Ref. Ares(2018)1734257 - 29/03/2018

H2020-FETHPC-01-2016

DEEP-EST

DEEP Extreme Scale Technologies

Grant Agreement Number: 754304

D1.2

Application use cases and traces

Final

Version: 1.0

Author(s): P. Martínez (BSC)

Contributor(s): H. E. Plesser (NMBU), P. Petkov (NCSA), V. Pavlov (NCSA), J. Romein (ASTRON), J. Amaya (KU Leuven), D. Gonzalez (KU Leuven), E. Erlingsson (UoI), H. Neukirchen (UoI), G. Cavallaro (UoI), S. Barakat (UoI) M. Riedel (UoI), M. Girone (CERN), V. Khristenko (CERN)

Date: 28.03.2018

Project and Deliverable Information Sheet

<table>
<tr><td rowspan="8">DEEP-EST Project</td><td colspan="2">Project Ref. №: 754304</td></tr>
<tr><td colspan="2">Project Title: DEEP Extreme Scale Technologies</td></tr>
<tr><td colspan="2">Project Web Site: http://www.deep-projects.eu</td></tr>
<tr><td colspan="2">Deliverable ID: D1.2</td></tr>
<tr><td colspan="2">Deliverable Nature: Report</td></tr>
<tr><td rowspan="2">Deliverable Level:
PU *</td><td>Contractual Date of Delivery:
31 / March / 2018</td></tr>
<tr><td>Actual Date of Delivery:
31 / March / 2018</td></tr>
<tr><td colspan="2">EC Project Officer: Juan Pelegrín</td></tr>
</table>

* - The dissemination levels are indicated as follows: PU = Public, fully open, e.g. web; CO = Confidential, restricted under conditions set out in Model Grant Agreement; CI = Classified, information as referred to in Commission Decision 2001/844/EC.

Document Control Sheet

<table>
<tr><td rowspan="6">Document</td><td colspan="2">Title: Application use cases and traces</td></tr>
<tr><td colspan="2">ID: D1.2</td></tr>
<tr><td>Version: 1.0</td><td>Status: Final</td></tr>
<tr><td colspan="2">Available at: http://www.deep-projects.eu</td></tr>
<tr><td colspan="2">Software Tool: Microsoft Word</td></tr>
<tr><td colspan="2">File(s): DEEP-EST_D1.2_Application_use_cases_and_traces_v1.0</td></tr>
<tr><td rowspan="4">Authorship</td><td>Written by:</td><td>P. Martínez (BSC)</td></tr>
<tr><td>Contributors:</td><td>H. E. Plesser (NMBU), P. Petkov (NCSA), V. Pavlov (NCSA), J. Romein (ASTRON), J. Amaya (KU Leuven), D. Gonzalez (KU Leuven), E. Erlingsson (UoI), H. Neukirchen (UoI), G. Cavallaro (UoI), S. Barakat (UoI) M. Riedel (UoI), M. Girone (CERN), V. Khristenko (CERN)</td></tr>
<tr><td>Reviewed by:</td><td>Valeria Bartsch (Fraunhofer ITWM), J. Kreutz (JUELICH), D. Alvarez (JUELICH)</td></tr>
<tr><td>Approved by:</td><td>BoP/PMT</td></tr>
</table>

Document Status Sheet

Version	Date	Status	Comments
1.0	28.03.2018	Final version	EC submission

Document Keywords

Keywords:	DEEP-EST, HPC, Exascale, Applications, Co-design

Copyright notice:

© 2017-2020 DEEP-EST Consortium Partners. All rights reserved. This document is a project document of the DEEP-EST Project. All contents are reserved by default and may not be disclosed to third parties without the written consent of the DEEP-EST partners, except as mandated by the European Commission contract 754304 for reviewing and dissemination purposes.

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

Table of Contents

Project and Deliverable Information Sheet ... 2
Document Control Sheet ... 2
Document Status Sheet ... 3
Document Keywords ... 4
Table of Contents ... 6
List of Figures ... 8
List of Tables ... 10
Executive Summary ... 12
1 Introduction ... 14
2 Task 1.2: Neuroscience (NMBU) ... 15
2.1 Benchmarking NEST ... 15
2.2 Benchmarking Arbor ... 25
2.3 Benchmarking Elephant ... 27
3 Task 1.3: Molecular dynamics (NCSA) ... 29
3.1 Use case description ... 29
3.2 Benchmarking metrics ... 30
3.3 MD benchmarking metrics ... 30
3.4 MD benchmarking results ... 31
4 Task 1.4: Radio astronomy (ASTRON) ... 34
4.1 What to measure ... 34
4.2 Measurement setup ... 34
4.3 Application parameters ... 36
5 Task 1.5: Space Weather (KU Leuven) ... 38
5.1 xPic ... 38
5.2 DLMOS ... 45
6 Task 1.6: Data analytics in Earth Science (UoI) ... 52
6.1 HPDBSCAN ... 52
6.2 piSVM ... 55
6.3 Deep learning with TensorFlow and Keras extension ... 58
7 Task 1.7: High Energy Physics (CERN) ... 62
7.1 Application description ... 62
7.2 Physics description ... 62
7.3 Application infrastructure and configuration ... 63
7.4 Benchmarking metrics ... 64
7.5 Benchmarking configuration ... 64
7.6 Benchmarking results ... 65
8 Global conclusion ... 67
8.1 Next steps ... 68
List of Acronyms and Abbreviations ... 70

List of Figures

Figure 1: Neuroscience simulation and analytics workflow for NEST with *in situ* computation of local field potentials using Arbor and HybridLFPy (left path) and *in situ* statistical analysis using Elephant (right path). 15

Figure 2: Major steps of build and simulation phase of NEST. The vertical black arrow on the left indicates single-threading and multi-threading (single and multiple lines, respectively), MPI communication (squares with bidirectional arrows), and the repetition of the main simulation cycle (dashed upward pointing arrow). Coloured and dark grey text highlighting corresponds to the data structures shown in Figures 2, 3, respectively; they indicate when the data structures are created, changed or accessed. From Kunkel and Schenck (2017) under CC BY license. 16

Figure 3: Fundamental data structures in NEST. Data structures on MPI rank 0 for (A) an example network of eight neurons with ring-like connectivity, which is simulated using two MPI processes and three threads per process. For simplicity, stimulating and recording devices are omitted. (B) Neuron infrastructure. For each local neuron (blue squares) the SparseNodeArray local_nodes (dark green) stores a struct of a pointer to the neuron and the neuron's GID. The two-dimensional vector nodes_vec (light green) stores a pointer to the local neurons sorted by thread. (C) Connection infrastructure. Each synapse is represented on the thread of its target neuron. Each thread owns a sparse table (dark orange), which stores a pointer to a Connector (orange) for every source neuron that has targets on the thread. The connectors hold the local synapses (pink) sorted by type (here only one static synapse per connector). From Kunkel and Schenck (2017) under CC BY license. 17

Figure 4: Five spike buffers for a simulation cycle with four neuronal update steps and for a network of eight neurons, which is simulated using two MPI processes and three threads per process. (A) During neuronal updates the three-dimensional vector spike_register stores the GIDs of the local neurons that spike (dark grey squares) sorted by thread and update step. (B) Before MPI communication each rank collocates its send buffer based on the entries in its spike_register. Communication markers (light gray squares) define update step and thread. Buffers may not be completely filled (white squares). (C) After MPI communication using MPI_Allgather, each rank holds the complete spike data in its receive buffer (global spike buffer), which is the concatenation of the send buffers of all ranks. Modified from Kunkel and Schenck (2017) under CC BY license. 19

Figure 5: Timeline of the imager application. PCIe transfer rates of 12.5 GB/s are not sufficient to keep the GPU busy (top figure), while NVLINK transfer rates of 68 GB/s are (bottom figure). 36

Figure 6: HPDBSCAN´s standard output, using the Bremen dataset running on the SDV partition with 128 cores. Intermediary time-measurements are provided for each relevant part of the application, as well as the whole execution. 53

Figure 7: Visualisation of the input data for the Inner city of Bremen dataset. Colours represent temperature values where blue is cold and red is warm (temperature is ignored for clustering) 54

Figure 8: A visualisation of the generated output (using a different perspective); each colour represents a single cluster. 54

Figure 9: Command line options available when running the svm-train executable. 56

Figure 10: Successful classification evaluation output obtained with pisvm-predict based on the partial example of the output produced by the training phase. 56

Figure 11: Visualisation of the Indian Pines dataset with the hyper-spectral layers on the left side, and its categorisation (labelling) on the right. 57

List of Tables

Table 1: JUBE output for NEST benchmarks. 25
Table 2: JUBE output for Arbor benchmarks. 27
Table 3: Gromacs use case description. 29
Table 4 Gromacs benchmarks. 31
Table 5: Gromacs benchmarking results. 32
Table 6: xPic benchmark 1. 38
Table 7: xPic benchmark 2. 40
Table 8: xPic benchmark 3. 42
Table 9: xPic benchmark 4. 43
Table 10: DLMOS benchmark 1. 45
Table 11: DLMOS benchmark 2. 47
Table 12: DLMOS benchmark 3. 48
Table 13: DLMOS benchmark 4. 50
Table 14: GEN-SIM results on AMD. 65
Table 15: RECO results on AMD. 66

Executive Summary

The main goal of the applications work package, namely WP1, in the DEEP Extreme Scale Technologies (DEEP-EST) project is to assess the Modular Supercomputing Architecture (MSA) developed in the project and to evaluate the DEEP-EST prototype. For this purpose, six applications from a wide range of scientific fields are chosen. These will show that the new architecture is beneficial for not only one specific kind of application, but for several ones and in different ways.

This second deliverable gathers a description of all the benchmarking necessary to track the increase in performance undergone by each application, in an effective way. Such performance gains will be a direct consequence of the applications' adaptation to the MSA. For this purpose, increasingly complex benchmarks have been carefully chosen to cover realistic test cases whilst keeping into consideration certain constraints such as runtime execution. WP2 ("Benchmarking and modelling") will carry out a continuous benchmarking as well as tracing of the applications throughout the duration of the project using the analysis tools introduced in early stages of the project.

DEEP-EST Design and Development Group (DDG) is currently evaluating potential architectures for the DEEP-EST prototype, especially for the Extreme Scale Booster (ESB) module, such as the AMD EPYC Naples 2S and the ARM Cavium Thunder X2 systems. A simpler version of the benchmarks that are described in this document (namely micro-benchmarks and not shown here for the sake of conciseness) is currently being deployed on such architectures to assess their appropriateness for the MSA. This highlights the importance of benchmarking for the achievement of the DEEP-EST co-design project.

1 Introduction

The first deliverable D1.1 reported on each application structure and their requirements[1], both in terms of hardware and software, for the co-design of the MSA to be built within the DEEP-EST project. This document aims at describing a series of benchmarks in order to track the progress, both in terms of performance and modularity, that each application will undergo throughout the lifespan of the project. This document is structured as follows: each application, which can consist of several programs, is reported in a separated section and a final section is reserved for the global conclusion followed by next steps.

Although each application case is presented in a slightly different manner due to different and particular requirements, a common structure is shared among all of them. When it is necessary, a detailed overview of the application is presented thus complementing the description given in the deliverable D1.1[1]. This overview clarifies concepts and particularities that are later applied to the benchmarking phase. After this, benchmarking metrics are defined to help identifying the final result of the benchmark. Benchmarks are then described from both a physical and a computational point of view in order to give a general idea of what they do represent. Instructions on how to access the application source code, compile it and run it via JUBE scripts are also provided herein. Additional comments on variables and parameters required by applications and benchmarks can also be found although a strong emphasis has been placed on automating all the involved procedures. Finally, results from previous benchmarking on other hardware are often provided as reference data, which might be helpful for colleagues of WP2 ("Benchmarking and modelling") who will carry out the benchmarking and compare data.

[1] A. Kreuzer, P. Martínez, H. E. Plesser, P. Petkov, V. Pavlov, J. Romein, J. Amaya, D. Gonzalez, M. Riedel, M. Girone. "Application co-design input", Deliverable D1.1, DEEP Extreme Scale Technologies (2017).

2 Task 1.2: Neuroscience (NMBU)

The long-term goal of the neuroscience task in DEEP-EST is to provide an optimised setup for the integrated simulation and analysis of large-scale brain activity. 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 the simulation to ensure that dynamics stay within relevant regimes. In DEEP-EST, we will focus on simulations of functional models of brain structure simulated using the NEST simulator combined with two types of *in situ* analysis: computation of electrical local field potentials using the Arbor and HybridLFPy packages on the one side, and statistical analysis of spike activity using the Elephant package on the other. NEST will be executed on the CM, Arbor/HybridLFPy on the ESB and Elephant on the DAM. NEST output will be communicated to the analysis packages using the MUSIC library.

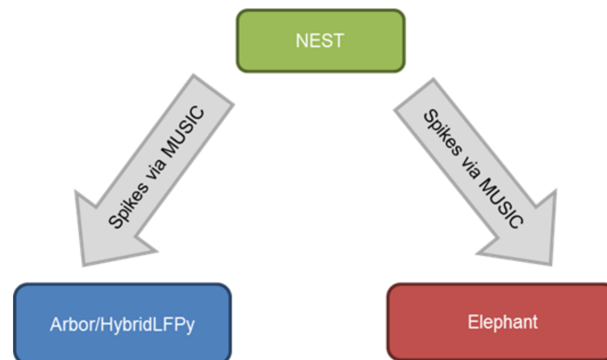


Figure 1: Neuroscience simulation and analytics workflow for NEST with *in situ* computation of local field potentials using Arbor and HybridLFPy (left path) and *in situ* statistical analysis using Elephant (right path).

Key performance constraints in the neuroscience workflow will be the performance of the NEST, Arbor and Elephant applications (HybridLFPy provides only configuration and limited postprocessing steps). These applications should therefore be benchmarked and traced independently of each other. Since the applications individually scale well, one can later adjust the resources devoted to each particular application to achieve balanced performance among all parts. We will therefore discuss the applications separately in this document. For each application, we describe the application itself, followed by a description of which quantities to measure and a description of the benchmark cases. Technical details on the benchmark configurations using JUBE are provided in the DEEP-EST Gitlab repository for benchmarks[2].

2.1 Benchmarking NEST

2.1.1 Application structure

2.1.1.1 Overview

NEST is a simulation code for the investigation of 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 which are connected to each other by the synapses.

[2] https://gitlab.version.fz-juelich.de/DEEP-EST/Benchmarks/blob/master/Applications/NEST/README.md

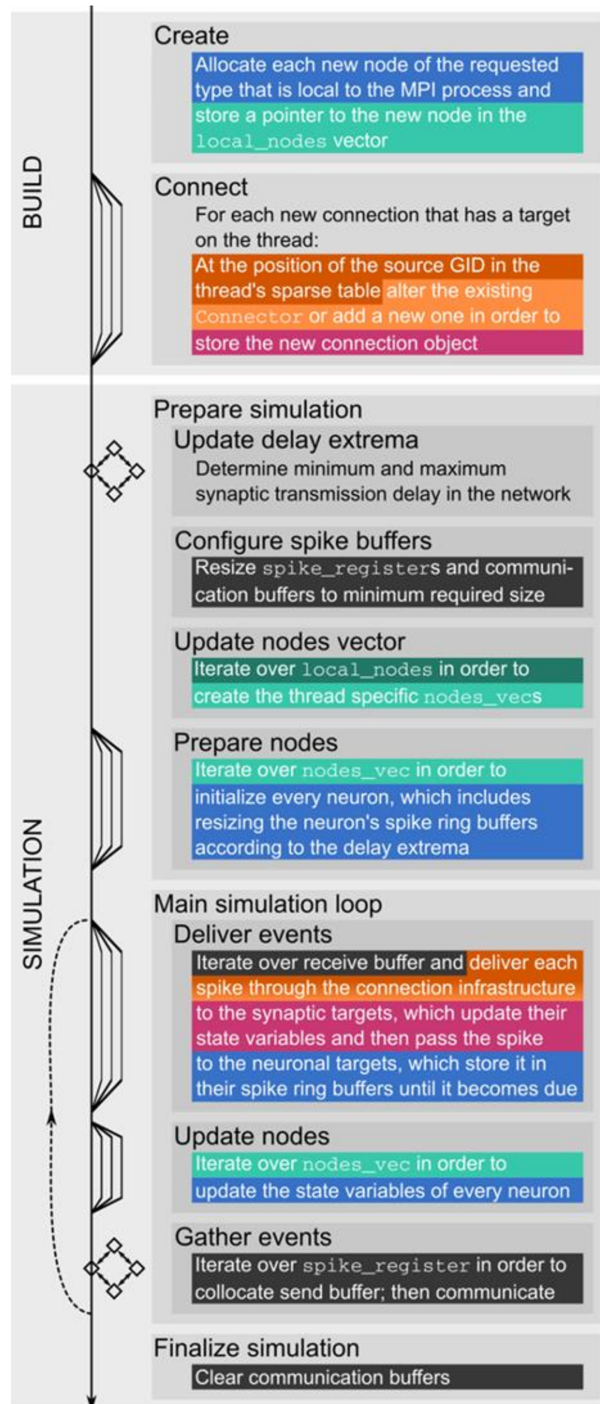


Figure 2: Major steps of build and simulation phase of NEST. The vertical black arrow on the left indicates single-threading and multi-threading (single and multiple lines, respectively), MPI communication (squares with bidirectional arrows), and the repetition of the main simulation cycle (dashed upward pointing arrow). Coloured and dark grey text highlighting corresponds to the data structures shown in Figures 2, 3, respectively; they indicate when the data structures are created, changed or accessed. From Kunkel and Schenck (2017) under CC BY license.

NEST considers the 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 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. In biology, each neuron provides input to $\sim 10^4$ other neurons and receives input from about as many. The largest NEST simulation to date simulated $1.86 \cdot 10^9$ neurons connected by $11.1 \cdot 10^{12}$ synapses using the full K computer in Kobe, Japan[3].

The code is written in idiomatic C++98, using object-oriented features and generic programming based on C++ templates. For parallelisation, a hybrid scheme combining MPI and OpenMP is used. Each of M MPI processes has the same number T of threads for a total number of $N_{VP} = M \cdot T$ virtual processes. For a fixed number N_{VP} of virtual processes (VP), any NEST simulation shall produce identical results regardless of how the virtual processes are divided between MPI processes and OpenMP threads.

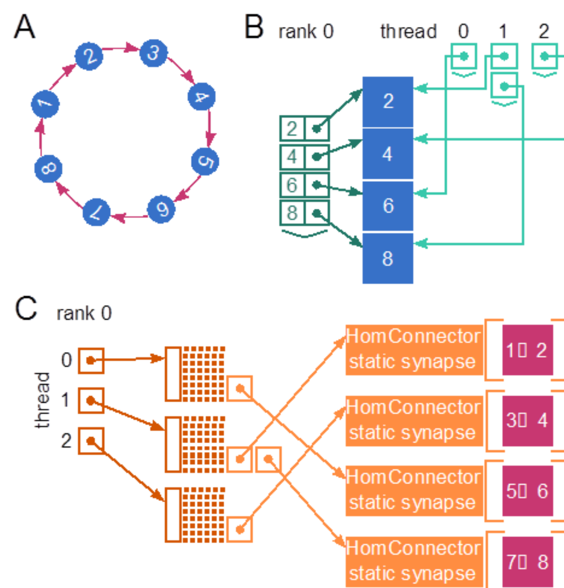


Figure 3: Fundamental data structures in NEST. Data structures on MPI rank 0 for (A) an example network of eight neurons with ring-like connectivity, which is simulated using two MPI processes and three threads per process. For simplicity, stimulating and recording devices are omitted. (B) Neuron infrastructure. For each local neuron (blue squares) the SparseNodeArray local_nodes (dark green) stores a struct of a pointer to the neuron and the neuron's GID. The two-dimensional vector nodes_vec (light green) stores a pointer to the local neurons sorted by thread. (C) Connection infrastructure. Each synapse is represented on the thread of its target neuron. Each thread owns a sparse table (dark orange), which stores a pointer to a Connector (orange) for every source neuron that has targets on the thread. The connectors hold the local synapses (pink) sorted by type (here only one static synapse per connector). From Kunkel and Schenck (2017) under CC BY license.

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 virtual process managing the connection's target neuron. For large simulations, this data structure dominates memory consumption. The NEST memory model can provide estimates of memory requirements

[3] S. Kunkel, M. Schmidt, J. M. Eppler, G. Masumoto, J. Igarashi, S. Ishii and et al., "Improved memory model and overhead reduction: Spiking network simulation code for petascale computers," in *Front. Neuroinform. 8:78. doi: 10.3389/fninf.2014.00078*, 2014.

based on a small number of parameters[2,4]. The build phase takes up a significant fraction of the overall time for a simulation experiment and can well be in the range taken up by the simulation phase.

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.

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[5,6], with PyNEST being the default interface.

2.1.1.2 Network representation

NEST represents a neuronal network as a directed graph. For currently relevant use cases, this graph has between 10^5 and 10^9 neurons[7], while future simulations of models of the human brain will comprise some 10^{11} neurons. The in- and out-degree of neurons is around 10^4, with connections (edges) spread widely throughout the entire graph, i.e., the graph can generally not be partitioned into weakly coupled subgraphs. The total number of connections in current use cases is thus 10^9–10^{13} connections, which need to be stored, distributed across compute nodes.

Each neuron is represented on exactly one virtual process (VP) by a C++ object with a typical size of around 1 KByte, although some neuron models have considerably larger neuron objects. Neurons are assigned to VPs in a round-robin fashion, and each neuron is identified by a globally unique global ID (GID, 64-bit integer).

The state of typical neurons is represented by a small number (< 10) of doubles governed by linear differential equations, which are updated using exact integration[8]. A single update step usually requires of the order of ten additions and multiplications. More complex neurons described by systems of non-linear ODEs are currently integrated using solvers from the

[4] S. Kunkel, T. C. Potjans, J. M. Eppler, H. E. Plesser, A. Morrison and M. Diesmann, "Initial memory model and overhead reduction: Meeting the memory challenges of brain-scale network simulation," in *Front. Neuroinform. 5:35. doi: 10.3389/fninf.2011.00035*, 2012.

[5] J. M. Eppler, M. Helias, E. Muller, M. Diesmann and M.-O. Gewaltig, "PyNEST: a convenient interface to the NEST simulator," in *Front. Neuroinform. 2:12. doi: 10.3389/neuro.11.012.2008*, 2008.

[6] Y. V. Zaytsev and A. Morrison, "CyNEST: a maintainable Cython-based interface for the NEST simulator," in *Front. Neuroinform. 8:23. doi: 10.3389/fninf.2014.00023*, 2014.

[7] In graph theoretical terminology, the neuronal network should be described in terms of nodes connected by edges. Since the use of "node" invariably will lead to confusion with compute nodes, it will be referred to the nodes of the neuronal network as "neurons" throughout, even though some of these "neurons" may represent devices, i.e., nodes injecting signals into or recording signals from the network. Also "connections" is used for the edges of the graph.

[8] S. Rotter and M. Diesmann, "Exact digital simulation of time-invariant linear systems with applications to neuronal modeling," in *Biol. Cybern. 81*, 1999, pp. 381-402.

GNU Scientific Library (GSL); the most complex model currently implemented has a 16-dimensional state vector.

Connections between neurons are represented exclusively on the VP storing the target node of the connection. This is (a) necessary for large-scale simulations where the memory required to store connections by far exceeds the memory available on a single compute node, (b) permits the connection construction in parallel on all VPs in most cases and (c) minimizes the amount of information that needs to be exchanged between processes; for details, refer to papers on the initial pure MPI implementation[9] and the current hybrid MPI-thread implementation[10].

The current data structures used to represent neurons and connections are based on systematic memory modelling and consequent optimisation[2,3]. There are also recent results on network construction and sensitivity to memory allocation issues for large numbers of threads[11].

The NEST kernel described so far and illustrated in Figure 2 is the so-called 4th-generation simulation kernel used in NEST releases 2.6.0–2.14.0. A new, 5th-generation NEST kernel is currently in prototype state and is expected to be integrated in the mainline NEST kernel by Q2/2018. The key step from NEST-2.14 to NEST-5G is a new connectivity representation and spike exchange scheme using directed communication based on `MPI_Alltoall()`; for details see Jordan et al. (2018)[12].

2.1.1.3 Network dynamics

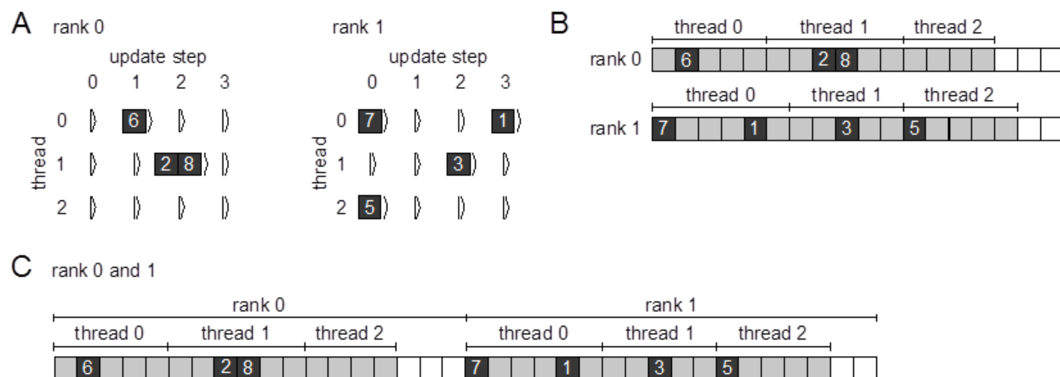


Figure 4: Five spike buffers for a simulation cycle with four neuronal update steps and for a network of eight neurons, which is simulated using two MPI processes and three threads per process. (A) During neuronal updates the three-dimensional vector spike_register stores the GIDs of the local neurons that spike (dark grey squares) sorted by thread and update step. (B) Before MPI communication each rank collocates its send buffer based on the entries in its spike_register. Communication markers (light gray

---

[9] A. Morrison, C. Mehring, T. Geisel, A. Artsen and M. Diesmann, "Advancing the boundaries of high connectivity network simulation with distributed computing," in *Neural Comput. 17 1776–1801. doi: 10.1162/0899766054026648*, 2005.

[10] H. E. Plesser, J. M. Eppler, A. Morrison, M. Diesmann and M.-O. Gewaltig, "Hybrid MPI-thread parallelisation: Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers," in *Euro-Par 2007: Parallel Processing, Vol. 4641, Lecture Notes in Computer Science, eds A.-M. Kermarrec, L. Bougé, and T. Priol (Berlin: Springer-Verlag) 672–681. doi: 10.1007/978-3-540-74466-5*, 2007.

[11] T. Ippen, J. M. Eppler, H. E. Plesser and M. Diesmann, "Improved network construction and memory allocation for massively parallel systems: Constructing Neuronal Network Models in Massively Parallel Environments," in *Front. Neuroinform. 11:30. doi: 10.3389/fninf.2017.00030*, 2017.

[12] Jordan J, Ippen T, Helias M, Kitayama I, Sato M, Igarashi J, Diesmann M and Kunkel S (2018) Extremely Scalable Spiking Neuronal Network Simulation Code: From Laptops to Exascale Computers. *Front. Neuroinform.* 12:2. doi: 10.3389/fninf.2018.00002

squares) define update step and thread. Buffers may not be completely filled (white squares). (C) After MPI communication using MPI_Allgather, each rank holds the complete spike data in its receive buffer (global spike buffer), which is the concatenation of the send buffers of all ranks. Modified from Kunkel and Schenck (2017) under CC BY license.

Neurons are usually updated on a fixed time grid and check at the end of each time step whether a threshold condition is fulfilled. In that case, the neuron emits an output pulse, it “fires a spike”. Since spikes are stereotyped pulses, the only relevant information about a spike event is (a) the GID of the neuron emitting the spike and (b) the time step at which the spike is emitted. Spikes are transmitted to all connection targets of a neuron with a finite, non-zero delay and a weight, which may differ from connection to connection. Because spikes are always with finite delay, updates on different VPs can be decoupled for as long as the minimal propagation delay without violating causality. Virtual processes therefore independently update their neurons throughout a full minimum-delay interval and buffer spikes emitted during this interval locally.

At the end of each minimum-delay interval, spikes are first aggregated across all threads on each MPI process and then exchanged between MPI processes using `MPI_Allgather()` or `MPI_Allgatherv()`. During this exchange, only the GIDs of the neurons that have spiked need to be communicated, while time-step information is provided by sentinels (see Morrison et al. (2005)[8] for details). This keeps the total amount of data to be exchanged small.

After the MPI exchange, each VP knows about all spikes emitted in the entire network during the previous minimum delay interval. Each VP now delivers these spikes to all their connection targets on the given VP. This requires (a) an almost random traversal of the large adjacency data structure representing connections (50% of all memory for mid-size networks, well over 90% for very large networks[2,10]), (b) almost random write access to the input buffers of neurons receiving spikes, and (c) for plastic connections, i.e., connections with weights changing over time, modifying access to the activity history of target neurons[11].

2.1.2 What to measure

The key performance measure for NEST is time to solution. This time includes the time for network construction (*build time*) and for simulating network dynamics (*simulation time*). Our benchmark JUBE scripts extract these numbers. For most applications, minimizing the simulation time has top priority, provided that the build time does not increase significantly as a side effect.

The main constraint on large-scale network simulations with NEST is the memory required to store network connectivity. Therefore, the total virtual memory size for the simulation must not increase significantly due to optimisations. This memory size is also reported by our JUBE scripts.

Past benchmarking and analysis efforts indicate that the "deliver events" phase of the simulation loop consumes most time. This phase requires repeated, and essentially random, lookups in the connectivity data structure illustrated in Figure 3C, followed by write operations into the input buffers of neurons receiving spikes, also in random order. The call path for this process is

```
SimulationManager::update_()
```

```
-> EventDeliveryManager::deliver_events()
-> ConnectionManager::send()
-> Connector::send()
-> <SomeConnectionClass>::send()
-> SpikeEvent::operator()()
-> iaf_psc_alpha::handle( SpikeEvent& )
```

In this chain,

- `deliver_events()` iterates over all spikes fired during the previous minimum delay interval.
- `ConnectionManager::send()` performs one sparse table lookup per spike (cf. Figure 3C); the sparse table has one entry per neuron in the network. Subsequent lookups will jump randomly through the sparse table.
- `Connector::send()` iterates over all connection targets that a given sender neuron has on a given thread. These targets are essentially random among the neurons managed by a thread.
- `<SomeConnectionClass>::send()` and `SpikeEvent::operator()()` essentially forward the spike event; for networks with plastic connectivity, this step may involve additional, complex operations.
- `iaf_psc_alpha::handle( SpikeEvent& )` registers the spike in the input buffer of the neuron (write operation).

The last step writes into a `RingBuffer`. Each neuron has three ring buffers. These buffers are `std::vector<double>` and are created/resized by the `init_buffers_()` method of each neuron model class during the "Prepare nodes" phase. The size of the ring buffers is given by the sum of the shortest and longest transmission delay in the network, expressed in simulation time steps; it is the same for all neurons in a simulation.

Initial attempts to trace performance of the spike delivery process using Extrae have run into problems due to the extremely large traces generated as the `handle()` method is called very frequently. We have not yet explored windowing or selective sampling.

2.1.3 Benchmarks

2.1.3.1 Code requirements

The DEEP-EST GitLab repository contains both NEST 2.14 and NEST 5G versions. While NEST 2.14 is a definite release and code will not be modified in the future, the source code in the NEST 5G repository may still be modified as we optimise and prepare it for mainline integration. NEST 5G is mainly provided to explore the effect of the new communication patterns used in that kernel.

We recommend using the thread-aware Jemalloc allocator with NEST 2.14 to achieve faster build times[4]. To obtain reproducible results for simulations using the same total number of threads but different combinations of number of tasks and number of threads per task, compiling NEST in a way that ensures strict IEEE754 compliance is required. In our experience so far, this has only marginal effect on performance, since NEST is heavy on memory access and relatively light on floating-point operations.

2.1.3.2 Benchmark configurations

Benchmarks are implemented using the native SLI language of the NEST simulator, a PostScript-like language with domain-specific extensions. We provide two different benchmarks at present, of which the second (simplified multi-area model benchmark) is the scientifically most relevant. Several JUBE configurations for each benchmark are provided, as described in detail in the `README.md` file in the NEST section of the Benchmark repository. Additional suggestions for scaling are given in Section 2.1.3.2.3, while Section 2.1.3.4 provides information on how to interpret benchmark output.

2.1.3.2.1 Standard HPC benchmark

A standard HPC benchmark has been used to evaluate NEST performance in several recent publications[2,10]. This benchmark simulates a network model with two neuronal populations (80% and 20%) in which each neuron has K=11,250 incoming synapses. The network size can be scaled freely from scale 10 upwards (112,500 neurons); for smaller network sizes, K is reduced. Average neuron firing rates are typically 5 to 10 Hz. All synaptic delays are fixed to 1.5 ms and 64% of the synapses are plastic, while the remainder are static.

Important properties of this benchmark case are

- the minimal delay is 1.5 ms, while the simulation time step is 0.1 ms. Therefore, thread synchronization and MPI communication is required only once every 15 simulation time steps;
- neurons in this benchmark have relatively small input ring buffers (3 buffers with 30 doubles per neuron). Simulating approximately 10000 neurons per process, the total size of the ring buffers is thus approximately 7MB per process;
- transmitting spikes via plastic synapses involves complex operations (spike history lookup in dequeues, computation of exponentials).

2.1.3.2.2 Simplified multi-area model benchmark

This benchmark closely mimics the behaviour of the neuroscientifically relevant multi-area model in terms of memory access patterns, computational load, and communication load.

The benchmark deviates from the standard benchmark in the following ways:

- the number of incoming synapses is reduced to half (K=5625);
- all synapses have static weights;
- the average firing rate is increased to approximately 14.8 spikes/s;
- transmission delays are uniformly distributed in [0.1, 50] ms.

Important consequences of these changes are

- significantly reduced memory requirement;
- a minimum delay of 0.1 ms and hence thread synchronization and MPI communication after each time step (15 times more often than in the standard benchmark);
- larger ring buffers with 501 elements each, so that ring buffers in total require approximately 43MB per process;
- no complex plasticity operations during spike delivery.

This hpc_mam benchmark therefore stresses crucial parts of the system (process synchronization, communication, random writes to ring buffers) more than the standard benchmark.

2.1.3.2.3 Scaling options

To create new benchmark configurations, copy one of the existing benchmark scripts in the folder jube_hpc and modify the following section (last three lines do not apply to hpc_mam):

```
<parameterset name="scale_check">
  <parameter name="SCALE" type="int">20</parameter>
  <parameter name="SIMTIME" type="float">1000</parameter>
  <parameter name="NUM_REC" type="int">1000</parameter>
  <parameter name="PLASTIC" type="string">false</parameter>
  <parameter name="D_MIN" type="float">0.1</parameter>
  <parameter name="D_MAX" type="float">1.5</parameter>
  <parameter name="NUMBER_OF_NODES" type="int">1</parameter>
  <parameter name="TASKS_PER_NODE" type="int">24</parameter>
  <parameter name="THREADS_PER_TASK" type="int">1</parameter>
</parameterset>
```

- `SCALE` determines the number of neurons: Scale 1 corresponds to 11,250 neurons.
- `SIMTIME` is the simulation time after equilibration in ms; this can be shortened.
- `NUM_REC` is the number of neurons to record activity from.
- `PLASTIC` determines whether synapses are plastic (true) or note (false).
- `D_MIN` is the minimum transmission delay in ms.
- `D_MAX` is the maximum transmission delay in ms.
- `NUMBER_OF_NODES` is the number of compute nodes to use (SLURM).
- `TASKS_PER_NODE` is the number of MPI tasks per compute node (SLURM).
- `THREADS_PER_TASK` is the number of OpenMP threads per MPI task (SLURM).

Remarks:

- For the standard benchmark: `PLASTIC==true`, `D_MIN=1.5`, `D_MAX=1.5`.
- For the mam benchmark: `PLASTIC==false`, `D_MIN=0.1`, `D_MAX=50.0` (not configurable).
- The total number of virtual processes is given by `N_VP = NUMBER_OF_NODES x TASKS_PER_NODE x THREADS_PER_TASK`.
- Benchmarks performed with the same `N_VP` but different splits between nodes, tasks and threads shall return the same simulation results (number of spikes, see section on output).
- Small values of `D_MIN` require more frequent thread synchronization and MPI communication.
- Ring buffer size is given by `10 x (D_MIN + D_MAX)`.

2.1.3.3 Benchmark stages

Both benchmarks proceed in the following stages and provide output after each stage:

1. **NEST startup**. NEST starts, including MPI_Init(), loads all libraries and prints welcome message.

2. **Create neurons and devices**. All network nodes created; expected additional memory use is small.
3. **Connect neurons**. Most memory allocation occurs during this phase.
4. **Initial simulation phase**. Prepare simulation (MPI communication to exchange minimal and maximal delay, resizing ring buffers to correct size, calibrate individual neurons; in NEST 5G also exchange of connectivity information) and simulate 10 ms.
5. **Presimulation phase**. Simulate 90 ms to allow initial transients in network activity to subside.
6. **Simulation phase**. Simulate for given SIMTIME, by default 1000 ms.

2.1.3.4 Benchmark output

All benchmark output is written to `stdout` by all MPI ranks. Output contains normal NEST startup and progress messages as well as the following benchmark log messages:

```
0 153236 # virt_mem_0

0 1.41 # build_time_nodes
0 752880 # virt_mem_after_nodes

0 96.51 # build_edge_time
0 70578416 # virt_mem_after_edges

0 70578416 # virt_mem_after_init
0 0.85 # init_time

0 70578416 # virt_mem_after_presim
0 6.41 # presim_time

0 70578416 # virt_mem_after_sim
0 71.33 # sim_time

0 7.531 # average rate
0 225000 # num_neurons
0 2531476000 # num_connections
0 1.5 # min_delay
0 1.5 # max_delay
0 1723039 # local_spike_counter
```

- The first number on each line is the rank reporting the information.
- `virt_mem` entries are virtual memory size in KB after the respective phases.
- `time` entries are time in seconds used for different phases.
- `average_rate` is averaged over locally recorded neurons, while `local_spike_counter` is the total number of spikes fired on the given rank; this number is more reliable than `average_rate`.
- `local_spike_counter` should be summed across all ranks for comparisons.
- `num_neurons` is the total number of neurons in the network.
- `num_connections` is the number of connections *on the rank*, it should be summed across ranks to obtain the total number of connections in the network.

JUBE output for NEST benchmarks is approximately as follows (slightly simplified for formatting purposes):

Table 1: JUBE output for NEST benchmarks.

N_NODES	TSKS_P_ND	THR_P_TSK	SCALE	T_nrns	T_conns	T_ini	T_equ	T_sim	VSize	N_spks	Rate	N_nrns	N_conns
1	24	1	4	0.02	2.87	0.29	1.12	13.09	6881856	1303925	28.9	45000	101284246

Here

- `T_nrns` is the time to create neurons.
- `T_conns` the time to create connections.
- `T_ini` the time for the initial simulation.
- `T_equ` the time for presimulation.
- `T_sim` the time for simulation SIMTIME.
- `VSize` the virtual memory size at the end of the simulation, summed across ranks.
- `N_spks` the total number of spikes fired during the simulation.
- `Rate` the average firing rate across ranks.
- `N_nrns` the total number of neurons in the network.
- `N_conns` the total number of connections in the network.

For `T_conns` and `T_ini`, both minimum and maximum across ranks are given. Differences are usually due to waiting times, so one usually has `T_conns_min + T_ini_max == T_conns_max + T_ini_min`. For NEST 5G, `T_ini` will contain the time required to exchange connectivity information between ranks; in general, one should therefore consider the sum `T_conns + T_ini` for comparisons.

2.2 Benchmarking Arbor

2.2.1 Application structure

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 the 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.

Arbor is a C++11 application that parallelises using MPI, C++11 threads and Intel TBB. It supports vectorisation using AVX2 and AVX512 as well as GPGPUs through CUDA. At present, Arbor simulations need to be hard-coded in C++, although a Python front-end is under development. Arbor benchmarks for DEEP-EST will therefore be based on a "miniapp" example included with the Arbor source code. Once the Python front-end becomes available, we will extend the range of benchmarks.

The present Arbor benchmarks rely on pure synthetic input data. Arbor simulations will provide by far the largest computational load of the Arbor/HybridLFPy toolchain. Arbor benefits significantly from vectorisation. Its CMake file supports compilation for AVX2 and AVX512, for which we provide JUBE build files, and also supports CUDA and TBB; see README.md for details. We strongly recommend benchmarking Arbor only with vectorisation support enabled, as Arbor is extremely slow without it.

2.2.2 What to measure

Similarly to NEST, time to completion is essential for Arbor. It is composed of a model-init and model-simulation time. Simulation time in particular should be minimized, provided that initialization time does not increase significantly. As Arbor will only be used to model a small subset of the full-scale NEST models, memory consumption does not play a significant role in Arbor benchmarks.

2.2.3 Benchmarks

2.2.3.1 Code requirements

Code installation and requirements are described in the README.md file in the DEEP-EST Gitlab repository for NEST benchmarks.

2.2.3.2 Benchmark configurations

All benchmark configurations use the Arbor example miniapp.exe executable, but with different neuron configurations.

2.2.3.2.1 Standard three-segment-dendrite benchmark

This is a simple benchmark case for Arbor. It implements a population of neurons, where each neuron consists of a soma and three dendritic segments: one starting at the soma and two branching at the end of the initial dendritic segment. Each dendritic segment is divided into a configurable number of compartments, which form a linear chain. Neurons are connected to each other in a random fashion through synapses. Started by an initial injected spike, neurons drive each other to spike.

2.2.3.2.2 Non-spiking pyramidal cell benchmark

This benchmark case reads the morphology (spatial structure) of a real neuron from file and creates a requested number of duplicates of this model neuron. The model neuron has some 200 dendritic segments forming a complex tree structure. The numerical problem to be solved is thus somewhat more complex, since segments at branching points in the tree have three neighbours instead of two, and at least one of those neighbours is likely to not be in the immediate vicinity in the data structure representing the segment. Neurons are also

connected in a random fashion, but the cell parameters are chosen such that the neurons do not respond with new spikes to the initial spike injected into the network.

2.2.3.2.3 Spiking pyramidal cell benchmark

This benchmark case reads a scaled version of the morphology of the cell used in the non-spiking benchmark case. As a consequence of this scaling, the neurons in this benchmark respond to the initial spike with output spikes of their own, leading to longer simulation times.

2.2.3.3 Benchmark output

JUBE scripts output for Arbor simulations is as follows (slightly simplified):

Table 2: JUBE output for Arbor benchmarks.

N_NODES	TSKS_P_ND	THR_P_TSK	SCALE	NUM_SYN	NUM_COMP	SIMTIME	T_setup	T_init	T_simulate	N_spikes
1	1	48	2.0	2000	7	100.0	0.0	19.389	119.878	1

- The first entries are the same as for NEST benchmarks, but Arbor benchmarks allow fractional `SCALE`.
- `NUM_SYN` is the number of synapses per neuron.
- `NUM_COMP` is the number of compartments *per segment*. To obtain a total of approximately 1000 compartments per neuron, comparable with relevant cases, this should be around 7 for the pyramidal cell cases and much larger for the three-segment benchmark.
- `SIMTIME` is the time simulated in milliseconds.
- `T_setup` is setup time.
- `T_init` is initialization time.
- `T_simulate` is simulation time.
- `n_spikes` is the number of spikes fired during SIMTIME, including the one injected spike.

In addition, Arbor writes a detailed report indicating the time spent on `stdout` procedures, but we do not currently extract this information in our JUBE scripts.

2.3 Benchmarking Elephant

2.3.1 Application structure

Elephant[13] 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. Herein 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

[13] https://elephant.readthedocs.io

provided by the NumPy convolution implementation, cross-correlation algorithms are purely serial at present.

Detection of synfire chains uses the ASSET algorithm[14] in a recently optimised version[15], replacing the non-optimised version currently included in the release version of Elephant. The optimised algorithm uses MPI4Py for parallelisation.

2.3.2 What to measure

Time to completion is the essential quantity for Elephant benchmarks as well. All Elephant benchmarks provided at present generate their own synthetic data. The time required to generate this synthetic data should not be taken into consideration when benchmarking Elephant.

2.3.3 Benchmarks

2.3.3.1 Code requirements

The benchmarks require the most recent version of Elephant (0.4.3) to be installed, as described in the Elephant documentation. In addition, the optimised ASSET algorithm, including a Cython extension module, must be installed from the NEST Benchmark section of the DEEP-EST Gitlab repository. For details, see the README.md file in that repository.

2.3.3.2 Benchmark configurations

2.3.3.2.1 Pure-Python cross-correlation

This benchmark computes cross-correlations between spike trains using a pure Python implementation. This benchmark is purely serial.

2.3.3.2.2 NumPy-supported cross-correlation histograms

This benchmark computes cross-correlation histograms between spike trains using NumPy convolution functions. This benchmark might benefit from thread-parallelisation, depending on the underlying NumPy libraries.

2.3.3.2.3 ASSET on random data

This benchmark applies the optimised ASSET algorithm to purely random data in which no synfire chains should be detected. This benchmark should benefit from MPI via MPI4Py.

2.3.3.2.4 ASSET on data with patterns

This benchmark applies the optimised ASSET algorithm to random data with an injected synfire chain. It should thus detect and report a chain. This benchmark should benefit from MPI via MPI4Py.

[14] Torre E, Canova C, Denker M, Gerstein G, Helias M, Grün S (2016) ASSET: Analysis of Sequences of Synchronous Events in Massively Parallel Spike Trains. PLoS Comput Biol 12(7): e1004939. doi:10.1371/journal.pcbi.1004939

[15] Carlos Canova, Wouter Klijn, Paul Baumeister, Alper Yegenoglu, Michael Denker, Dirk Pleiter, Sonja Grün (2017) ASSET for JULIA: executing massive parallel spike correlation analysis on a KNL cluster. Poster presented at HBP Summit 2017.

3 Task 1.3: Molecular dynamics (NCSA)

A molecular dynamics (MD) simulation, such as Gromacs, generally tracks the trajectories of many particles (atoms) evolving over time. It solves differential equations of motion in time steps. The coordinates and velocities of the particles are calculated by using the coordinates and velocities from the previous time step. In each time step one has to calculate forces acting on each atom. This is the most time consuming operation. Usually pairs of atoms are defined in a predefined cut-off radius calculating short range interactions, while the long range interactions are calculated using FFT based algorithms.

3.1 Use case description

MD simulations of different size and topology are set up to test and explore both the computer architecture and software optimisations if any. The prepared benchmark aim at exploring and test the hybrid MPI/OpenMP programing model of Gromacs, which utilises the most widely used families of vector instruction sets. The main benchmark set consist of three atomic systems described in the following table:

Table 3: Gromacs use case description.

Name	Description	Number of atoms Simulation box dimensions	DEEP partition (number of nodes)
Magainin	Antimicrobial peptide Magainin solvated in water	34k 8x8x5.3 nm^3	KNL (1,2,4) SDV (1,2,4,8)
Bombinin	27 antimicrobial peptide Bombinin molecules forming 2 aggregates in water solutions	325k 15x15x15 nm^3	KNL (1,2,4) SDV (1,2,4,8)
Ribosome	Ribosome unit solvated in water	2.2M 31.6x31.6x22.3 nm^3	KNL (1,2,4) SDV (1,2,4,8)

The three benchmarks are chosen to explore the machines architecture and the corresponding interconnect. The molecular dynamics parameters (cut-off radii and long-range electrostatics treatment) match the force field requirements while other parameters have typical values used by researchers. The v-rescale thermostat and Parrinello-Rahman barostat are used to control temperature and pressure, respectively, and the time step is consistent with the level of restraining (h-bonds, all-bonds, v-sites). The Particle Mesh Ewald (PME) algorithm is used for the long-range electrostatics calculation. The reader is referred to the Gromacs documentation[16] for a detailed description of algorithms and parameters.

3.1.1 Magainin system

The Magainin system has both the smallest number of atoms and box size and it is designed to scale on a couple of nodes. V-sited is used when applying force field parameterisation and a fixed time step value of 5 fs is chosen (1fs is equal to 10^{-15}s). Considering the limit of 100 atoms per thread as an absolute lower limit of Gromacs efficiency, the magainin case should scale up to four SDV partition nodes. The data fit mostly in the CPU core's cache and the benefit of using dedicated SIMD kernels should be visible. The size of communication

[16] http://manual.gromacs.org/documentation/2018/manual-2018.pdf

messages is relatively small and therefore the parameter limiting scalability is the latency of the node interconnect network.

3.1.2 Bombinin system

Nowadays, the majority of researchers use systems of this size to do their studies. The bombinin system should scale up to tens of nodes. Only the bonds connecting the hydrogen atoms to heavy atoms are restrained and the time step is fixed to 2 fs. As the number of nodes increases, the number of atoms per CPU core decreases. Depending of the node interconnect network one should find the beneficial strategy of using balance between the number of MPI ranks per node and the number of OpenMP threads per MPI rank.

3.1.3 Ribosome system

This is a relatively big system and, for sufficiently long trajectories, one needs a supercomputing platform. All bonds are constrained and the time step value is 2 fs. Scalability starts degrading with a relative high number of nodes due to the increase in (i) all-to-all communications between a dedicated subset of MPI ranks (PME algorithm) and (ii) point-to-point communications (particle domains – particle domain and particle domain – PME domain).

3.2 Benchmarking metrics

The quantity which we use to measure the performance is the simulated time per wall clock time units, namely nanoseconds per day [ns/day]. The faster the time step execution, the higher the performance. The time step value is determined by the physics.

3.3 MD benchmarking metrics

Our developed benchmark suite allows us to explore the performance of the main Gromacs simulation tool (mdrun) on both SDV nodes (`sdv` keyword) and KNL nodes (`knl` keyword). It consists of JUBE scripts and two bash shell scripts to execute run the JUBE benchmarks in a convenient way. Gromacs source code, benchmark systems, JUBE configuration files and shell scripts are hosted in the DEEP-EST GROMACS repository[17].

Firstly one should clone the git repository to get the source code and the rest of the files:

```
git clone ssh://git@gitlab.version.fz-juelich.de:10022/DEEP-EST/GROMACS.git
```

The current version of Gromacs is located in `src/gromacs-2018` subdirectory and the corresponding inputs for the test simulations can be found in the `tests` directory. JUBE xml files for compiling the MD package are:

- `gmx-2018-compile-knl.xml` and
- `gmx-2018-compile-sdv.xml`

Each test simulation can be run on a KNL partition via the following JUBE scripts:

- `gmx-2018-magainin-knl.xml`,
- `gmx-2018-bombinin-knl.xml` and

[17] https://gitlab.version.fz-juelich.de/DEEP-EST/GROMACS

- `gmx-2018-ribosome-knl.xml`.

For the SDV partition the following JUBE scripts are available:

- `gmx-2018-magainin-sdv.xml`,
- `gmx-2018-bombinin-sdv.xml` and
- `gmx-2018-ribosome-sdv.xml`.

As already mentioned, the process of compiling, running and results gathering can be simplified with dedicated shell. Those scripts will display help messages (usage and detailed description) by passing the argument `-h`. The entire benchmark procedure, applied on a specific partition (`sdv` for SDV nodes or `knl` for KNL nodes), consists of executing the following commands:

```
./compile-gromacs-2018-jube.sh <partition>

./submit-benchmarks.sh <partition>

./analyse-show-result-benchmarks.sh <partition>
```

The above compilation steps should not take more than 15 minutes.

Each benchmark case consists of a number of runs shown in the following table (wall clock execution time of 180 seconds):

Table 4 Gromacs benchmarks.

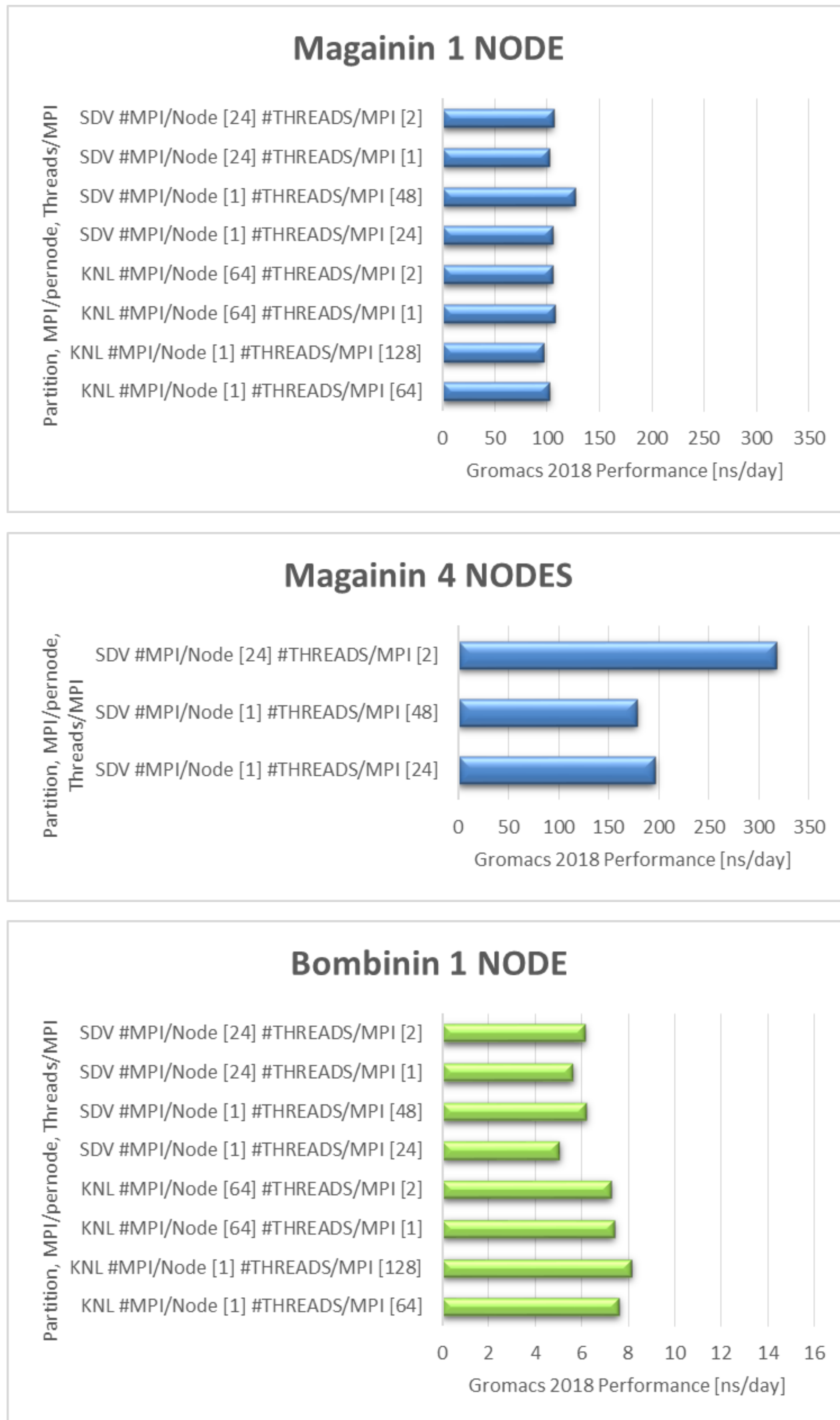
Partition	Benchmark case	Number of runs	Nodes/(MPIs per node)/(threads per MPI)
KNL	Magainin	10	1/1/64, 1/1/128, 1/64/1, 1/64/2, 2/1/64, 2/1/128, 2/64/1, 2/64/2, 4/1/64, 4/1/128
KNL	Bombinin	12	1/1/64, 1/1/128, 1/64/1, 1/64/2, 2/1/64, 2/1/128, 2/64/1, 2/64/2, 4/1/64, 4/1/128, 4/64/1, 4/64/2
KNL	Ribosome	12	1/1/64, 1/1/128, 1/64/1, 1/64/2, 2/1/64, 2/1/128, 2/64/1, 2/64/2, 4/1/64, 4/1/128, 4/64/1, 4/64/2
SDV	Magainin	14	1/1/24, 1/1/48, 1/24/1, 1/24/2, 2/1/24, 2/1/48, 2/24/1, 2/24/2, 4/1/24, 4/1/48, 4/24/1, 4/24/2, 8/1/24, 8/1/48
SDV	Bombinin	16	1/1/24, 1/1/48, 1/24/1, 1/24/2, 2/1/24, 2/1/48, 2/24/1, 2/24/2, 4/1/24, 4/1/48, 4/24/1, 4/24/2, 8/1/24, 8/1/48, 8/24/1, 8/24/2
SDV	Ribosome	16	1/1/24, 1/1/48, 1/24/1, 1/24/2, 2/1/24, 2/1/48, 2/24/1, 2/24/2, 4/1/24, 4/1/48, 4/24/1, 4/24/2, 8/1/24, 8/1/48, 8/24/1, 8/24/2

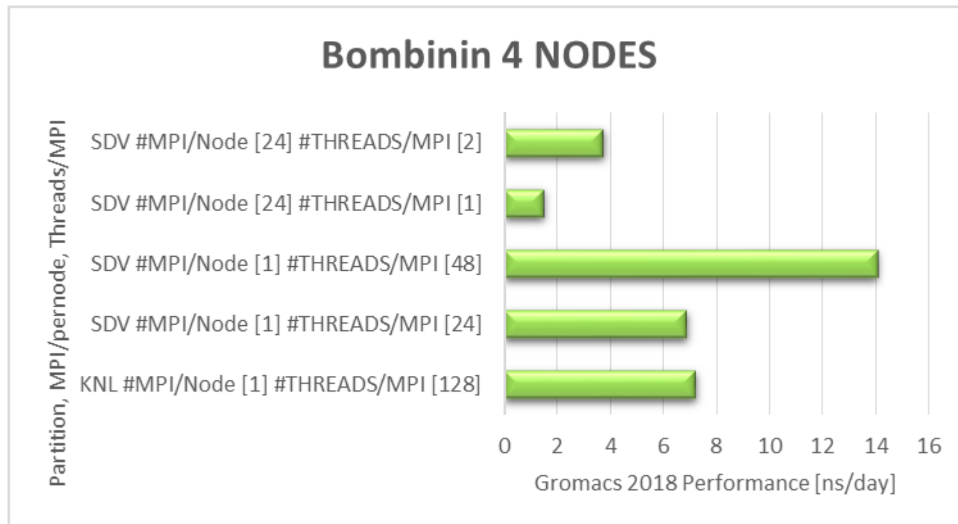
The time needed for running benchmarks on 4 KNL nodes or 8 SDV nodes should not exceed one hour.

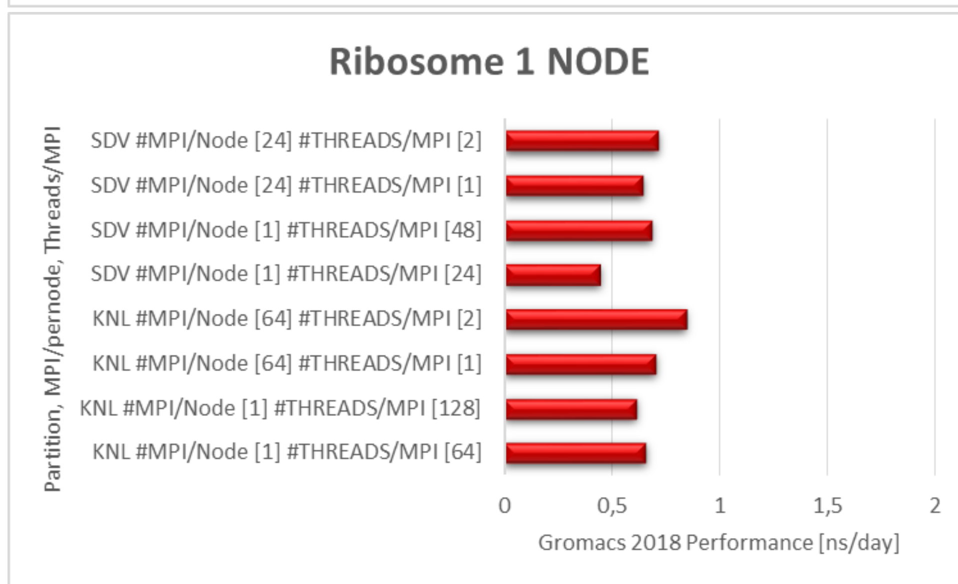
3.4 MD benchmarking results

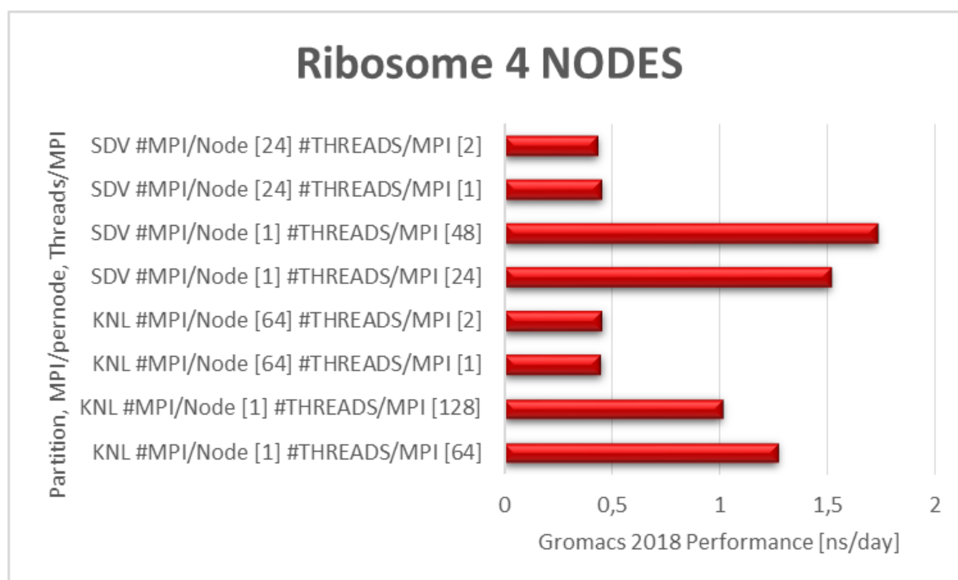
Some results of Gromacs performance on KNL and SDV partitions with different number of MPI processes per node and number of threads per MPI process are show in the following table:

Table 5: Gromacs benchmarking results.




Bombinin 4 NODES
Partition, MPI/pernode, Threads/MPI
SDV #MPI/Node [24] #THREADS/MPI [2]
SDV #MPI/Node [24] #THREADS/MPI [1]
SDV #MPI/Node [1] #THREADS/MPI [48]
SDV #MPI/Node [1] #THREADS/MPI [24]
KNL #MPI/Node [1] #THREADS/MPI [128]
0 2 4 6 8 10 12 14 16
Gromacs 2018 Performance [ns/day]


Ribosome 1 NODE
Partition, MPI/pernode, Threads/MPI
SDV #MPI/Node [24] #THREADS/MPI [2]
SDV #MPI/Node [24] #THREADS/MPI [1]
SDV #MPI/Node [1] #THREADS/MPI [48]
SDV #MPI/Node [1] #THREADS/MPI [24]
KNL #MPI/Node [64] #THREADS/MPI [2]
KNL #MPI/Node [64] #THREADS/MPI [1]
KNL #MPI/Node [1] #THREADS/MPI [128]
KNL #MPI/Node [1] #THREADS/MPI [64]
0 0,5 1 1,5 2
Gromacs 2018 Performance [ns/day]


Ribosome 4 NODES
Partition, MPI/pernode, Threads/MPI
SDV #MPI/Node [24] #THREADS/MPI [2]
SDV #MPI/Node [24] #THREADS/MPI [1]
SDV #MPI/Node [1] #THREADS/MPI [48]
SDV #MPI/Node [1] #THREADS/MPI [24]
KNL #MPI/Node [64] #THREADS/MPI [2]
KNL #MPI/Node [64] #THREADS/MPI [1]
KNL #MPI/Node [1] #THREADS/MPI [128]
KNL #MPI/Node [1] #THREADS/MPI [64]
0 0,5 1 1,5 2
Gromacs 2018 Performance [ns/day]

4 Task 1.4: Radio astronomy (ASTRON)

ASTRON has two applications for which will be set a performance baseline: the correlator and the imager. The correlator combines the signals from tens or hundreds of receivers, while the imager creates sky images from these correlations (after removing bad data affected by interference and after correlations are calibrated to compensate for instrumental and environmental effects). Both applications are described in the Application Description document.[18]

4.1 What to measure

Fortunately, both applications can benchmark themselves: they contain code that measures the runtimes (in seconds) and the performance (in TFLOPS) of all relevant compute kernels, as well as energy efficiency (in GFLOP/J) using either LIKWID[19] or PowerSensor[20]. Measuring energy efficiency is optional: if these libraries are not available, the applications are benchmarked without energy efficiency measurements. Yet, we are interested in energy efficiency, as this is not only a fair way to compare different processor architectures (more fair than just comparing FLOP counts), but also because the energy budget for imaging of the Square Kilometre Array (SKA) is tight: 5 MW per site for the entire Science Data processor. Imaging (gridding, degridding, FFTs) will need a large fraction of this, because this is computationally the most expensive operation.

Within the DEEP-EST project, we develop new implementations of our applications for FPGAs (using Intel's OpenCL/FPGA toolkit) and a new CUDA-based correlator using tensor core operations (tensor cores are special-purpose hardware designed to boost deep learning performance by roughly a factor of eight, but as they do just mixed-precision matrix multiplication, they should boost signal processing by a similar amount). These new implementations are still highly experimental and incomplete, and therefore they cannot be used to set a performance baseline. Instead, we will use versions of our applications that are somewhat more mature.

4.2 Measurement setup

Both applications can be run in such a way that they generate synthetic input, on the fly. This simplifies benchmarking and allows isolating compute performance from I/O performance. In the particular case of the correlator, it is difficult to set up I/O experiments because the high data rates require manual binding of network interrupt handlers and application threads to disjoint sets of CPU cores (which requires superuser privileges). This is further complicated in the presence of NUMA restrictions. Hence, we propose to only benchmark the computational part, even though we are well aware that I/O is challenging.

Neither of the applications uses MPI and thus run on a single machine. However, both applications perform poorly on CPU-only machines for different reasons. For the correlator, it would be fine to do the multiplications in 8 or 16 bits (short int or half float) depending on the

[18] https://bscw.zam.kfa-juelich.de/bscw/bscw.cgi/d2411360/application-description-ASTRON.pdf

[19] J. Treibig, G. Hager, and G. Wellein. LIKWID: A Lightweight Performance-oriented Tool Suite for x86 Multicore Environments. International Conference on Parallel Processing Workshops (ICPPW'10), pp. 207-216, San Diego, CA, September 2010.

[20] J.W. Romein and B. Veenboer. PowerSensor 2: a Fast Power Measurement Tool. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'18), Belfast, Northern Ireland, United Kingdom, April 2018, to appear.

telescope and the RFI (Radio Frequency Interference) environment; the minimum 32-bit operand size of present-day CPUs is overkill. Preliminary experiments with GPUs that support smaller operand sizes show that 40-80 TFLOPS can be achieved while high-end CPUs achieve about 2 TFLOPS (single precision floating point). The imager needs very good sine/cosine performance, as in the critical path, the imager does one sine/cosine operation for every 17 fused-multiply-add operations. Current CPUs cannot perform such transcendental operations using single hardware vector instructions. Hence on CPUs the imager spends roughly 85% of the time computing sines and cosines. In contrast, GPUs can perform these operations with dedicated hardware (some GPUs even concurrently with fused-multiply-add instructions). The performance/energy efficiency gap between GPUs and CPUs is typically about a factor of 15. As a result, we think that setting a performance baseline on CPUs (like the SDV) is not meaningful: it would be too easy to improve on this in the course of this project, and the SKA Science Data Processor cannot be built within its power constraints using CPU-only systems. We therefore propose to use a GPU-based system like JURON as reference platform, as accelerators are indispensable on our path to exa-scale.

Both applications are implemented in OpenCL and CUDA. For the correlator, the OpenCL implementation is more complete and robust (it is in operational use by the AARTFAAC telescope), while the CUDA implementation is an experimental mock-up, more suitable for research. For the imager, the OpenCL and CUDA implementations are similar. On NVIDIA GPUs, it is better to use the CUDA implementation, as it performs better (even though there is no fundamental reason why the OpenCL implementation could not perform as well as the CUDA implementation) and because the NVIDIA Visual Profiler does not support OpenCL applications anymore.

Creating traces with extrae remains a challenge. The CUDA implementations of the imager and correlator cannot be profiled out of the box, because extrae only supports applications that are written using the CUDA runtime API (which makes sense, as a vast majority of the CUDA applications are written using this runtime API). Our applications, in contrast, use the CUDA driver API, which is more low level, but gives more control over the GPU. In particular, we need the driver API to perform runtime compilation. As all observation parameters (e.g., the number of receivers, frequency channels and integration times) are constant for a particular observation, the GPU kernels are compiled at run time, so that these variable parameters in fact become constants. This does not only result in more efficient code, it also results in much more readable code, because in CUDA and OpenCL multi-dimensional arrays can only be declared when their dimensions are fixed. Indexing a fixed-sized multi-dimensional array as

```
samples[receiver][channel][time],
```

is much more readable than indexing it as a flat, variable-sized, single-dimensional array like

```
samples[(receiver*nr_channels + channel)*nr_times + time].
```

Instead of extrae, the NVIDIA Visual Profiler (or command-line profiler) could be used to create traces.

Profiling the OpenCL implementations of our applications with extrae should have been possible, but unfortunately it is not possible at this moment and would need some non-trivial fixes in extrae first. On AMD GPUs, we also found that profiling the OpenCL code with CodeXL is highly problematic; in fact, CodeXL 1.2 and 1.7 were the only versions that we

managed to get working, and these versions only work on older systems. Especially remote profiling, where the GPUs and the screen are not in the same machine, does not always work as it should do.

Figure 5: Timeline of the imager application. PCIe transfer rates of 12.5 GB/s are not sufficient to keep the GPU busy (top figure), while NVLINK transfer rates of 68 GB/s are (bottom figure).

Both the correlator and the imager are PCIe bandwidth bound when running on high-end GPUs that are connected by PCIe. The maximum throughput that can be achieved on PCIe-bases systems is normally 12-13 GB/s, but to keep a high-end GPU busy, both applications need 30-40 GB/s of input data (see Figure 5). This limitation can be overcome by using a system with NVLINK between the CPU and the GPU, like the systems on Juron.

There is no need to profile the applications for a long time, especially for the correlator, which shows highly jitter-free runtime behaviour: each integration time (typically about a second) the same operations are performed on equal-sized blocks with input data, so there is little variability. The only caveat is that GPUs need a few minutes to heat up, while a hot GPU typically runs at a slightly lower clock frequency than a cold GPU (due to leakage, a hot GPU needs more power, while the clock frequency is dynamically adjusted to prevent the GPU from drawing more power than it is allowed to do).

4.3 Application parameters

For the correlator, we normally simulate a large number of receivers (stations). A typical number is 576, the number of antennas in the AARTFAAC telescope. This version of the correlator is not optimised for “small” numbers of receivers (e.g., for LOFAR, different GPU kernels are used that are more optimised for up to 80 stations). A typical number of frequency channels that the PolyPhase Filter bank should create is 64, a typical number of time steps over which is integrated is 3072. Also, we assume dual polarized receivers (X and Y), and the correlations contain all polarization pairs (XX, XY, YX, YY). These numbers are representative for a real radio telescope, while the corresponding data structures fit in (GPU) memory. As all these numbers are the default values, there are no parameters that need to be changed when running the correlator application.

For the imager, the parameter space is close to infinite, and for a given observation it is virtually impossible to determine the most optimal parameter set a priori (for example, because there are techniques that trade memory usage for compute effort, and because some parameters change the balance between the computational costs for gridding and the costs for FFTs). How these parameters affect the imaging process requires a deep understanding of the algorithm, and is beyond the scope of this document. So to establish a performance baseline, we select a “reasonable” parameter set. All parameters can be changed by setting the environment variables below, prior to starting the application:

- `NR_STATIONS` 120
- `NR_CHANNELS` 16
- `NR_TIME` 8192
- `NR_TIMESLOTS` do not set
- `IMAGESIZE` do not set
- `GRIDSIZE` 8192
- `SUBGRIDSIZE` 32
- `KERNELSIZE` do not set
- `NR_CYCLES` 4

These numbers are already beyond what is needed for LOFAR observations (up to 80 stations). Eventually, we will scale these numbers to what is required for the Square Kilometre Array, (e.g., 512 stations and 100,000x100,000 grid sizes), but to scale to these sizes, the imager needs to use GPU page migration (unified memory) to dynamically map parts of the grid into GPU device memory. As support for unified memory is still experimental in the imager, we will not use this, but use the GPU-only imager (cuda-generic.x) for setting the performance baseline, with parameters chosen such that everything fits in GPU device memory. When benchmarking the application, the runtime can be increased by running multiple cycles that repeat the whole process of gridding, FFTing, and degridding. The execution times, FLOP counts, and possibly energy measurements can be averaged then.

5 Task 1.5: Space Weather (KU Leuven)

In this section we present the benchmark cases that are used for the applications of KU Leuven. These benchmarks are used to test the basic features of the codes, focusing in particular in memory use, performance and I/O.

For each application we propose four benchmarks. One of them represents a real-size operational run, and requires a longer runtime. This large benchmark cannot be used for profiling with optimisation tools: it is only used to gather the runtime of the code under realistic conditions.

Two other types of benchmarks have been included: micro-benchmarks and mid-size-benchmarks. The former feature run-times of the order of less than one minute while the later feature run-times up to 15 minutes. These can be used for quick node testing.

5.1 xPic

The following tables present the four benchmarks of the code xPic that will be performed all along the duration of the DEEP-EST project. They will be used to stress different parts of the DEEP-EST system and to check that new developments have a positive impact in the code performance. We defined four types of benchmarks:

1. A micro-benchmark for comparisons with iPic3D.
2. A large benchmark to check the code under realistic conditions.
3. A mid-size benchmark to test the code under realistic conditions and test I/O.
4. A micro-benchmark to test the effects of cache misses.

Benchmarks 3 and 4 feature two different input files to perform the comparisons.

Table 6 shows the benchmark 1 conditions using the input file: `test_02` (the names of the tests correspond to the names of the input files included in the code, and not to the order in this document). The objective of this benchmark is to test code performance on conditions similar to those used in the iPic3D code.

The number of blocks has been selected to fit the cache memory size. This value is architecture-specific and thus might need to be readjusted.

Table 6: xPic benchmark 1.

Parameter	Description	Observations [unit of measurement]
Name of the input	`test_02`	Production run that uses the work load typical of an iPic3D simulation. Used for comparisons with that code and for quick performance tests
Type of benchmark	Micro-benchmark	Test that can be run in a couple of minutes in a single core
Type of parallel efficiency	Weak scaling	

Objective	Check compute efficiency	

Metrics

Total runtime	Computed as the added time of the fields, particles, moment gathering and I/O runtimes	[sec] The estimated total runtime in one core for this test: ~ 120 sec = 2 minutes
Fields runtime		[sec]
Particles runtime		[sec]
Moment gathering runtime		[sec]
I/O runtime		[sec]

Additional metrics

Objective	Extracted only for major revisions of the code	These metrics are not gathered continuously. They are obtained each time a deliverable is due, to show more details on the performances of the code.
IPC	Instructions per cycle	[IPC] Obtained using Extrae/Paraver
FLOPS	Floating points per second	[FLOPS] Obtained using Vtune as part of the roofline analysis
Bytes/FLOP	Arithmetic intensity	[Bytes/FLOP] Obtained using Vtune as part of the roofline analysis

Simulation conditions

Type of run	iPic3D performance comparison	
Memory use	Blocks of size 920KB Total memory use 1.3 GB	Block sizes fit the L2-cache of Intel Skylake 8180 processor. Please change the number of blocks to fit the data in the cache of the targe t directory
Number of cells	16384	
Number of particles per cell per species	100	2 species
Number of blocks per MPI process	256	Selecting the number of blocks must follow the following formula: `(C/P)/B*n*s*f*r < cache size` where:

		`C` = Total number of cells `P` = Number of MPI processors `B` = Number of blocks per MPI process `n` = Number of particles per cell `s` = number of species `f` = 9 = number of large particle vectors `r` = size of double in the target architecture
# of iterations	10	
Numerical method	IMM	Using Maxwellian particle initialization. Drifting plasma with thermal velocity. Field solver tolerance of 1e-3
I/O frequency	No I/O	

Table 7 shows the benchmark 2 conditions using the input file: `test_03`. The objective of this benchmark is to gather the peak performances of the code under realistic production conditions, using large memory loads. The setup is similar to the previous test, but uses much more memory (more cells and many more particles per cell).

Table 7: xPic benchmark 2.

Parameter	**Description**	**Observations [unit of measurement]**
Name of the input	`test_03`	Production run that uses the work load typical of an xPic simulation. Used for peak performance tests
Type of benchmark	Benchmark	Test that can be run in the order of hours. Not inteded for fast benchmarking
Type of parallel efficiency	Weak scaling and strong scaling	This test can be used to test strong scaling in a single node. If the scaling involves changing the number of MPI processes, please make sure that the cache load is performed as suggested bellow
Objective	Check compute efficiency	
Metrics		
Total runtime	Computed as the added time of the fields, particles, moment gathering and I/O runtimes	[sec] The estimated total runtime in one core for this test: ~ 1800 sec = 30 minutes
Fields runtime		[sec]
Particles runtime		[sec]
Moment gathering		[sec]

runtime		
I/O runtime		[sec]

Additional metrics

None	Do not use external profilers for this test	We do not recommend to gather metrics from this run as it would produce extremely large files

Simulation conditions

Type of run	xPic performance test	
Memory use	Blocks of size 200KB Total memory use 16 GB	Block sizes fit the L3-cache of Intel Skylake 8180 processor. Please change the number of blocks to fit the data in the cache of the targe t directory
Number of cells	24576	
Number of particles per cell per species	1000	2 species
Number of blocks per MPI process	512	Selecting the number of blocks must follow the following formula: `(C/P)/B*n*s*f*r < cache size` where: `C` = Total number of cells `P` = Number of MPI processors `B` = Number of blocks per MPI process `n` = Number of particles per cell `s` = number of species `f` = 9 = number of large particle vectors `r` = size of double in the target architecture
# of iterations	10	
Numerical method	IMM	Using maxwellian particle initialization. Drifting plasma with thermal velocity. Field solver tolerance of 1e-3
I/O frequency	No I/O	

Table 8 shows the benchmark 3 conditions using the input files: `test_04` and `test_05`. These tests contain exactly the same parameters as the previous one, but with a much smaller domain, in order to make quick tests under realistic conditions. The number of iterations has been extended to present meaningful runtimes including I/O in `test_05`. These benchmarks can be used for quick checks of the code.

Please notice that test_05 contains the same features as test_04, but it also includes I/O.

Table 8: xPic benchmark 3.

Parameter	Description	Observations [unit of measurement]
Name of the input	test_04 test_05	Production run that imposes a realistic work load on the particle solver. Used for peak performance tests of the particle solver, including the I/O performances
Type of benchmark	Mid-size-benchmark	Test that can be run in a few minutes to 15 minutes
Type of parallel efficiency	Weak scaling	
Objective	Check compute efficiency and I/O performance	
Metrics		
Total runtime	Computed as the added time of the fields, particles, moment gathering and I/O runtimes	[sec] The estimated total runtime in one core for this test: ~ 180 sec = 3 minutes
Fields runtime		[sec]
Particles runtime		[sec]
Moment gathering runtime		[sec]
I/O runtime		[sec]
Additional metrics		
Objective	Extracted only for major revisions of the code.	These metrics are not gathered continuously. They are obtained each time a deliverable is due, to show more details on the performances of the code
IPC	Instructions per cycle	[IPC] Obtained using Extrae/Paraver
FLOPS	Floating points per second	[FLOPS] Obtained using Vtune as part of the roofline analysis
Bytes/FLOP	Arithmetic intensity	[Bytes/FLOP] Obtained using Vtune as part of the roofline analysis
Simulation conditions		

Type of run	Small size production run of xPic with realistic workloads	
Memory use	Blocks of size 7000KB Total memory use 0.5 GB	Block sizes fit the L3-cache of Intel Skylake 8180 processor. Please change the number of blocks to fit the data in the cache of the targe t directory
Number of cells	768	
Number of particles per cell per species	1000	2 species
Number of blocks per MPI process	16	Selecting the number of blocks must follow the following formula: `(C/P)/B*n*s*f*r < cache size` where: `C` = Total number of cells `P` = Number of MPI processors `B` = Number of blocks per MPI process `n` = Number of particles per cell `s` = number of species `f` = 9 = number of large particle vectors `r` = size of double in the target architecture
# of iterations	51	
Numerical method	IMM	Using maxwellian particle initialization. Drifting plasma with thermal velocity. Field solver tolerance of 1e-3
I/O frequency	`test_04` has no I/O `test_05` Writes the field files every 10 iterations, and the particle files every 25 iterations	1 particle file and 1 field file are written at iteration 0 A total of 6 field files and 3 particle files are written Field file size: 343 KB Particle file size: 71 MB

Table 9 shows the benchmark 4 conditions using the input files: `test_06` and `test_07`. The goal of this micro-benchmark is to test the effects of the block size in the performances of the code. For optimal performances, the block data should be as close as possible to the CPU. In Intel Skylake processors this means blocks sizes smaller than 256KB to fit in the L2 cache, or 39424KB to fit in the L3 cache.

Table 9: xPic benchmark 4.

Parameter	Description	Observations [unit of measurement]

Name of the input	`test_06` `test_07`	Files that test the efficiency improvement by using good cache sizes
Type of benchmark	Micro-benchmark	Test that can be run in a couple of minutes in a single core
Type of parallel efficiency	Weak scaling	
Objective	Check compute memory management efficiency	

Metrics

Total runtime	Computed as the added time of the fields, particles, moment gathering and I/O runtimes	[sec] Estimated total runtime in one core for this test: ~ 180 sec = 3 minutes
Fields runtime		[sec]
Particles runtime		[sec]
Moment gathering runtime		[sec]
I/O runtime		[sec]

Additional metrics

Objective	Extracted only for major revisions of the code	These metrics are not gathered continuously. They are obtained each time a deliverable is due, to show more details on the performance of the code
IPC	Instructions per cycle	[IPC] Obtained using Extrae/Paraver
FLOPS	Floating points per second	[FLOPS] Obtained using Vtune as part of the roofline analysis
Bytes/FLOP	Arithmetic intensity	[Bytes/FLOP] Obtained using Vtune as part of the roofline analysis

Simulation conditions

Type of run	Small size production run of xPic with realistic workloads	
Memory use	Blocks of size: * 630 MB (`test_06`) * 1 MB (`test_07`) Total memory use 4 GB	Block sizes fit the L3-cache of Intel Skylake 8180 processor. Please change the number of blocks to fit the data in the cache of the targe t directory

Number of cells	7680	
Number of particles per cell per species	1000	2 species
Number of blocks per MPI process	2 (`test_06`) 64 (`test_07`)	Selecting the number of blocks must follow the following formula: `(C/P)/B*n*s*f*r < cache size` where: `C` = Total number of cells `P` = Number of MPI processors `B` = Number of blocks per MPI process `n` = Number of particles per cell `s` = number of species `f` = 9 = number of large particle vectors `r` = size of double in the target architecture
# of iterations	10	
Numerical method	IMM	Using Maxwellian particle initialization. Drifting plasma with thermal velocity. Field solver tolerance of 1e-3
I/O frequency	None	This benchmark does not test I/O

5.2 DLMOS

The second application from KU Leuven is the DLMOS suite. By the end of the project, the DLMOS suite will be a Python package that includes all the models developed by KU Leuven for the modelling of the solar wind using deep learning algorithms. In the meantime, we have designed a test suite that will benchmark multiple features of the DEEP-EST system, including the performance of Python packages, TensorFlow efficiency and I/O efficiency.

In Table 10, we have detailed the conditions for the baseline benchmark 1. This test runs a Multi-Layer Perceptron (MLP) fully written in Python using only NumPy and Pandas and that will allow testing the performance of the Python environment without any additional deep learning framework.

Table 10: DLMOS benchmark 1.

Parameter	Description	Observations
Name of the function	`dlmos.dstmlp('test')`	Multi-Layer Perceptron written without a ML framework. Uses numpy and pandas
Type of benchmark	Mid-size-benchmark	Test that can be run in a few minutes to 15 minutes
Type of parallel efficiency	Strong and weak scaling	Uses data parallelism, distributing batches of data in different processors

Objective	Check python pure performance Check Data loading times	

Metrics

Total runtime	In seconds. Computed as the addition of the preprocessing, training and I/O run-times	Estimated total runtime in one core for this test: ~ 2 to 5 minutes
Preprocessing runtime	In seconds	
Training runtime	In seconds	
I/O runtime	In seconds	
Data throughput	In bytes/sec	Total input data size in bytes divided by the processing runtime
Time to convergence	In seconds	Time that the script takes to reach the value of 0.8 in the R^2 metric

Additional metrics

Objective	Extracted only for major revisions of the code	These metrics are not gathered continuously. They are obtained each time a deliverable is due, to show more details on the performances of the code.
FLOPS	Floating points per second	Obtained using VTune as part of the roof-line analysis
Bytes/FLOP	Arithmetic intensity	Obtained using VTune as part of the roof-line analysis

Simulation conditions

Type of run	Regression problem. Small size Multi-Layer Perceptron that forecasts the Dst geomagnetic index from solar wind information	
Memory use	280 MB	When all input data is loaded in memory
Number of inputs	52000	Estimated number of inputs after data preprocessing
Number of nodes in the input layer	40	
Number of layers	3	Including the output layer

Nodes per layer	20 x 10 x 1	Including the output layer
Number of epochs	200	
I/O	Size of the data sets: 16 MB Size of the NN weights: 13 KB	

The “dstmlp” case uses data from OMNIWeb containing information about the solar wind conditions at 1 AU, and geomagnetic indexes on the surface of the Earth. The DLMOS model forecasts the geomagnetic index Dst from measurements of the conditions of the solar wind. The data must be first read from a file (or downloaded from the Internet), pre-processed and organised. Then the model trains a basic multilayer neural network that has only one output value.

The second benchmark presented in Table 11 has the exact same parameters as the previous benchmark, but uses two deep learning frameworks: TensorFlow (tf) and Keras (kr). The I/O conditions are kept the same in each case. This benchmark allows comparing the performance of the frameworks relative to the base run of benchmark 1.

Table 11: DLMOS benchmark 2.

Parameter	**Description**	**Observations**
Name of the function	`dlmos.dsttf('test')` `dlmos.dstkr('test')`	Multi-Layer Perceptron written without a ML framework. Uses numpy and pandas
Type of benchmark	Mid-size-benchmark	Test that can be run in a few minutes to 15 minutes
Type of parallel efficiency	Strong and Weak scaling	Uses data parallelism, distributing batches of data in different processors
Objective	Check TensorFlow/Keras performance. Check data throughput times	
Metrics		
Total runtime	In seconds. Computed as the addition of the preprocessing, training and I/O run-times	Estimated total runtime in one core for this test: ~ 2 minutes
Preprocessing runtime	In seconds	
Training runtime	In seconds	
I/O runtime	In seconds	
Data throughput	In bytes/sec	Total input data size in bytes divided by the processing runtime
Time to	In seconds	Time that the script takes to reach the value

convergence		of 0.8 in the R^2 metric
Additional metrics		
Objective	Extracted only for major revisions of the code	These metrics are not gathered continuously. They are obtained each time a deliverable is due, to show more details on the performances of the code.
FLOPS	Floating points per second	Obtained using VTune as part of the roof-line analysis
Bytes/FLOP	Arithmetic intensity	Obtained using VTune as part of the roof-line analysis
Simulation conditions		
Type of run	Regression problem. Small size Multi-Layer Perceptron that forecasts the Dst geomagnetic index from solar wind information	
Memory use	280 MB	When all input data is loaded in memory. (Calculated using memory_profiler)
Number of inputs	52000	Estimated number of inputs after data preprocessing
Number of nodes in the input layer	40	
Number of layers	3	Including the output layer
Nodes per layer	20 x 10 x 1	Including the output layer
Number of epochs	200	
I/O	Size of the data sets: 16 MB Size of the NN weights: 13 KB	

Table 12 below shows the conditions for benchmark 3. This benchmark is the same as benchmark 1 (using only Python, Pandas and NumPy) but uses many more input nodes and more hidden layers. The input/output raw data is also the same, but here we use a large number of historic points to make the forecasting prediction. The preprocessing generates a much larger input set. This emulates the workload of a larger neural network, with a much bigger weights matrix.

Table 12: DLMOS benchmark 3.

Parameter	Description	Observations
Name of the	`dlmos.dstmlp('bigtest')`	Multi-Layer Perceptron written without a ML

function		framework. Uses numpy and pandas
Type of benchmark	Mid-size-benchmark	Test that can be run in a few minutes to 15 minutes
Type of parallel efficiency	Strong scaling	Uses data parallelism, distributing batches of data in different processors
Objective	Check python pure performance. Check Data loading times	
Metrics		
Total runtime	In seconds. Computed as the addition of the preprocessing, training and I/O run-times	Estimated total runtime in one core for this test: ~ 4 minutes
Preprocessing runtime	In seconds	
Training runtime	In seconds	
I/O runtime	In seconds	
Data throughput	In bytes/sec	Total input data size in bytes divided by the processing runtime
Time to convergence	In seconds	Time that the script takes to reach the value of 0.8 in the R^2 metric
Additional metrics		
Objective	Extracted only for major revisions of the code.	These metrics are not gathered continuously. They are obtained each time a deliverable is due, to show more details on the performances of the code.
FLOPS	Floating points per second	Obtained using VTune as part of the roof-line analysis
Bytes/FLOP	Arithmetic intensity	Obtained using VTune as part of the roof-line analysis
Simulation conditions		
Type of run	Regression problem. Small size Multi-Layer Perceptron that forecasts the Dst geomagnetic index from solar wind information	
Memory use	600 MB	When all input data is loaded in memory

Number of inputs	52000	Approximate number of inputs after data preprocessing
Number of nodes in the input layer	192	8 features x 24 historical values
Number of layers	5	Including the output layer
Nodes per layer	100 x 40 x 20 x 10 x 1	Including the output layer
Number of epochs	2005	
I/O	Size of the data sets: 80 MB Size of the NN weights: 194 KB	

Table 13 shows the setup for the large benchmark case 4. This will be used to test CNN script under more operational conditions. This benchmark performs a CNN training using images from the SDO satellite as input, and data from the OMNIWeb database. The image resolution used here is only 512x512, but it helps to test the data retrial and loading procedures.

Table 13: DLMOS benchmark 4.

Parameter	Description	Observations
Name of the function	`dlmos.swtf('test')`	Simple CNN that uses SDO data to forecast solar wind speed at 1AU
Type of benchmark	Benchmark	Test that can be run in the order of hours.
Type of parallel efficiency	Strong scaling	Uses data parallelism, distributing batches of data in different processors
Objective	Check CNN performance in TensorFlow. Check Data loading times. Check memory performance	
Metrics		
Total runtime	In seconds. Computed as the addition of the preprocessing, training and I/O run-times	Estimated total runtime in one core for this test. Currently unknown. Estimated: 30-60 minutes
Preprocessing runtime	In seconds	
Training runtime	In seconds	
I/O runtime	In seconds	
Data throughput	In bytes/sec	Total input data size in bytes divided by the processing runtime

Additional metrics

None		

Simulation conditions

Type of run	Deep CNN with multiple images as input and one value, solar wind speed, as output	
Memory use	~ 3 GB	With input/output data loaded in batches of 16
Number of inputs	260 000	One year of data. Images taken every 2 minutes
Number of nodes in the input layer	7.8 million	Image size: 512 x 512 pixels. 30 channels per input. Batches of size 16 inputs
Number of layers	3 convolutional, and 2 fully connected	Including the output layer. Dropout, down-sampling, flattening functions are not included
Number of channels or nodes per layer	8, 12, 24 (channels) + 128, 8 (nodes)	Channels for the convolutional layers and nodes for the fully connected nodes, including the output layer
Number of epochs	1	We use batch training for this case with only one passage through all the inputs
I/O	Size of the data sets: 600 GB Size of the NN weights: 600 KB	

This full operational benchmark case is also used to test some of the features of TensorFlow and/or Keras used for CNN computation. It will also test the scripts for data movement and will stress the system memory.

Some of the parameters of this benchmark may be adjusted in the following deliverables, as the script is not yet available and will require adaptations to the selected architecture (in particular disk and memory space). If disk conditions do not allow performing the benchmark, we will first reduce the total number of inputs by half and double the number of epochs, until the disk can be used. If the limitation comes from memory constraints, we will reduce the number of channels in the input layer. Any modifications to the conditions presented in Table 13 will be reported.

6 Task 1.6: Data analytics in Earth Science (UoI)

The University of Iceland´s contribution consists of data analytics within the realm of Earth sciences, with three machine learning applications demonstrating different approaches within the field, effectively covering a wide range of the techniques employed in contemporary machine learning, namely:

- Clustering using HPDBSCAN (6.1).
- Support vector machines with PiSvM (6.2).
- Deep learning with TensorFlow and the Keras extension (6.3).

6.1 HPDBSCAN

HPDBSCAN is a highly parallel implementation of the established DBSCAN clustering algorithm. It takes points of an arbitrary dimension as one of its input argument and labels each point either as part of a cluster ID or identifies it as noise. The DBSCAN algorithm determines this through the use of input parameters: the minimal number of points required to form a cluster and the maximum neighbourhood search radius. HPDBSCAN has been developed by researchers from the University of Iceland and the Jülich Supercomputing Centre.

6.1.1 Implementation

The HPDBSCAN implementation is written in C++ and is optimised for high parallelism, using both shared and distributed memory parallelism via OpenMP and MPI respectively. Further parallelism is also employed by exploiting the HDF5 API when the input data is read and the output data is written. The application scales well and is able to practically operate on any number of nodes, cores and threads.

The source code is split into two folders, jsc_mpi, which contains the hybrid MPI/OpenMP code relevant to our use-case and the DEEP-EST project, and also jsc_openmp, a OpenMP-only variant without HDF5 outside the scope of the project.

The application can be compiled by using the Makefile with the deep option, i.e. “make deep” in the jsc_mpi folder. More information is available in the Readme file located in the application´s Gitlab repository[21] .

6.1.2 Binary

HPDBSCAN’s executable, *dbscan*, has three required input parameters:

- *minpoints*, the minimum amount of points required to form a cluster,
- *epsilon*, the maximum neighbourhood search radius for each point,
- *filename*, a file reference to the input dataset.

Note that simply executing the binary without parameters will display concise information on all of the input parameters that are possible (but are not needed). Finding the parameter values that give the best clustering results for a given dataset can be a time-consuming process because they do depend on the dataset itself and are application specific.

[21] https://gitlab.version.fz-juelich.de/DEEP-EST/UoI-HPDBSCAN

6.1.3 I/O

Input files for HPDBSCAN must adhere to the HDF5 format. The HDF5 API is used by each MPI process to access the input data in parallel, where the data is divided into equally sized chunks corresponding to the number of MPI processes, thus enabling each MPI process to read just the chunk it is going to process. The application labels each point to a cluster ID and writes the labels for each point into the input file near the end of its execution. Additionally, the application writes the execution time of each part of the application and its status to the standard output, as can be seen in the image below. Note that the input parameters have been approximated to give a good “clusters” vs. “noise points” ratio, i.e. it is desirable to keep the noise at a minimum without giving up too much cluster granularity.

```
Calculating Cell Space...
        Computing Dimensions... [OK] in 0.057296
        Computing Cells...      [OK] in 0.175638
        Sorting Points...       [OK] in 1.540659
        Distributing Points...  [OK] in 8.107850
DBSCAN...
        Local Scan...           [OK] in 69.104443
        Merging Neighbors...    [OK] in 0.356783
        Adjust Labels ...       [OK] in 0.087154
        Rec. Init. Order ...    [OK] in 16.265062
        Writing File ...        [OK] in 25.732853
Result...
        9105    Clusters
        81234457        Cluster Points
        164353  Noise Points
        81067186        Core Points
Took: 156.043796s
-bash-4.2$
```

Figure 6: HPDBSCAN´s standard output, using the Bremen dataset running on the SDV partition with 128 cores. Intermediary time-measurements are provided for each relevant part of the application, as well as the whole execution.

6.1.4 Input datasets

HPDBSCAN has been used to perform clustering on numerous datasets, including ‘Inner city of Bremen’: a digital 3D cartography of the city of Bremen, Germany, using point-clouds generated with 3D laser scans. This dataset demonstrates how HPDBSCAN can be used to identify real-world structures in three dimensions, see Figure 7 and Figure 8.

The “Inner city of Bremen” dataset (including a smaller, sub-sampled dataset) is available online[22]. Note that the online dataset link contains another dataset with a collection of 2D Twitter geo-tags in the United Kingdom. However, that dataset was deemed unsuitable for the DEEP-EST project due to the low number of dimensions and detail.

Using a *minpoint* input value of 20 and an *epsilon* value of 30 gives an adequate result for the Bremen dataset, i.e. a good number of clusters with little noise.

As the DEEP-EST project progresses, we will eventually need even larger datasets to fully take advantage of the new and better performing hardware. For this, we have several promising prospects such as the LiDAR point-cloud AHN (Actueel Hoogtemodel Nederland) dataset of the whole Netherlands, collected and maintained by the Dutch government. A collaboration with TU Delft in this context has been started.

[22] https://b2share.eudat.eu/records/7f0c22ba9a5a44ca83cdf4fb304ce44e

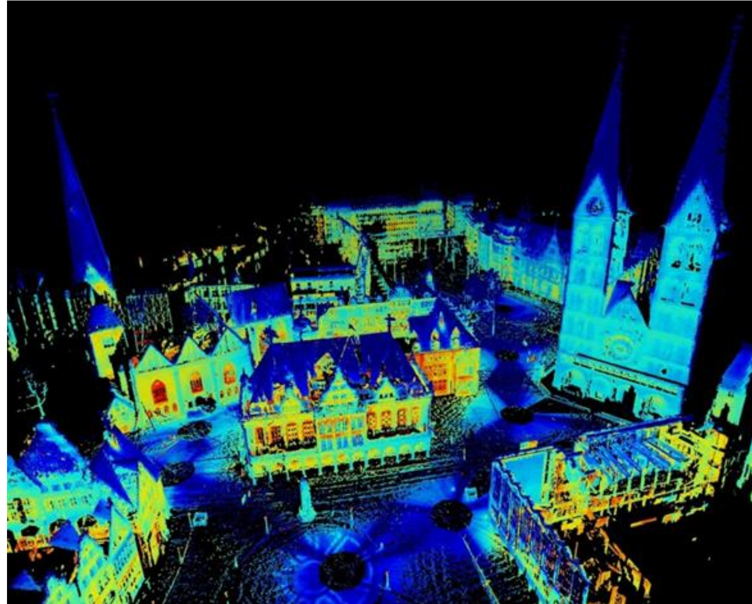


Figure 7: Visualisation of the input data for the Inner city of Bremen dataset. Colours represent temperature values where blue is cold and red is warm (temperature is ignored for clustering).

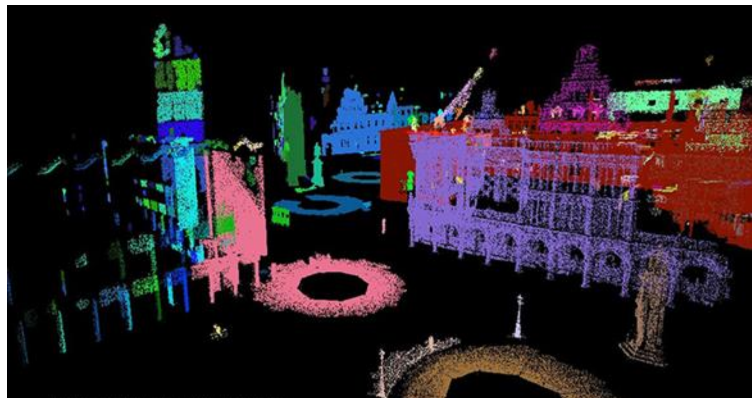


Figure 8: A visualisation of the generated output (using a different perspective); each colour represents a single cluster.

6.1.5 Proposed benchmarks on DEEP system

If we assume that HPDBSCAN is executed with reasonable input parameters, i.e. parameters that produce suitable clusters with excluding outliers as noise, the most important benchmarking metric is the total execution time. Other important metrics are how well the application scales *strongly* and *weakly*. We therefore propose benchmarking the application with two datasets and according different hardware configurations.

For the first benchmark we investigate weak scaling, i.e. how well it scales when you increase the problem size. For this we use the sub-sampled Bremen dataset on a low number of cores to obtain a “yard-stick” time measurement which can be compared against the same measurement made using the full dataset. One possible hardware configuration on the DEEP-EST system would be to select the SDV partition with two nodes and two threads per process, using the sub-sampled Bremen dataset. This is then followed with using the full Bremen dataset, which is approximately 32 times larger than its sub-sampled counterpart and should therefore use 32 times the amount of cores, e.g. 8 nodes, 8 MPI processes, and 16 threads per process.

For the second benchmark we investigate strong scaling, i.e. how it scales with a varying number hardware resources using a fixed problem size. This can for example be realised on the DEEP-EST by executing the application on the SDV partition with an exponentially growing number of cores.

All sample DEEP-EST micro-benchmarks JUBE scripts are available in the application´s Gitlab repository.

6.2 piSVM

PiSvM is a parallel support vector machine (SVM) implementation which we use to classify hyper-spectral data of natural and man-made land covers via supervised learning. First, the input data is pre-processed with feature engineering before the trained models are produced using different kernels or datasets, performing cross-validation when necessary. Finally, the models' classification accuracies and error-rates are determined by performing predictions on a separate dataset.

6.2.1 Implementation

The PiSvM implementation is written in C++ and uses MPI for distributed memory parallelism. It is not a highly optimised implementation with observable performance bottlenecks in both its MPI collective communication pattern and I/O routines. Furthermore, PiSvM does not use shared memory parallelism such as OpenMP, and thus only relies on MPI. However, it is able to operate with an arbitrary number of processes. While PiSvM is the best known parallel SVM implementation, it needs to be optimised as part of DEEP-EST and will thus serve as an example how to make a non-optimised application ready for the DEEP-EST hardware.

PiSvM currently requires a C/C++ compiler and MPI. In the future, the application will also need a HDF5 module capable of parallel I/O. It is possible to compile and run all PiSvM executables with the default modules.

The source code[23] can be built by simply executing the Makefile, via the `make` command, in the project's source folder, which will produce three binary executables.

6.2.2 Binary

The three executable binaries of PiSvM are:

1. `pisvm-train` to train the model,
2. `pisvm-predict` for inference,
3. `pisvm-scale` which is outside the scope of this document.

As it can be seen in Figure 9, there are numerous options which can be set when running the pisvm-train executable:

```
srun pisvm-train -D -o 1024 -q 512 -c 100 -g 8 -t 2 -m 1024 -s
0 $TRAINDATA
```

The prediction executable, pisvm-predict, however, only requires the model file generated by training and a test dataset which is mutually exclusive to the training dataset, i.e.

[23] https://gitlab.version.fz-juelich.de/DEEP-EST/UoI-piSVM

```
srun pisvm-predict $TESTDATA $MODELDATA $RESULTS
```

```
-bash-4.2$ ./pisvm-train
Usage: svm-train [options] training_set_file [model_file]
options:
-s svm_type : set type of SVM (default 0)
        0 -- C-SVC
        1 -- nu-SVC
        2 -- one-class SVM
        3 -- epsilon-SVR
        4 -- nu-SVR
-t kernel_type : set type of kernel function (default 2)
        0 -- linear: u'*v
        1 -- polynomial: (gamma*u'*v + coef0)^degree
        2 -- radial basis function: exp(-gamma*|u-v|^2)
        3 -- sigmoid: tanh(gamma*u'*v + coef0)
-d degree : set degree in kernel function (default 3)
-g gamma : set gamma in kernel function (default 1/k)
-r coef0 : set coef0 in kernel function (default 0)
-c cost : set the parameter C of C-SVC, epsilon-SVR, and nu-SVR (default 1)
-n nu : set the parameter nu of nu-SVC, one-class SVM, and nu-SVR (default 0.5)
-p epsilon : set the epsilon in loss function of epsilon-SVR (default 0.1)
-m cachesize : set cache memory size in MB (default 40)
-e epsilon : set tolerance of termination criterion (default 0.001)
-h shrinking: whether to use the shrinking heuristics, 0 or 1 (default 1)
-b probability_estimates: whether to train a SVC or SVR model for probability estimates, 0 or 1 (default
-wi weight: set the parameter C of class i to weight*C, for C-SVC (default 1)
-v n: n-fold cross validation mode
-o n: max. size of working set
-q n: max. number of new variables entering working set
flags:
-D: Assume the feature vectors are dense (default: sparse)
```

Figure 9: Command line options available when running the svm-train executable.

6.2.3 I/O

Currently, the input datasets for PiSvM are simple text files with space-separated values. PiSvM's input-output procedures are not optimised, allowing future work for improvement. In particular, an updated version is planned with improved I/O that enables parallel input data processing by every MPI process, using the HDF5 API and MPI parallel I/O.

The I/O procedure can be described in two steps, in the correct order:

1. Training uses the optimised Indian Pines dataset (see Section 6.2.4) as input and generates a model file with the trained model. Verbose information relevant to a data-scientist using the application is streamed to the standard output.
2. Prediction uses the model file generated in step 1 above and another larger test dataset for inference. A simple text file is generated with the predicted labels for each respective test sample and the models accuracy is streamed to the standard output.

```
Accuracy = 97.633% (681341/697859) (classification)
Mean squared error = 0.279428 (regression)
Squared correlation coefficient = 0.955522 (regression)
```

Figure 10: Successful classification evaluation output obtained with pisvm-predict based on the partial example of the output produced by the training phase.

6.2.4 Input dataset

Our use case processes the "Indian Pines" dataset[24] gathered by the AVIRIS sensor on board of an aircraft over a test site in North-western Indiana in the US. It is made up of

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

145x145 pixels and 224 spectral reflectance bands of labelled data. It mostly consists of agriculture although it also includes forests, roads and some large structures.

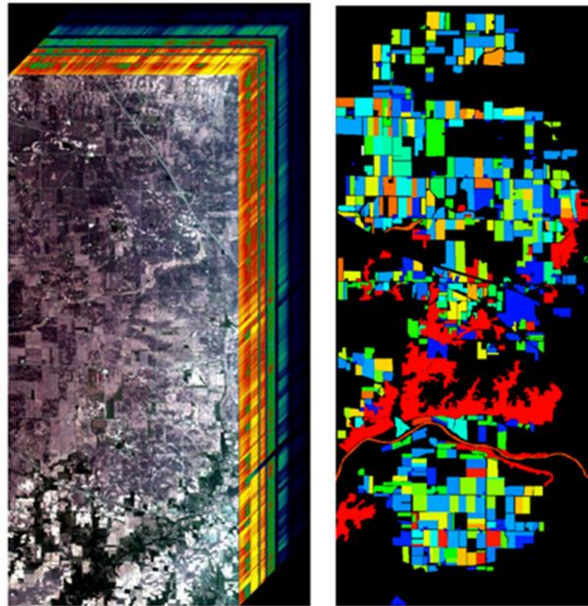


Figure 11: Visualisation of the Indian Pines dataset with the hyper-spectral layers on the left side, and its categorisation (labelling) on the right.

Note that the link contains two versions of the dataset, an unmodified raw dataset and its optimised version which has undergone feature engineering in order to reduce the number of features and ensure convergence during training.

The Indian Pines dataset is of medium size which is not large enough to put a considerable strain on the DEEP-EST hardware, therefore new, larger datasets are currently being investigated to give the opportunity to make use of the full power of the future DEEP-EST hardware. These datasets include several high-resolution national LiDAR datasets compromising mostly of terrain, one such example being the national LiDAR dataset of the Netherlands maintained by the Dutch government.

6.2.5 Proposed benchmarks on the DEEP system

The most important metrics to be measured on the DEEP system is the execution time of the training and prediction phase, respectively, and the measured accuracy of a trained model that is reported after the prediction has finished its execution. Previous tracing and profiling has revealed that the application's load-balancing is sensitive to hyper-threading; therefore, we recommend not using it.

For the training, we recommend the processed Indian Pines training dataset, which is pre-processed in order to ensure convergence and speed-up the execution by reducing the number of features. The number of cores and benchmarks should be selected as to measure how well the application scales strongly, i.e. benchmark with an increasing number of cores while the problem size remains fixed.

For the prediction, we recommend using the same number of cores as in the training phase above to be able to directly compare the execution times of these two phases and how strongly they scale. For inference the Indian Pines test dataset is used, which is significantly larger than the training dataset.

Note that sample JUBE scripts for the requested DEEP-EST micro-benchmarks are available in the application´s Gitlab repository.

6.2.6 Future Improvements

There are several scheduled improvements to the current PiSvM implementation that will be explored throughout the lifespan of the DEEP-EST project:

- The standard serialised I/O routines will be replaced with parallel HDF5 API calls, which let MPI processes to access the input dataset concurrently.
- The training phase will be improved through the use of cascade support vector machines, which offers a significant speed-up of training time.
- Tracing and profiling via Extrae and Paraver has revealed several performance bottlenecks which can be significantly improved with a smarter implementation of distributed memory parallelism.

DEEP-EST modules will be used to speed-up the application´s execution, and improve its workflow, by using the Network Access Memory (NAM) module as a storage target, the Global Collective Engine (GCE) to enhance MPI collective operations, and the Booster module for inference.

6.3 Deep learning with TensorFlow and Keras extension

TensorFlow is an open-source deep learning framework that is used to create traditional Artificial Neural Networks (ANN) and more recent deep learning networks (e.g. Long-Short Term Memory or Convolutional Neural Networks (CNN)) for all kinds of applications. The core of TensorFlow is implemented in C++ but it offers several APIs on top of the core, with Python being the most popular variant. It is common to run TensorFlow on accelerators such as GPUs due to the intrinsic parallelisation possibilities.

Keras is a user-friendly TensorFlow extension library that allows using high-level Python code to create machine learning applications, based on TensorFlow, using a layer-wise approach and an easy way to add/remove regularisation methods (e.g. weight decay, etc.).

We use the same machine learning application as described in Section 6.2 (except we add 6 more classes compared to the original SVM problem because that 6 classes had too little samples for PiSvM to learn them), i.e. the classification of hyper-spectral data of natural and man-made land covers via supervised learning. However, instead of a support vector machine (SVM), we use a deep 3D convolutional neural network (3DCNN). While the spatial resolution is 2D, the hyper-spectral dimension of the data adds a further dimension, thus requiring a 3D convolutional neural network for an effective classification. Using a deep neural network has the advantage of enabling the use of the unmodified raw dataset as input, instead of requiring the use of a pre-processed, feature engineered dataset as above.

6.3.1 Implementation

The current implementation of our application uses TensorFlow/Keras and runs most efficiently on GPUs. To make the most of the Intel CPUs offered by the DEEP/DEEP-ER SDV, we tried to make use of the speed-up provided by the Intel Math Kernel Library (MKL) to exploit acceleration offered by Intel CPUs. Unfortunately, the 3D convolution used by our

application is not supported by Intel MKL[25] yet. While we were just finalising this deliverable, the Intel Math Kernel Library for Deep Neural Networks MKL-DNN started to support Intel AVX-512-accelerated 3D convolution[26]. That library was not available on time but improvements on using this library are expected to appear in the next coming weeks. Another interesting path in the implementation would be the use of specialised Neural Network accelerator hardware such as the Intel Nervana Neural Network Processor (NNP) as outlined in the Deliverable D3.1 - System Architecture.

In order to start with a bottom-up approach, we use the GPU-centric JURON cluster at Jülich Supercomputing Centre (JSC) as a realistic reference platform for the time being while considering other options in the near future. Once acceleration on other hardware is available on the future DEEP-EST MSA platform, we will investigate the acceleration offered by that new hardware (and the accompanying accelerated software library) for our deep neural network application. The deep learning application for classification of remote sensing data makes use of Keras as library on top of TensorFlow, i.e. the application itself is implemented in Python but runs as CUDA code using multiple GPUs in parallel on the same node, but currently not on multiple nodes (which would require MPI communication between the different nodes that are equipped with GPUs OpenMPI).
The performance analysis of GPU applications is still a matter of active research and thus using the Extrae tool to create traces of our GPU executions is not possible: Extrae supports only tracing GPU applications that make use of the CUDA runtime API, but TensorFlow uses the low-level CUDA driver API.

After having loaded the needed modules (`module load`) and Python modules (`pip install`), the Python interpreter can be used to run the Keras code of the application.

6.3.2 I/O

The raw input dataset is pre-processed to fit the needs of the deep learning application, i.e. out of the original text based file format, several HDF5 files are produced to enable fast I/O via the h5py Python module. This includes:

- The training samples and their corresponding labels,
- The test samples and their corresponding labels,
- The 2D pixel coordinate of every sample,
- Statistical information, i.e. dataset standard deviation and mean.

The training/testing datasets are created with a randomized 90/10 % split of the raw dataset.

Similarly, the output model file is written in the HDF5 format, containing the weights of all of the directed edges in the deep neural network. Additional output is written to text files with information on the neural network´s setup and also internal information that facilitates debugging when necessary. Additionally, the application takes advantage of the check-pointing mechanism (hooks) offered by TensorFlow.

[25] https://github.com/tensorflow/tensorflow/issues/11802
[26] https://github.com/intel/mkl-dnn/commit/1a4d45d90a3889a08a857d5896f61e23be5a50b4

6.3.3 Input datasets

We use the same dataset as described in Section 6.2.4, i.e. the "Indian Pines" dataset[27] made up of 1417x614 pixels and 224 spectral reflectance bands of labelled data except 6 more classes that have been removed in the dataset used with PiSvM in section 6.2. It mostly consists of agriculture although it also includes forests, roads and some large structures. Due to the deep learning approach, we use from that dataset the unmodified raw, non-feature engineered data and the whole set of classes.

As the deep neural network-based classification is more compute intensive then SVM-based classification, this data is probably big enough for the final DEEP-EST MSA platform, but we will nonetheless also explore larger datasets, i.e. the same datasets as we´ll be using for the SVM-based classification described in Section 6.2.4. This also opens new possibilities of investigating the scalability of the deep learning application. In particular, it is planned to use time series classification of remote sensing data (e.g. change of land cover over time vs. single image classification) that adds another complexity to the application problem. This will be with a temporal resolution of 5 days with data sizes from 23 TB / day[28] recorded since June 2015.

6.3.4 Proposed Benchmarks on the DEEP system

We use the same benchmarks as those in Section 6.2, i.e. the execution time of the training and prediction phase, respectively, and the measured accuracy of a trained model that is reported after the prediction has finished its execution. As described in Section 6.3.1, we are not using the DEEP system for this deliverable, but rather the JURON GPU cluster as it is the most efficient system that we have access to for our application.

- Execution time: The benchmark of our application on JURON makes use of a single node equipped with four NVIDIA TESLA P100 GPUs optimised for HPC: the connection between the GPUs and to the host CPU is via NVLink interconnects (NVLink offers a 160 GB/s bandwidth in comparison to 32 GB/s offered by PCIe x16).
  The benchmark results are as follows (training and test data obtained from the Indian pines data set by dividing it in a random way using a 90% / 10% split):
  - Training time: 2 hours, 5 minutes and 39 seconds,
  - Testing time: 16 minutes and 57 seconds.
- Accuracy: The nice property of deep learning is that it learns properties of the features from the raw data. Hence, our deep learning application can work on the raw data, but still achieves an accuracy that is comparable to applying SVMs on the pre-processed, feature-engineered data set. Apart from the speed-up of training the networks, the intrinsic feature engineering thus significantly reduces time of scientists to perform manual feature engineering.

In the near future, we will use the most suitable neural network accelerator technology available in the DEEP-EST MSA. Examples include GPUs with dedicated tensor cores or even specialised neural network hardware accelerators.

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

[28] https://sentinel.esa.int/web/sentinel/missions/sentinel-2

6.3.5 Future improvements

We foresee the following future changes to improve the performance of our deep learning application:

- Making our deep learning application runnable on the future DEEP-EST MSA hardware platform whether this offers CPU-based acceleration, GPU-based acceleration, or specialised Neural Network accelerator hardware such as the Intel Nervana Neural Network Processor (NNP).
- Currently, only a single GPU node (with 4 GPUs) is used. Whatever accelerator technology is used in the future DEEP-EST MSA, using MPI will enable further parallelisation across nodes.

7 Task 1.7: High Energy Physics (CERN)

7.1 Application description

The CMS Experiment is one of the four large experiments at the Large Hadron Collider. The enormous amounts of collision data recorded need to be processed and analysed as efficiently as possible. Optimisation of the throughput is of utter importance for the experiment as it allows to utilize vast compute resources better and to probe new physics in a shorter amount of time.

The overall collection of software used by the CMS Experiment for data analysis, referred to as CMSSW, is built around a Framework, an Event Data Model (EDM), and Services needed by the simulation, calibration and alignment, and reconstruction modules that process event data so that physicists can perform analysis. The primary goal of the Framework and EDM is to facilitate the development and deployment of reconstruction and analysis software.

The CMSSW event processing model consists of one executable, called cmsRun, and many plug-in modules which are managed by the Framework. All the code needed in the event processing (calibration, reconstruction algorithms, etc.) is contained in the modules. The same executable is used for both detector and simulation data.

The CMSSW executable, cmsRun, is configured at run time by the user's job-specific configuration file. This file tells cmsRun:

- which data to use,
- which modules to execute,
- which parameter settings to use for each module,
- what is the order of the execution of modules, called path,
- how the events are filtered within each path, and
- how the paths are connected to the output files.

The CMS Event Data Model (EDM) is centred around the concept of an Event. An Event is a C++ object container for all raw and reconstructed data related to a particular collision. During processing, data are passed from one module to the next via the Event, and are accessed only through the Event. All objects in the Event may be individually or collectively stored in ROOT files, and are thus directly browsable in ROOT. This allows tests to be run on individual modules in isolation. Auxiliary information needed to process an Event is called Event Setup, and is accessed via the Event Setup object.

One of the main building blocks of CMSSW and HEP Data Analysis is the ROOT framework. ROOT is a platform for big data analytics, which provides primitives for serialization, storage, histogramming, drawing and graphics functionality. ROOT framework is written entirely in C++ and has Python bindings for more flexible user interface.

7.2 Physics description

The main idea behind any High Energy Physics (HEP) data analysis is the process of comparing and testing of the recorded collision events (real data) against some known physics models (simulation) that describe the interactions of elementary particles (e.g. the Standard Model of Physics). The purpose is to establish (or exclude) the validity of a particular model by comparing distributions of various reconstructed physics quantities/observables.

Both, data and simulation, have almost identical sequences of steps to arrive at a point of comparison. However, where real collision events are recorded and are directly available for reconstruction, simulated events have to be generated and passed through a very detailed simulation of the detector geometry first. It is worth mentioning that once event generation and simulation steps are completed, the reconstruction sequences that run on previously generated events involve the same algorithms (with different conditions) employed in the reconstruction of recorded collision events. The CMSSW workflows are:

- **GEN**. GEN is the workflow that involves the generation of various elementary particle interactions and simulation of the proton-proton collision environment. This is the very first step of any simulation chain. The output of GEN consists of EDM events (with generator level products).
- **SIM**. SIM is the workflow that involves simulation of detector and electronics response. Generator level products produced in the previous step (GEN) are passed through simulated detector geometry components and the induced detector response is simulated. Various electronics effects are approximated as well. This step is trying to simulate the response of the detector given the collision environment modelled by the physics included in the GEN step.
- **RAW2DIGI**. RAW2DIGI is the workflow responsible for unpacking/converting RAW data representation into a more friendly analysis form. Within the CMS experiment, DIGI is a nomenclature for the digitized products, collections of electronics outputs (charge) for all of the detector regions.
- **RECO**. RECO is the workflow with a collection of algorithms that reconstruct the physical content of the event. Physicists performing analysis would like to manipulate high level familiar objects such as muons, electrons, photons, jets, etc. This dramatically simplifies the process of data analysis by abstracting away the details of various detector components and concentrating on the actual physics of interest.

There is quite a large variety of algorithms employed in this step: from simple regression procedures to complex clusterisation and pattern recognition techniques. Therefore, with the increased complexity of HL LHC collision events, the percentage of time spent on reconstruction will become larger than the one spent on the GEN-SIM step, which has traditionally been more time consuming.

7.3 Application infrastructure and configuration

This is the current infrastructure:

- **CMSSW**. The CMS Experiment software framework, CMSSW, is a software distribution of a substantial size with multiple internal components and various external dependencies. For example, for the purpose of I/O (Input/Output) CMS utilizes ROOT data analysis framework and for detector modelling and response simulation the GEANT toolkit is currently used.
- **CernVM-FS**. In order to facilitate the distribution of the software stack, CMS Experiment uses CernVM-FS, a service which simplifies the distribution of software components. CernVM-FS assists LHC experiments and collaborations in deploying their applications across the world-wide computing infrastructure. CERNVM-FS usually contains prebuilt binary distributions together with the source code itself (mainly for x86_64 architecture, but there are also ARM and Power PC binary releases). Since it is foreseen that a given source code needs to be recompiled either

for different architectures or with different optimisation flags, a given release can be checked out into some local area and rebuilt.

- **Squid**. For the purpose of storing information about experiment conditions and meta-data about a recorded or simulated event (e.g. various detector calibrations, weights to be used for inference, detector channels that are not working for a given time period), CMS employs an ORACLE database, which is physically located on the servers at CERN. However, due to the fact that the actual data processing happens on the distributed world-wide infrastructure and fetching conditions metadata substantially increases the startup time of the application, a frontier distributed database caching system is used. It is mainly based on the Squid caching service and allows reducing substantially the startup time of the application, which is critical for successful profiling of a given configuration.
- **File system**. Another important point to clarify is the I/O (Input/Output) and data locality. For the purpose of simplifying the benchmarking and profiling, input data files will be located locally on the cluster, either on General Parallel File System (GPFS) or on BeeGFS file system. It is also worth noting that CMSSW is equipped with a capability of reading across the network via the XRootD protocol. However, this will not be used in order to facilitate the setup.
- **Parallelisation**. Given that CMSSW is a multithreading application, but not a multiprocessing one, there is no MPI communication among the cmsRun processes (the application is embarrassingly parallel). The configuration that is currently being used by the CMS experiment is to run with eight threads per single cmsRun process.

7.4 Benchmarking metrics

Throughput is the benchmarking metric. The majority of the High Energy Physics (HEP) analyses require a large number of collision events to be simulated and/or processed in order to be able to perform proper statistical analysis. Various hypothesis testing techniques usually require substantial amount of statistics to be accumulated so the results of such tests can be properly interpreted. We define the throughput to be the number of simulated and/or processed collision events per unit of time (e.g. second). Optimisation and thorough profiling of this metric is of ultimate importance, especially in the view of the High Luminosity LHC (the upgrade of the LHC is foreseen to start in 2024 and will have a factor 2.5 in peak instantaneous luminosity together with the increased High Level Trigger rates), where the collision environment is going to explode algorithm runtime on current infrastructure.

7.5 Benchmarking configuration

A set of well-defined workflows has been selected that follow as much as possible the standard CMS data processing pipelines. Both event generation with simulation and reconstruction will be benchmarked, however the emphasis will be on the reconstruction algorithms. For the simulation workflows, no input data files are required as collision events will be simulated along the way.

The choice of the datasets selected for benchmarking is dictated by the knowledge of the physics present in these events and the relation of these datasets to each other. For example, for the reconstruction of simulated events, we know that TTBar events have richer content and therefore it will take more time to simulate and reconstruct them. In fact, the runtime associated with the reconstruction of TTBar events can be considered an upper

bound and used as a measure to judge how much compute resources we need to process other kinds of events.

- **Monte Carlo generation, simulation, reconstruction**. For the event generation and simulation, we selected two types of events: MinBias and TTBar. As already mentioned, TTBar is considered as the upper bound on the runtime for any type of workload. Therefore, by profiling the performance of it, we automatically infer the robustness of other workflows. In its current configuration, the LHC collides proton beams every 25ns and usually multiple collisions happen per bunch crossing (per 25ns). As a consequence, one needs to reconstruct an event that comes from several proton-proton collisions. This effect is known as in-time Pile Up (PU). For the purpose of simulating the PU environment, MinBias events are used because they represent the most statistically frequent background events (known physics interactions are not of primary interest for researches).
- **Collision data reconstruction**. CMS Datasets recorded during the 2017 production campaign will be used as they reflect the most upgraded detector so far. A certain number of events need to be considered in order to guarantee the use of enough computing nodes for the sake of benchmarking. The collision datasets to be run with the reconstruction benchmark are:
  1. `/ZeroBias/Run2017F-v1/RAW`,
  2. `/JetHT/Run2017F-v1/RAW`,
  3. `/SingleEl/Run2017F-v1/RAW`.

7.6 Benchmarking results

A continuous profiling and benchmarking of CMSSW applications across different kinds of architectures and optimisation levels requires the same scientific care that physicists use to discover new physics. Results obtained by scanning through the phase space of possible optimisations will point towards a better and more efficient configuration that will yield to higher throughput and lower computational cost. Provided below is a description of several single node test benchmarks that will serve as examples of what kind of results and numbers are expected to be measured.

- **GEN-SIM results on AMD**. A set of benchmarks similar to the ones to be measured have been tested on a single machine equipped with 32 cores (2x for hyper-threading) AMD Opteron 6376 processor with 128 GB of main memory. For the GEN-SIM sequence, results are summarised in the following table:

Table 14: GEN-SIM results on AMD.

Gen Type	Processes	Threads	Ev. per Thread	Ev. per process	Total Events	Total Throughput	Sum Max RSS	Sum Max VMM
TTbar	32	1	250	250	8000	0.570 ev/sec	25,361 Mb	31,560 Mb
TTbar	64	1	200	200	12800	0.881 ev/sec	48,568 Mb	78,274 Mb
TTbar	16	4	200	800	12800	0.855 ev/sec	14,334 Mb	27,998 Mb

TTbar	8	8	200	1600	12800	0.861 ev/sec	10,243 Mb	19,888 Mb
MinBias	64	1	1000	1000	64000	4.63 ev/sec	45,343 Mb	76,290 Mb
MinBias	16	4	1000	4000	64000	4.03 ev/sec	14,622 Mb	28,833 Mb
MinBias	8	8	1000	8000	64000	4.01 ev/sec	7,563 Mb	20,650 Mb

- **RECO results on AMD**. The previous AMD node has also been employed to run a similar set of GEN-SIM benchmarks. They measure the runtime of the reconstruction workflows of TTBar events mixed with MinBias in order to accommodate the Pile UP effects. Results are summarised in the following table:

Table 15: RECO results on AMD.

Process	**Processes**	**Threads**	**Ev. per Thread**	**Ev. per Process**	**Total Events**	**Total Throughput**	**Sum Max RSS**	**Sum Max VMM**
Reco	42	1	100	100	4200	1.430 ev/sec	108,723 Mb	163,543 Mb
Reco	16	4	100	400	6400	1.693 ev/sec	76,440 Mb	121,598 Mb
Reco	8	8	100	800	6400	1.474 ev/sec	59,417 Mb	97,910 Mb
Reco	48	1	800	800	51200	1.556 ev/sec	133,349 Mb	205,392 Mb
Reco	32	1	800	800	25600	1.203 ev/sec	91,832 Mb	136,149 Mb
Reco	32	2	800	1600	51200	1.868 ev/sec	114,233 Mb	192,650 Mb
Reco	16	4	800	3200	51200	1.839 ev/sec	82,974 Mb	149,697 Mb
Reco	8	8	800	6400	51200	1.805 ev/sec	67,431 Mb	128,296 Mb

8 Global conclusion

The objective of this document is to present a series of benchmarks to track the performance gains undergone by the applications as they adapt to the Modular Supercomputing Architecture built within the DEEP-EST project. This conclusion gives a brief summary of the most important aspects regarding benchmarking.

Overall, it is clear that the most important metric to be measured during benchmarking is the time to solution or runtime, which is expressed in time units (e.g. seconds). Some applications like GROMACS measure the time step execution instead. In the case of CERN, it is the number of processed collisions per unit of time (throughput). All these concepts are ultimately related to the speed of the application. In the particular case of ASTRON, it is equally important to measure the performance (in TFLOPS) and energy efficiency (GFLOP/J) of their applications due to a constrained energy budget. KU Leuven also proposes to measure memory and I/O in addition to performance. Finally, machine and deep learning applications must be also benchmarked against model accuracy and, when relying on external packages or deep learning frameworks, one needs to measure the influence of these as well.

Each application proposes different benchmarks that are especially adapted to its own requirements within the DEEP-EST project (cf. D1.1[1]):

- NMBU proposes an *in-situ* analysis-simulation framework where NEST, Arbor and Elephant run simultaneously in the MSA. NEST simulation provides relevant data to be analysed by the other two applications. Two benchmarks are proposed for NEST: a standard HPC benchmark based on a system of two neuronal populations as well as a simplified version of it whose objective is to stress crucial parts of the system such as process synchronisation, communication and random writes. Arbor and Elephant are benchmarked in a slightly different manner because they are based on purely synthetic input data. In the case of Arbor, three synthetic benchmarks are proposed, being the last one the most computationally intensive. Elephant is a Python-based library and the four proposed benchmarks are intended to measure the efficiency of cross-correlations using various Python implementations (e.g. Numpy to evaluate convolution functions) and the performance of the ASSET algorithm.
- NCSA benchmarks aim at exploring the performance of Gromacs on various computer architectures (DEEP-SDV and KNL), with a hybrid parallel programming model (OpenMP and MPI) in the presence or not of hardware optimisations like vector instruction sets. The three proposed benchmarks increase in complexity both in terms of number of atoms and problem size. While the latency of the network is the limiting scalability factor for the smallest benchmark, the increase in MPI communications associated with the larger benchmark starts degrading scalability, which requires finding a balance between the number of MPI ranks per node and the number of OpenMP threads per MPI rank.
- The correlator and imager applications of ASTRON have the particularity of being able to benchmark themselves, by generating synthetic input on the fly, and have been ported recently to FPGAs and made compatible with CUDA tensor cores since the beginning of the DEEP-EST project. It is clear that ASTRON applications largely benefit from GPUs or FPGAs performance over traditional CPUs, which justifies its

choice of using a GPU-based system as reference baseline to conduct their benchmarks.

- KU Leuven proposes four benchmarks for xPic and another four for DLMOS. In the case of xPic, one benchmark is used for comparison between xPic and iPic3D utilised in the DEEP-ER project. Other two benchmarks aim at testing xPic under realistic conditions, i.e. using large memory loads, with and without taking into account I/O operations. A final micro-benchmark is proposed to highlight the impact of cache misses. The DLMOS benchmarks are oriented towards the efficiency of Python packages (similarly to what is proposed for Elephant) as well as TensorFlow and Keras frameworks. It is worth noting that benchmarking deep learning applications with such frameworks might require additional tuning, depending on the architecture, in order to avoid constraints due to memory and/or disk usage.
- University of Iceland proposes two different benchmarks for its machine learning application HPDBSCAN. Each benchmark uses a different input size and a different number of hardware resources allowing them to be compared against each other in order to study the weak scalability of the application. As already discussed in deliverable D1.1[1], the piSVM application is purely based on MPI. Moreover, parallel load imbalance was detected in piSVM during the training on Extrae and Paraver that took place at BSC in November 2017 (piSVM's imbalance is expected to be addressed in the near future). The two proposed piSVM benchmarks for the training and inference phases, respectively, are designed to evaluate the performance of the application over time with the integration of new features. Finally, further work needs to be done for UoI's deep learning approach based on TensorFlow and Keras frameworks since there is not support for 3D convolution hardware-accelerated operations within Intel's MKL library.
- CERN is focused on two particular benchmarks intended to measure the throughput of high energy physics experiments, which is expected to sky rocket in 2024 after the Large Hadron Collider upgrade. The first benchmark is concentrated on the generation, simulation and reconstruction phases characteristic of high energy physics experiments while the second one focuses on the data reconstruction phase. Three collision data sets from an early experiment of 2017 are provided to guarantee that the systems are stressed during benchmarking.

8.1 Next steps

Next steps include the cooperation with colleagues from WP2 to ensure that benchmarking will be carried out smoothly during the project. Application's tracing will be done in parallel with benchmarks, e.g. using Extrae. Finally, support for 3D convolution operations with TensorFlow using Intel's MKL library is currently being discussed with colleagues at Intel.

Furthermore, micro-benchmarking of the applications is currently being carried out in the AMD EPYC Naples 2S system by Intel. JUELICH and BSC will start benchmarking the applications against the ARM Cavium Thunder X2 architecture in order to determine the best architecture choice for the MSA, which will be presented in the future deliverable D3.2 at project month 12.

The results obtained from benchmarking and tracing of the different applications (WP2) will help each application partner to develop its own strategy to use the DEEP-EST prototype in the most efficient manner. This will be presented in deliverable D1.3 at project month 12.

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

ASTRON: Netherlands Institute for Radio Astronomy, Netherlands

B

BN: Booster Node (functional entity)

BoP: Board of Partners for the DEEP-EST Project

BSC: Barcelona Supercomputing Centre, Spain

BSCW: Repository used in the DEEP-EST Project to share all project documentation.

C

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

CN: Cluster Node (functional entity)

CNN: Convolutional Neural Networks

CPU: Central Processing Unit

D

DAM: Data Analytics Module: with nodes (DN) based on general-purpose processors, a huge amount of (non-volatile) memory per core, and support for the specific requirements of data-intensive applications

DDG: Design and Developer Group of the DEEP-EST Project

DEEP: Dynamical Exascale Entry Platform (project FP7-ICT-287530)

DEEP-ER: DEEP - Extended Reach (project FP7-ICT-610476)

DEEP-EST: DEEP - Extreme Scale Technologies

Dimemas: Performance analysis tool developed by BSC

DIMM: Dual In-line Memory Module

DSP: Digital Signal Processor

DN: Nodes of the DAM

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

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

Extrae: Performance analysis tool developed by BSC

F

FFT: Fast Fourier Transform

FMA: Fused Multiply Add; an operation of the form A * B + C

FP7: European Commission 7th Framework Programme

FPGA: Field-Programmable Gate Array, Integrated circuit to be configured by the customer or designer after manufacturing

G

GCE: Global Collective Engine, a computing device for collective operations

GFLOP/S: Gigaflop, 10^9 Floating point operations per second

GFLOPS/W: Giga ($10^{\wedge 9}$) Floating-Point Operations per Second per Watt, or alternatively: Giga Floating-Point Operations per Joule (GFLOPS/J)

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

HBM: High Bandwidth Memory

HDL: Hardware Description Language

HPC: High Performance Computing

HPDBSCAN: A clustering code used by UoI in the field of Earth Science

HW: Hardware

I

Intel: Intel Germany GmbH, Feldkirchen, Germany

I/O: Input/Output. May describe the respective logical function of a computer system or a certain physical instantiation

J

JUELICH: Forschungszentrum Jülich GmbH, Jülich, Germany

JURON: JUelich and NeuRON supercomputer created by IBM and NVIDIA featuring Telsa graphics and POWER8 processors with the NVLink technology

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

LLNL: Lawrence Livermore National Laboratory

LOFAR: Low-Frequency Array, an instrument for performing radio astronomy built by ASTRON

M

MPI: Message Passing Interface, API specification typically used in parallel programs that allows processes to communicate with one another by sending and receiving messages

MSA: Modular Supercomputer Architecture

N

NAM: Network Attached Memory

NCSA: National Centre for Supercomputing Applications, Bulgaria

NEST: Widely-used, publically available simulation software for spiking neural network models developed by NMBU.

NMBU: Norwegian University of Life Sciences, Norway

NN: Neural Network

NUMA: Non-Uniform Memory Access

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

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

P

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.

piSVM: Parallel classification algorithm

PMT: Project Management Team of the DEEP-EST Project

R

RAM: Random-Access Memory

S

SCR: Scalable Checkpoint/Restart. A library from LLNL

SDV: Software Development Vehicle: HW systems to develop software in the time frame where the DEEP-EST Prototype is not yet available.

SIMD: Single Instruction Multiple Data

SIONlib: Parallel I/O library developed by Forschungszentrum Jülich

SKA: Square Kilometre Array

SSSM: Scalable Storage Service Module

SVML: The Short Vector Math Library

SW: Software

T

TCP: Transmission Control Protocol; a reliable, stream-based network protocol

TFLOP/S: 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

UoI: Háskóli Íslands – University of Iceland, Iceland

W

WP: Work package

X

xPic Programming code developed by the KULeuven to simulate space weather