NextSVM. For model inference (i.e. testing the trained model), only a few GPUs and

not a lot of RAM are needed; hence, the DAM (i.e. fewer GPUs) or the ESB (i.e. smaller RAM) can be used. Figure 6.2 depicts in the upper part a mapping where model training is performed using the many CPUs of the CM and model inference is then done using the ESB's GPUs and the model is locally stored in the ESB. The lower part depicts model training using the many GPUs of the ESB and then model inference on the DAM using also DAM's DCPMM for model storage. (These stored models can then be re-used for additional training via transfer learning and/or further inference.) An alternative, CPU-based mapping has been described in a research paper[79].

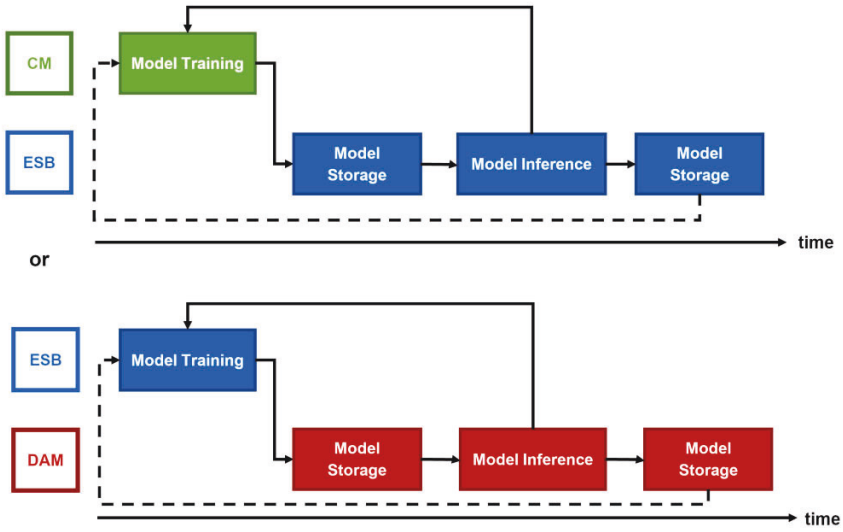


**Figure 6.2: Two different schematic workflows of NextSVM in the MSA**

UoI's Deep Learning application also explores two different MSA mappings to support performance comparisons: both mappings use the same MSA modules, namely the ESB and DAM (see Figure 6.3). The main difference between these "mirrored" mappings lies in which MSA module trains the neural networks, and which infers their quality. After training, the obtained models are stored at two locations, where one supplies the inference with input data and the other is enhanced by metadata and can be used for any subsequent training, e.g. transfer learning.

[79] Ernir Erlingsson, Gabriele Cavallaro, Morris Riedel, Helmut Neukirchen. Scaling Support Vector Machines Towards Exascale Computing for Classification of Large-Scale High-Resolution Remote Sensing Images. IEEE International Geoscience and Remote Sensing Symposium (IGARSS) 2018. DOI: 10.1109/IGARSS.2018.8517378 IEEE 2018.

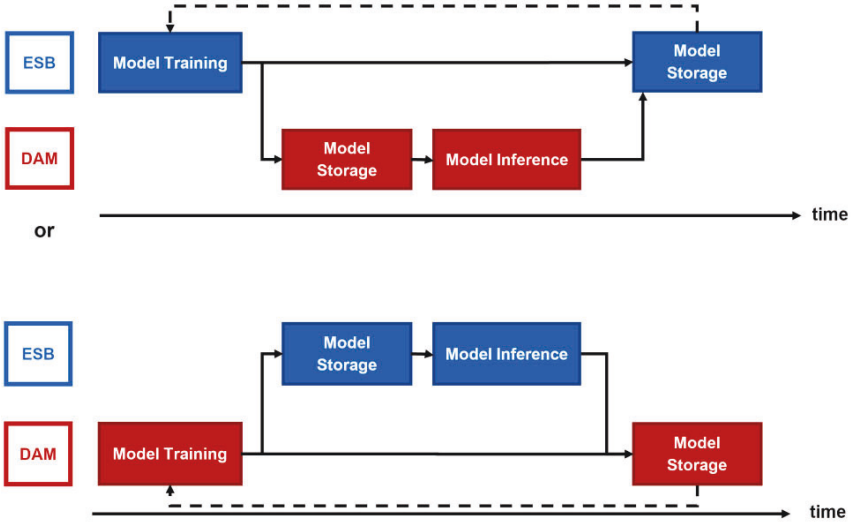


**Figure 6.3: Two different schematic workflows of deep learning application in the MSA**

Benefits of using of the NAM prototype has been investigated and published in research papers[80, 81].

## 6.4 Porting experience

At the start of the DEEP-EST project, we expected the porting procedure for our applications to be straightforward. However, this assumption proved false as we discovered a critical scaling problem with our initial Support Vector Machine application (PiSVM), which is explained in Section 6.7. Furthermore, we found that our initial density-based clustering application (HPDBSCAN) was inherently incompatible with the use of GPUs, as it was per-design unable to exploit the parallelism offered by accelerators. Therefore, our porting efforts revolved mostly around re-designing these applications from the ground-up, thereby producing NextSVM and NextDBSCAN, to provide necessary compatibility with distributed memory GPUs and the MSA. Our deep learning application implementation, however, was virtually unaffected by the project's

[80] Ernir Erlingsson, Gabriele Cavallaro, Morris Riedel, Helmut Neukirchen. Scalable Workflows for Remote Sensing Data Processing with the DEEP-EST Modular Supercomputing Architecture. IEEE International Geoscience and Remote Sensing Symposium (IGARSS) 2019, DOI: 10.1109/IGARSS.2019.8898487, IEEE 2019.

[81] Ernir Erlingsson, Gabriele Cavallaro, Andreas Galonska, Morris Riedel, Helmut Neukirchen. Modular Supercomputing Design supporting Machine Learning Applications. International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO 2018), DOI: 10.23919/MIPRO.2018.8400031, IEEE 2018.

focus shift towards GPUs, due to its high-level implementation in TensorFlow and Keras that anyway supports CPUs and accelerators.

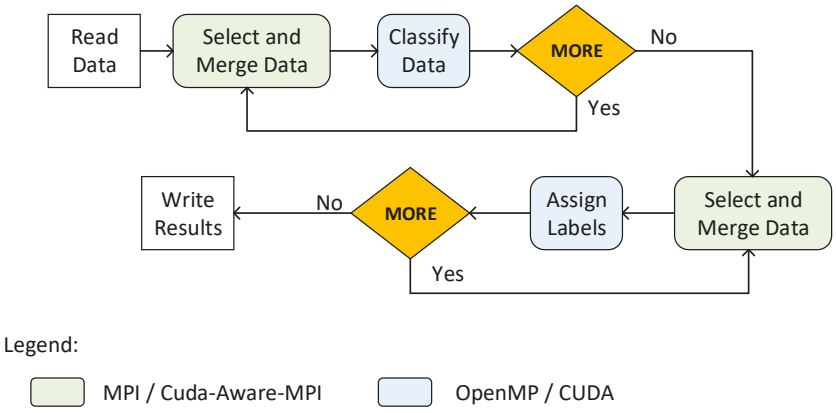


**Figure 6.4: An algorithmic flowchart of NextDBSCAN and its numerous versions (OpenMP, CUDA, MPI)**

Codebase fragmentation was one of our key concerns as we had to develop multiple application versions to support both CPUs and GPUs. This increases the development time, source code size, and the risk of human errors, which could endanger the quality of the application and its performance results, e.g., when the CPU and GPU versions produce different results due to an error in one of them, or both. To mitigate this problem, we decided to create a single cross-platform version of our application, which can exploit the project's CPUs and GPUs (see Figure 6.4 for a single flowchart of NextDBSCAN supporting MPI, OpenMP, and CUDA). To this effect, we developed a library (called Magma) that mimics the C++ standard library blueprint, and subsequently wrote both NextDBSCAN and NextSVM with it. The main benefit of the library is that it takes care of linking the source code to either the C++ STL (in case of CPU) or CUDA Thrust (in case of GPU) libraries, as depicted in Figure 6.5.

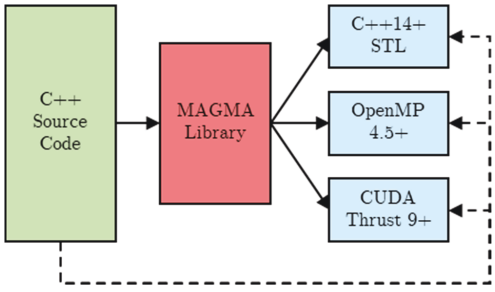


**Figure 6.5: Magma Library Schematic Overview**

Technically, Magma is a C++ header library that makes extensive use of C++ templates to offer compile-time polymorphism for increased usability at the expense of a small compile-time overhead. Specific compiler flags dictate which header files are used, and therefore which internal libraries are used. Currently our Magma library encapsulates the C++ STL, OpenMP 4.5+, and CUDA Thrust 9+. However, it can be expanded to cover more software libraries as there is nothing in its inherent design that limits the number of supported internal libraries. The Magma library is available to all as FOSS via a public GitHub repository[82].

```
magma::for_each(n_offset_size, v_coord_cell_size.size() - 1, [=]
#ifdef CUDA_ON
    __device__
#endif
    (auto const &i) -> void {
        it_coord_cell_size[i] = it_coord_cell_offset[i + 1] - it_coord_cell_offset[i];
    });
```

**Figure 6.6: An example of the usage of a Magma library for each-loop, taken from NextDBSCAN source code using C++ pass-by-value lambdas**

By developing NextDBSCAN and NextSVM using the Magma library we could construct a single code-base for each application while still supporting the C++ STL, OpenMP and CUDA (via Thrust). The source code is identical for both the CPU and GPU platform with the exception of the necessary host and/or device annotations which CUDA requires to specify the execution target, as is outlined in Figure 6.6. C++ functors and/or lambdas can be used to specify kernels with or without parameters, which are copied to the kernel (or passed by value). This greatly facilitated the development of our applications by allowing us to target multiple platforms, while simultaneously maintaining a single, compact, code-base accompanied with unit tests. The reusable Magma library is ~2000 lines of code. NextDBSCAN is ~1500 lines, NextSVM is ~1000 lines, and our deep learning scripts ~500 lines of high-level Keras/TensorFlow code (which does not use Magma). The majority of our PMs went into the software development of our applications and the Magma library, following an iterative development process including in each iteration analysis, design, development, and testing. New versions were conceptualised and often scrapped when they proved inadequate. It is difficult to quantify the time spent directly on porting to the DEEP-EST MSA as it was one part of a bigger scope of developing NextDBSCAN and NextSVM from scratch, independently from the MSA. However, we estimate that a quarter of our development process can be attributed to MSA porting, as part of analysis, design and testing. However, developing an application from

[82] https://github.com/ernire/magma

scratch is not the same as porting exiting applications with good scalability onto the MSA. We believe that with the knowledge we now possess, we could retrofit any such an application onto the MSA in a matter of weeks, followed by an arbitrary amount of time for optimisations.

## 6.5 Scalability

We examined the scalability of our three applications with numerous modular benchmarks and present the highlights of our findings in this section. In Subsection 7.3.1., we also outline the path of our applications towards Exascale. Note that we use the term parallel efficiency according to standard practice, i.e. the speedup of the respective number of nodes when compared to using a single node, divided by the respective number of nodes. For our experiments, we used different dataset suites with multiple input parameters, where applicable, and re-ran each experiment three times, reporting the median of the measured results to remove the effect of outliers. We applied NextDBSCAN on large LiDAR point-cloud datasets, i.e. the Dutch AHN3[83] and Bremen[84] datasets. For NextSVM, we employed the Rome[85] and Indian Pines[86] datasets with their accompanying classification maps. Finally, we trained neural networks with deep learning using Sentinel-2 imagery tiles[87].

Figure 6.7 depicts NextDBSCAN's strong scaling properties when measuring the time-to-solution (TTS). We observe that for an equal number of nodes, the ESB significantly outperforms the CM (GPU vs. CPU), consistently reporting ~50x faster time-to-solution (TTS). These results were consistent for all experiments with big data. However, for smaller datasets the runtime difference between the modules shrunk as a function of its size, as the amount of parallel computations simply are not enough to sustain its scalability with GPUs, and the application's serial processing overhead becomes more and more dominant.

NextDBSCAN's parallel efficiency is depicted in Figure 6.8. On the CM, it remains relatively high for this strong scaling case, but drops more quickly on the ESB. We examined the cause and determined that most of it stems from each ESB node spending much less time doing computations compared with its CM counterpart, but a near equal amount performing MPI communications. Therefore, the MPI communication's latency and inherent scalability affects the parallel efficiency to a

[83] https://downloads.pdok.nl/ahn3-downloadpage/

[84] http://doi.org/10.23728/b2share.7f0c22ba9a5a44ca83cdf4fb304ce44e

[85] https://b2share.eudat.eu/records/daf6c389e54340b4b1416cf874251e77

[86] https://b2share.eudat.eu/records/8d1fbbba69944fc5a5ae01d1c141c37a

[87] https://scihub.copernicus.eu/

higher degree on the ESB. By increasing the problem size, the ESB's parallel efficiency remains higher as the computational load grows then faster than the distributed communication. However, for our comparison, we already selected the largest possible problem size that a single CM node can solve within a reasonable time duration.

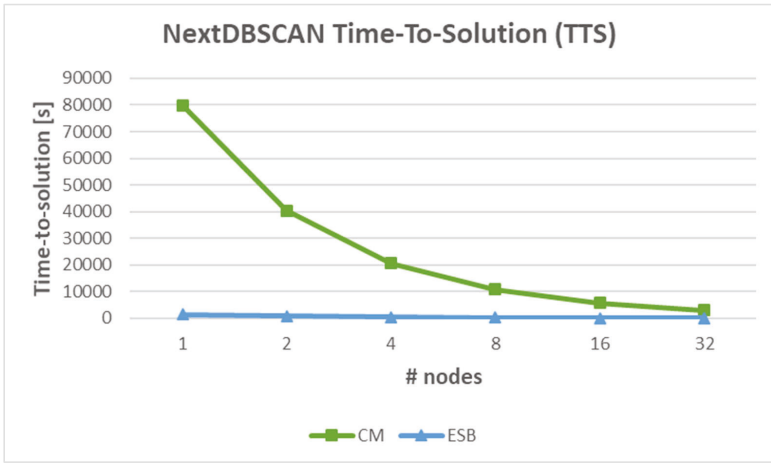


**Figure 6.7: NextDBSCAN's time-to-solution, measuring strong scaling on CM (CPU) and ESB (GPU)**

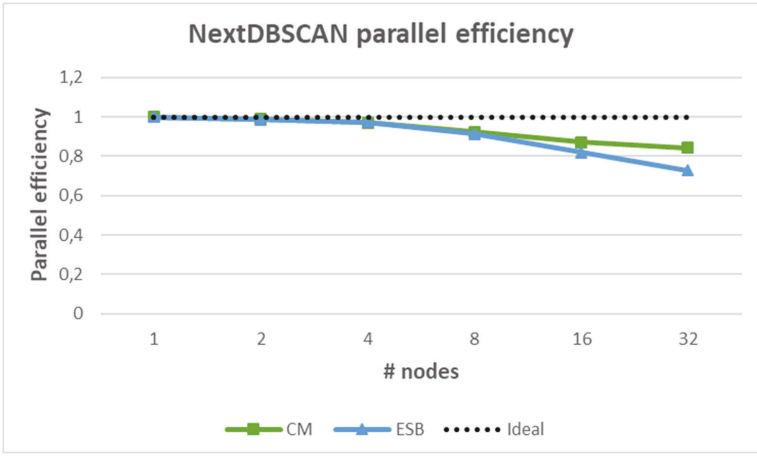


**Figure 6.8: NextDBSCAN's parallel efficiency, measuring strong scaling on the CM (CPU) and ESB (GPU)**

By using a heterogeneous approach, as is depicted in Figure 6.1, we were able to slightly improve the ensemble TTS performance of a typical grid-search, which executes a shared-memory version of NextDBSCAN concurrently on multiple nodes, using a different pair of parameters. Figure 6.9 shows the aggregated runtime values

using six different epsilon values, where each value represents eight different minPoint values, i.e. each column represent the total runtime of eight different doubling parameter pairs which share the same epsilon. The GPU performed better for most, but not all, parameter pairs, which leads to the optimal performance of the total aggregate being a heterogeneous combination using both CM and ESB.

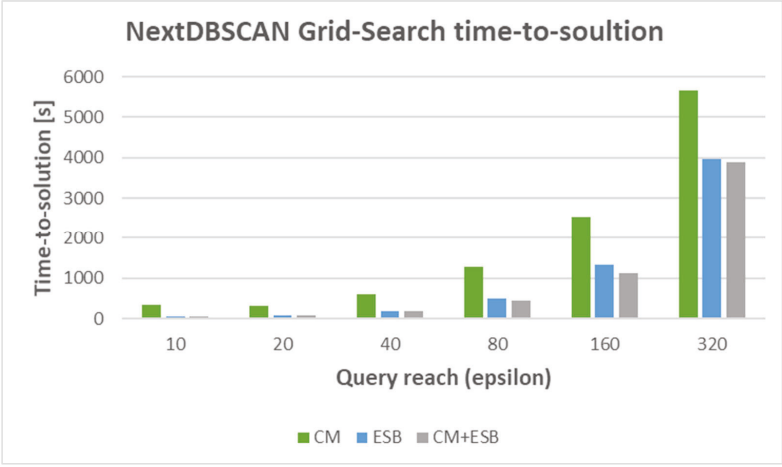


**Figure 6.9: Grid-search runtimes aggregated for each epsilon value, which doubles every iteration**

Figure 6.10 illustrates our main finding when using NextVSVM, measuring the time-to-solution for model training. We observe that when running NextSVM on a single-module the ESB starts with a faster baseline performance but then drops behind the CM after only 16 nodes. Here, we are implicitly comparing CPUs vs. GPUs, as NextSVM is in this scenario only using the GPU on the ESB. NextSVM, however, uses a Sequential Minimization Optimizer (SMO) solver that enforces a strong serial order of computations for a small part of the iterative algorithm. This part is bound by single core performance, which forms a severe bottleneck for NextSVM running exclusively on the GPU. The best time-to-solution performance was achieved by using both the CPU and GPU on the ESB module, as depicted by the light blue line, despite having higher offloading costs, as data is transferred more frequently between the GPU and CPU memories. Overall, 99% of the execution takes place on the GPU, but the remaining 1% CPU-time is critical to better maintain the speedup curve, as we can observe in the figure. Although the CM offers CPUs with a stronger single-core performance than the ESB, we still attained the best results using only the latter, as the gain with stronger cores did not overcome the cost of transmission at each iteration, mainly due to the very low number of necessary computations.

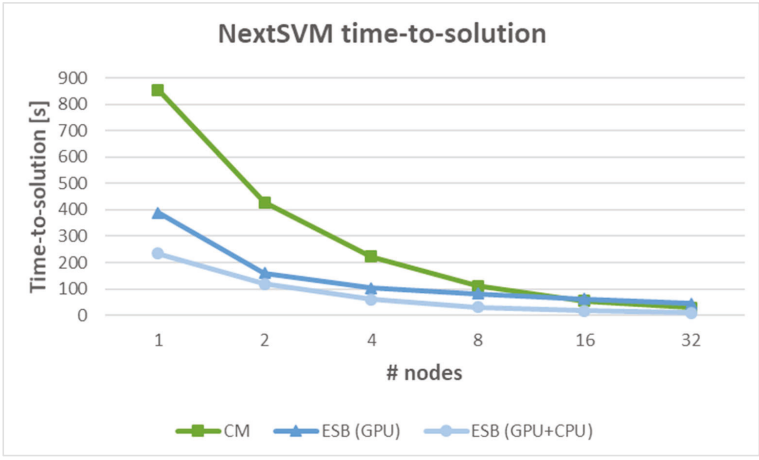


**Figure 6.10: NextSVM's strong-scaling time-to-solution, measured on the CM and the ESB, where the former uses CPUs and the latter solely GPUs**

Figure 6.11 shows the parallel efficiency of the two NextSVM versions that exhibit the best scaling properties, i.e., running on the CM, or ESB using both CPU and GPU. As the figure illustrates, the application running on the CM maintains its scalability better than the ESB. The main cause of this discrepancy is due to the offloading cost, as data is transferred to and from the GPU, which has a fixed size irrespective of the number of nodes, a consequence of the SMO solver algorithm employed by NextSVM (see also later discussion on fixing limitations).

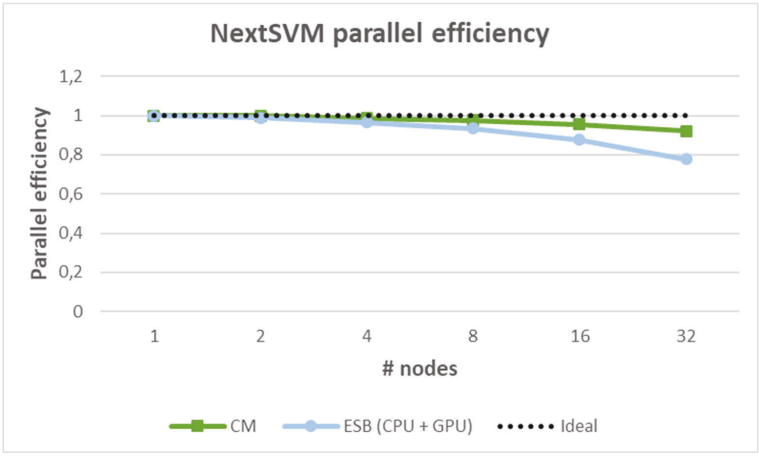


**Figure 6.11: NextSVM parallel efficiency while performing strong scaling on the CM and the best ESB scaling with CPU + GPU**

Finally, Figure 6.12 and Figure 6.13 depict the strong scalability on the ESB module while training a neural network with satellite imagery, using Keras/TensorFlow and the Horovod framework, for distributed computing. The first figure shows the effect of modifying the image size for each training batch, i.e. it is possible to cause some variations to the scalability curve by loading different batch sizes at once (using HDF5). However, when scaling up the number of nodes this variance will decrease over time, as is illustrated by the parallel efficiency. Overall, our experiments with parameters and I/O configurations had an insignificant effect on the training performance and scalability. Additionally, Horovod's scalability was worse than what we anticipated, making us doubt its suitability as a deep learning vehicle for satellite imagery on Exascale systems (see also later discussion on fixing limitations).

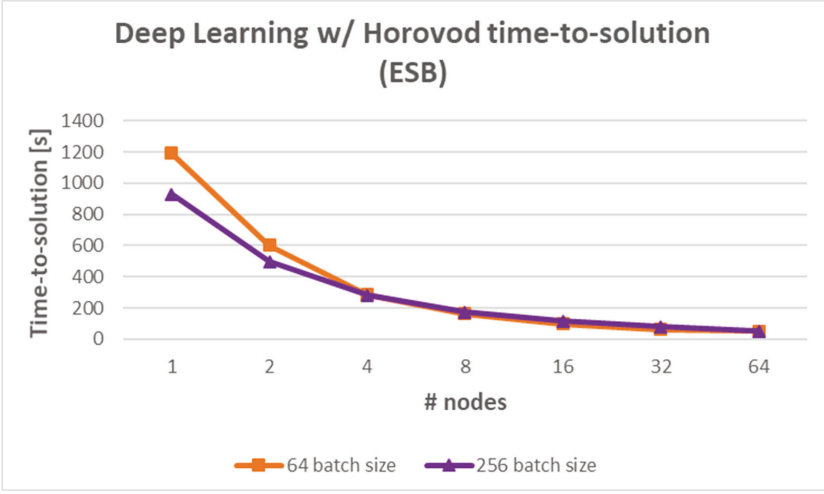


**Figure 6.12: Deep Learning w/ Horovod strong-scaling time-to-solution using different training batch sizes**

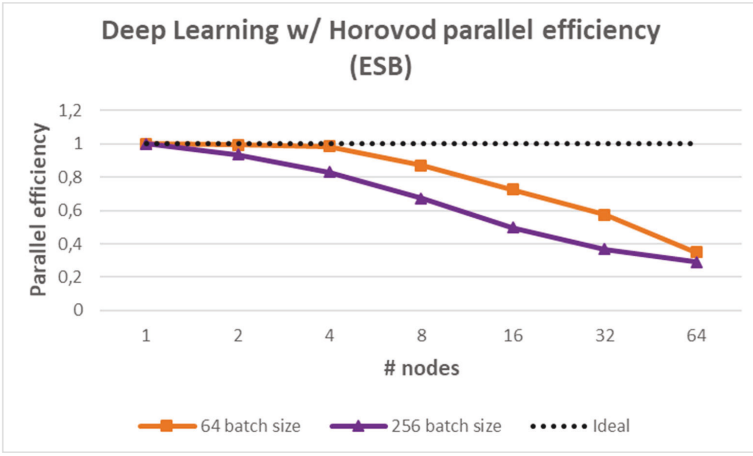


**Figure 6.13: Deep Learning w/ Horovod parallel efficiency using different training batch sizes**

### *6.5.1 Our path to Exascale*

In summary, we achieved the above results in the following manner:

- We redesigned and reconstructed the MPI communication of our DBSCAN and SVM applications, optimizing them for Exascale HPC systems. We abandoned the master-slave paradigm such as was employed by PiSVM (where a single node controls the execution process and collects all the data) and replaced it with MPI collectives where each node's role is identical. We achieved the best results by limiting our applications to in-place `MPI_Allgather` and `MPI_Allreduce` communication, as much as possible.
    - For NextDBSCAN, we inferred that our initial usage of `MPI_Alltoallv` would scale significantly worse than using in-place `MPI_Allgather` with a fixed buffer size, i.e., using buffer padding where necessary. This was due to internal processing of the send and receive count buffers, which increase linearly with the number of nodes, coupled with the overhead of each node communicating its buffer size.
    - For NextSVM, we completely re-designed the communication strategy towards MPI collectives instead of point-to-point transmissions, using in-place `MPI_Allgather` with a fixed buffer size.
- By minimizing the number of memory allocations, we were able to significantly improve the shared-memory parallel efficiency of our applications. This was especially effective for NextDBSCAN where we managed to replace all

intermediate dynamic buffers with a single fixed buffer allocated at the start of the execution.

- Using the GPU accelerators, we were able to greatly reduce the time-to-solution (TTS) for our applications, with an even lower cost of energy, as depicted in Figure 6.14 and Figure 6.15.

#### *6.5.1.1 What are the limitations? – Can they be fixed?*

- NextDBSCAN is ready to be applied to Exascale systems without special limitations. The only requirement is that the aggregated memory is sufficiently large to store the input dataset. To the best of our knowledge, NextDBSCAN is the first non-approximate DBSCAN application that is a viable candidate, and the first to support distributed GPUs.
- NextSVM can also be applied to Exascale systems, but has limited usability as it currently only supports a single linear kernel and does not include the option of cross-validation. Note that our scalability results are kernel agnostic. To fix this, more development time is needed, and it is our hope that our public repository can attract additional open-source developers.
- Deep Learning model training with satellite imagery and Horovod failed to meet our expectations. There is no easy fix, as Horovod would have to be probably partially re-written to meet the demands of Exascale systems. Given the rapid progress of multiple deep learning libraries and frameworks these past years, it is not unlikely that another solution, better suitable for Exascale systems, is nascent.

#### *6.5.1.2 How to use future Exascale systems*

- NextDBSCAN can be used as-is for both CPU and GPU clusters. Its source code is compartmentalized to facilitate its usage for heterogeneous systems. However, our evaluation in DEEP-EST indicates that an arbitrarily sized and homogenous GPU cluster should be used for fastest results.
- NextSVM can also be used as-is for both CPU and GPU clusters using standard compilers and software stacks. However, according to the scalability results presented in Subsection 6.5, it is only realistically applied at a large scale to CPUs, not GPUs.
- Our research indicates that deep learning with Horovod is not a good fit for Exascale systems. However, its development continues and future versions, or other uses, could provide a more realistic option.

#### *6.5.1.3 Where did the DEEP-EST project help on the way to Exascale?*

The DEEP-EST project was instrumental in enabling us to improve our applications, both their performance and usability.

- The access to state-of-the-art hardware and software resources was critical to the development and optimization of our applications.
- The tools and workshops provided by BSC, combined with expert help from the consortium (in particular JSC, BSC, EPCC, and Intel), gave us the insight we needed to make critical decisions to improve the scalability and usability of our applications. Tracing and profiling with Extrae and Paraver, respectfully, visualized the scalability problems of PiSVM, as outlined in previous deliverables, which prompted the development of NextSVM to supersede it.
- The shift towards GPUs helped us improving scalability of our applications as it revealed bottlenecks which were more difficult to detect with the same number of CPU nodes. As an example, we discovered that we had underestimated the scalability impact of some short sequential code areas in NextDBSCAN. The strong single-core performance of the CPU coupled with the small number of nodes (relative to Exascale) had obscured the fact, but with GPU parallelism the adverse impact of the sequential code areas became apparent and after careful improvements we managed to eliminate them from the source code.
- The DEEP-EST MSA provided us with the modules we needed to study our applications across different CPUs, interconnects, and accelerators, which strengthened our performance claims. Additionally, it allowed us to design novel workflows across different hardware platforms.

## 6.6 Energy consumption

The total energy consumption was measured using the resources at our disposal in the DEEP-EST project. Figure 6.14 illustrates the aggregate energy consumption of NextDBSCAN, measured alongside strong scaling benchmarks depicted in Figure 6.7. Note that NextDBSCAN on the ESB runs near-exclusively on the GPU, using CUDA-aware MPI, requiring the CPU only for file system I/O. We observe that the difference between the CM and ESB module's energy consumption is similar to the runtime results. However, the difference is slightly larger for the energy consumption, as the ESB uses up to 60× less energy than the CM; i.e. for 32 nodes: 327k vs 18M (Joule), respectively. As the difference increases slightly with the number of nodes, it can be expected that the difference will be even greater for a higher number of nodes, although this hypothesis should be validated on larger systems than the DEEP-EST system.

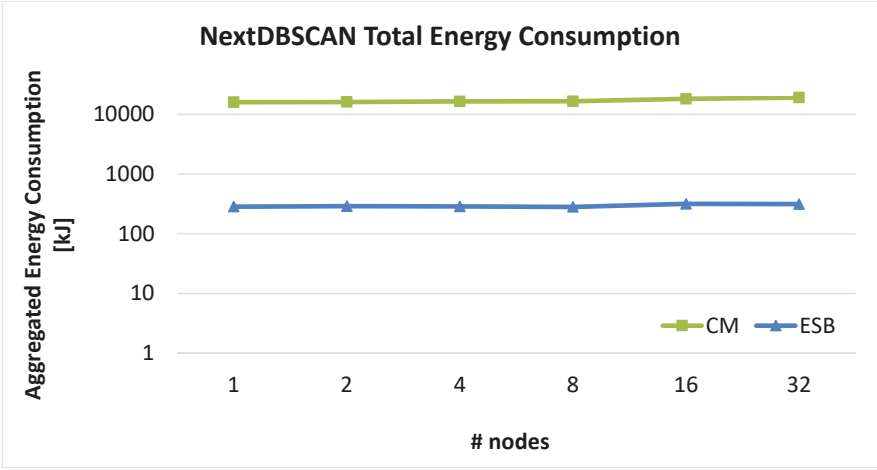


**Figure 6.14: NextDBSCAN energy consumption using CM (CPU) and ESB (GPUs)**

The total energy consumption of NextSVM is depicted in Figure 6.15. There is some correlation to the runtime of Figure 6.10: we can observe that the energy consumption increases for both the CPU and GPU as its scalability starts to flatten. We also note that the ESB's energy consumption increases faster than the CM, also corresponding to Figure 6.10. Additionally, we can see yet again that the runtime with GPUs requires less energy than using CPUs, however, this should flip when using more nodes, as the GPU runtime will scale progressively worse.

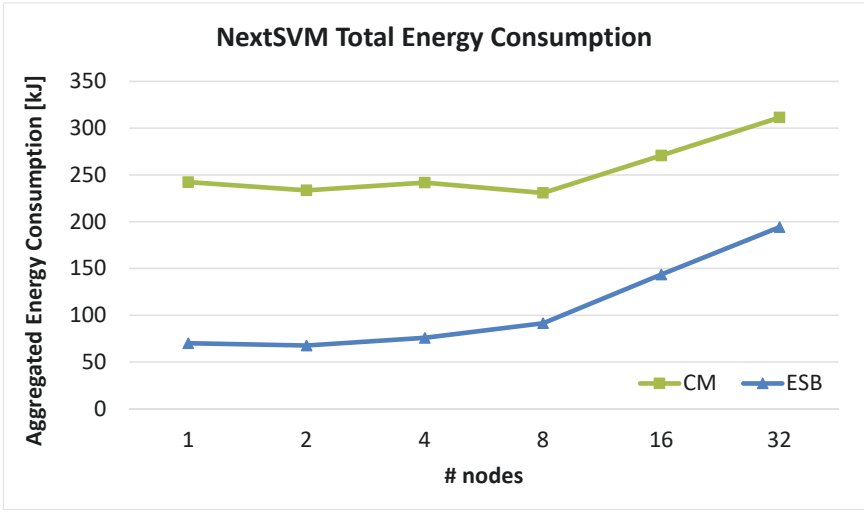


**Figure 6.15: NextSVM energy consumption using CM and ESB (GPUs)**

## 6.7 Performance comparison

After over three years of development, this subsection compares our current application status with their status at the start of the DEEP-EST project. Most of our effort has been spent on developing NextDBSCAN and NextSVM, which have already been outlined in previous sections. Their effectiveness is best demonstrated by comparing them to their predecessors, HPDBSCAN and PiSVM respectively. One of the greatest weaknesses of our initial applications was their surprising lack of scalability, as shown in Figure 6.16, which can in turn be compared to Figure 6.8 and Figure 6.11 to see the effect that DEEP-EST has had on our DBSCAN and SVM applications.

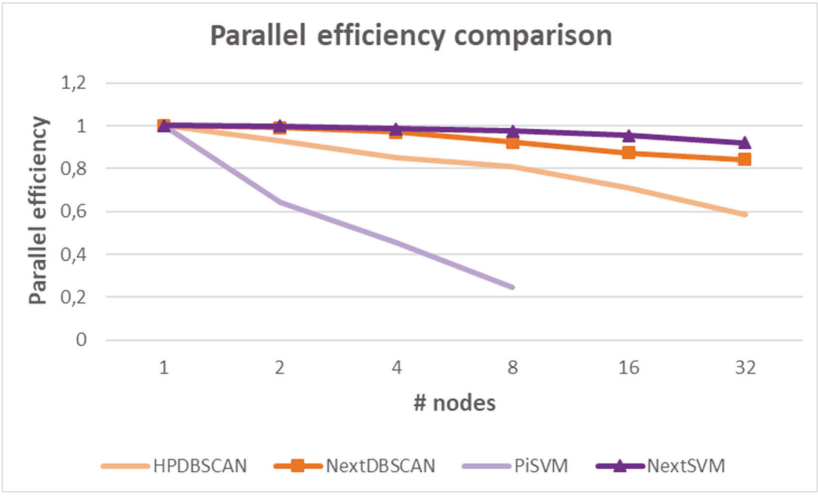


**Figure 6.16: Strong scaling parallel efficiency for HPDBSCAN and PiSVM**

For PiSVM the parallel efficiency was consistent for model training, mostly irrespective of the dataset used. After studying the application and its underlying algorithms, we found that the problem is caused by a poor distributed communication strategy in combination with too few computations being performed at each iteration. These problems had not yet surfaced, as no large scale performance measurements had been performed on the application for several years, and slower CPUs managed to mask the problem sufficiently by spending a larger portion of the total runtime doing computations, and less doing communication. PiSVM uses the well-known sequential minimal optimisation (SMO) solver, which solves the quadratic programming (QP) problem. The QP problem arises during the training of support vector machines by breaking the problem down to its smallest possible sub-problems which involves optimizing a sequence of pairs of Lagrange multipliers from the problem's dual form expression. This solver is the backbone of most SVM application produced in the last

two decades, where the computation is parallelised by distributing necessary computations to optimize one pair of Lagrange multipliers among the computational nodes being employed. Therefore, the amount of computation which can be achieved from a single pair of Langrangian is limited.

For HPDBSCAN, the parallel efficiency is more mixed, ranging from good to poor, depending on the dataset and input parameters but always degrading fairly rapidly in scalability with an increase in node cardinality. We analysed its code and could determine that the problem occurs in a grid-based data structure which is tightly integrated in the application, using it with heuristics both for data redistribution and thread load-balancing. However, this structure always leads to an imbalance, especially in datasets of high-dimensionality and/or when using a large number of nodes. Additionally, we found that HPDBSCAN's overall usability is limited by design, as it employs a tessellation indexing structure whose range is limited to 64-bit integers, i.e. the total number of possible cells cannot exceed the maximum value of an unsigned 64-bit integer. Although this is a large number, it is easily exceeded for even low dimensional datasets with a low epsilon input parameter.

The Deep Learning application with Horovod is a new application in the form of a high-level Keras/TensorFlow script, and could therefore not be compared to a previously developed application, as we could do with our DBSCAN and SVM applications.

## 6.8 Conclusion

Work in the DEEP-EST project took a very different path than originally anticipated. Instead of spending most of our effort tailoring already proven applications to the project's MSA platform, their intrinsic scalability and portability limitations led to the necessity to scrap most of them and start from scratch. However, our new applications, NextDBSCAN and NextSVM respectively, are much stronger than the previous applications. We discovered new algorithmic approaches to enhance the scalability of our algorithms and their baseline performance, consistently outperforming the best available algorithms that are freely available to perform the same task.

To our knowledge, NextDBSCAN is the first non-approximate DBSCAN clustering algorithm supporting distributed GPU computing. Furthermore, it also exhibits good scaling properties, as we have shown in previous sections. Our research indicates that NextDBSCAN is a candidate application for Exascale systems, using both CPUs and GPUs. With NextSVM, we managed to refit the old and proven SMO solver to parallel systems, with good scaling using CPUs, and show that there is potential to use GPUs to accelerate even via a heterogeneous approach.

Our results using the Horovod framework with TensorFlow on DEEP-EST are underwhelming and demonstrate that more work must be done in order for it to reach Exascale system potential. Additionally, the landscape of deep learning is still changing rapidly and it is difficult to predict which deep learning library will be utilized on future Exascale systems.

Overall, the GPU accelerator was a key component on our path towards Exascale: all our applications gained a significant performance benefit by employing it, also in terms of less energy used. Last but not least, we also published numerous research articles to further research in HPC and machine learning, with some of our greatest algorithmic findings, namely NextDBSCAN and NextSVM, still pending publication.

# 7 High Energy Physics with CMSSW

**Viktor Khristenko, Maria Girone,**
European Laboratory for Particle Physics, CERN, Switzerland
viktor.khristenko@cern.ch

## 7.1 Introduction

The Compact Muon Solenoid (CMS) detector located at the Large Hadron Collider from CERN is a general-purpose particle detector consisting of several components: tracker, electromagnetic and hadronic calorimeters, magnet and muon systems. Each component (usually addressable as sub-detectors) accomplishes a different task. For instance, tracker (both Pixel and Strip parts) is the closest sub-detector to the interaction point and responsible for identifying the trajectories of charged particles. Calorimeters measure energy depositions of the particles passing through.

Two different applications were evaluated on the DEEP-EST Modular Supercomputer Architecture (MSA): CMS event reconstruction and CMS event classification. CMS event reconstruction refers to the Compact Muon Solenoid Software framework (CMSSW) data processing pipeline aiming to reconstruct a full LHC collision event. CMS event classification is an analytics workflow, which aims to train several Machine Learning (ML) models and perform a multi-event classification

## 7.2 Application structure

### *7.2.1 CMS event reconstruction*

The process of reconstruction consists of three consecutively applied stages: digitization, local and global reconstruction. Each stage has a mix of GPU and CPU based algorithms.

#### *7.2.1.1 Digitization*

Upon recording the response of the CMS detector, physics data is packed in a highly efficient binary format that requires unpacking before it can be dealt with. The actual content of this format is the raw electrical signals that correspond to the amount of digitized charge. Digitization is the first phase in the reconstruction chain.

#### *7.2.1.2 Local reconstruction*

In order to perform physics analysis, it is necessary to reconstruct the actual physical quantities of interest. Therefore, digitized signals are converted (or reconstructed) into physical quantities such as energy, time and position. This conversion is performed on a per sub-detector base.

Local reconstruction applies to a particular component of the CMS detector (sub-detector). For instance, a hadronic calorimeter contains thousands of channels and energy deposition, within each is computed a sophisticated regression procedure. Regression algorithms are typically implemented using third party libraries (e.g. Eigen) which incorporate optimised linear algebra routines. However, certain functionality has to be manually ported to CUDA/OpenCL in order to preserve the algorithm itself and utilize the heterogeneous resources provided with the MSA.

#### *7.2.1.3 Global reconstruction*

Global reconstruction is the process of combining information from several components of the CMS detector in order to build high-level physics objects such as electrons, photons, jets, etc.

This operation drastically improves the precision of the measurements of properties of high-level objects.

### *7.2.2 CMS event classification*

A typical Machine Learning (ML) pipeline consists of three phases: feature engineering, model training (including cross-validation) and evaluation (inference).

#### *7.2.2.1 Feature engineering*

In a typical ML application, input data does not correspond one-to-one to the model's input. Therefore, a certain transformation algorithm has to be applied in order to prepare the input in a certain format. Apache Spark is used to perform Extracting Transforming and Loading (ETL) operations. The transformation involves taking collections of various particles (photons, electrons, etc.) and building an abstract two-dimensional image representing an event.

#### *7.2.2.2 Model training*

The model training phase is usually the most time-consuming part of the analytics workflow. At the current stage, GPUs provide the highest performance.

#### *7.2.2.3 Model evaluation*

Upon completing the training phase and finding the appropriate hyper parameters, inference is performed. The input data needs to be split at the previous stage so that a classifier does not see the data on which the inference is to be performed. The goal is to find the model giving the highest classification accuracy.

## 7.3 Application mapping

CMS event reconstruction workflow is a completely data parallel workload, where each event is independent, therefore the distribution of processing across MSA is quite trivial, i.e., there is no communication. Each node processes a completely different set of events and produces output data products. Within the DEEP-EST project, several time consuming parts of CMSSW (i.e. Hadron and Electromagnetic calorimeters) were identified and ported to utilize NVIDIA GPUs. Figure 7.1 below provides a basic overview of the distribution strategy. The idea is to use all the available resources and if possible the more performant one, i.e. if both CPUs and GPUs are available, the latter are chosen to run the codes parts that support them. It is important to note that overall adapting CMSSW to heterogeneous computer architectures is an ongoing activity.

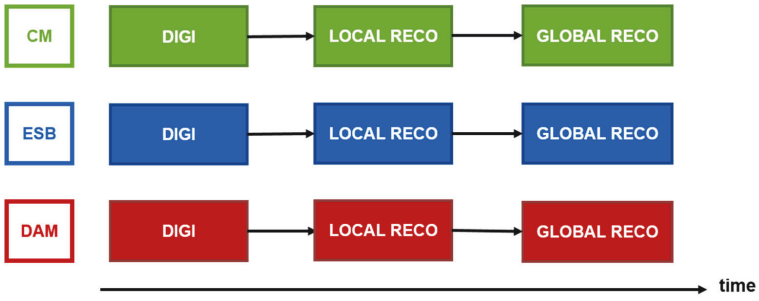


**Figure 7.1: Schematic workflow of the CMS event reconstruction in the MSA**

CMS event classification workflow (Figure 7.2) is a distributed deep learning training workflow that utilizes PyTorch for the training part. The distribution is implemented using the NNLO package[88], which uses MPI to communicate the weights. Furthermore, it also incorporates the use of Horovod for the purpose of distribution and communication to enhance the more basic Master-Worker approach implemented in NNLO package. More specifically, the model tested out on the DEEP-EST prototype

[88] https://github.com/vlimant/NNLO

is called JEDI-net, which stands for Jet Identification algorithm based on interaction networks. Jets are typically thought of as collimated cascades of particles which are abundant in hadron collisions, such as proton-proton collisions at LHC. Within the CMS event classification, we employed the JEDI-net neural network, which is trained to identify different types of such jet clusters.

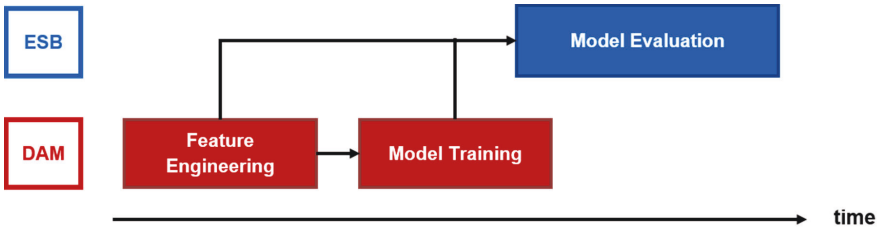


**Figure 7.2: Schematic workflow of the CMS event classification in the MSA**

## 7.4 Porting experience

For the CMS event reconstruction, the main porting effort was adapting the CMSSW framework to run on GPUs and optimizing selected time-consuming workflows to NVIDIA V100 in particular. At the start of the project, CMSSW contained CPU-based algorithms only, and there was minimum machinery available that had to be implemented in order to optimize it for heterogeneous resources (e.g., minimize host-device transfers, minimize device memory allocations, etc…). Furthermore, since CMSSW is a framework by itself, it already had certain intrinsic architectural choices and it was important to evolve without breaking the existing infrastructure.

The algorithms to be ported were those on the most time-consuming parts of the code. Figure 7.3 shows a breakdown of how much time is spent in a particular algorithmic part. Hadron and Electromagnetic calorimeters (labelled *HCAL local reconstruction* and *ECAL local reconstruction* in the figure) are two similar parts of the reconstruction that constitute around 24% of the total. The core parts of both algorithms were mathematically identical and employed Fast NNLS for the purpose of energy regression, although there was still quite a large amount of source code porting that completely differs for respective calorimeters. Therefore, these calorimeters were selected for porting and optimization to CUDA.

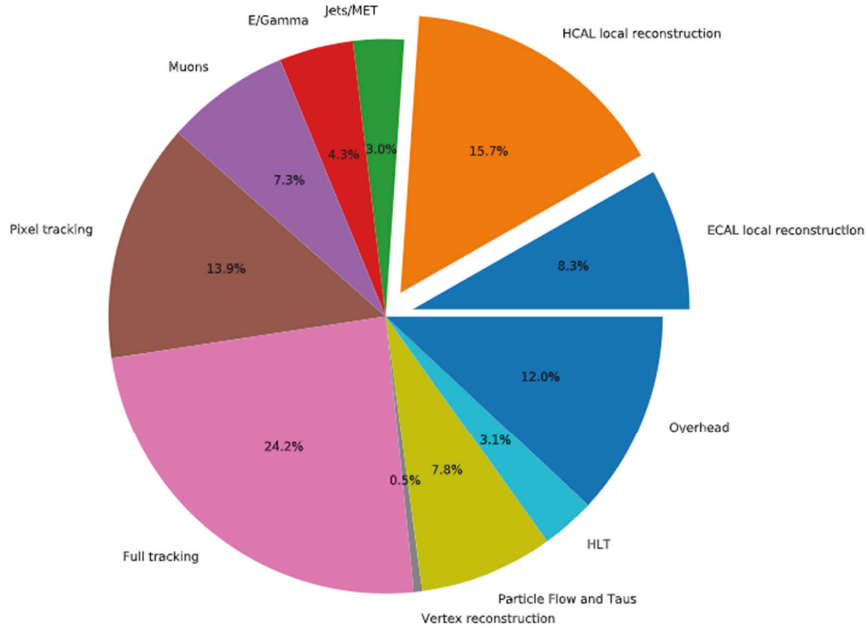


**Figure 7.3: Time spent in different algorithmic parts of CMSSW**

Before describing the necessary changes and efforts, we outline the problems faced. There are two main issues. First, the pure size and complexity of the source codebase. It is, of course, possible to use isolated small mockups, but then transferring the results to the CMS experiment and its community would be very difficult. The initial code for either Hcal or Ecal calorimeters amounted to O(10K) lines each. Even if isolated computational parts produce the largest impact, it is crucial to stress the importance of integration for software in the scale of CMSSW. Second, the existing software stack was written years ago (HEP experiments tend to last decades) and there was limited documentation about the actual algorithms and time vs. space complexities. In other words, the algorithms first had to be reverse engineered in order to deduce all of the available parallelism, which is quite an important aspect when dealing with GPUs.

The efforts required to port CPU-based source code to CUDA vary depending on the nature of the algorithms. For instance, if there are for-loops with large independent computations per iteration (i.e. data parallel), this trivially maps to CUDA kernel invocations, provided that all the code inside of this for-loop is supported by CUDA (e.g., the C++ version matters) or can be made supportable (e.g. mark functions as

either `constexpr` or `__device__`). This was actually the case with Hcal and Ecal source codes. However, the problem with this approach is that, although trivial to port, it might not lead to the desired performance (e.g., because memory was not aligned to suit GPUs) and the optimizations needed quickly became a full code-rewrite. Nevertheless, we believe that such a simple approach is a very good starting point, especially for people who are not experienced with CUDA and NVIDIA GPUs.

Therefore, here we outline the steps performed to arrive at an implementation that proved to be a good starting point for further optimizations:

1) Profile/Trace CPU code to identify hot spots.
2) In parallel with 1) deduce (reverse engineer) the algorithm and identify all the available parallelism. This also requires reasoning about how to align data in memory to better utilize GPU's compute units.
3) Typically, as the result of identifying the available parallelism, it will be apparent how many kernels are required and what the dependencies are between them.
4) Implement the required kernels for the GPUs.

Once point 4) is completed and implementations of separate kernels are available, it is crucial to evaluate these unoptimised versions to make sure that results (in the case of CMS event reconstruction it is physics quantities like energy) are validated with respect to what was obtained using CPU-based reconstruction algorithms. This allows to debug the ports early on before starting the optimization, which very often requires rewriting some compute-heavy routines, e.g., various mathematical operations. At this point, it is important to add that both CMS Hcal and Ecal CPU-based algorithms employed the Eigen library for the linear algebra computations. Although Eigen does feature some CUDA support, it did not cover the routines that were used in our algorithms. Therefore, we extended the functionality that was required to make the code work on GPUs (a very similar procedure was later applied to enable using Eigen from within Intel oneAPI kernels and is described below).

Once we verified that results of CPU vs. GPU reconstruction either match or are within certain precision (note that results of floating point computations on CPU and GPU can differ: they should not differ dramatically, but differences at certain tiny precision could be expected and were observed), we employed NVIDIA profiling tools, NVIDIA Nsight Systems and Nsight Compute. The first one, Nsight Systems, gives an overall system level view of what runs on a single node and identifies kernels that are the most time-consuming and therefore would be the first ones to undergo further optimization. For the purpose of kernel-level optimization we employed NVIDIA Nsight Compute. The most time-consuming kernel (originally ~90% of the total time) was the kernel responsible for the actual Fast NNLS energy regression. There are many potential approaches with regard to what to look for and how during the optimization. It is also

important to consider the amount of resources to be utilized during the execution – in other words, we could give more GPU resources to a kernel, but given that we run multiple streams (multiple events are reconstructed and run the same kernels but on different data), the overall performance would not go up, just the performance per single kernel/stream. Therefore, for the purpose of optimization, we tried to keep the amount of resources used per kernel fixed and minimize the runtime of this kernel when running a single CUDA stream. One of the first things we did was to rewrite the majority of Eigen operations, which were generating quite large stack frames. Also, given that Eigen is a header-only template-heavy library, it featured quite deep function call stacks. This made it very difficult to use the sampling-based features of Nsight Compute, which can help navigate to lines of code sampled more often and identify reasons for pipeline stalls, e.g., due to memory dependencies. Another important optimization was to not just use shared memory, but rather reuse it for different stages of the kernel execution, reducing the stack frame size of the kernel. All of that allowed, in turn, to reduce the number of registers used per thread, which is really important for parallelism of warps (more warps could be running on a given streaming multiprocessors (SM).

Before moving forward to describe the experience with our second application, we would like to outline our experience with Intel oneAPI as an approach for portability, which is based on the SYCL C++ language extension. First, a few words about why one would need such a layer, using the CMSSW framework as an example for this discussion. Consider that we rewrote and optimized substantial amount of C++ to CUDA (O(10K) lines of code). For the CERN/CMS collaboration, this essentially implies that these parts can only be run on nodes equipped with NVIDIA GPUs – in other words we are stuck with the choice of the vendor (critical for large collaborations such as LHC experiments). Of course, within the project we are optimizing for the given DEEP-EST prototype, but CMS will always strive to exploit as many compute resources as possible, therefore locking into a single vendor is not optimal, especially considering that there will be machines with AMD and Intel GPUs in the future. Another aspect is that given large source bases, rewriting parts for different accelerators could take months, which implies a lengthy development process, sometimes requiring some reengineering effort as some accelerators might not expose the same primitives. Overall, portability frameworks should prove useful in particular for large-scale scientific software development targeted to run on accelerators, provided the performance drop is minimal when using a portability framework with respect to using the native toolchain.

For the purpose of performing a minimal evaluation of Intel oneAPI, we used a standalone electromagnetic calorimeter reconstruction implementation that does not require the CMSSW framework and was ported from plain C++ to CUDA in the

beginning of the project. In short, it was a rather easy process to turn a CUDA-based implementation into oneAPI-compatible one. Here are the steps performed:

1. Employ the Intel DPC++ Compatibility Tool to transform our CUDA-based implementation into DPC++ compliant;
2. Fix all the issues raised by the Compatibility Tool;
3. Fix all the issues that come up during the compilation/linkage stages.

The very first step is probably the easiest one here, basically we just need to run a single command with all the initial host/device sources, and the Compatibility Tool will produce new sources that are now based on oneAPI. This conversion takes care of things like device memory buffers, allocations and transfers. CUDA streams and kernel invocations are mapped to the usage of queues and command groups. In terms of error handling, CUDA uses C style by reporting errors through error codes, whereas oneAPI is more C++ like and uses exceptions. The conversion between both is automatically handled by the tool as well, and step 2 above essentially refers to fixing whatever the compatibility tool was not able to convert. Kernels that use certain features specific to NVIDIA GPUs, for instance Tensor Cores, must be reimplemented. This is actually an important point overall and will require further investigation. For our small standalone evaluation we did not observe any difficulties after the conversion – the code was converted almost to 100% and required minimal changes. The third step was a bit more involved for us and should be of interest to other developers. Our implementation makes significant use of Eigen's primitives and therefore it is important that Eigen's routines work inside of the kernel and are supported. We found a couple of things that had to be adapted; first, Eigen is a header-only template-heavy library, therefore it utilizes advanced features of C++ templates and also makes heavy use of macros to configure the compilation process. When compiling things with DPC++, similar to when using the NVCC compiler, there are essentially two modes of compilation: kernel and host modes. The idea is that pre-processor directives will be configured differently based on the mode. The host mode in Eigen is for regular CPU execution and kernel mode is what sits inside of our oneAPI kernels. The main issue overall is that there is a set of restrictions applied to the code that goes into the kernel part, which can be cumbersome to fix when trying to use functionality from some library in your kernels. For the case of Eigen, one of the most difficult features to resolve, we had to disable dynamic stack allocation explicitly and also make sure that inline assembly instructions are not added, not even in comments. For the most part, we utilized the Ahead Of Time (AOT) compilation flow of Intel oneAPI toolchain. One of the minor drawbacks of the kernel mode compilation is quite poor error reporting: just reporting the error without explicitly stating parts of the code causing it makes it really difficult to debug, especially if trying to port a library one is not the author of.

In terms of results, we compared an implementation that is pure C++ based with one that is oneAPI based. For the purpose of evaluation, we just used a Virtual Machine (VM) from CERN's cloud. We have been able to reproduce exactly the same physics results (energy) and performance-wise the two versions were comparable, considering that oneAPI was able to utilize the multi-core VM and the plain C++ was written having single thread execution in mind.

For the CMS event classification, there was minimal porting effort as it is mostly an ML-based workload and does not require changing a lot of application logic in the code, compared to the CMS event reconstruction workload, where we had to produce O(20K) lines of code from scratch in order to have CMSSW-compliant implementations. However, here emphasis is more on the system integration and on the proper installation and configuration of the required software. For instance, Horovod requires a multithread aware version of MPI, which might be unavailable by default. Furthermore, the usage of the underlying communication method is quite important, i.e. using RDMA versus TCP for MPI. In other words, proper usage of the system is crucial for workloads that perform Deep Learning and similar activities, and a lot of time can be spent identifying why certain things do not perform as expected even if the actual porting of the code was trivial.

Overall, approximately 24 PMs were spent doing the actual development, porting, testing, validation for both of the applications combined. In terms of the sheer code size, around 20K lines of code were developed just for CMS Hcal and Ecal Local Reconstruction as the final footprint, not including the iterations involving the optimizations. These numbers do not include scripting side code for the purpose of analysis and benchmarking of applications.

## 7.5 Scalability

As it has already been emphasized, the CMS event reconstruction is a data parallel workload. As a consequence, there is no real communication across compute nodes. Therefore, the critical metric for this application is weak scaling because our goal is to use as many resources as we can get on the system without degrading the performance per unit. In addition, the CMSSW data processing constantly requires getting more and more input data (when running in production), by reading either from shared storage or from a remote location, and also produces output. This input/output data flow is where the bottlenecks for scalability are lying at the moment, and this is currently being investigated outside the DEEP-EST project.

Figure 7.4 shows how, by using all three types of compute nodes (CM, ESB, and DAM) and almost all of the corresponding nodes, the throughput increases. This figure does

not demonstrate the scalability, but it allows to view how adding more nodes does increase throughput and also dissects the contributions of different types of nodes.

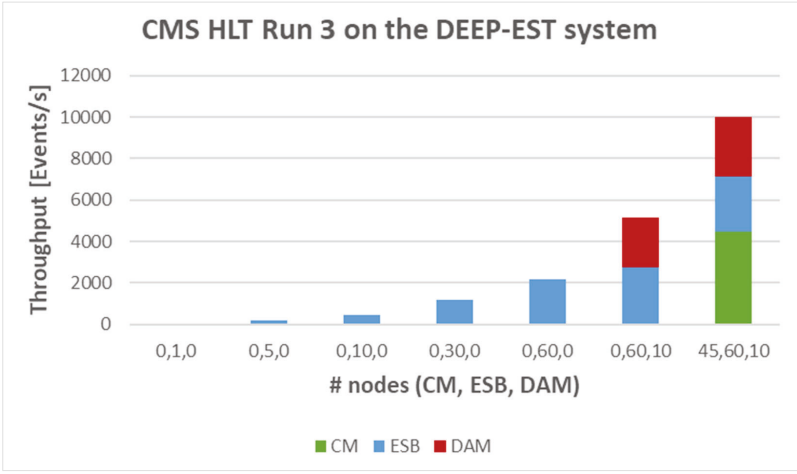


**Figure 7.4: CMS reconstruction using the whole DEEP-EST system**

For the purpose of demonstrating weak scaling for the CMS Event Reconstruction workflow, we essentially tried loading as many available nodes as possible, in the expectation that performance per node does not degrade. Throughout all the measurements we used the BeeGFS shared storage system for storing/ingesting input data. Figure 7.5 shows the distribution of throughput when employing all the CM nodes. As we can see, there are very few outliers and for the most part performance per node is quite stable.

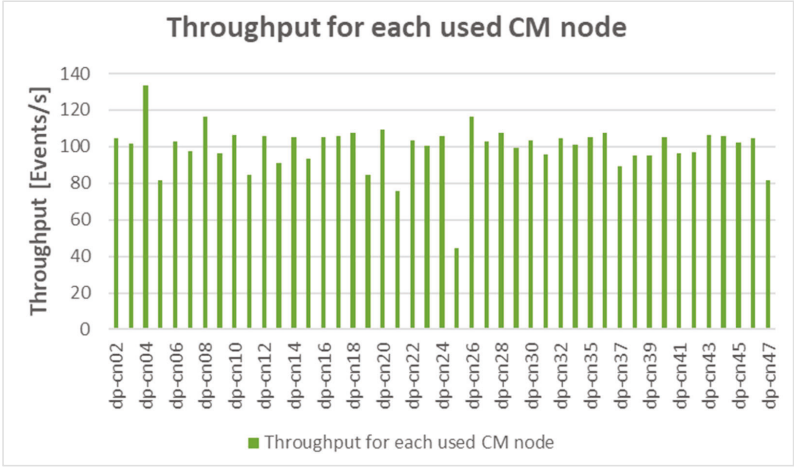


**Figure 7.5: Throughput distribution by node**

CMS Event classification is a typical distributed ML training workflow with a strong scaling objective. Figure 7.6 shows the execution time as a function of the nodes used for training. The training was performed using the ESB nodes. The most important outcome of these measurements is the fact that this distributed training workload shows good strong scaling features when using more and more nodes. In particular this is important when we compare running training on ESB to other systems that contain special NVIDIA Inter-GPU links and other optimizations. Employing the ESB we can scale up quite flexibly the number of nodes available for training.

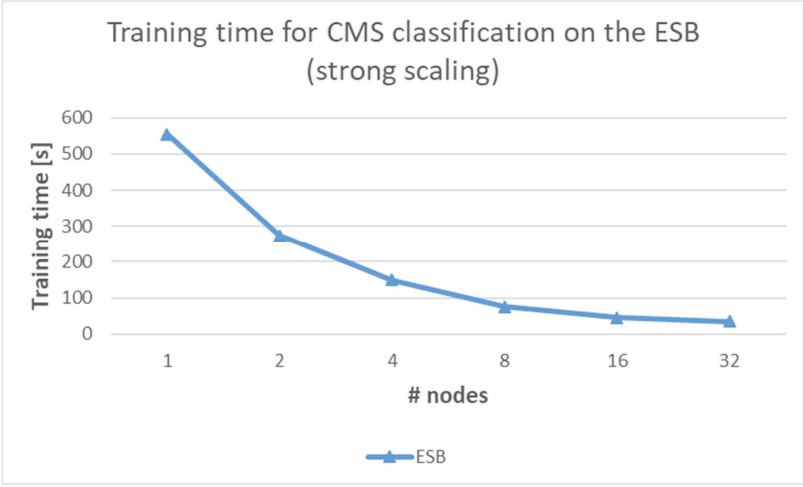


**Figure 7.6: Training time for CMS event classification on the ESB**

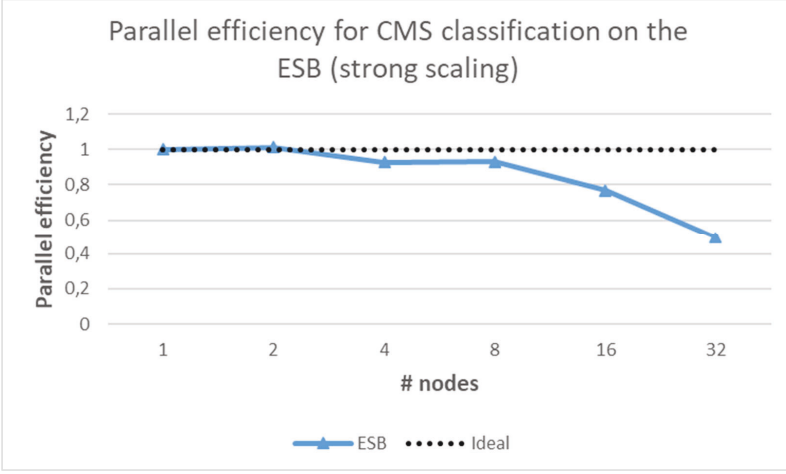


**Figure 7.7: Parallel efficiency for CMS event classification on the ESB**

### *7.5.1 Our path to Exascale*

Overall, we think that it is important to understand first what the future is going to look like before addressing our path towards that future. For LHC, the CMS experiment and HEP in general, the future will bring significantly more data and more complex structure of collision events, both of which in turn mean that physicists have to find more efficient ways of handling these massive amounts of information. To quantify this, more data means processing O(500PB) by the CMS experiment per year. These numbers are obtained using the existing production level workflows.

Within the DEEP-EST project we tackled the very first part of this journey – making our software more efficient by using heterogeneous resources available (or soon to be available) at HPC facilities. Our contributions have been integrated into the CMS Experiment's framework and will be used in production starting with this year's production campaign. It is crucial to note that these algorithms are not just running at an HPC centre in an offline manner (where someone launches jobs and collects results), but are also used at CMS experiment's online farm during the data-taking, in order to identify events of interest for the future physics analysis, which means that they must be robust, performant, and error-free all the time. Heterogeneous resource utilization is one of the key features for any scientific software when targeting Exascale as it allows us to make the code more efficient. For CMS, it means higher throughput per single node.

For the purpose of data processing, all large scale LHC experiments, CMS included, utilize the world-wide distributed computing grid, which allows us to perform data processing across many different computing sites. The reason for this distribution is that a single site would not be able to cope with requirements imposed by HEP workflows. Furthermore, by dividing the processing infrastructure into multiple sites, the movement of data has to be properly handled. This problem of having to move around O(500PB) data per year is one of the central challenges of the path to Exascale for the CMS experiment.

#### ***7.5.1.1 What are the limitations? – Can they be fixed?***

From the perspective of an LHC (or probably even High Luminosity LHC) experiment, it is not a question of limitations, but rather a question of what has to be developed in order to unleash the full scientific potential of the upgraded detectors when moving to High Luminosity LHC. As was mentioned for the CMS event reconstruction workflow, it is necessary to keep ingesting data and also storing the output data products, which means that I/O subsystem could be seen as a limitation at Exascale. Therefore, here are two important areas of work that are currently being investigated by the whole HEP community:

- At Exascale it is important to consume computing resources without degrading the performance per node. There could be two potential limitations in here: the network and the storage itself. For instance, there could be too many clients trying to do I/O from the shared storage system. It is true that data could be pre-staged to the compute nodes first and then processed, but again the limit on the network could be reached with the huge number of nodes at Exascale. On the other hand, there are new types of shared storage systems (e.g., object storage) that are significantly more robust for applications that heavily utilize the I/O subsystem. To run production-type of workflows these “limitations” need to be further tested and resolved.
- The previous point covered the network and storage that are fully internal to an HPC facility. The next item to consider is the external connection of an HPC facility. Considering that HEP experiments cannot store all of their data at an HPC site, it is crucial that there are efficient means of enabling data flow to/from such HPC centres. There are two main reasons why all of the data cannot be preserved at an HPC site. First, there will be as much as O(500PB) for a single experiment. Second, HEP collaborations by themselves are quite large entities and data collected is one of their main products, therefore storage and preservation of this information is of crucial importance to HEP. Delegating this responsibility will not be feasible. Overall, this means that it will be necessary to find efficient ways to enable dynamic and scalable flow of data to/from HPC centres. Such a system would need to be capable of:
    - Overlapping processing with bringing in more data;
    - Talking to the outside world: request more data, inform of what is missing, etc.;
    - Being aware of what is running, which data can be purged, what has to be requested and brought in or taken out.

Overall, this requires having a system in place that runs at an HPC site and handles the external data flow activity.

#### *7.5.1.2 How to use future Exascale systems*

Overall, the way for HEP community to exploit Exascale systems is by maximising the efficiency every available compute node. Having in mind that our target is weak scaling, we have to stay efficient when scaling out. This is mainly achieved by embracing the usage of accelerators and software reengineering, which has been successfully done within the DEEP-EST project for the CMS experiment’s workflows.

Another important aspect to stress is data. We will not have less data in the future, only more. And this applies to many other data driven sciences, not just HEP (e.g., SKA,

see Section 4 of this volume). The network bandwidth, although increasing as well, will not keep up with the pace of data volumes. This is in particular crucial for the large scale experiments that cannot store their data at an HPC site. The dynamic component that is responsible for bringing data in and taking processed output outside of an HPC centre in a scalable fashion will play an important role for such applications. We would also like to note that this is an area of work for HPC centres as well, as they are aware of the shifting requirements of the applications that would like to utilize their facilities.

#### *7.5.1.3 Where did the DEEP-EST project help on the way to Exascale?*

Within the DEEP-EST project, one of the more crucial development contributions overall was the porting of 20-25% of CMS High Level Trigger (HLT) to utilize NVIDIA V100 GPUs. There are several reasons why this contribution is important:

- The algorithm implementations developed in DEEP-EST will be running in data-taking production for the CMS experiment starting this year. They are not just performant, but they reproduce physics results with good precision, therefore having no effect on downstream physics performance.
- The very positive experience serves as motivation for other members of the CMS collaboration to join the effort of code modernization and porting to heterogeneous architectures. In parallel to the CMS Hcal/Ecal ports, the software from another detector was ported by other members of the CMS collaboration, which allowed developers to interact and discuss ideas for implementations and optimizations.
- Finally, developed functionality will not just be running for the CMS HLT, but also for offline production workflows, which will be utilizing HPC resources. Through this effort, we increased the efficiency (i.e. throughput) of our applications when targeting future Exascale systems.

Given that HEP community traditionally did not use HPC machines for reconstruction workflows, this work allowed to gain overall confidence that we can run efficiently these data-driven types of workflows on such large machines, not just simulation workloads that do not really require any input data. Another aspect is the ability to test out our production workflows when running on larger node counts using a prototype system, not a production one – we are able to tweak things, test, change, and do it again, which is quite important during the development life cycle. This applies to data analytics type of workflows even more, as these require tweaking the configuration to find the appropriate parameters on a given system.

## 7.6 Energy consumption

Energy consumption is an important metric in particular when thinking of Exascale workflows. It is clear that in order to be sustainable, we cannot just think of performance without addressing the issue of power consumption, as well as costs associated with resource utilization. For HEP production types of workflows, with a weak scaling objective, it is important to be as efficient as possible per node, given the tendency to consume as many resources as there are available. This is where software reengineering potentially helps not only performance but also energy efficiency.

All the energy utilization metrics were collected using various DEEP-EST prototype sensors, which allow for frequent probing of quantities, which in turn enables fine-grained downstream analysis. Figure 7.8 shows the total energy consumed when running the full CMS Event Reconstruction workflow on CM nodes (green) and ESB nodes (blue) as a function of the number of nodes used for the reconstruction.

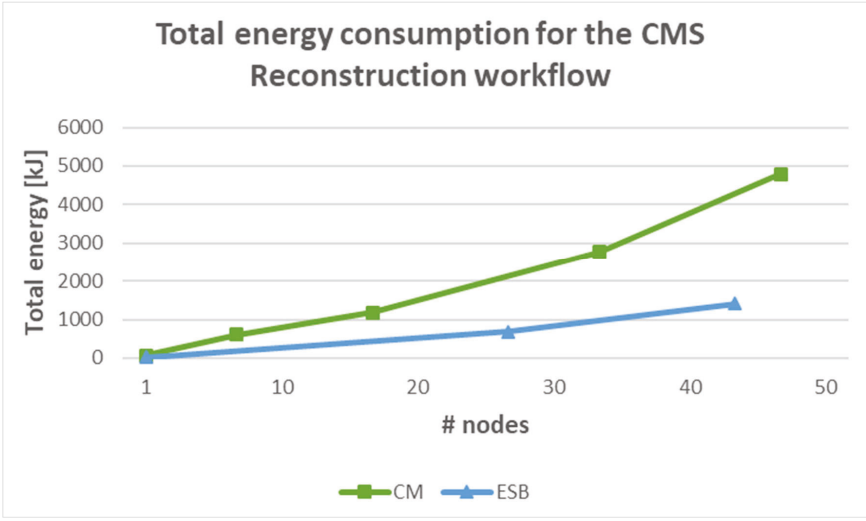


**Figure 7.8: Total energy consumption running for the CMS Event Reconstruction**

Figure 7.9 in turn shows the same values but averaged over the number of nodes. Given that we are after stable resource utilization when scaling out the number of nodes, here we observe a slight increase when going to 32 and 49 nodes on the CM. This is not too dramatic and could be a result of the outliers observed in Figure 7.5.

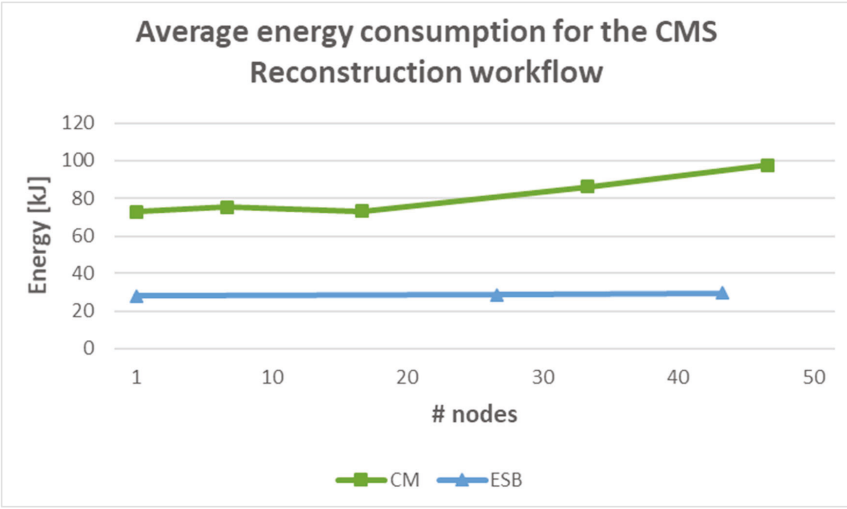


**Figure 7.9: Average energy consumption for the CMS Event Reconstruction**

When employing NVIDIA GPUs for the CMS Event Reconstruction, the ESB shows nearly perfect linear scaling in total energy when scaling out the data processing from 1 node to 48 (blue line in Figure 7.8). The averaged energy utilization per node, which is the more relevant metric for the workload that aims to scale weakly, is shown in the blue line in Figure 7.9. Overall, here we observe stable resource utilization per node when scaling out our production workload to many nodes equipped with NVIDIA V100 GPUs.

## 7.7 Performance comparison

### *7.7.1 CPU vs GPU*

For the purpose of performance comparison we were trying to evaluate the maximum throughput, defined in terms of events per second that could be achieved either on CPU or GPU. Although the CMS software framework by itself is multi-threaded, the original CPU-based algorithm implementations are single threaded (i.e. task-based parallelism), therefore they basically scale with the number of available cores (provided there are no other limitations). However, for the case of a GPU-based implementation it is not so straightforward and we essentially are trying to push as many concurrent events into a single card as possible until we reach a limit. Figure 7.10 shows the result of comparing the CMS Ecal Local Reconstruction using NVIDIA V100 vs. using 2-socket Intel Xeon Gold 6148 (32 cores total). The reason for comparing against this particular model of the CPU is that most of the comparisons which we make when

targeting heterogeneous hardware architecture are done against models similar to the ones currently employed at CMS High Level Trigger (HLT). Here we observe a factor of 3-4× speedup with respect to the CPU version.

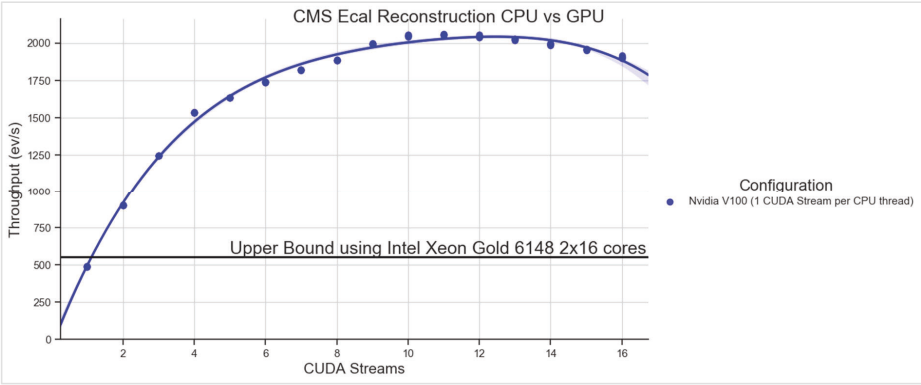


**Figure 7.10: CPU vs GPU for CMS Ecal Reconstruction**

Similarly, for Hadron calorimeter reconstruction, Figure 7.11 displays the comparison of throughput when running the CMS Hcal Local Reconstruction on NVIDIA V100 GPU vs 2-socket Intel Xeon Gold 6148 (32 cores in total). Here we observe factors of 7-8× speedup when comparing to the baseline CPU version. This could come as a surprise, given that we indicated that the Hcal and Ecal algorithms share the same core routines. However, as mentioned above the core routines, although crucial, are not the only element needed for these algorithms to run successfully within the CMSSW framework. Furthermore, most of the optimizations were first tested using Hcal workflow and then applied to Ecal, which means that Hcal was more heavily optimized. Also the percentage of time spent in different kernels is slightly different for Hcal and Ecal. All this leads to slightly different speed up factors.

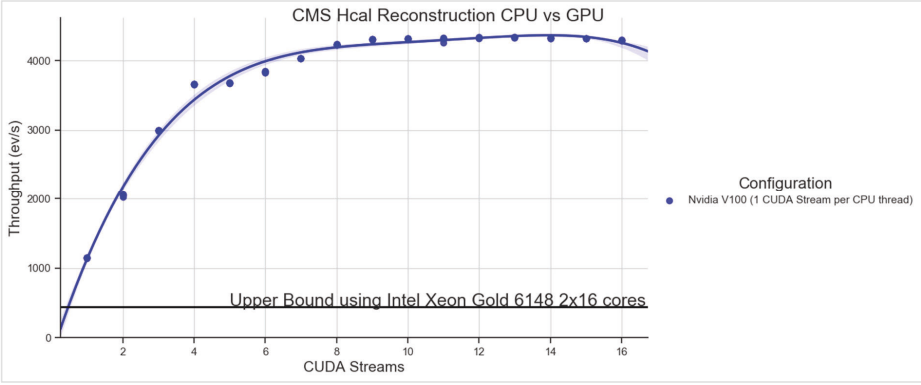


**Figure 7.11: CPU vs GPU for Hcal Reconstruction**

Finally, one of the more interesting things to test was to use the full CMS HLT workload on each type of the nodes available on the DEEP-EST prototype. This workflow consists of running O(1000) algorithms including Ecal and Hcal. We used the CMS Open Dataset for the purpose of these measurements. Figure 7.12 shows the throughput (events per second) when running CMS event reconstruction on different types of nodes on the DEEP-EST MSA and also either CPU only or combining CPU and GPUs.

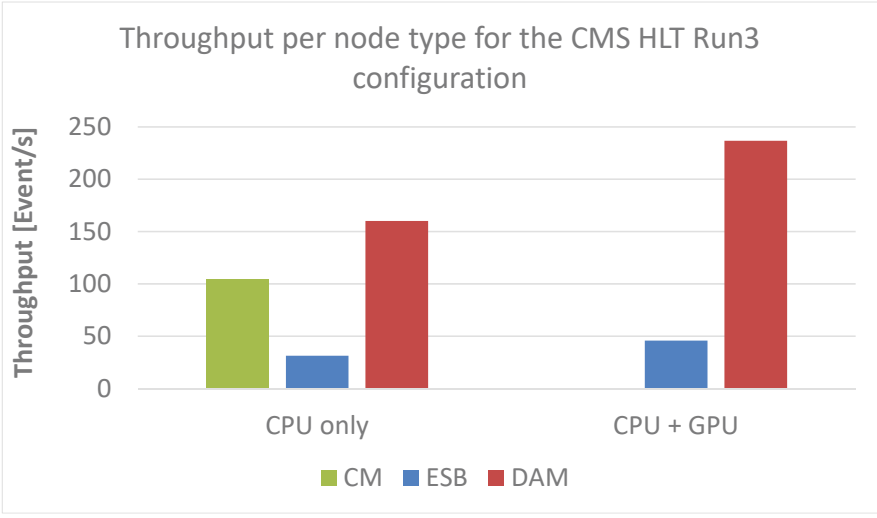


**Figure 7.12: Throughput per node type on the DEEP-EST MSA**

We have used CPU-only configuration on the CM nodes, but when we targeted either ESB or DAM we could employ both CPU-only and CPU+GPU configurations. With CPU+GPU configurations we achieved 50% speed up in throughput from a single node. This result is quite important to justify the advantage of porting your software to heterogeneous hardware, keeping in mind that not all algorithms do benefit from being ported to accelerators.

## 7.8 Conclusion

The work carried out in the DEEP-EST project paves the way for the High Energy Physics community to successfully exploit the future Exascale HPC systems for both production data processing workloads, and for analytics driven applications requiring usage of Deep Learning techniques. It is a highly non-trivial task to take such a large software stack such as CMSSW and be able to efficiently run, measure and understand the results when employing high node counts at an HPC site. Although we did not tackle issues related to the I/O subsystem, experience gained within the project about what worked well and which components are going to be required for a successful utilization of large Exascale machines is invaluable for our further investigations.

Experience gained within the project has been conveyed to other members of the CMS collaboration working on porting scientific software to heterogeneous platforms. In particular, the knowledge about NVIDIA Profiling tools, Nsight Systems and Compute, allowed us to optimize CMS Hcal and Ecal Local Reconstruction GPU implementations and to achieve significant speedups (3-4× for Ecal and 7-8× for Hcal) with respect to the CPU-based implementations.

Finally, the developed algorithmic implementations have successfully been integrated into the CMS software framework. Physics validation has been performed and GPU-based algorithms will be exploited during the upcoming data-taking campaign of the CMS Experiment at LHC.

# 8 Best Practices Guide

**Anke Kreuzer, Jochen Kreutz, Benedikt Steinbusch, Zia Ul Huda**
Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH, Leo Brandt Strasse, 52428 Jülich, Germany

a.kreuzer@fz-juelich.de

**Germán Llort, Julita Corbalan, Lau Mercadal Melià, Pedro Martinez**
Barcelona Supercomputing Center, BSC, Spain

**Jorge Amaya**
Katholieke Universiteit Leuven, KU Leuven, Belgium

Hans-Christian Hoppe[89]
Intel, Germany

**John Romein**
Netherlands Institute for Radio Astronomy, ASTRON, Netherlands

**Carsten Clauss**
ParTec AG, Germany

## 8.1 Introduction

The DEEP-EST system implements the Modular Supercomputing Architecture (MSA) which has evolved within the DEEP project family over several years[90]. One of the most important advantages of the MSA is its flexibility: MSA systems can target a wide range of applications with widely different characteristics and system requirements. This guide shows how to port applications to the DEEP-EST system (described in Chapter 1 of this volume) and gives advice on how to get good performance out of it. Each kind of application (as with the different co-design applications within the DEEP-EST project) may have different ways to use the DEEP-EST system.

In this document, several use cases will be explained, and advice will be given about how an application can benefit most from the system architecture. Examples of the improvements that could be achieved for demonstrator applications will be shown. This guide is structured in the following way:

[89] Now at scapos AG, Germany.

[90] https://deeptrac.zam.kfa-juelich.de:8443/trac/wiki/Public/User_Guide/Tutorial1/MSA_Idea

- First in Section 8.2 we describe how to analyse an application and figure out which modules to use in Sections 8.2 and 8.3.
- Once this decision is taken, Section 8.4 focus on the real porting work (mostly porting to GPU, plus a short introduction on the FPGA porting).
- The next topic is how to partition the application code to enable it to run across multiple modules. This is covered by Section 8.5.
- Section 8.6 describes several different file systems which are provided in the DEEP-EST system.
- Section 8.7 covers certain additional features provided on the DEEP-EST system.

Last, but not least, Section 8.8 summarizes the most important lessons learned by the application developers in the DEEP-EST project, which refer to their experience adapting the codes to MSA, but also more in general when preparing them to exploit heterogeneous computing at the Exascale era.

## 8.2 Analysis

The three DEEP-EST prototype modules were designed to fit the needs of different kinds of applications. The ESB has the highest node count and is equipped with GPGPU accelerators coupled to relatively weak CPUs in the interest of energy efficiency. Highly scalable applications or codes with data and control structures suited to GPGPU computation can run perfectly on the ESB, yet it is essential that the computation happens exclusively on the GPGPU, and that all data structures do fit within the 32 GB of GPGPU high-bandwidth memory. Codes or code parts that require high amounts of memory, for example, should run on the DAM with 384 GB DRAM and 3 TB Persistent Memory attached to each node. There are also different ways to distribute the code parts depending on the individual application. There might even be applications using only one module, with the choice of the module depending, among others, on the problem size. Applications which combine parts best suited for different modules have the option of running simultaneously across multiple modules, while other codes with a workflow structure will run different steps on different modules, for instance as a job chain.

A detailed analysis of the code is essential to get to know which parts of the code can benefit from which parts of the architecture. Without this it is not possible to get all the benefits out of the DEEP-EST system. Here are some recommended profiling tools (all available on the DEEP-EST system):

- Intel VTune Amplifier [91]
- Intel Vector Advisor [92]
- JSC Scalasca [93]
- BSC Extrae/Paraver (for basic instruction please see Section 8.2.1) [94] [95]

These tools will help determining which are the most time-consuming parts, whether the application is compute, memory or bandwidth bound, and how well balanced the application is. With this insight the developer can decide how to map the application to the MSA: for example, time consuming parallel code parts, should exploit the scalable GPU nodes on the ESB, whereas code parts that need a large amount of (fast) memory should use the DAM with Intel Persistent Memory.

### *8.2.1 Performance analysis tools, Extrae, Paraver & Dimemas (BSC)*

**Extrae** is a dynamic instrumentation package to trace programs. It generates trace files that can be later visualized with **Paraver**. To use Extrae on the DEEP-EST system load first a compiler and the MPI distribution that you want to use, e.g. GCC and ParaStationMPI, and then load the Extrae module:

```
ml GCC
ml ParaStationMPI
ml Extrae
```

#### *8.2.1.1 Using Extrae in 3 steps*

##### 8.2.1.1.1 Adapt the job script to use Extrae

The job script needs to be adapted in three aspects (Figure 8.1):

- Load the above mentioned modules
- Specify the name for the output traces (optionally)
- Run with Extrae

[91] https://software.intel.com/content/www/us/en/develop/download/intel-vtune-amplifier-2019-help.html

[92] https://software.intel.com/content/www/us/en/develop/articles/quick-analysis-of-vectorization-using-intel-advisor-2019.html

[93] https://apps.fz-juelich.de/scalasca/releases/scalasca/2.5/docs/UserGuide.pdf

[94] https://tools.bsc.es/doc/html/extrae

[95] https://deeptrac.zam.kfa-juelich.de:8443/trac/wiki/Public/User_Guide/Tutorial2

```
#!/bin/bash
#SBATCH --job-name=lulesh2.0_27p
#SBATCH --output=\%x_\%j.out
#SBATCH --error=\%x_\%j.err
#SBATCH --ntasks=27
#SBATCH --nodes=2
#SBATCH --cpus-per-task=1
#SBATCH --exclusive
#SBATCH --time=00:10:00
#SBATCH --partition=dp-cn

ml GCC
ml ParaStationMPI
ml Extrae

export TRACE_NAME=lulesh2.0_27p.prv

srun ./trace.sh ./lulesh2.0 -i10-s65
```

**Figure 8.1: Job script with Extrae**

The `trace.sh` wrapper loads Extrae. The user needs to select the proper tracing library depending on their application type (MPI, OpenMP, CUDA, hybrid, etc.) and language (C, Fortran). The available libraries can be found under $EBROOTEXTRAE/lib.

```
#!/usr/bin/env bash

export EXTRAE_ENFORCE_FS_SYNC=1

# Configure Extrae
export EXTRAE_CONFIG_FILE=./extrae.xml

# Load the tracing library (choose C/Fortran)
export LD_PRELOAD=$EBROOTEXTRAE/lib/libmpitrace.so
#export LD_PRELOAD=$EBROOTEXTRAE/lib/libmpitracef.so

# Run the program
$*
```

**Figure 8.2: `trace.sh` to extract traces with Extrae**

#### 8.2.1.1.2 Extrae XML configuration

Within the `trace.sh` file you have to specify the XML file containing your Extrae configuration. In Figure 8.2 it is called *extrae.xml*. Here you can configure what will be traced, e.g. if you want to trace the MPI calls and the call-stack the file should look like this:

```
<mpi enabled="yes">
    <counters enabled="yes" />
</mpi>
...
<callers enabled="yes">
    <mpi enabled="yes">1-3</mpi>
    <sampling enabled="no">1-5</sampling>
</callers>
```

There are several other options which can all be found in the Extrae documentation[96].

#### 8.2.1.1.3 Run it

Now you can submit your job as usual:

```
sbatch job.slurm
```

Please note: Always run your job from the /work directory not from $HOME!

Once the job finishes you will have the trace (3 files):

- lulesh2.0_27p.pcf
- lulesh2.0_27p.prv
- lulesh2.0_27p.row

### *8.2.1.2 First steps of analysis*

To analyse the traces first copy them to your local computer and then load them with Paraver. Several guided demos are included with Paraver, which walk the users through the first steps of analysis with real applications examples. These are available for download clicking on *Help → Tutorials → Download* and *install tutorials*. Following the tutorials is as easy as clicking on the hyperlinks which open pre-generated example traces and different analysis views.

For new users it is recommended to start with Tutorial 1 which explains basic control and navigation with the tool; and Tutorials 4 & 5 which show two examples of complete analyses with pre-generated traces from real applications. More advanced users will find Tutorial 3 interesting as it describes an analysis methodology that focuses on detecting work and performance imbalances. If the users already have a trace of their own application and load it on Paraver, the tutorials can be likewise applied on their traces, and the analysis views will be computed on the users' application. For more information on how to analyse the traces and using Paraver, we refer to the Tutorial[95].

[96] https://tools.bsc.es/doc/html/extrae

#### *8.2.1.3 Simulations with Dimemas*

Dimemas is a simulator that reconstructs the time behaviour of a parallel application on a machine modelled by a set of performance parameters. Performance experiments can be done easily changing the target architecture by modifying network and CPU parameters. For communications, a linear performance model is used, but some non-linear effects such as network conflicts are taken into account. The simulator allows specifying different task-to-node mappings.

This simulator is useful to predict the behaviour of applications on non-existent machines, perform parametric sweeps (e.g., mass-evaluate different BW and latencies), and conduct 'what if' analyses to answer questions like: *"Does the application have load balanced and dependence problems?"*, *"Would we benefit from grouping messages?"*, *"Is bandwidth the problem?"*, *"Is network contention the problem?"*.

Dimemas generates Paraver trace files enabling the user to conveniently examine any performance problems indicated by a simulator run. The Paraver Tutorial 2 contains an introduction to the use of Dimemas with an example and guidelines to get started with this tool. For more information on the architecture and use of the simulator, the user may refer to the tool website[97].

[97] https://tools.bsc.es/Dimemas

## 8.3 MSA Usage Models

Within the DEEP-EST project we identified 6 different usage models for our applications, which can be sorted into two different categories: Single and multi-module usage. In each category we have three different usage models (see Figure 8.3).

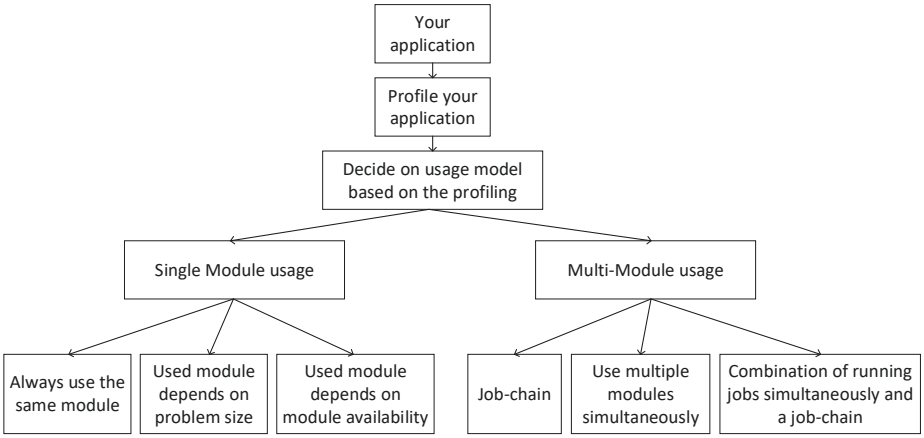


**Figure 8.3: Different usage models on the MSA**

### *8.3.1 Single module usage*

**Always use the same module:** Examples for this usage model are NEST, or the GPU/FPGA Imager.

- NEST needs strong CPUs and cannot take advantage of GPGPU accelerators so it can either use the CM or DAM. Since NEST does not makes use of GPUs, FGPAs, or a huge amount of memory, the DAM nodes are somewhat over dimensioned. The CM is therefore the best suited for executing NEST.
- The GPGPU and FGPA Imagers used in radio astronomy need to run on the DAM. For the FPGA imager this is obvious since only the DAM nodes are equipped with FPGA accelerator. The GPU imager needs a huge amount of memory, thus running on the GPUs in the ESB nodes is not an option.

**Used module depends on use case:** This is the case in the single module version of GROMACS. The CM is used for small size problems, whereas the GPUs on the ESB are needed for the larger use cases. GROMACS could also use the GPUs on the DAM but the ESB is a better choice because the code does not need much memory and is scalable over many nodes.

**Used module depends on module availability:** Finally, there are applications, such as the CMS Reconstruction, which can run on all the modules. The CMS Reconstruction has a CPU version for the CM and a GPU version for ESB and DAM. Since the execution runs in all nodes independently it can just utilize any kind of nodes that are available at any given time.

### *8.3.2 Multi module usage*

**Job chain:** An example for the "Job chain" model are the coupled versions of NEST plus Arbor (Figure 8.4) and NEST plus Elephant (Figure 8.5). NEST first runs on the CM (as explained in Section 8.3.1) and after that Arbor starts to work on the output from NEST using the GPUs of the ESB. Similarly, Elephant starts the data analysis on the output from NEST on the DAM.

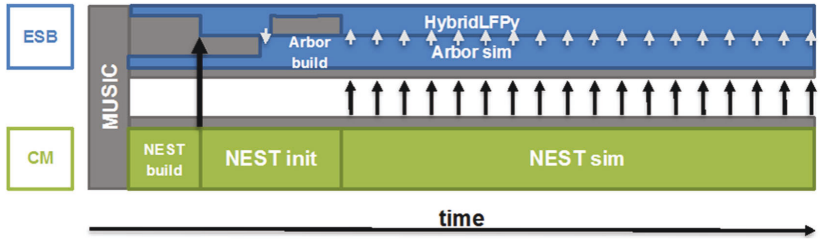


**Figure 8.4: NEST plus Arbor workflow**

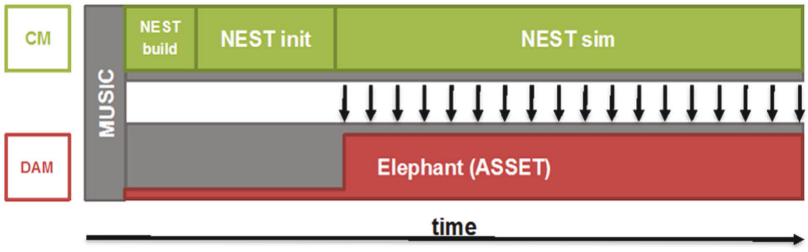


**Figure 8.5: NEST plus Elephant workflow**

**Use multiple modules simultaneously:** A good example for this fifth usage model is the xPic and GMM combination: xPic's particle solver runs on the GPUs of the ESB and its field solver runs on the CPUs of the CM, while GMM runs on the DAM. Particle and field solvers from xPic run simultaneously and exchange data during runtime. The data produced by the particle solver is analysed by GMM on the DAM nodes (Figure 8.6).

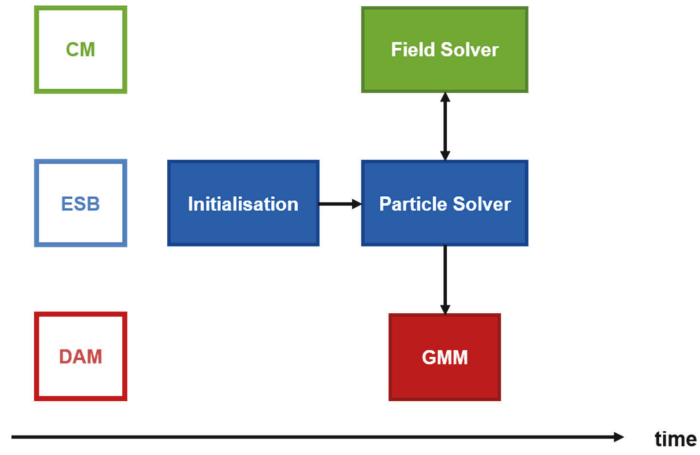


**Figure 8.6: xPic plus GMM workflow**

Another example is GROMACS in the offload version. It simultaneously uses the CM and the ESB within one job (Figure 8.7).

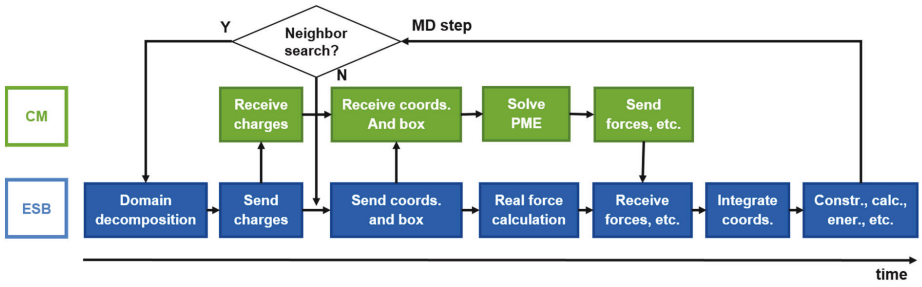


**Figure 8.7: GROMACS workflow for the offload version**

**Combination of job chain and jobs running simultaneously:** An example of this last usage model is the workflow of DLMOS (DAM) → xPic (CM+ESB) → GMM (DAM). The ML codes DLMOS and GMM will run on the DAM and between these two jobs xPic will run on CM+ESB (Figure 8.8).

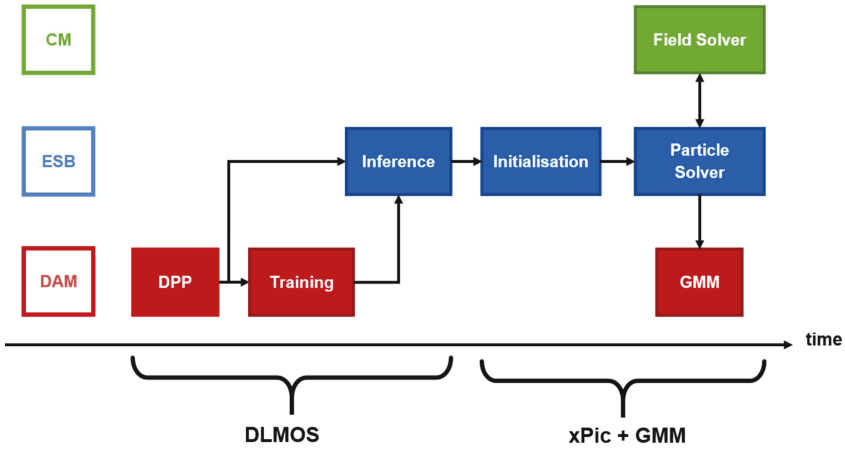


**Figure 8.8: Workflow of DLMOS plus xPic plus GMM**

## 8.4 Porting

The DEEP-EST system provides the GCC (8.3.0, 9.3.0, 10.2.0), Intel (2019.5.281) and NVHPC (20.9, experimental) compilers for C, C++ and Fortran. There are also different MPI versions available (ParaStationMPI, Intel MPI and OpenMPI), but it is recommended to use ParaStationMPI because it enables all the MSA features on the system. If the code needs specific software packages, it should be checked if they are provided on the DEEP-EST system. Detailed information on all available packages and the module environment used on the system can be found in the DEEP-EST Wiki[98]. Jobs can be submitted to the job queue for all compute modules (CM, ESB, DAM) via the Slurm resource manager.

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

### 8.4.1 The resource manager

Slurm supports both interactive and batch jobs (scripts submitted into the system). This is an example on how to allocate an interactive session on the CM (`-p dp-cn`) with 4 nodes (`-N 4`) and 2 tasks per node (`-n 8`) for 30 minutes (`-t 00:30:00`):

srun -p dp-cn -N 4 -n 8 -t 00:30:00 --pty /bin/bash –i

The following example shows a job script for submitting a batch job using the same parameters (number of nodes, runtime etc.) as before:

```
#!/bin/bash
#SBATCH --partition=dp-cn
#SBATCH -A deep
#SBATCH -N 4
#SBATCH -n 8
#SBATCH -o /Path/to/output
#SBATCH -e /Path/to/erroutput
#SBATCH --time=00:30:00
```

**Figure 8.9: Job script example**

For more details (all available partitions, `srun` and `sbatch` options and useful Slurm commands) refer to the batch system section in the DEEP-EST Wiki [99].

### 8.4.2 Code porting and optimisation on the CM

Porting the codes to the CM should be straightforward since the CM is equipped with standard, general purpose CPUs and every code targeting multi-core CPUs should work.

### 8.4.3 Code porting and optimisation on the ESB

The ESB is equipped with NVIDIA Tesla V100 GPUs. The code parts that were identified to be compute intensive and can be parallelized should be ported to the GPUs. If serial code parts are just included to manage the GPU computation, they do not need a high computing capacity, and can fit all application data into the 32 GB high-bandwidth GPGPU memory, using only ESB nodes with its comparatively weak CPUs is sufficient. If the serial code parts need stronger CPUs, the developer should strongly consider dividing the code onto CM (strong CPU) and ESB (GPU). The DAM would also be an option for strong CPU plus GPGPU runs, but because there are only 16 nodes, running on the CM plus ESB (see Section 8.5) makes more sense to scale the application.

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

Porting code to the GPU can be done with different programming models. On the DEEP-EST system we support the following: CUDA, OpenACC, OmpSs, and OpenMP5.0. Below we give some introductory information on how to use these programming models. For more details we refer to the specific user guides and documentation, since detailed explanations of the programming models and their usage would go far beyond the scope of this document.

#### *8.4.3.1 Using CUDA*

Since the GPGPUs in the DEEP-EST system are NVIDIA GPGPUs, using CUDA is likely the way to get the maximum performance out of the code. However, it should be kept in mind that CUDA code is not the best option for non-NVIDIA GPUs. In addition, a lot of effort may be required to port an application to CUDA if one has to start from scratch. As an example, we will use a simple vector addition (see Figure 8.10).

First, the computations that should run on the GPGPU have to be turned into CUDA kernels. For this the `__global__` keyword has to be added to the affected functions. If the Host device needs the results from the GPU, it must be ensured that the host waits for the GPGPU to finish the calculations. For this the function `cudaDeviceSynchronize()` can be used.

In addition, one has to manage memory and potentially data placement. With the available Unified Memory, a memory space can be allocated and then be used by the CPU as well as by the GPU, so that the data does not need to be transferred manually anymore. To allocate and later free Unified Memory, two functions need to be called (as replacement for ‘`malloc`’ and ‘`free`’): `cudaMallocManaged(…)` and `cudaFree(…)`

Now the code is ready to run on the GPU, so finally the kernel can be launched with `vectoradd<<<x, y>>>(n, a, b)`.

```cpp
#include <iostream>
#include <math.h>

// function to add the elements of two arrays
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20; // 1M elements

  float *x = new float[N];
  float *y = new float[N];

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 1M elements on the CPU
  add(N, x, y);

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  delete [] x;
  delete [] y;

  return 0;
}
```

**Figure 8.10: C++ version of the vector addition**

Figure 8.11 shows a code including the above changes. Although these changes make the code run on the GPU, there is plenty of room for optimization, so the basic code should then be analysed with a profiler, e.g. `nvprof`, to get an idea of what to optimize. There are plenty of tutorials, guides and courses on how to write and/or optimize CUDA codes: here are just a few examples[100] [101] [102].

[100] https://developer.nvidia.com/blog/even-easier-introduction-cuda/

[101] https://fz-juelich.de/SharedDocs/Termine/IAS/JSC/EN/courses/2020/ptc-gpu-cuda-2020.html

[102] https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

It should be noted that for certain codes, manually managing the location of data objects (on host or GPU memory) can extract more performance than relying on the Unified Memory mechanisms; this is akin to cache optimizations on traditional CPU systems. For the ESB, it is critical to ensure that all application data objects are located in GPU memory.

```
#include <iostream>
#include <math.h>

// Kernel function to add the elements of two arrays
__global__
void add(int n, float *x, float *y){
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}

int main(void){

  int N = 1<<20;
  float *x, *y;

  // Allocate Unified Memory - accessible from CPU or GPU
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 1M elements on the GPU
  add<<<1, 1>>>(N, x, y);

  // Wait for GPU to finish before accessing on host
  cudaDeviceSynchronize();

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  //Free memory
  cudaFree(x);
  cudaFree(y);

  return 0;
}
```

**Figure 8.11: Vector addition example in the CUDA version**

#### 8.4.3.1.1 The new Magma library

Within the DEEP-EST project, UoI faced the concern of having to significantly refactor their codebase, so that they could develop multiple application versions supporting both CPUs and GPUs. To mitigate this problem, they developed a library (called “Magma”) that mimics the C++ standard library blueprint. The library takes care of linking the source code to either the C++ STL (in case of CPU) or CUDA Thrust (in case of GPGPU) libraries.

The Magma library is available to all as free-open-source-software via a public GitHub repository[103] and its functionality is detailed in Section 6.4 of this book.

```
magma::for_each(n_offset_size, v_coord_cell_size.size() - 1, [=]
#ifdef CUDA_ON
    __device__
#endif
    (auto const &i) -> void {
        it_coord_cell_size[i] = it_coord_cell_offset[i + 1] - it_coord_cell_offset[i];
    });
```

**Figure 8.12: Example of a for-each loop with Magma**

Figure 8.12 shows an example on how the Magma library is used in the NextDBSCAN application of UoI. The source code is identical for both the CPU and GPGPU platform with the exception of the necessary host and/or device annotations which CUDA requires to specify the execution target.

### *8.4.3.2 Using OpenACC*

Another option to offload code to the GPUs is using OpenACC via the NVIDIA NVHPC compiler. OpenACC is a directive-based performance-portable parallel programming model. With OpenACC applications can be ported to a wide variety of heterogeneous HPC hardware platforms and architectures with significantly less programming effort than required for a low level model such as CUDA. Programming with OpenACC should happen in 4 steps:

1. Identify parallelism (already done in Section 8.2)
2. Parallelize code parts with OpenACC
3. Express data locality
4. Optimize performance

After the analysis phase described in Section 8.2, it is known which code parts should be parallelized on the GPU. These code parts will be put within a pragma region as shown in Figure 8.13 for a small Jacobi iteration. The two nested inner loops (over `i`

[103] https://github.com/ernire/magma/tree/master

and `j`) can be parallelized. The `kernels` directive tells the compiler to analyse the code and look for parallel loops in the specified region. In this case, the compiler identifies two regions of code to generate an accelerator kernel. The compiler also analyses which arrays are used in the calculation and generates code to move `A` and `Anew` into GPU memory. The compiler even detects that it needs to perform a *max reduction* on the `error` variable.

The next step is to express the data locality. Sometimes not everything needs to be copied on and from the device. With the `data` pragma the relevant data locations can be specified. The `copy` clause in the `data` pragma tells the compiler that it should copy the `A` array to and from the device as it enters and exits the region, respectively. Since the `Anew` array is only used within the convergence loop, the `create` clause is used to request the compiler to create temporary space on the device, since we do not care about the initial or final values of that array.

```
#pragma acc data copy(A) create(Anew)
while ( error > tol && iter < iter_max ){
  error = 0.0;
  #pragma acc kernels
  {
    for( int j = 1; j < n-1; j++) {
      for( int i = 1; i < m-1; i++ ) {
        Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                            + A[j-1][i] + A[j+1][i]);
        error = fmax( error, fabs(A[j][i] - Anew[j][i]));
      }
    }
    for( int j = 1; j < n-1; j++) {
      for( int i = 1; i < m-1; i++ ) {
        A[j][i] = Anew[j][i];
      }
    }
  }
  if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);
  iter++;
}
```

**Figure 8.13: Jacobi example in the OpenACC version**

In following references the reader can find more detailed guides and courses to get started with OpenACC[104,105,106].

For the ESB, data objects have to stay in GPGPU memory as long as possible, so the programmer should radically limit copying between CPU and GPGPU as long as it is not strictly necessary for code correctness.

#### *8.4.3.3 Using OmpSs-2*

The OmpSs-2 task-based programming model supports message-passing libraries (MPI) and improved GPU programming of the MSA.

Herein we will cover the well-known N-Body benchmark, which numerically approximates the evolution of a system of bodies in which each body continuously interacts with every other body. A familiar example is an astrophysical simulation in which each body represents a galaxy or an individual star and all bodies attract each other through gravitational force.

In the benchmark presented here the particle space is divided into smaller blocks. Similarly, MPI processes are also divided into two groups: CPU processes and GPU processes. Firstly, GPU processes are responsible for computing the forces between each pair of blocks of particles; secondly, these forces are sent to CPU processes, where each process updates its blocks of particles using the received forces. The blocks of particles and forces are equally distributed amongst each MPI process within each group. Thus, each MPI process is in charge of computing the forces or updating the particles of a consecutive chunk of blocks.

[104] https://developer.nvidia.com/blog/getting-started-openacc/

[105] https://www.fz-juelich.de/SharedDocs/Termine/IAS/JSC/DE/Kurse/2020/ptc-gpu-openacc-2020.html?nn=2320772

[106] https://www.openacc.org/sites/default/files/inline-files/OpenACC_Programming_Guide_0.pdf

```
void nbody_solve(nbody_t *nbody, int num_blocks, int timesteps, float
    time_interval)
{
  assert(nbody != NULL);
  assert(timesteps > 0);
  . . .

  for (int t = 0; t < timesteps; t++) {
    if (nbody->gpus_group) { // CODE FOR GPU NODES
      recv_particles(local, num_blocks, peers, group_rank, MPI_COMM_WORLD); //
          Receive updated particles from CPU peers
      reset_forces(forces, num_blocks); // Reset the forces to zero
      . . .

      for (int r = 0; r < group_size; r++) { // Compute the forces between
          particles
        calculate_forces(forces, local, sendbuf, num_blocks);

        if (r < group_size - 1) {
          const int tagbase = get_shift_tagbase(r, num_blocks);

          exchange_particles(sendbuf, recvbuf, num_blocks, group_rank,
              group_size, tagbase, nbody->COMM_GROUP);
        }

        . . .
      }

      send_forces(forces, num_blocks, peers, group_rank, MPI_COMM_WORLD); //
          Send forces block to the CPU peer
    } else { // CODE FOR CPU NODES
      send_particles (local , num_blocks, peers, group_rank, MPI_COMM_WORLD); //
          Send updated particles to GPU peers
      recv_forces (forces, num_blocks, peers, group_rank, MPI_COMM_WORLD); //
          Receive forces from GPU peers
      update_particles(local , forces , num_blocks, time_interval); // Update
          particles in the CPU
    }
  }
  #pragma oss taskwait
  MPI_Barrier(MPI_COMM_WORLD);
}
```

**Figure 8.14: NBody solver**

The computation pattern in the code (Figure 8.14) is repeated during multiple time steps. The communication pattern during each time step consists of GPGPU processes, which exchange their particles with each other in a circular manner in order to compute the forces between their own particles against those from other GPGPUs. For the purpose of simplifying this pattern, this benchmark uses a different MPI communicator for the circular exchange. Once a GPU process finishes the computation of its forces, it sends the forces to the corresponding CPU process(es)

and then it receives the updated particles. MPI sends/receives are performed separately on each block.

The actual GPGPU computation takes place within the function `calculate_forces`. A closer look to this function in Figure 8.15 reveals that the programmer has indeed the choice of using either the CPU or the GPU version of this function.

```
void calculate_forces(forces_block_t *forces, const particles_block_t *block1,
                    const particles_block_t *block2, int num_blocks)
{
       for (int i = 0; i < num_blocks; i++) {
              for (int j = 0; j < num_blocks; j++) {
                     #ifdef USE_CUDA
                     calculate_forces_block_cuda(forces+i, block1+i, block2+j,
                         BLOCK_SIZE);
                     #else
                     calculate_forces_block(forces+i, block1+i, block2+j);
                     #endif
              }
       }
}
```

**Figure 8.15: Calculate_forces function**

The code part in Figure 8.16 shows the CPU version of the kernel associated to the computation of forces inside a block. It is worth noting that the programmer is responsible for annotating this function with OmpSs-2 pragmas in order to convert this kernel into a regular task, to be later executed in parallel by the CPU.

```
#pragma oss task label(calculate_forces_block) in(*block1, *block2)
    inout(*forces)
static void calculate_forces_block(forces_block_t *forces, const
    particles_block_t *block1, const particles_block_t *block2)
{
       float *x = forces->x;
       . . .

       for (int i = 0; i < BLOCK_SIZE; i++) {
              float fx = x[i], fy = y[i], fz = z[i];
              for (int j = 0; j < BLOCK_SIZE; j++) {
                     const float diff_x = pos_x2[j] - pos_x1[i];
                     . . .
                     }

                     fx += force * diff_x;
                     . . .
              }
              x[i] = fx;
              . . .
       }
}
```

**Figure 8.16: CPU kernel**

More interesting is to see which modifications are now necessary to convert the previous, original CPU code into its equivalent GPU code and, at the same time, render it compatible with OmpSs-2. In Figure 8.17 we add the CUDA kernel declaration in the header file `kernel.h`. It is important to highlight that the programmer is responsible for annotating this CUDA kernel as if it were a regular (i.e., CPU) function that can later be invoked by the OmpSs-2 runtime. Note, for instance, that now it is necessary to indicate the clauses `device` and `ndrange`. It can be readily seen that this procedure eases the development of GPU programming and is rendered possible thanks to the OmpSs-2 runtime, which takes care of data movements and correct synchronization between the host (CPU) and device (GPU) tasks and kernels following a true data-flow execution model.

```
#include "nbody.h"

#pragma oss task label(calculate_forces_block) in(*block1, *block2)
    inout(*forces) \
                device(cuda) ndrange(1, block_size, 128)
__global__ void calculate_forces_block_cuda(forces_block_t *forces, const
    particles_block_t *block1,
                                        const particles_block_t *block2, const
                                            int block_size);
```

**Figure 8.17: CUDA header file defining OmpSs-2 tasks for GPU**

Finally, the code in Figure 8.18 shows the `calculate_force_block_cuda` CUDA C kernel from the N-Body application. This kernel is almost identical to the CPU kernel illustrated in Figure 8.16. It is important to point out that the CUDA kernel code is located in a different file that is separately compiled by the CUDA C compiler. For completeness, the definition of the `forces_block_t` has been added to highlight that it is a `struct` of static arrays, thus suitable for host–device data movement. Data movement makes use of the CUDA unified memory. The OmpSs-2 runtime has been extended to explicitly manage data transfers, so that unified memory is no longer a hard requirement.

```
#include "kernel.h"

__global__ void calculate_forces_block_cuda(forces_block_t *forces, const
    particles_block_t *block1,
                                        const particles_block_t *block2, const
                                            int block_size)
{
        int id = (blockDim.x * blockIdx.x) + threadIdx.x;
        if (id >= BLOCK_SIZE) return;

        float *x = forces->x;
        . . .

        for (int j = 0; j < BLOCK_SIZE; j++) {
                const float diff_x = pos_x2[j]  pos_x1[id];
                . . .
                }

                fx += force * diff_x;
                . . .
        }
        x[id] = fx;
        . . .
}
```

**Figure 8.18: CUDA kernel**

#### *8.4.3.4 Using OpenMP 5.0*

GPGPU (or more generally, accelerator) offloading was introduced into the OpenMP standard with version 4.5 and enhanced functionality is provided with OpenMP 5.0. Like OpenACC and OmpSs, OpenMP relies on using directives and supports Fortran, C and C++. The fork-join model used by OpenMP 5.0 is similar to OpenACC, but OpenACC is more descriptive and OpenMP 5.0 is more prescriptive.

This section shows multiple steps to transform a basic "*SAXPY*" code into a fully functional OpenMP 5.0 application. It covers the most common and useful approaches to offload a computationally intensive loop to the GPU accelerators in the ESB and DAM modules of the DEEP-EST system.

The basic code contains two `for` loops, one for the initialization of the values and a second for the main operations. In this example we have wrapped the main `for` loop in a function in order to show how such externally defined methods can be called from within OpenMP sections. All vectors are allocated dynamically.

Porting the SAXPY code to the GPU using OpenMP 4.5/5.0 requires the addition of only one line of code (see Figure 8.19). This new line performs the memory transfers

between Host and Device and divides the computation of the for loop among the different threads in the GPU.

```
void saxpy(long long int n, float s, float* x, float* y, float* z){
  #pragma omp target teams distribute parallel for map(to:x[:n],y[:n])
      map(from:z[:n])
  for (auto i=0; i<n; i++)
    z[i] = s*x[i] + y[i];
}
int main(){

  long long int n = 1000000000;
  int s = 2.0;
  float* x = new float[n];
  float* y = new float[n];
  float* z = new float[n];
  for (auto i=0; i<n; i++){
    x[i] = i;
    y[i] = i%3;
  }
  saxpy(n, s, x, y, z);

  free(x);
  free(y);
  free(z);
}
```

**Figure 8.19: Example for OpenMP 5.0 offload**

Here we explain each one of the terms in the pragma call:

- `#pragma omp`: signals to the compiler that the following code section will be processed by OpenMP.
- `target`: tells the compiler that the following section of code will be executed on the GPU. This is equivalent to the definition of a kernel function around the for loop as shown in the CUDA code in Figure 8.11.
- `teams`: instructs the main thread in the Device to spawn multiple, isolated, threads associated with the different processor blocks (SMs) in the GPU.
- `distribute`: instructs the GPU to decompose the loop iterations and assign different chunks to the different teams requested.
- `parallel`: instructs each team master thread to spawn a group of threads for each team.
- `for`: distribute the loop iterations in each teams' chunk across the threads in the team.
- `map(to:…)`: perform a data transfer of the listed vectors from the Host to the Device on entering the OpenMP section.

- `map(from:…)`: perform a data transfer of the listed vectors from the Device to the Hoist on exiting the OpenMP section.

There are several metadirectives, declare target, macro defined directives, and device allocations, which will be explained below.

#### 8.4.3.4.1 Declare target

In the previous section one single `for` loop was offloaded to the accelerator, but in most useful cases the programmer wants to offload more complex code, usually encapsulated in functions (or kernels). Functions that can be called from within an accelerated `target` region must be defined by opening and closing `declare target` pragmas, as shown in Figure 8.20.

```
#pragma omp declare target
void saxpy(long long int n, float s, float* x, float* y, float* z){
  #pragma omp distribute parallel for
  for (auto i=0; i<n; i++)
    z[i] = s*x[i] + y[i];
}
#pragma omp end declare target
int main(){

  long long int n = 1000000000;
  int s = 2.0;
  float* x = new float[n];
  float* y = new float[n];
  float* z = new float[n];
  for (auto i=0; i<n; i++){
    x[i] = i;
    y[i] = i%3;
  }
  #pragma omp target teams map(to:x[:n],y[:n]) map(from:z[:n])
  saxpy(n, s, x, y, z);
  free(x);
  free(y);
  free(z);
}
```

**Figure 8.20: OpenMP 5.0 offload: Declare target example**

With this change, it is possible to offload the `saxpy` function to the accelerator in any location of the code. The function must only be called from within a `target` region (in the snippet above the scope of the target region contains only one line). The offloading line used in Figure 8.19 has been divided here in two parts: 1) the `target teams` pragma that spawns a set of master teams in the accelerator and performs all memory transfers to the device, and 2) the `distribute parallel for` pragma, called from

within the accelerator in the `saxpy` function, that segments the `for` loop and distributes it among the teams and the corresponding threads.

#### 8.4.3.4.2 Declare target + declare variant

```
void gpu_saxpy(long long int, float, float*, float*, float*);

#pragma omp declare variant (gpu_saxpy) match(construct={target})
void saxpy(long long int n, float s, float* x, float* y, float* z){
  #pragma omp for simd
  for (auto i=0; i<n; i++)
    z[i] = s*x[i] + y[i];
}
#pragma omp declare target
void gpu_saxpy(long long int n, float s, float* x, float* y, float* z){
  #pragma omp distribute parallel for
  for (auto i=0; i<n; i++)
    z[i] = s*x[i] + y[i];
}
#pragma omp end declare target
int main(){

  long long int n = 1000000000;
  int s = 2.0;
  float* x = new float[n];
  float* y = new float[n];
  float* z = new float[n];
  for (auto i=0; i<n; i++){
    x[i] = i;
    y[i] = i%3;
  }
  #pragma omp parallel
  saxpy(n, s, x, y, z);
  #pragma omp target teams map(to:x[:n],y[:n]) map(from:z[:n])
  saxpy(n, s, x, y, z);
  free(x); free(x);
  free(y);
  free(z);
}
```

**Figure 8.21: OpenMP 5.0 offload: Declare target + declare variant example**

If the programmer wants to use the `saxpy` function both in the host (CPU) and the device (accelerator), it is possible to create alternative kernels of the function with their respective OpenMP pragmas. Figure 8.21 shows that in the main code the function is called, first in the Host (without `target` pragma) and once in the Device (inside a `target` pragma). Two different versions of the routine are activated for each case. The `declare variant` call instructs the compiler to look for an alternative version of the code following the `match` conditions. In this case, if the function is called within a target region, the variant `gpu_saxpy` function is called.

This division of work is interesting for applications that want to perform the same procedure both on the Host and on the Device. This could allow workload balancing

between CPU and Accelerator, maximizing the use of the available computational resources.

#### 8.4.3.4.3 Macros

```
void saxpy(long long int n, float s, float* x, float* y, float* z){
  #ifdef __GPU__
  #pragma omp target map(to:x[:n],y[:n]) map(from:z[:n])
  #endif
  {
    #ifdef __GPU__
    #pragma omp teams distribute parallel for
    #else
    #pragma omp parallel for simd
    #endif
    for (auto i=0; i<n; i++){
      z[i] = s*x[i] + y[i];
    }
  }
}
```

**Figure 8.22: OpenMP 5.0 offload: Example for Macro usage**

The problem with the use of the `declare variant` clause is that important parts of the code need to be duplicated. This is a potential source of bugs and can complicate its maintenance. To avoid code duplication the OpenMP pragmas can be surrounded by macros defined by the user. In Figure 8.22 the Host and Device versions of the `saxpy` function have been separated by the use of the compile-time variable `__GPU__`. At compile-time it is possible to generate one version with offloading or a different version without offloading. This approach also allows the inclusion of details of the architecture. For example, the programmer can define the flags `__INTELCPU__`, `__AMDCPU__`, `__AMDGPU__`, `__NVIDIAGPU__`, corresponding to the four most common hardware architectures today. Each one will encompass a different OpenMP pragma line before the for loop in the `saxpy` code.

#### 8.4.3.4.4 Metadirectives

```
void saxpy(long long int n, float s, float* x, float* y, float* z){
  #pragma omp target map(to:x[:n],y[:n]) map(from:z[:n]) device(dvc)
  {
    #pragma omp metadirective when (device={arch("nvptx")}: teams distribute
        parallel for) default (parallel for simd)
    for (auto i=0; i<n; i++){
      z[i] = s*x[i] + y[i];
    }
  }
}
```

**Figure 8.23: OpenMP 5.0 offload: Metadirectives example**

The previous approach is very useful but can become cumbersome and it almost feels like OpenMP should support such a use case scenario. The OpenMP 5.0 standard does provide an alternative called *metadirectives*. Instead of using compilation macros the `metadirective` is built using the following schema:

```
#pragma omp metadirective \
        when (<condition> : teams distribute parallel for) \
        default (parallel for simd)
```

This structure allows the programmer to get rid of macro definitions and uses `<conditions>` to choose one OpenMP line instead of the `default` OpenMP line. In Figure 8.23 the condition selects the outcome of the `metadirective` based on the type of hardware architecture in which the loop is running.

This is a very handy option but presents two drawbacks: 1) it currently allows only two options, the one selected by the `<condition>` and the `default`, and 2) it is not currently supported by most compilers, including LLVM (March 2021).

#### 8.4.3.4.5 Macro defined directives

```
#ifdef __GPU__
#define _OMP_DIRECTIVE_ teams distribute parallel for
#define _OFFLOAD_ true
#else
#define _OMP_DIRECTIVE_ for simd
#define _OFFLOAD_ false
#endif

void saxpy(long long int n, float s, float* x, float* y, float* z){
  #pragma omp target if (_OFFLOAD_) map(to:x[:n],y[:n]) map(from:z[:n])
  {
    #pragma omp _OMP_DIRECTIVE_
    for (auto i=0; i<n; i++){
      z[i] = s*x[i] + y[i];
    }
  }
}
```

**Figure 8.24: OpenMP 5.0 offload: macro defined directives example**

A workaround to avoid code duplication and simplify its main structure, without using metadirectives, is to define the OpenMP lines with a global macro that can then be referenced inside the code as shown in Figure 8.24. This approach makes the code much cleaner but requires the programmer to specify all the possible OpenMP calls at the beginning of the code. This could lead to a large number of macros that can be included in a separate file. Although this approach can complicate the maintenance of

the code, the final result is much cleaner and easier to follow for non-experts in OpenMP.

#### 8.4.3.4.6 Device allocation

```
#include <omp.h>

#ifdef __GPU__
#define _OMP_DIRECTIVE_ teams distribute parallel for
#else
#define _OMP_DIRECTIVE_ parallel for simd
#endif
void init(long long int n, float* x, float* y, int ndvc, int dvc){
  #pragma omp target if (ndvc>0) is_device_ptr(x,y) device(dvc)
  {
    #pragma omp _OMP_DIRECTIVE_
    for (auto i=0; i<n; i++){
      x[i] = i;
      y[i] = i%3;
    }
  }
}
void saxpy(long long int n, float s, float* x, float* y, float* z, int ndvc,
    int dvc){
  #pragma omp target if (ndvc>0) map(from:z[:n]) is_device_ptr(x,y) device(dvc)
  {
    #pragma omp _OMP_DIRECTIVE_
    for (auto i=0; i<n; i++){
      z[i] = s*x[i] + y[i];
    }
  }
}
int main(){
  long long int n = 1000000000;
  int s = 2.0;
  float* x;
  float* y;
  float* z;
  int ndvc = omp_get_num_devices();
  int dvc = omp_get_default_device();
  int hst = omp_get_initial_device();
  x = (float*) omp_target_alloc(sizeof(float) * n, dvc);
  y = (float*) omp_target_alloc(sizeof(float) * n, dvc);
  z = (float*) malloc(sizeof(float) * n);
  init(n, x, y, ndvc, dvc);
  saxpy(n, s, x, y, z, ndv, dvc);
  omp_target_free(x, dvc);
  omp_target_free(y, dvc);
  free(z);
}
```

**Figure 8.25: OpenMP 5.0 offload: Device allocation example**

One of the most important optimizations in OpenMP, and in general for any code that uses offloading, is to minimize the transfers of information between the Host and the Device. Up until now we have used the `map(…)` clause. This performs a memory transfer between the CPU and the Accelerator. To avoid such transfer, the programmer can allocate memory directly on the Accelerator with the API functions `omp_target_alloc()` and `omp_target_free()`. These two functions work in almost the same way as C/C++ `malloc()` and `free()` functions, but require also the number of the target accelerator device. The memory allocation function returns a pointer that is associated with memory in the accelerator. Any access to this pointer from code outside a target region will produce a memory access error.

In the initialization and in the `saxpy` functions shown in Figure 8.25, the pointer corresponding to the dynamically allocated accelerator memory is identified by the clause `is_device_ptr(…)`. The allocation functions must be called at any point outside the target region, but the pointers must only be referenced inside them.

In this snippet we show how the `saxpy` function receives the addresses of the two dynamically allocated accelerator vectors and returns the result by memory transfer to the Host device using the `map(from:…)` clause. This version of the `saxpy` test results in the best performance. It is also the cleanest version and the easiest to maintain. We recommend other programmers to understand the sections above, but to use the pattern presented in this section as a starting point of their code porting procedure.

### *8.4.4 Code porting and optimisation on the DAM*

The DAM nodes are equipped with two different kinds of accelerators: NVIDIA Tesla V100 GPUs and Intel Stratix10 FPGAs. Section 8.4.3 already covers porting to GPUs. This section will have a look at the FPGAs.

#### *8.4.4.1 oneAPI*

Intel oneAPI[107] is an open, unified programming model. It is used to simplify programming across CPUs, GPUs, FPGAs and other accelerators. On the DEEP-EST system oneAPI is interesting for either working on FPGAs or CPUs. Information on how to work with it on FPGAs can be found here[108]. Section 7.4 of this book explains how to use it for GPU portable programming.

[107] https://software.intel.com/content/www/us/en/develop/tools/oneapi.html

[108] https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/fpga.html

#### 8.4.4.2 OpenCL

OpenCL is an industry standard for programming systems that contain several heterogeneous devices and memory spaces. Like CUDA, the standard uses a kernel language to specify optimized code parts that run on accelerators like GPGPUs or FPGAs, an API to define and direct code parts to be run on a specific device, and an API to manage the (usually disjoint) memory spaces of devices and transfer data between them. OpenCL is used in non-HPC applications, such as heterogeneous embedded or mobile systems, and it has emerged as the method of choice to program FPGAs if the significant additional effort to develop RTL or VHDL code is seen as not worth the potential performance gain.

OpenCL for Intel CPUs and the FPGAs of the DAM module is provided by the Intel® FPGA SDK for OpenCL™ [109], which is currently in version 20.4, complemented by a BSP (board support package) matching the installed Stratix 10 devices. Figure 8.26 shows an example of an OpenCL kernel to compute and print out the Fibonacci numbers on the FPGA. A very detailed programming guide with information on how to build and optimize your OpenCL kernels, how to adapt your host program, and how to compile the code, can be found here[110].

```
__kernel void print_fibonacci_number(int n)
{
  int2 history = (int2) (1, 1);

  for (int i = 0; i < n; i ++)
    history = (int2) (history.y, history.x + history.y);

  printf("fib(%d) = %d\n", n, history.x);
}
```

**Figure 8.26: Fibonacci OpenCL kernel**

OpenCL is also supported on a range of GPUs, including the NVIDIA Tesla V100.

### 8.4.5 Data Analytics & Machine Learning frameworks

The DEEP-EST system also provides specific frameworks targeting Data Analytics and Machine Learning applications:

[109] https://www.intel.de/content/www/de/de/programmable/products/design-software/embedded-software-developers/opencl/support.html

[110] https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807965224.html

- TensorFlow: an end-to-end platform that makes it easy for developers to build and deploy ML models.[111] On the DEEP-EST system TensorFlow versions 2.2 and 1.13.1 based on Python 3.6.8 are deployed.
- PyTorch: a Python package that provides two high-level features: Tensor computation (like NumPy) with GPU acceleration and Deep Neural Networks built on a tape-based autograd system[112]. On the DEEP-EST system PyTorch versions 1.1.0 and 1.4.0 based on GCC are deployed.
- Horovod: a distributed deep learning training framework for TensorFlow, Keras, PyTorch, etc.[113]. On the DEEP-EST system Horovod version 0.16.2 based on GCC and ParaStationMPI is deployed.

These frameworks can be used on either CPUs or GPUs, so in theory they can run on all three compute modules. But since for data analytics (in most cases) a large amount of memory is necessary, the DAM would in general be the best suited module.

For trained networks there is the option of generating an interoperable ONNX[114] version which can be used for inference on many platforms including accelerators, which do not support the full-blown neural network development platforms listed above. This is a potential migration path to the FPGA accelerators of the DAM nodes, should users be interested in performing inference there.

## 8.5 Use of multiple modules

To run an application on multiple modules, it has to be partitioned: the code parts optimized for the different modules need to be separated and communication between the different parts has to be coded (preferably using MPI or files). As shown in Figure 8.3 there are three ways of using multiple modules: running on multiple modules at the same time (multi-module jobs), running consecutively on different modules (job chains and workflows), or a combination of both.

There might be jobs that need more than one module either at the same time or consecutively. In both cases one has to first divide the code in the parts for each module, and then make sure that both parts can communicate if necessary (either with MPI or through the file system).

[111] https://www.tensorflow.org/tutorials?hl=en

[112] https://github.com/pytorch/pytorch; the term refers to reverse-mode automatic differentiation.

[113] https://github.com/horovod/horovod#usage

[114] https://onnx.ai/, term refers to Open Neural Network Exchange

### 8.5.1 Running on Multiple Modules at the Same Time – Multi-Module Jobs

The Slurm resource manager supports allocating heterogeneous jobs (using more than one module). Figure 8.27 shows an example how to allocate one node on the CM and one node on the DAM and executing the `hostname` command on both.

```
[zitz1@deepv ~]$ srun --account=deep --partition=dp-cn -N 1 -n 1 hostname :
    --partition=dp-dam -N 1 -n 1 hostname
dp-cn01
dp-dam01
```

**Figure 8.27: `Srun` command to allocate a heterogeneous job**

Heterogeneous jobs can also be launched in a batch script using the `packjob` keyword. For information on functionalities regarding heterogeneous jobs in Slurm please see the DEEP-EST Wiki[115].

#### 8.5.1.1 Using MPI

After an MPI application has started its processes as shown above, they can determine their module affiliation and thus coordinate their work accordingly. For this purpose, the processes can query on which module they are currently running by looking it up as a *Module ID* in the `MPI_INFO_ENV` object, which is provided by the MPI standard for environmental adaptations (see Figure 8.28).

This assignment between ID and modules is not fixed, but can be set by the user according to the needs of the application by using the environment variable `PSP_MSA_MODULE_ID`. However, if the user does not set such a module affiliation, the assignment of the IDs is performed automatically according to the order of the modules in the `srun` call: the first module gets ID 0, the second module ID 1, and so forth. Hence, it is the user's responsibility to match the respective `srun` call with an appropriate evaluation of the queried module IDs at application level.

[115] https://deeptrac.zam.kfa-juelich.de:8443/trac/wiki/Public/User_Guide/Batch_system#HeterogeneousjobswithMPIcommunicationacrossmodules

```c
#include <mpi.h>
int main(int argc, char* argv[])
{
int world_rank, world_size, flag;
    char value[MPI_MAX_INFO_VAL];
    MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Info_get(MPI_INFO_ENV, "msa_module_id", MPI_MAX_INFO_VAL, value, &flag);
    if (flag) { /* This MPI environment is modularity-aware! */
        printf("(%d) \"msa_module_id\" found. ID is %s\n", world_rank, value);
    } else {
        printf("(%d) Found no entry for \"msa_module_id\"\n", world_rank);
    }
    MPI_Finalize();
    return 0;
}
```

**Figure 8.28: MPI standard**

### *8.5.1.2 Topology-aware MPI Communicator Creation*

In addition to querying explicitly for the module affiliation, it is possible to split the `MPI_COMM_WORLD` communicator into sub-communicators reflecting the module affinity of processes by using the new communicator spilt type `MPIX_COMM_TYPE_MODULE`. However, please note that this split type is an extension in ParaStationMPI and that it is hence *not* part of the official MPI standard! One may use the macro `MPIX_TOPOLOGY_AWARENESS` to test whether this feature is available or not:

```c
#if defined(MPIX_TOPOLOGY_AWARENESS) && MPIX_TOPOLOGY_AWARENESS
/* MPIX_COMM_TYPE_MODULE available as split type: */
MPI_Comm_split_type(MPI_COMM_WORLD, MPIX_COMM_TYPE_MODULE, 0,
MPI_INFO_NULL, &split_comm);
#else
split_comm = MPI_COMM_WORLD;
/* MPIX_COMM_TYPE_MODULE _not_ available... */
#endif
```

**Figure 8.29: MPI_Comm_split_type**

Please also note that to use these extensions, the so-called *Topology Awareness* of ParaStationMPI must be enabled, which has to be done at compile time of the MPI library by using the configure switch `--with-topology-awareness`, plus explicitly setting the environment variable `PSP_MSA_AWARENSS=1` for the MPI sessions.

#### *8.5.1.3 Using MSA-aware Patterns for Collectives*

When topology awareness is enabled for ParaStationMPI, locality information as well as module affiliations can be taken into account by collective MPI operations for choosing optimized communication patterns, for example, for global reduction algorithms. In doing so, the locality awareness can be two-fold: (1) with respect to intra-node vs. inter-node communication (*SMP awareness*), and (2) with respect to inter-module vs. intra-module communication (*MSA awareness*). The following environment variables can be used for enabling these different degrees of topology awareness:

- `PSP_SMP_AWARENESS=1` – Generally, take locality information into account, e.g. for a meaningful use of `MPI_Win_allocate_shared`. This feature is enabled by default.
- `PSP_MSA_AWARENESS=1` – Generally activate the consideration of modular topologies. This feature is *not* enabled by default (see also Section 8.5.1.2).
- `PSP_SMP_AWARE_COLLOPS=1` – Enable the use of MPICH's SMP-aware collectives. This feature is disabled by default and requires SMP awareness in general (see above).
- `PSP_MSA_AWARE_COLLOPS=0|1|2` – Select the feature level for MSA-aware collectives:
  - `0` – Disable MSA awareness for collective MPI operations.
  - `1` – Enable MSA awareness for collective MPI operations. This feature is enabled by default if `PSP_MSA_AWARENESS=1` is set.
  - `2` – Apply MSA awareness *recursively* in multi-level topologies. For MSA plus SMP awareness, this requires that also `PSP_SMP_AWARENESS=1` is enabled.

The benefits of these different feature levels will depend on the patterns and settings of the applications. Therefore, at this point the user is advised to test the different options and check for which setting the application achieves the best performance. Moreover, it has to be emphasized that only a suitable subset of the MPI collectives actually do provide topology awareness. These are: `MPI_Barrier`, `MPI_Bcast`, `MPI_Reduce`, `MPI_Allreduce` and `MPI_Scan`, as well as their respective non-blocking counterparts.

#### *8.5.1.4 Realizing Workflows on MPI Level*

To pass data between workflow steps, the DEEP-EST project supports different approaches – for instance, using fast local storage, and/or using the global parallel file system. In this subsection, a further approach will briefly be introduced: the use of the standardized `MPI_Comm_connect`/`accept` API for passing data directly via MPI

messages between workflow steps. According to this approach, the preceding step of a workflow application opens a so-called *port* and forwards this port information to the subsequent step, which then in turn can connect to it so that both MPI sessions can communicate directly via an inter-communicator. A good approach for passing the port information is the use of a small file, where the preceding workflow step puts the port name when the end of this phase is reached. The next workflow step can wait for this file to be created and then connect to receive the data directly via MPI communication, which avoids the considerable overhead of storing and retrieving volume data via a storage device. The two functions in Figure 8.30 show draft code for realizing this between two steps of a workflow.

```
int comm_accept(void){
/* Predecessor step in workflow: */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) { /* = root */
    char port_name[MPI_MAX_PORT_NAME];
    std::ofstream portfile;
        MPI_Open_port(MPI_INFO_NULL, port_name);
portfile.open("portfile.txt");
portfile << port_name;
portfile.close();
}
MPI_Comm intercomm;
int remote_size;
MPI_Comm_accept(port_name, MPI_INFO_NULL, 0 /*= root */, MPI_COMM_WORLD,
         &intercomm);
    MPI_Comm_remote_size(intercomm, &remote_size);
/* Pass data to successor step in workflow: */
MPI_Send(..., intercomm);
/* ... */
}

int comm_connect(){
/* Successor step in workflow: */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) { /* = root */
        char port_name[MPI_MAX_PORT_NAME];
        std::ifstream portfile;

        do {
            portfile.open("portfile.txt");
            sleep(1);
} while (!portfile.is_open());
getline(portfile, line);
strncpy(port_name, line.c_str(), line.size()+1);
portfile.close();
}

MPI_Comm_connect(port_name, MPI_INFO_NULL, 0 /* = root */, MPI_COMM_WORLD,
          &intercomm);
    MPI_Comm_remote_size(intercomm, &remote_size);
    /* Receive data from predecessor step in workflow: */
    MPI_Recv(..., intercomm, MPI_STATUS_IGNORE);
/* ... */
}
```

**Figure 8.30: MPI_comm_accept and MPI_comm_connect**

How such steps of a workflow can be orchestrated at job level is described in the following sections.

### 8.5.2 Running on Multiple Modules Consecutively – Workflows

There are two ways of running jobs consecutively on the system: Using the `--delay` switch (where the jobs can have an overlap, e.g., for data exchange via MPI) or using Slurm job dependencies (where jobs start one after another).

#### 8.5.2.1 `--delay` switch

The Slurm version running on the DEEP-EST system allows overlapping jobs inside a workflow: with the `--delay n` option the start of jobs in a job pack can be delayed by `n` minutes from the start of the first job of the job pack. Figure 8.31 shows a small example.

After submission of this job pack, Slurm divides it into separate jobs, and ensures that the delay is respected by using reservations, rather than the usual scheduler. Using this approach, the user has to estimate the duration of each sub-job to make a good choice of the interval that the jobs will be delayed. As the user-provided delay values tend to be not so accurate, we also provide API calls that a job can use to request Slurm to change the start times of the remaining jobs in the workflow it belongs to.

```
#!/bin/bash
#SBATCH -p sdv -N2 -t3
#SBATCH packjob
#SBATCH -p sdv -N1 -t3 --delay 2
srun hostname
```

**Figure 8.31: Example for --delay switch**

#### 8.5.2.2 Slurm job dependencies

With this approach the jobs will not have a guaranteed overlap, yet will still run in a specified sequence. Using the Slurm dependencies, jobs can be chained with the following script:

```
!/usr/bin/env bash

if [ $# -lt 3 ]
then
    echo "$0: ERROR (MISSING ARGUMENTS)"
    exit 1
fi

LOCKFILE=$1
DEPENDENCY_TYPE=$2
shift 2
SUBMITSCRIPT=$*


if [ -f $LOCKFILE ]
then
    if [[ "$DEPENDENCY_TYPE" =~ ^(after|afterany|afterok|afternotok)$ ]]; then
        DEPEND_JOBID=`head -1 $LOCKFILE`
        echo "sbatch --dependency=${DEPENDENCY_TYPE}:${DEPEND_JOBID}
            $SUBMITSCRIPT"
        JOBID=`sbatch --dependency=${DEPENDENCY_TYPE}:${DEPEND_JOBID}
            $SUBMITSCRIPT`
    else
        echo "$0: ERROR (WRONG DEPENDENCY TYPE: choose among 'after',
            'afterany', 'afterok' or 'afternotok')"
    fi
else
    echo "sbatch $SUBMITSCRIPT"
    JOBID=`sbatch $SUBMITSCRIPT`
fi

echo "RETURN: $JOBID"
# the JOBID is the last field of the output line
echo ${JOBID##* } > $LOCKFILE

exit 0
```

**Figure 8.32: Script for job chains**

Job scripts can then be submitted in the following way:

```
./chain_jobs.sh lockfile afterok simple_job.sh
```

This creates a chain of jobs with the dependency type `afterok`. This halts the allocation of such jobs until the independent job finishes with success. The currently running independent job, when it deems fit, calls an API function to change the dependency type of all its dependent jobs to the type `after`. This enables Slurm to consider these jobs for allocation, provided that the resources are available.

For more details, see the DEEP-EST Wiki[116].

[116] https://deeptrac.zam.kfa-juelich.de:8443/trac/wiki/Public/User_Guide/Batch_system#Workflows

## 8.6 File system and Storage

A number of different storage locations and file systems are available on the DEEP-EST system:

- JSC GPFS file systems, provided via the JUST storage system and mounted on all JSC systems.
- Parallel BeeGFS `/work` file system, available on all the nodes of the DEEP-EST system and hosted on the SSSM module.
- Parallel BeeOND file systems, created for the lifetime of Slurm jobs on demand and using local node storage devices (SSDs or Persistent Memory).
- Local ext3/ext4 file systems hosted on the CM, DAM and ESB nodes.

The next subsections will briefly describe each file system. More details can be found in the DEEP-EST wiki[117].

### *8.6.1 Permanent Storage (GPFS)*

In the usage model of JSC, each user has different home directories for each of the systems that they are using, so for the DEEP-EST system there will be a directory located in

```
/p/home/jusers/username/deep
```

These home folders have a low space quota and are meant to be used for configuration files, ssh keys, etc.

Data and computational resources are assigned to projects. As a consequence, each user can create folders within each of the projects that they are part of. For the DEEP project, the project folder is located under

```
/p/project/cdeep/username
```

Here is where the user should place data. Both `/p/home` and `/p/project` are provided by the shared GPFS file systems.

All data stored in the GPFS file system is regularly backed up by JSC.

### *8.6.2 Shared Fast Storage (BeeGFS) on SSSM*

The SSSM module hosts a total of 304 TB of storage managed by the BeeGFS parallel file system[118]. The data is stored in two RAID arrays with 24 disks each, using a RAID6 storage scheme. Four file system data servers provide access to these data, which are

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

[118] https://www.beegfs.io/docs/BeeGFS_Flyer.pdf

handled through BeeGFS clients on each of the CM, DAM and ESB nodes via standard POSIX I/O interfaces. The SSSM is connected to the rest of the system using 40 Gbit/s Ethernet technology, and data are passed on to the InfiniBand fabric via IP gateways. Metadata is handled by two additional servers and resides in two SSD RAID arrays.

Users just have to move data into the `/work` file system tree to use the SSSM BeeGFS – standard POSIX interfaces are supported in all the relevant programming frameworks.

As the name implies, the SSSM is considered a temporary storage device mainly to serve data required by applications, which run on the DEEP-EST system. Users are free to leave data on that system, but there is no backup and in case of resource shortage, data will be deleted.

### *8.6.3 Local Storage*

The compute nodes of the different modules expose some local storage devices that can be used (via ext3/ext4 file systems) during job execution. On the CM, DAM and ESB, local SSDs on each node are available via `/scratch` directory. It is meant to be used instead of `/tmp` (which should be avoided). Please, consider that `/scratch` is local to each node, hence data in `/scratch` cannot be shared between nodes. Additionally, data in `/scratch` will be removed once the job is finished. The size of `/scratch` is:

- CM and ESB nodes: ~380 GB
- DAM nodes: ~128 GB

On the DAM nodes there is additional local storage available through NVMe devices in:

- /nvme/scratch: ~1.5 TB (formatted with xfs)
- /nvme/scratch2: ~1.5 TB (formatted with ext4)

As for the data in `/scratch`, the data in the `/nvme/scratchX` directories will be removed at job termination. The DAM nodes furthermore expose some very fast persistent memory which can (depending on the operation mode) directly be used by applications and is described in Section 8.7.1.

### *8.6.4 Local storage – BeeOND*

The Slurm installation on the system provides a new switch `--beeond` for `sbatch` / `srun` / `salloc` commands. When this command is used, Slurm triggers the mechanism to start for this job the BeeOND server and clients on each assigned node

at allocation time. The server and clients are then properly removed at the end of a job and all the data is deleted.

BeeOND provides the same POSIX interfaces as the standard BeeGFS, but the data is actually stored across node-local devices. Depending on the partition size and fabric used on the module(s), significantly higher I/O bandwidths are available compared to the BeeGFS system on the SSSM.

In contrast to the use of `/scratch` devices, the BeeOND data is available to all nodes in a job partition, regardless of its physical location.

## 8.7 Using DEEP-EST specific features

### *8.7.1 Persistent memory*

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

In Memory Mode no changes to the application are required: the installed DRAM acts as a memory cache and the Intel Optane Persistent Memory transparently offers its full memory capacity to the OS and to applications. However, memory contents is volatile here. In DEEP-EST, the partner ASTRON has made use of this mode for applications running on the DAM nodes. No specific changes or adaptations were required to the base OS of the DAM or other SW packages -- Memory Mode is enabled via UEFI/BIOS settings and requires a node reboot. To get DAM nodes configured for Memory Mode, please contact the DEEP-EST support[119] to reconfigure some DAM nodes and create a reservation for you.

In App Direct Mode, DRAM and persistent memory are mapped onto separate memory address spaces (seen as memory nodes by Linux), and applications have to be modified in order to exploit the different characteristics of the two memory technologies. Access to the persistent memory occurs through regular load and store operations. Intel has released the Open Source Persistent Memory Development Kit (PMDK[120]) as Open Source, and recent Linux distributions do fully support it.

[119] sup@deep-est.eu

[120] Information about PMDK is available from https://pmem.io/pmdk/, including links to open repositories for source code and binaries

A special use case of App Direct mode is to map a file system onto a non-volatile memory partition; for this, the `fs-dax` layer provided by PMDK enables file system access while avoiding the need to go through a block device chain. For I/O-heavy applications, this usage mode can provide significant speed-ups, as for instance reported by the NextGenIO project[121]. The BeeOND parallel file system has been adapted to use the persistent memory as a storage target, enabling a job running on *n* DAM nodes to use a transient BeeGFS file system placed onto the n×3 TByte of persistent memory at a bandwidth significantly exceeding those achievable for the NVMe SSDs.

App direct mode and PMDK are in principle supported by the current base OS of the DAM (CentOS 7), which runs the 3.x Linux kernel. Newer OS versions (such as CentOS 8 with kernel 4.x) however provide significantly better performance, and experiments were run with a back-ported 4.19 kernel to establish whether the DAM nodes would be fully functional with a combination of CentOS 7 and such kernel. Therefore, access to BeeOND using the persistent memory will be available once the kernel version has been updated accordingly.

### *8.7.2 SIONlib (MSA features)*

SIONlib[122] is an I/O concentrator library which can significantly speed up large-scale parallel I/O. It allows users to read and write binary data to/from several thousands of processors into one or a small number of physical files. SIONlib provides simplified file handling for parallel programs which logically read or write binary data in parallel into separate files (task-local files), yet want to avoid the significant management overhead caused by having thousands of these files.

For general information on SIONlib please see the SIONlib documentation[123]. During the DEEP-EST project, three new features were added to the library: MSA aware collectives, I/O forwarding, and a CUDA-aware interface. Within this document we will concentrate on those three new features. The basics will be explained in the following subsections, but there is a more detailed description in the DEEP-EST Wiki[124].

#### *8.7.2.1 MSA aware collectives*

Recent versions of SIONlib allow all parallel processes to take part in the I/O operation which enables an exchange of I/O data between the processes, allowing a subset of

[121] Information about the NEXTGenIO project can be found at http://www.nextgenio.eu/.

[122] https://www.fz-juelich.de/ias/jsc/EN/Expertise/Support/Software/SIONlib

[123] https://apps.fz-juelich.de/jsc/sionlib/docu/current/index.html

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

all processes (the `collector` processes) to perform the transfer of data to the storage for all the processes. To achieve the maximum performance benefit, the `collector` processes should be the ones that are placed on parts of the MSA with a high bandwidth link to the parallel file system holding the large files created by SIONlib.

The new feature adds a MSA algorithm for the selection of collector processes. This algorithm is portable and relies on platform specific plug-ins which are specified during the installation of SIONlib (so this is nothing the user on the DEEP-EST system has to worry about). Through these plug-ins processes, which run on parts of the system that are well suited for the role of I/O, the collector processes are identified.

The MSA aware collective I/O has to be enabled when opening a file. This is done using in the open function the `file_mode` argument, which contains a string that consists of a comma-separated list of keys and key value pairs. The word `collmsa` must appear in that list to select MSA aware collective I/O, so the open-call should look like this:

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

The next step is to specify the nodes that should be used by setting an environment variable. For example, to select nodes from the DAM:

```
export SION_MSA_COLLECTOR_HOSTNAME_EREGEX="dp-dam.*"
```

#### *8.7.2.2 I/O forwarding*

The collective approach mentioned above has some constraints that make it inapplicable in certain scenarios:

- By design, collective I/O operations force application tasks to coordinate. This is at odds with SIONlib's world view of separate files per task that can be accessed independently.
- Collector tasks in general have to be application tasks, i.e. they have to run the user's application. This can generate conflicts on MSA systems, if the nodes that are capable of performing I/O operations efficiently are part of a module that the user application does not map well onto.

The new feature, called I/O forwarding, helps in both scenarios. It works by relaying calls to low-level I/O functions (e.g. `open`, `write`, `stat`, etc.) via a remote procedure call (RPC) mechanism from a client task (running the user's application) to a server task (running a dedicated server program), which then executes the functions on behalf of the client. Because the server tasks are dedicated to performing I/O, they can dynamically respond to individual requests from client tasks rather than imposing

coordination constraints. Also, on MSA systems the server tasks can run on different modules than the user application.

```
#!/bin/bash
# Slurm's heterogeneous jobs can be used to partition resources
# for the user's application and the forwarding server, even
# when not running on an MSA system.
#SBATCH --nodes=32 --partition=dp-cn
#SBATCH packjob
#SBATCH --nodes=4 --cpus-per-task=1 --partition=dp-dam

module load "some compiler" ParaStationMPI SIONlib/1.7.6

# Defines a shell function sionfwd-spawn that is used to
# facilitate communication of MPI ports connection details
# between the server and the client.
eval $(sionfwd-server bash-defs)

# Work around an issue in ParaStationMPI
export PSP_TCP=0

# Spawns the server, captures the connection details and
# exports them to the environment to be picked up by the
# client library used from the user's application.
sionfwd-spawn srun --pack-group 1 sionfwd-server

# Spawn the user application.
srun --pack-group 0 user_application

# Shut down the server.
srun --pack-group 0 sionfwd-server shutdown

# Wait for all tasks to end.
wait
```

**Figure 8.33: Sample job script to use I/O forwarding with SIONlib**

To use the I/O forwarding within the application it has to be selected when opening the file. This is done by adding the word `sionfwd` to the `file_mode` argument of SIONlib's open functions:

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

Be aware that the server processes are not spawned by MPI, so the server tasks have to be launched from the user's job script before the application tasks are launched. A typical job script is shown in Figure 8.33.

#### *8.7.2.3 CUDA aware interface*

To more closely match the programming interface offered by other libraries (such as ParaStationMPI), the SIONlib functions have been made CUDA aware. This means that applications are allowed to pass device pointers, which point to device-memory, to the various read and write functions of SIONlib without the need to manually copy their contents to the host memory. The user may pass the device pointers as the `data` argument to SIONlib's write and read functions.

## 8.8 Summary of lessons learned

The DEEP-EST project has demonstrated the potential of the MSA. The flexibility of the MSA concept allows very different usage models, so that a wide range of different applications can be addressed. This was shown and evaluated by a selection of large-scale, real-world applications from research fields relevant for the European research arena. Most of the DEEP-EST applications combine HPC computation with advanced data processing and analytics and therefore represent the HPC as well as the HPDA areas. Thus, they do consist of multiple parts with different resource requirements, which is suitable to assess the potential of the MSA and the DEEP-EST system.

### *8.8.1 Achievements of each application development team*

During the project lifetime the application teams showed some very promising results which made the DEEP-EST project a part on their way towards Exascale:

- NMBU – Neuroscience: The focus on performance and scalability in the DEEP-EST project has allowed NMBU to enhance performance of their applications. This has driven the development in NEST of the optimised spike-delivery algorithm and the advanced dry-run mode. The work in the DEEP-EST project on the NEST-Arbor and NEST-Elephant couplings to combine on the one hand simulations at different levels of description and on the other hand simulations and analysis has shown the potential of distributing different parts of a workflow across different modules of a MSA.
- NCSA – Molecular Dynamics: During the DEEP-EST project NCSA came to some very important conclusions: in computer-aided drug design or life sciences on the MSA one can optimise the price/performance ratio by choosing the appropriate configuration of compute nodes for each particular task. The multi-GPU FMM was developed as part of this project because FMM starts to become beneficial for large volumes of the simulation box (more than the previously used PME algorithm). This new functionality allows the utilization of FMM on large number of GPUs and opens new possibilities for GROMACS to perform very

large simulations in fields like material science, polymer science, molecular biology, nanostructures and condensed systems. For biological systems in life science research, the existing PME method already provides excellent performance on the MSA.

- ASTRON – Radio Astronomy: During the DEEP-EST project, ASTRON has made significant improvements to both of their applications: the Correlator and the Imager. The use of tensor-core technology will have a disruptive impact on correlators, due to their order-of-magnitude increase in performance and significant reduction in energy consumption when compared to the use of regular GPU cores. It was also shown that for newer generation GPUs the benefit even increases. ASTRON explored a new technique, called W-tiling. This significantly reduces the amount of memory used to create (very) large sky images, at the expense of a minor increase in computations, so that the painstaking effort of stitching hundreds of facets together belongs to the past. All in all, the DEEP-EST project enabled ASTRON to improve the overall performance of the imager and brings them a big step closer to Exascale imaging. Even if the results for the FPGA imager were not as positive as originally expected, the experience that was obtained with the OpenCL/FPGA toolkit has been very useful. ASTRON now uses this experience for other applications where FPGAs are indispensable, such as in the upgrade of the LOFAR stations.
- KU Leuven – Space Weather: very valuable experience has been gained in the usage of OpenMP5.0 to offload code to the GPU. As a result the xPic code is now accelerated in a portable, vendor-independent manner. KU Leuven showed the nearly perfect scalability for the accelerated particle solver and the code was also identified as a good candidate for Exascale scalability. On the road towards Exascale, KU Leuven believes in the continuous development of the code xPic and coupling its execution with multiple on-the-fly machine learning analysis tools. KU Leuven has already applied for a pilot program with the LUMI supercomputer centre where the Cluster-Booster architecture will be deployed using AMD CPUs and GPUs. All developments during the DEEP-EST project led to a good energy balance of the code.
- UoI – Data Analytics in Earth Science: By completely rewriting two of their codes (NextDBSCAN and NextSVM) UoI made a huge step towards Exascale. Both applications are now much stronger than the previous applications. Research within the project indicates that NextDBSCAN is now a good candidate application for Exascale systems, using both CPUs and GPUs. The results with the Horovod framework for distributed deep learning show that more work must

be done in order for it to reach Exascale system potential. Another achievement during the DEEP-EST project was the development of the Magma library to ease the porting efforts.

- CERN – High Energy Physics: The DEEP-EST project was an important part for the High Energy Physics community on the way towards Exascale HPC systems for CMS reconstruction workloads as well as for CMS classification. Porting the most time critical parts of the reconstruction to NVIDIA GPUs resulted in a significant performance gain. The work done in DEEP-EST has already been included in the official CMSSW stack.

In addition to the evaluation of the MSA concept the DEEP-EST project allowed to gain many valuable experiences:

### *8.8.2 Portability nearly as important as performance*

In the beginning of the project, and so also during the planning phase of the project, the idea was to equip the ESB with many-core CPUs of the Intel Xeon Phi series. After a few months within the project it became clear that this would not be possible, so the plan changed to using GPUs. This led to some difficulties for some of the applications, because they did not have GPU-code available. For example, KU Leuven had only a version of xPic optimised for Intel Xeon Phis. Also UoI and CERN had only CPU based code (with multi- and many-core versions). Each one of the three application partners used a different approach to implement a new GPU version, all of them striving towards a portable solution to become vendor independent.

KU Leuven used the pragma based OpenMP 5.0 offload (Section 5.4.3.2 and Section 8.4.3.4 of this book). UoI developed Magma, a C++ header library, that makes extensive use of C++ templates to offer compile-time polymorphism for increased usability at the expense of a small compile-time overhead (Section 6.4), and CERN made use of the oneAPI framework as a portability platform (Section 7.4).

### *8.8.3 Different code versions for different purposes*

During the project we noticed that for some of the applications it makes sense to have different code versions for different purposes:

- If we take a look at GROMACS from NCSA we see that the non-offload version is very efficient for small and medium scale problems, whereas the offload version is very efficient for large scale problems, and the version using FFM is more efficient for extremely big problems than the PME version.

- For NEST from NMBU different optimisations are needed to achieve a good performance on runs on a small number of nodes, than when targeting a large amount of nodes.

### *8.8.4 FPGA challenges*

It turned out that programming the FPGAs was more complicated than expected. In ASTRON's case, a complete code restructuring was needed to port the imager from one generation of FPGAs (Arria10) to the new one (Stratix10). Compiling FPGA code takes a very long time (in ASTRONs case sometimes up to 24 hours) which makes it a really time consuming work. A detailed explanation on the experience with the FPGA programming is given in Section 4.4.6. Nevertheless the experiences gained are very helpful for other applications where FPGAs are indispensable, such as in the upgrade of the LOFAR stations

### *8.8.5 Conclusion*

This report on applications experience clearly shows that the DEEP-EST system is flexible enough to accommodate the requirements coming from different problem domains. Each co-design application has benefitted from the experience made by other applications, as well as from the support from the technical consortium members who developed the hardware and software in the project. DEEP-EST has also shown that an important investment in effort and time is required to enable highly complex HPC applications to run efficiently on the next generation supercomputers, but that these efforts definitely do pay off. After over three years of joint work, the DEEP-EST applications are better prepared to exploit heterogeneous supercomputers as those expected in the Exascale era.

# 9 Critical Analysis of the Modular Supercomputing Architecture

Estela Suarez(1), Norbert Eicker(1,2), Thomas Moschny(3), Thomas Lippert(1,4)

(1) Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH, Leo Brandt Straße, 52428 Jülich, Germany

(2) Fakultät für Mathematik und Naturwissenschaften, Bergische Universität Wuppertal, Gaußstraße 20, 42119 Wuppertal, Germany

(3) ParTec AG, Possartstraße 20, 81679 Munich, Germany

(4) Goethe-Universität Frankfurt, Frankfurt Institute for Advanced Studies (FIAS). Ruth-Moufang-Straße 1, 60438 Frankfurt am Main, Germany

e.suarez@fz-juelich.de

## 9.1 Introduction

The modular supercomputing architecture (MSA) is a novel approach to implement heterogeneous supercomputing[125]. MSA's major differentiation to other types of approaches is that it defines a new intermediate level in the computer architecture hierarchy, which is located between the node- and the system levels. In MSA, subsets of nodes are grouped into special "computational modules" according to their common characteristics and algorithmic features of the corresponding subtasks.

For example, CPU-only nodes are put together into a cluster module, GPU accelerated nodes into a booster module, or quantum devices constitute a quantum module. A Modular Supercomputer is born when these modules, each of which is a high performance computer in its own right, are interconnected via a high-speed network, and are integrated by a common software stack that allows the dynamical allocation of resources from and between all modules.

This meta-architecture allows to dynamically reserve and allocate hardware resources and enables users to select the most suitable mix of resources at each time, respecting the characteristics and requirements of their code portions.

In this chapter, the MSA concept is explained in more detail to dispel some frequent misconceptions. For better understanding, MSA is contrasted to the conventional,

[125] E. Suarez, N. Eicker, Th. Lippert, "Modular Supercomputing Architecture: from idea to production", Chapter 9 in Contemporary High Performance Computing: from Petascale toward Exascale, Volume 3, pp 223-251, Ed. Jeffrey S. Vetter, CRC Press. (2019) [ISBN 9781138487079]

approach of tightly integrating all possible kinds of compute and memory elements within each node, and then replicating this entity several thousand times to build up a "monolithic" HPC system. We argue that the two architectural lines are not mutually exclusive, but that their combination by "integrating" a tightly integrated module into MSA can be beneficial to end users and operators.

## 9.2 Partitions vs. modules

Very diverse application profiles of HPC users, various kinds of processor types, and pressure on budgets for both procurement and operational costs have made heterogeneity of computers the rule rather than an exception (e.g. [126,127,128]). HPC providers deploy systems that combine different kinds of CPUs and accelerators (in general GPUs), organized in various node configurations. Frequently, supercomputers have multiple compute partitions, with different amounts of memory per node, with or without accelerators, even with different numbers or generations of GPUs.

Often the two fundamental questions are raised: when is a heterogeneous computer considered to be an MSA system? What is the difference between heterogeneous computing and modular supercomputing? The answer to these questions lies more in the manner the system can be operated rather than on its specific hardware configuration. It is the software stack that "modularizes" a heterogeneous supercomputer.

As a principle, MSA strives for a homogeneous internal configuration within each hardware module and achieves global heterogeneity by interconnecting the different modules enabling dynamical allocation of compute resources from several modules from a given program or workflow. One reason for this approach is that combining too many different computing resources within a single node makes it very difficult to share them efficiently between users with different requirements for those resources. In addition, many programs use only one variant of processors on such a "fat node" in a given part of code. All of this results in many elements in the supercomputer being idle and potentially continuing to consume power. Such underutilization can be avoided by MSA.

The first MSA system deployed in the DEEP project was a cluster-booster platform where the cluster was composed of general-purpose (Intel Xeon) CPUs on an InfiniBand network, and the booster consisted of many-core accelerators (host-less

[126] http://www-hpc.cea.fr/en/complexe/tgcc-JoliotCurie.htm

[127] https://docs.nersc.gov/systems/cori/

[128] https://www.bsc.es/marenostrum/marenostrum/mn3

Intel Xeon Phi) on an Extoll network[129]. However, this maximal separation (disaggregation) of CPUs and accelerators is one of many potential hardware realizations but it is not the defining criterion of the MSA. As a matter of fact, in most recent modular supercomputers (e.g. JUWELS[130] and MELUXINA[131]) the booster is a GPU-accelerated platform where management-CPUs are used to orchestrate the GPUs. Here, the booster node itself obviously is a heterogeneous system, but the computational power, to the largest extent, is delivered by the GPUs, while the host-CPUs clearly play a secondary role in so far as they mainly support the GPUs to fulfil their task.

It is indeed possible to choose a different interconnect technology for each module, as was the case in the first DEEP prototype, but this is not a criterion for defining modularity. Avoiding gateways and network bridges between modules, as of course expected and experienced on physical systems, leads to better performances. For this reason, the latest MSA systems use a homogenous interconnect and integrate modules in a common fabric.

Therefore, from the hardware point of view, a supercomputer with two or more distinctive partitions can be considered as a modular supercomputer. The decisive criterion for modularity is whether users can, at the same time, reserve resources on multiple modules and can run their applications across them in a distributed manner, performing communication and data transfers between these modules at runtime. What is more, modularity allows dynamically changing the size of the partitions on the modules according to the needs of the codes at runtime.

Modularity as operational and usage mode requires a software stack and programming environment that supports its requirements. The scheduler and resource manager must be aware of the hardware partitions and their features, and provide an interface enabling users to define the mix of resources to be employed in each partition. In the ideal case, dynamic allocation of the diverse resources is supported, so that each compute element is assigned to the job, when the execution of the application phase that needs it, starts, and only then. Outside these phases, these computing resources are available for other jobs. For example, for applications organized as job chains, different time windows can be set up for reserving the individual partitions. These features as well as multi-tenant use of partitions are important topics of research for the effective realization of modularity.

---

[129] N. Eicker, Th. Lippert, Th. Moschny, and E. Suarez, "T*he DEEP Project - An alternative approach to heterogeneous cluster-computing in the many-core era*, Concurrency and computation: Practice and Experience, Vol. 28, p. 2394–-2411 (2016), doi = 10.1002/cpe.3562.

[130] https://apps.fz-juelich.de/jsc/hps/juwels/configuration.html

[131] https://luxprovide.lu/technical-structure/

Modularity must also be enabled in the programming environment and the runtime system. Sections of the application's code have to be programmed and compiled to run on the hardware of the modules where they shall be executed. The various executables must be enabled to communicate with each other (e.g., via MPI or some other communication interface). This requires changes at the lower layers of the programming models that interface to the different kinds of compute (and possibly interconnecting) hardware. All these features were developed and implemented in the ParaStation Modulo[132,133] software suite in the course of the European DEEP projects. Furthermore, profiling and performance analysis tools running on MSA systems must be capable of collecting hardware counters across partitions and understand the correlation between them for modularity-enabled applications.

All these software components together have a common goal: enable each part of an application to utilize the best suitable selection of resources. This goal, aiming at globally maximizing the use of a heterogeneous set of closely interconnected supercomputers, is what characterizes a Modular Supercomputer.

## 9.3 Data movement

Dividing computing resources into different modules as strived for in MSA could raise concerns about performance degradation in communication and data transfers between computing elements that are separated from each other. We have already argued in Section 9.2 that such segregation is not necessary in a strict sense when one computes in a "modular" manner. Nevertheless, we would like to adduce some arguments addressing concerns about data-movement. Such concerns are often brought forward to favour monolithic supercomputers that integrate many different kinds of compute resources within each node, colloquially called "fat" node.

Let us first state that in most situations of parallel data processing data movements between nodes cannot be avoided. Only so-called embarrassingly parallel problems can work entirely without significant inter-node data movement. For the rest, simple to sophisticated strategies are used to minimize the surface-to-volume ratio, particularly for regular problems. There are data-centric concepts as well to avoid data movement – at the expense of more computational operations or increased memory consumption. All these strategies must be and indeed are applied within system modules in the MSA. Therefore, in the following, we focus on the particular case of inter-module communication only.

[132] ParaStation Modulo. https://par-tec.com/software/

[133] S. Pickartz, Virtualization as an enabler for dynamic resource allocation in HPC, Dissertation, RWTH AachenUniversity, Aachen, 2019. https://doi.org/10.18154/RWTH-2019-02208 .

When switching between different accelerator types, the impact of data movement on performance depends on the volume and frequency of data exchange. For a given application, these factors are correlated with the computational size of the code sections involved in the communication:

i. **Small kernels**: a typical example is often given by the innermost loop of an application, where a small but computationally intensive calculation is repeated at high frequency for a given number of iterations. This kind of computation requires very small latencies and directly profits from intra-node acceleration. Such type of computations are in fact the traditional target of CPU-GPU systems, where the main program is executed on the host CPU and the small – in the sense that they fit on the GPU memory – but computationally intensive kernels are offloaded to the GPU.

ii. **Large code parts**: in complex applications, and especially those that simulate multi-scale or multi-physics phenomena, code partitioning is done at a much coarser level. Different larger portions of the code are responsible for computing specific parts of the overall problem. They most of the time communicate internally within the respective code part, exchanging information with the rest of the code parts relatively infrequently and mainly to share intermediate results and to update parameters. As the different code parts might also have very different structures and requirements, they might profit from different types of hardware. This is where inter-module communication in MSA is required, which happens between larger code elements such as (library) functions. Between such a coarse-grained code partitions, data movement (off module) involves a rather small amount of data to be exchanged compared to module-resident (on-module) data movement.

Therefore, intra-node heterogeneity applies to on-node and on-module computation of smaller code elements (case (i)), while MSA operates off-module on bigger code-structures of an application or workflow, i.e., large code elements (case (ii)).

Increasing the compute-power of a single node by including multiple (heterogeneous) accelerators can be very helpful to speed-up the execution of small code kernels. However, this makes the supercomputer more imbalanced, and therefore less efficient as to running system-wide problems scalably. A very strong pressure is set on the system network, which cannot increase the bandwidth between nodes at the same rate as the increasing computational power inside the nodes does. In consequence, data movement off the node must be avoided, or the advantage gained by the kernel speed-up may be nullified.

Furthermore, data movements between different accelerators inside a highly heterogeneous node are not necessarily cheap either. They would be if all accelerators

could access the same (high-bandwidth) main memory in the node. However, if the main memory is standard DDR-RAM it will always be faster to stay within one single accelerator's (HBM) memory. The situation is even worse when PCIe is involved in linking the memories of the various accelerators, as is the case today. The communication performance between accelerators is then only marginally different from off-node communication. All current monolithic heterogeneous HPC systems connect their computing elements via PCIe, which requires expensive cross-element transfers and leads to a similar impact on data movement as the inter-module communication in MSA does.

The strongest caveat one often hears as to separation of resources in MSA is the occurence of increased latency for inter-module communication. This effect certainly is most acute when the data have to pass network gateways, i.e., when the modules utilize different interconnect technologies and are connected via a network bridge. However, in case the same or a fully compatible network technology is used across the entire MSA and gateways do not need to be involved, the inter-module communication capabilities are indeed comparable in capability and latency to the inter-node communication as given within an HPC module.

But even on a homogeneous network it is obvious that the latency between a CPU on the cluster and a GPU on the booster, is slightly higher than if they were located inside the same node, where they save the hop over the interconnect. It is for this reason that it is not advisable to offload small kernels between modules in MSA. Therefore, as already stated, in contrast to offloading small kernels as done on node (see in case (i)), in MSA code-partitioning is carried out at a much coarser granularity (see case (ii)). Moreover, on these coarse structures, one can benefit from algorithmic methods in order to reduce data movement between the MSA modules. For example, when running larger code parts on the different modules in parallel, communication between the modules is required much less frequently than within the module, dramatically reducing the impact of the inter-module latency. Finally, to accelerate small compute kernels, MSA can resort to exactly the same strategy as one does on the monolithic fat node system (case (i)). MSA can thus take full advantage of the standard strategy for accelerating small computational cores, while providing a massive improvement in speed when accelerating large compute kernels.

In conclusion, the communication and data movement strategy of MSA relies on executing fine-grained communication within the modules, while only coarse-grained state-exchange information is transferred between modules. This allows both the individual application kernels within a module to be accelerated on the nodes, and the entire application workflow between modules to be boosted via a matching set of resources for each large section of code. In contrast, a monolithic system composed

of identical nodes each containing a diversity of compute and acceleration resources has no means to efficiently accommodate the coarse-grained granularity of case (ii), which leads to resource under-utilization.

## 9.4 Energy efficiency

Many strategies are applied today in HPC centres to optimize energy efficiency. They comprise the use of low-power computing elements and/or accelerators, shutting-down unused resources, holistic system monitoring, optimizing the site-infrastructure and system cooling (e.g., through direct liquid cooling), actively re-using waste heat, etc. All these approaches can profit from MSA in the same manner as known from any other heterogeneous architecture. What is more, MSA operates at a coarse-grained scale that naturally matches the sub-second timescales handled by monitoring and cooling systems. Heterogeneous System-on-Chip (SoC) approaches – which represent the smallest form of intra-node heterogeneity – are governed by much smaller spatial-scales and shorter time-scales (micro- to nanoseconds). Holistic monitoring starting out from this level would require a vertical integration of monitoring capabilities from very deep (SoC-level) up to very high (infrastructure-level). This ambition constitutes a complicated technological challenge and may not be feasible due to timescales involved differing by orders of magnitude[134].

On top of the general methodology to save energy as mentioned above, MSA can increase energy efficiency by applying three additional strategies:

1.) **Targeted hardware scale-out**: the dimensions of the individual MSA-modules are determined by the requirements of the user-portfolio running on the MSA system, as well as by the energy efficiency of its components. For instance, a cluster module, where applications in need for high single-thread performance run, is composed of relatively power-hungry general purpose CPUs and is therefore kept rather small. The booster, on the other hand, which runs highly-scalable applications (or parts thereof) achieves a very high compute performance using more energy-efficient accelerators. In MSA, only this part of the system is scaled-out to thousands or tens-of-thousands of nodes, if needed, in contrast to fat node systems where complex and expensive fat nodes need to be scaled out.

2.) **Tailor system to application needs**: by running each part of the user code on the kind of node that allows best performance, improved application efficiency

[134] An additional aspect is the fact that, due to the electrical capacities in the hardware, neither accurate power measurement nor adequate power and cooling management seem realistic on a time scale of less than a millisecond.

and performance is achieved. The speed-up gained by the individual applications translates into a shorter execution time, which typically leads to lower overall energy consumption.

3.) **Maximize use of resources**: MSA enables dynamic scheduling, reservation and allocation of resources and makes them available for the job only for the relevant time window, while the rest of the time they are free to be used by others. This enables more efficient resource sharing, and therefore achieves a higher utilization of the individual components, reducing idle time and unnecessary energy waste. In contrast, on a monolithic supercomputer with fat nodes, all resources of all utilized fat nodes are blocked during a job's runtime. Sharing of nodes is expected to be inefficient due to the impact of jitter effects induced by co-utilization[135] on such fat nodes.

As far as system scaling is concerned, one might argue against point (1) that in a booster built as a GPU-accelerated system, the necessary amount of (power hungry) host-CPUs also grows with system size. This issue is, however, readily avoided by choosing a suitable, low-power CPU for the booster, as the CPU only needs to manage the GPUs and not to perform relevant application computation. It is expected that the market will offer GPU designs with integrated orchestrator CPU cores in the near future. This would make GPUs much more independent and allow building a true GPU-only booster.

Building "lean" booster nodes employing low-power management-CPUs (or host-less GPUs) also addresses point (3), as it minimizes (eventually even down to zero) the energy consumption of host CPUs, which are among the very few resources in an MSA system that are prone to be idle, since they are less intensively used for application computation.

Here it is worth mentioning that maximum resource utilization (3) is an important advantage of MSA compared to monolithic systems based on highly-heterogeneous (fat) nodes. An increased intra-node heterogeneity leads to underutilization of resources, since for a given job either CPUs or GPUs, but very rarely different accelerators, are simultaneously in use. The unused node-elements run idle and continue to consume power. Given a broad portfolio of applications, this problem cannot simply be overcome by choosing the best-suited accelerator mix for the heterogeneous node, as this will always introduce a fixed ratio between CPUs and accelerators. This ratio will support only a few applications optimally while others have their sweet-spot at higher or lower ratio. MSA, on the other hand, is fully flexible and

[135] F. Petrini, D. J. Kerbyson, and S. Pakin, "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8192 Processors of ASCI Q," in ACM Supercomputing, 2003.

dynamic in the assignment of resources even during program execution, which is its most characteristic new feature of MSA.

In order to compensate these limitations of fat nodes, some chip-designers propose the idea of so-called “dark silicon”. It leverages the concept of integrating an amount of computational resources that deliberately would exceed the chip’s actual power envelope, while selectively switching some resources on and off when possible. In principle, this strategy can be equally applied to heterogeneous chip designs by powering off unused accelerators units. However, it is questionable if steering the power is possible at such extremely small time-scales (see case (i) in Section 9.3) required by the tight integration of accelerators within a chip. More importantly, even if the power for the processing elements is switched on only during operation, the investment made for the switched-off elements is lost for this idle time. Taking into account that during the lifetime of an HPC system, the hardware investment is about two thirds of the total cost of ownership, the energy adjustment as just described can only partially compensate for the underutilization. We argue that maximizing resource utilization by MSA is a fundamentally better approach to increase energy efficiency and reduce cost, and increases the total scientific throughput of HPC systems.

Beyond that, the central assumption behind the dark-silicon strategy is that the cost of transistors’ silicon is negligible when compared to the power-consumption of running them. Reaching the end of Moore’s law by now and observing the worrying situation of the silicon industry since 2020 lets us have serious doubts on this underlying assumption of the dark-silicon strategy.

The challenge of connecting the additional transistors should not be neglected either.

Highly integrated systems are widely used in the mobile and embedded markets, where space and power constraints play a crucial role. Need for high energy efficiency together with moderate prices of mass-market components have been arguments for applying similar strategies in HPC. However, mobile and embedded markets are completely different from the HPC market. In mobile devices, a small number of heterogeneous elements (thin cores, fat cores, GPU, memory, flash, modem, AI,…) are interconnected via standardized interfaces and integrated on an SoC. Until now, HPC has not yet settled on a standard interface for the hardware elements, which limits the possible combinations of elements, and the bandwidth demand in between the elements is significantly higher than on the mobile devices. The main motivation for a SoC in mobile devices is the level of integration and low production costs, rather than bandwidth and latency as in HPC. In HPC, high bandwidth and latency requirements lead to the use of highly sophisticated interposers. Considering the technological challenges and the economic constraints, which these intermediate layers are subject to, their feasibility has not really been proven to date. Therefore, the amount of dark

silicon elements is limited by both the technology of the interposer and the cost of the silicon.

## 9.5 System integration

For more than a decade, standard accelerators have been integrated within fat nodes to achieve very high peak performance. The main disadvantages of this approach, i.e., underutilization of resources and shared network interfaces, have been discussed extensively above. Today, its strongest advantage as to closer integration with the CPU resources is still diminished by the lack of a technology, where CPU and accelerators have access to shared high-bandwidth memory. Heterogeneous chips (e.g., GPUs with integrated CPU cores and dynamical mutual assignment), which are under development, promise access to shared high-bandwidth memories. If such chips reach the market, they will benefit both monolithic and modular architectures that, for example, could build a cheaper and more energy-efficient booster by getting rid of management CPUs.

Interestingly, the MSA technology also enables the coupling of modules that are operated by GPUs from different manufacturers, for example. In this case, it is not so much about accelerating computations in cluster-booster mode, but rather about equipping the overall system with various accelerator technologies. This strategy makes it possible, on the one hand, to make the most suitable technology available to the user in the workflow and, on the other hand, to still make the entire system accessible to applications that have very large memory and computing requirements. Such type of HW requirements can currently only be delivered by MSA.

From a physical system integration perspective, building, maintaining, and operating MSA platforms are just as complex as monolithic systems: the single modules itself are similar to monolithic systems, they just use slim nodes in contrast to fat complex nodes. Interconnecting them is a problem that is solved by using the right system software, as proven by JUWELS. MSA-modules can also be adapted over time to meet new user requirements by substituting modules or adding new ones when enhanced technology emerges. In fact, MSA also opens up opportunities to integrate presumably disruptive technologies into HPC systems, such as neuromorphic devices or quantum computers. They are still in very early development stages, but have already demonstrated impressive performances for some specific applications.

The inclusion of neuromorphic or quantum modules in the MSA might facilitate their adoption by the wider user communities. For example, it has been demonstrated that quantum computers are extremely efficient to solve specific kinds of problems such as high-dimensional optimization scenarios. While it is very unlikely to see a large-scale HPC application executed fully on a quantum computer anytime soon, it seems

worthwhile to explore an application running e.g., on the cluster module of an MSA, which offloads an optimization problem as part of its code to be solved by a quantum module. These types of embedded optimization problems are ideal for MSA, as they consist of a well-isolated and large part of the code, with only small amounts of data being exchanged between the cluster and the quantum module – which is again ideal for a quantum computing system allowing for small data rates only. This coarse-grained quantum-hybrid strategy allows for the exploration of quantum computing especially for applied problems from science and industry already today, in particular when a quantum annealer like a D-Wave system is exploited.

## 9.6 Application scalability

Another frequently expressed misconception about MSA is the fear of hindering application-scalability by the need to spread the code components across vastly different module architectures until all available compute resources are occupied. However, for the analysis of the scalability of codes on the MSA, only the booster module should be considered. As with Amdahl's law, the maximum problem size and maximum scalability is always given by the highly scalable part of the code that, in MSA, runs on the scalable booster. In addition to that, decoupling the less-scalable code parts from the high-scaling ones and running them on the cluster improves the overall application scalability: the high-scaling part can scale unhindered on the booster, while the low-scaling part is speed-up through the high single-thread performance of the cluster module.

On the other hand, a justified criticism of MSA – or rather of the current software environment – is that it imposes a relatively high burden on application developers to prepare their codes for execution in a multi module mode. First of all, it is emphasized that such code-distribution is an opportunity in MSA not a general obligation. To give an example, highly scalable applications with an intrinsic monolithic structure (e.g., tightly coupled differential operations on regular lattice systems) should never be spanned across modules, but rather run entirely within the booster.

Candidate MSA codes from multiphysics and multiscale applications to be coarse-grained assigned to modules must undergo a series of analyses and transformations: any such application has to be analyzed as to its internal structure and potential performance bottlenecks, code parts need to be identified and ported to the given module architectures using a suitable programming model (e.g., CUDA or OpenACC for GPUs), and scaling studies need to be performed with relevant and suitable use-cases to find their best modus operandi and the appropriate number of nodes on each module. All these steps are summarized in the best practices guide provided as Chapter 8 of this book. Many of the adaptations to optimize application performance

on specific modules (e.g., increase vectorisation, keeping data locality, proper organisation of data structures, etc.) are necessary on any modern heterogeneous compute platform, not only on MSA. The additional MSA-specific considerations are those related to the implementation of a coarse-grained code partition.

The additional effort of porting codes to MSA might scare application developers. While so far, only a few applications are enabled to run in multi-module mode, from a user and computing centre perspective MSA is even beneficial for single-module operations, as the different modules provide a variety of computing resources for a diverse application portfolio of an HPC centre, even if each code runs on only one type of node. Still, in order to improve user experience and to promote the modularization of HPC applications, the MSA software stack is in continuous development in order to make the MSA-specific and the more general code porting actions as comfortably as possible: this is the goal of the EU-funded DEEP-SEA project, which started in April 2021 and will run for three years[136]. It will continue the software development efforts of the DEEP project series, which already delivered an MSA-enabled runtime system (ParaStation Modulo), as well as a scheduler and a resource manager targeting heterogeneity at system level. Advanced features for a better support of compute and memory heterogeneity, enhanced malleability and interoperability features, co-scheduling aspects, and performance portability will be developed in DEEP-SEA.

## 9.7 Conclusion

The goals of MSA are to offer the best system configuration to a portfolio of applications with very different profiles and requirements, to assign the best suited hardware resources to each of them (and each of their code-parts), and to maximize system usage and energy efficiency by enabling an efficient sharing of compute resources overall. Most of the reservations for which MSA is often criticized and contrasted with other alternative heterogeneous computing approaches have their roots in simple misunderstandings about basic MSA principles.

The MSA is fundamentally different from other heterogeneous computing approaches, and in particular from highly integrated monolithic systems, in that system-level heterogeneity is achieved by combining a set of (rather) homogeneous computational modules, which allows coarse-grained partitioning of application code among these modules. Multi-module execution is foreseen mainly for applications with an intrinsic multi-physics or multi-scale nature. The associated large code parts run within the modules exchanging a limited amount of data between each other at relatively low frequency. Performance is therefore not impacted by the slightly increased inter-

[136] www.deep-projects.eu

module latency. Intra-node heterogeneity, on the other hand, is suitable for low-granularity operations, such as the execution of small but computationally intensive kernels. Here data is exchanged at a much higher rate and low latency is very crucial to achieve performance.

Because the operational levels of both approaches to heterogeneous computing (MSA and highly-integrated node designs) are so different, it suggests itself to combine them. Therefore, the MSA welcomes the inclusion of heterogeneous modules, and, in fact, current MSA systems do contain them. The combination of different modules with diverse node configurations, some homogenous, some heterogeneous, makes MSA extremely flexible and adaptable to any application portfolio. Further benefits include the possibility to scale out only the most energy-efficient modules of the system, keeping the power-hungry modules at a relatively low node count but still available for the user codes that require them, and the ability to include modules based on disruptive computing technologies such as quantum technologies.

## 9.8 Acknowledgements

The authors thank all the institutions and individuals involved in the DEEP series of projects and, in particular, all the members of the DEEP-EST team who have contributed to the development of the MSA architecture, its prototype hardware implementation, and its software environment.

This work has been partially funded by the European Union’s Seventh Framework (FP7/20017-2013) and Horizon 2020 Framework Programmes for Research and Innovation under grant agreements 287530 (DEEP), 610476 (DEEP-ER), 754304 (DEEP-EST), and 955606 (DEEP-SEA). The present publication reflects only the authors' views. The European Commission is not liable for any use that might be made of the information contained therein.

## List of Acronyms and Abbreviations

# A

**AARTFAAC:** The Amsterdam-ASTRON Radio Transients Facility And Analysis Center; a LOFAR-based, all-sky radio telescope

**API:** Application Programming Interface

**ASIC:** Application Specific Integrated Circuit, Integrated circuit customised for a particular use

**ASTRON:** Netherlands Institute for Radio Astronomy, Netherlands

**AVX:** Advanced Vector Extensions

**AVX-512:** Intel 512-bit SIMD instructions

# B

**BeeGFS:** The Fraunhofer Parallel Cluster File System (previously acronym FhGFS). A high-performance parallel file system.

**BeeOND:** BeeGFS-on-demand, parallel storage based on BeeGFS

**BoP:** Board of Partners for the DEEP-EST project

**BSC:** Barcelona Supercomputing Centre, Spain

**BW:** Bandwidth

# C

**CERN:** European Organisation for Nuclear Research / Organisation Européenne pour la Recherche Nucléaire, International organisation

**CM:** Cluster Module: with its Cluster Nodes (CN) containing high-end general-purpose processors and a relatively large amount of memory per core

**CMS:** Compact Muon Solenoid experiment at CERN's LHC

**CMSSW:** Physical toolset software for the CMS experiment at CERN

**CNN:** Convolutional Neural Networks

**CPU:** Central Processing Unit

# D

**D:** Deliverable, followed by a number, term to designate a deliverable (document) in the DEEP-EST project

**DAM:** Data Analytics Module: with nodes (DN) based on general-purpose processors, a huge amount of (non-volatile) memory per core, and support for the specific requirements of data-intensive applications

**DBSCAN:** Density-based Spatial Clustering for Applications with Noise

**DCDB:** Data Centre Data Base (a tool developed in DEEP)

**DCPMM:** Intel Optane DC Persistent Memory Module, a non-volatile/persistent memory in DDR4 DIMM form factor

**DDG:** Design and Developer Group of the DEEP-EST project

**DDR:** Double Data Rate

**DDR4:** Double Data Rate fourth-generation

**DEEP:** Dynamical Exascale Entry Platform (project FP7-ICT-287530)

**DEEP-ER:** DEEP - Extended Reach (project FP7-ICT-610476)

**DEEP-EST:** DEEP - Extreme Scale Technologies

**DIMM:** Dual In-line Memory Module

**DL:** Deep Learning

**DLMOS:** A Deep Learning Model of the Solar Wind to forecast the plasma conditions at the orbit of the Earth from images of the Sun developed by KU Leuven

**DNN:** Deep neural network

**DRAM:** Dynamic Random Access Memory. Typically describes any form of high capacity volatile memory attached to a CPU

# E

**EC:** European Commission

**EEP:** European Exascale Projects

**ESB:** Extreme Scale Booster: with highly energy-efficient many-core processors as Booster Nodes (BN), but a reduced amount of memory per core at high bandwidth

**EU:** European Union

**Exascale:** Computer systems or Applications, which are able to run with a performance above $10^{18}$ Floating point operations per second

**EXTOLL:** High speed interconnect technology for HPC developed by UHEI

**Extrae:** Performance analysis tool developed by BSC

## F

**Fabri$^3$:** Interconnect technology based on EXTOLL (pron. “Fabri-Cube”)

**FFT:** Fast Fourier Transform

**Flop/s:** Floating point operation per second

**FP7:** European Commission 7th Framework Programme

**FPGA:** Field-Programmable Gate Array, Integrated circuit to be configured by the customer or designer after manufacturing

**FTI:** Fault Tolerant Interface, a checkpoint/restart library

## G

**GB/s:** Gigabyte per second, $10^9$ Byte transfer rate

**GbE:** Gigabit Ethernet**,** $10^9$ Bit transfer rate

**GFlop/s:** Gigaflop, $10^9$ Floating point operations per second

**GFlop/w:** Giga ($10^9$) Floating point operations per second per Watt, or alternatively: Giga Floating point operations per Joule

**GPFS:** IBM General Parallel File System

**GPU:** Graphics Processing Unit

**GROMACS:** A toolbox for molecular dynamics calculations providing a rich set of calculation types, preparation and analysis tools

# H

**H2020:** Horizon 2020

**HPC:** High Performance Computing

**HPDA:** High Performance Data Analytics

**HPDBSCAN:** A clustering code formerly used by UoI in the field of Earth Science

# I

**IB:** see InfiniBand

**InfiniBand:** A networking communication standard for HPC clusters

**Intel:** Intel Germany GmbH, Feldkirchen, Germany

**I/O:** Input/Output. May describe the respective logical function of a computer system or a certain physical instantiation

**IP:** Intellectual Property

# J

**JUELICH:** Forschungszentrum Jülich GmbH, Jülich, Germany

# K

**KNL:** Knights Landing, second generation of Intel® Xeon Phi™

**KU Leuven:** Katholieke Universiteit Leuven, Belgium

# L

**LHC:** Large Hadron Collider (LHC), the world's most powerful accelerator providing research facilities for High Energy Physics researchers across the globe

**LOFAR:** Low-Frequency Array, an instrument for performing radio astronomy built by ASTRON

# M

**MB:** Mega Bytes

**Megware:** Megware Computer Vertrieb und Service GmbH, Chemnitz, Germany

**ML:** Machine Learning

**MPI:** Message Passing Interface, API specification typically used in parallel programs that allows processes to communicate with one another by sending and receiving messages

**MPICH:** MPI implementation maintained by Argonne National Laboratory

**MSA:** Modular Supercomputer Architecture

**MT:** Multi-Threading

**MUSIC:** Multisimulation Coordinator (MPI-based library for coupled codes)

# N

**NCSA:** National Centre for Supercomputing Applications, Bulgaria

**NEST:** Widely-used, publically available simulation software for spiking neural network models developed by NMBU.

**NextDBSCAN:** A next generation, accelerator enabled parallel clustering code developed by UoI with applications in the field of Earth Science

**NextSVM:** A next generation, accelerator enabled parallel classification algorithm by UoI with applications in the field of Earth Science

**NMBU:** Norwegian University of Life Sciences, Norway

**NN:** Neural Network

**NVM:** Non-Volatile Memory. Used to describe a physical technology or the use of such technology in a non-block-oriented way in a computer system

**NVMe:** Non-Volatile Memory Express interface and protocol

**NVRAM:** Non-Volatile Random-Access Memory

# O

**OmpSs:** BSC's Superscalar (Ss) for OpenMP

**OpenCL:** Open Computing Language, framework for writing programs that execute across heterogeneous platforms

**OpenMP:** Open Multi-Processing, Application programming interface that support multiplatform shared memory multiprocessing

**Open MPI:** MPI implementation maintained by the Open MPI Project

**OS:** Operating System

# P

**ParaStation Modulo:** Software for cluster management and control developed by JUELICH and its linked third party ParTec

**Paraver:** Performance analysis tool developed by BSC

**ParTec:** ParTec Cluster Competence Center GmbH, Munich, Germany. Linked third Party of JUELICH in DEEP-EST

**PCIe:** Peripheral Component Interconnect Express; a bus that is often used to connect CPUs to GPUs, network devices, etc.

**PCIe3:** Peripheral Component Interconnect Express third-generation

**PDU:** Power Distribution Unit

**PFlop/s:** Petaflop, $10^{15}$ Floating point operations per second

**Phi:** see Xeon Phi

**piSVM:** Parallel classification algorithm formerly used by UoI

**PME:** Particle mesh Ewald

**PMT:** Project Management Team of the DEEP-EST project

**POSIX:** Portable Operating System Interface

**PRACE:** Partnership for Advanced Computing in Europe (EU project, European HPC infrastructure)

**PSU:** Power Supply Unit

# Q

## R

**RAID:** Redundant Array of Inexpensive Disks

**RAM:** Random-Access Memory

**RDMA:** Remote Direct Memory Access

**RM:** Resource Manager

## S

**SIMD:** Single Instruction Multiple Data

**SIONlib:** Parallel I/O library developed by Forschungszentrum Jülich

**SKA:** Square Kilometer Array

**Slurm:** Job scheduler that will be used and extended in the DEEP-EST prototype

**SMP:** Symmetric Multi-Processing

**SSD:** Solid-State Drives

**SSSM:** Scalable Storage Service Module

**SVM:** Support Vector Machine

## T

**TCP:** Transmission Control Protocol; a reliable, stream-based network protocol

**TensorFlow:** Open-source software library for dataflow programming

**TFlops:** Teraflop, $10^{12}$ Floating point operations per second

**Tk:** Task, Followed by a number, term to designate a Task inside a Work Package of the DEEP-EST project

## U

**UDP:** User Datagram Protocol; an unreliable, packet-based network protocol

**UEDIN:** University of Edinburgh, UK

**UHEI:** Ruprecht-Karls-Universitaet Heidelberg, Germany

**UoI:** Háskóli Íslands – University of Iceland, Iceland

## V

## W

**WP:** Work package

## X

**x86:** Family of instruction set architectures based on the Intel 8086 CPU

**Xeon:** Non-consumer brand of the Intel®x86 microprocessors (TM)

**Xeon Phi:** Brand name of the Intel®x86 manycore processors (TM)

**xPic:** Programming code developed by partner KU Leuven to simulate space weather

Band / Volume 35
**Understanding the formation of wait states in one-sided communication**
M.-A. Hermanns (2018), xiv, 144 pp
ISBN: 978-3-95806-297-9
URN: urn:nbn:de:0001-2018012504

Band / Volume 36
**A multigrid perspective on the parallel full approximation scheme in space and time**
D. Moser (2018), vi, 131 pp
ISBN: 978-3-95806-315-0
URN: urn:nbn:de:0001-2018031401

Band / Volume 37
**Analysis of I/O Requirements of Scientific Applications**
S. El Sayed Mohamed (2018), XV, 199 pp
ISBN: 978-3-95806-344-0
URN: urn:nbn:de:0001-2018071801

Band / Volume 38
**Wayfinding and Perception Abilities for Pedestrian Simulations**
E. Andresen (2018), 4, x, 162 pp
ISBN: 978-3-95806-375-4
URN: urn:nbn:de:0001-2018121810

Band / Volume 39
**Real-Time Simulation and Prognosis of Smoke Propagation in Compartments Using a GPU**
A. Küsters (2018), xvii, 162, LIX pp
ISBN: 978-3-95806-379-2
URN: urn:nbn:de:0001-2018121902

Band / Volume 40
**Extreme Data Workshop 2018**
Forschungszentrum Jülich, 18-19 September 2018
Proceedings
M. Schultz, D. Pleiter, P. Bauer (Eds.) (2019), 64 pp
ISBN: 978-3-95806-392-1
URN: urn:nbn:de:0001-2019032102

Band / Volume 41
**A lattice QCD study of nucleon structure with physical quark masses**
N. Hasan (2020), xiii, 157 pp
ISBN: 978-3-95806-456-0
URN: urn:nbn:de:0001-2020012307

Schriften des Forschungszentrums Jülich
IAS Series

Band / Volume 42
**Mikroskopische Fundamentaldiagramme der Fußgängerdynamik – Empirische Untersuchung von Experimenten eindimensionaler Bewegung sowie quantitative Beschreibung von Stau-Charakteristika**
V. Ziemer (2020), XVIII, 155 pp
ISBN: 978-3-95806-470-6
URN: urn:nbn:de:0001-2020051000

Band / Volume 43
**Algorithms for massively parallel generic *hp*-adaptive finite element methods**
M. Fehling (2020), vii, 78 pp
ISBN: 978-3-95806-486-7
URN: urn:nbn:de:0001-2020071402

Band / Volume 44
**The method of fundamental solutions for computing interior transmission eigenvalues**
L. Pieronek (2020), 115 pp
ISBN: 978-3-95806-504-8

Band / Volume 45
**Supercomputer simulations of transmon quantum computers**
D. Willsch (2020), IX, 237 pp
ISBN: 978-3-95806-505-5

Band / Volume 46
**The Influence of Individual Characteristics on Crowd Dynamics**
P. Geoerg (2021), xiv, 212 pp
ISBN: 978-3-95806-561-1

Band / Volume 47
**Structural plasticity as a connectivity generation and optimization algorithm in neural networks**
S. Diaz Pier (2021), 167 pp
ISBN: 978-3-95806-577-2

Band / Volume 48
**Porting applications to a Modular Supercomputer**
Experiences from the DEEP-EST project
A. Kreuzer, E. Suarez, N. Eicker, Th. Lippert (Eds.) (2021), 254 pp
ISBN: 978-3-95806-590-1

Weitere ***Schriften des Verlags im Forschungszentrum Jülich*** unter
http://wwwzb1.fz-juelich.de/verlagextern1/index.asp

IAS Series
Band / Volume 48
ISBN 978-3-95806-590-1

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
Forschungszentrum