H2020-FETHPC-01-2016

DEEP-EST

DEEP Extreme Scale Technologies

Grant Agreement Number: 754304

D1.1

Application co-design input

Final

Version: 1.0

Author(s): A. Kreuzer (JUELICH), 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), M. Riedel (UoI), M. Girone (CERN), V. Khristenko (CERN)

Date: 31.10.2017

Project and Deliverable Information Sheet

DEEP-EST Project	**Project Ref. №:** 754304	
	Project Title: DEEP Extreme Scale Technologies	
	Project Web Site: http://www.deep-projects.eu	
	Deliverable ID: D1.1	
	Deliverable Nature: Report	
	Deliverable Level: PU *	**Contractual Date of Delivery:** 31 / October / 2017
		Actual Date of Delivery: 31 / October / 2017
	EC Project Officer: Juan Pelegrín	

* - 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

Document	**Title:** Application co-design input	
	ID: D1.1	
	Version: 1.0	**Status:** Final
	Available at: http://www.deep-projects.eu	
	Software Tool: Microsoft Word	
	File(s): DEEP-EST_D1.1_Application_co-design_input_v1.0	
Authorship	**Written by:**	A. Kreuzer (JUELICH), P. Martínez (BSC)
	Contributors:	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), M. Riedel (UoI), M. Girone (CERN), V. Khristenko (CERN)
	Reviewed by:	P. Niessen (ParTec), I. Schmitz (ParTec)
	Approved by:	BoP/PMT

Document Status Sheet

Version	Date	Status	Comments
1.0	31/October/2017	Final version	EC submission

Document Keywords

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

Table of Contents

List of Figures

List of Tables

Executive Summary

The main goal of the Applications Work Package (WP1) in DEEP Extreme Scale Technologies (DEEP-EST) is to assess the Modular Supercomputing Architecture (MSA) developed in the project and to evaluate the DEEP-EST Prototype. Therefore six applications from a wide range of scientific fields were chosen. The applications all are very different. These will show that the new architecture is beneficial for not only one specific kind of application, but for several ones in different ways.

This deliverable will mark the first step towards this goal. The document collects the requirements of the different applications. Firstly the Design and Development Group (DDG) of DEEP-EST created a questionnaire that covers all important topics, where each software and hardware Work Package provided questions to find out what the application needs are. Secondly there was a F2F meeting at the end of September gathering all the application developers and the people responsible for the software and hardware in the project. Additional questions were answered and clarified during the meeting.

The results of the questionnaire and F2F discussions will help to make design decisions for the DEEP-EST Prototype and the software stack.

1 Introduction

The Cluster/Booster architecture introduced in the Dynamical Exascale Entry Platform (DEEP) Project was improved in the DEEP - Extended Reach (DEEP-ER) Project and will now be extended to a Modular Supercomputing Architecture (MSA) in the DEEP-EST Project. More information on the DEEP Projects can be found here [1]. The first step towards this is a co-design discussion about the requirements (hardware and software) of all the applications within the project.

This document gives a short introduction to each of the applications and summarises all the requirements that are important for the DDG of the project to make design decisions for the DEEP-EST Prototype. For this each application is reported following this structure:

- Presentation of the application structure
- Requirements of the application

The “Application structure” subsection describes the application. It is explained what the application is used for, what a typical workflow is and how it is initially planned to distribute it over the whole system. Of course this distribution strategy can change after a detailed application analysis. Planned changes will then be described in the following deliverables.

The second subsection “Application requirements” gives an overview on all the gathered information from the answers to the applications questionnaire. All important topics for hardware as well as software aspects are covered here so that the DDG should get a clearer picture on what is needed.

The document continues with a global conclusion which reflects what the requirements that all applications have in common are. Note that these requirements may differ or even converse.

2 Task 1.2: Neuroscience (Task leader: NMBU)

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.

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 ~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 [2].

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 $N = M \cdot T$ virtual processes. For a fixed number `N` 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.

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 provides realistic estimates of memory requirements based on a small number of parameters [2], [3]. 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 [4], [5], with PyNEST being the default interface.

2.1 Application structure

2.1.1 Network simulations in NEST

NEST represents a neuronal network as a directed graph. For currently relevant use cases, this graph has between 10^5 and 10^9 neurons[1], 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.

2.1.1.1 Network structure

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 [6]. A single update step usually requires O(10) additions and multiplications. More complex neurons described by systems of non-linear ODEs are currently integrated using solvers from the 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 (a) permits the connection construction in parallel on all VPs in most cases and (b) minimizes the amount of information that needs to be exchanged between processes; see [7] for the initial pure MPI implementation and [8] for the current hybrid MPI-thread implementation.

The current data structures used to represent neurons and connections are based on systematic memory modelling and consequent optimisation as described in [2] and [3]. For recent results on network construction and sensitivity to memory allocation issues for large numbers of threads, see [9].

2.1.1.2 Network dynamics

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.

[1] 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.

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 [7] 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; see [2], [9]), (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 (see [10]).

2.1.2 LFP predictions integrating NEST, Arbor and HybridLFPy

Local field potentials (LFP) are valuable measures of the activity of neuronal populations, especially of input to neuron populations. LFPs are typically recorded using an array of electrodes inserted into the brain, low-pass filtering the signals. Significant progress in theory and simulation allows us today to make detailed, biophysically correct predictions of LFP signals in brain models [11]. This proceeds in a three-stage process: Full network activity is simulated using NEST, and resulting spike trains are written to file. The spike trains are then read by the HybridLFPy package [12], which maps spikes generated by point neuron models in NEST onto spatially detailed (compartmental) neuron models represented in other simulators, such as Neuron. Neuron simulations of uncoupled neurons then generate detailed, spatially resolved information about ionic currents into and out of neurons. Based on these current traces, Python code in the HybridLFPy package generates LFP predictions solving electrostatic equations and writes the results to file.

In DEEP-EST, this will be converted into a single workflow, where spikes will be streamed directly from NEST point-neuron simulations into compartmental neuron simulations; instead of Neuron, the Arbor simulation package [13] will be used which is currently under development in a collaboration between the SimLab Neuroscience at Jülich Supercomputing Centre (JSC) and CSCS, the Swiss Supercomputing Centre, as part of their activities in the Human Brain Project. Arbor has a modern architecture based on Intel Thread Building Blocks and is thus expected to be much more suited to the DEEP-EST Extreme Scale Booster (ESB) architecture than Neuron.

In this setup, NEST will run on the HPC Cluster Module (CM), while Arbor and HybridLFPy will run on the ESB.

Spike transfer from NEST to Arbor/HybridLFPy will use the MPI-based MUSIC library [14]. As long as Arbor/HybridLFPy can consume all spikes provided by NEST simulations, there are no synchronization requirements between NEST on the CM and Arbor/HybridLFPy on the ESB. Bandwidth requirements will be quite limited, since spikes are transferred in a compact address-event representation.

2.1.3 In-situ data analysis integrating NEST and Elephant

Understanding the dynamics and function of large neuronal networks requires an analysis of their activity, which is represented by spike trains, i.e., sequences of pulses with which neurons communicate. Statistical analysis of large numbers of spike trains obtained from many neurons in a network is thus essential to interpreting simulation results. The Elephant package [15] is a standard toolkit for such analysis developed by Forschungszentrum Jülich (FZJ) in collaboration with research partners in the Human Brain Project and elsewhere. Several parts of Elephant have been parallelized in recent years and further parallelisation efforts are ongoing.

Initial focus in using Elephant on DEEP-EST will be in situ data correlation analysis of up to 1000 spike trains recorded in parallel using sliding windows, and the ASSET technique for the analysis of sequences of synchronous events in massively parallel spike trains [16]. The technique has recently been optimised for Intel Broadwell and KNL architectures by scientists at FZJ.

In this case, NEST will run on the CM, Elephant on the Data Analytics Module (DAM) and communication will use MUSIC. Latency and bandwidth requirements will be limited.

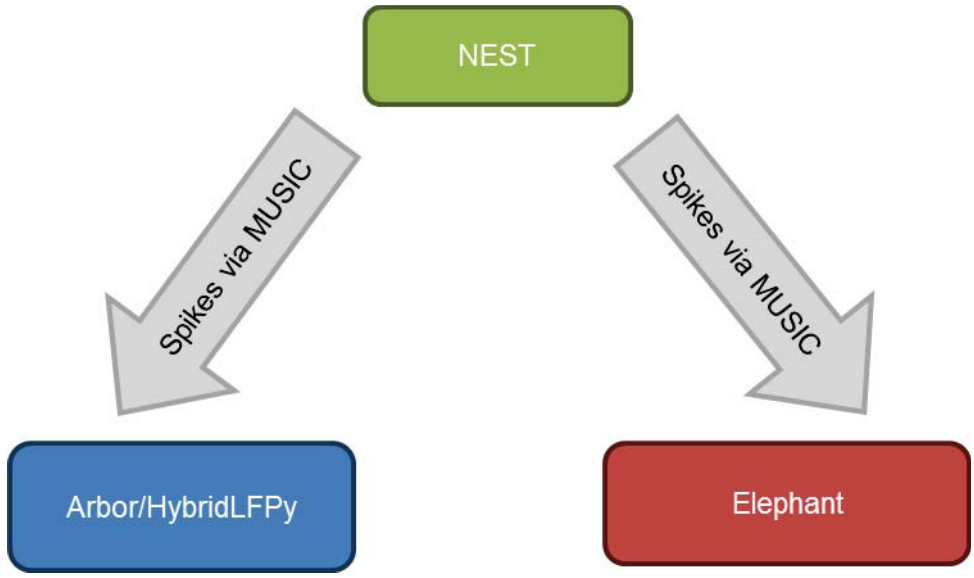


Figure 1: Workflow of NEST with Arbor/HybridLFPy (left path) and NEST with Elephant (right path) (NMBU).

2.2 Application requirements

2.2.1 Use case description

2.2.1.1 NEST Standard Benchmark

A standard benchmark has been used to evaluate NEST performance in several recent publications [2], [9]. This benchmark simulates a network model with two neuronal populations (80/20 split) in which each neuron has an in-degree of `K=11250`, i.e., it receives input from 11250 other neurons (may be modified); 64% of connections are plastic (weight updated during simulation), while the remainder are static. The network size can be scaled freely from `10·K = 112500` neurons upwards; for smaller network sizes, `K` needs to be

reduced). Average neuron firing rates are typically 5–10 Hz. Minimum delay in the network is 1.5 ms or 15 time steps, i.e., communication between MPI processes is required only every 15 time steps. For details see [9]. The benchmark is included in NEST as `examples/nest/hpc_benchmark.sli`.

This use case is mainly used to compare performance across architectures and software generations. It uses only the CM.

2.2.1.2 Potjans-Diesmann Network Model

This is a widely used simplified model a $1mm^3$ of cerebral cortex [17]. It differs from the standard benchmark in the following ways:

- Eight instead of two neuronal populations, approximately 90.000 neurons in total.
- Variable in- and out-degrees, average 3750.
- Delays are randomized with minimum delay 0.1 ms, requiring MPI communication after every time step.
- All connections are static.

It is the simplest use case included here that is directly scientifically relevant in computational neuroscience.

This use case forms part of the HybridLFPy and Elephant use cases described below, but is also relevant for its own sake. In that case, it uses only the CM.

2.2.1.3 Multiarea Network Model

This is an extension of the Potjans-Diesmann model to include 32 brain areas of the visual system [18]. Each individual area is represented by a modified version of the Potjans-Diesmann model, differing in neuron numbers and connection probabilities. Key model properties are:

- 4.13×10^6 neurons.
- 2.42×10^{10} connections, all static.
- Average in-/out-degree 5.960.
- Average spike output rate per neuron 14.6 spikes/s.
- Minimal delay/MPI communication interval 0.1 ms simulated time.
- Simulation length: 100 s simulated time.

So far, this model has been simulated on the JUQUEEN supercomputer using 1024 MPI processes of 64 threads each for a total of 65536 virtual processes [18]. The memory model for the NEST simulator [2] indicates that six SDV cluster nodes provide sufficient memory (RAM) to store and simulate this model. To achieve acceptable simulation speeds, though, the entire SDV cluster will be required.

This use case forms part of the HybridLFPy and Elephant use cases described below, but is also relevant for its own sake. In that case, it uses only the CM.

2.2.1.4 HybridLFPy use case

HybridLPFy [12] computes mesoscopic electrical brain signals, so called local field potentials (LFPs) based on the network dynamics simulated using NEST. Specifically, spike trains generated by neurons in a NEST simulation using highly connected point neurons are fed into detailed models of unconnected neurons simulated using Arbor to compute the electrical currents passing through the cell membrane at different locations. From these currents, HybridLFPy then computes the LFP at different locations in a piece of brain tissue using electrostatic principles. Currently, HybridLFPy is run only after a NEST simulation is complete. This use case focuses on computing LFP signals in situ, while the network simulation is running. To this end, the Neuron simulator currently used by HybridLFPy to simulate electrical currents will be replaced through neuronal membranes with Arbor, a software currently under development at JSC and CSCS and optimised for modern hardware architecture.

HybridLFPy will either be driven by spikes from the entire Potjans-Diesmann model or a small number of areas from the multiarea model. Spikes will be communicated from NEST to HybridLFPy/Arbor using the MPI-based MUSIC library, with NEST running on the CM and HybridLFPy/Arbor on the ESB.

2.2.1.5 Elephant use case

Elephant is a widely used package for statistical analysis of spiking neuronal network activity, such as rate and cross-correlation estimates and the detection of repeating patterns in neuronal activity. Currently, such analysis is performed after a simulation has completed. In this use case, the focus will be on in situ analysis, in particular sliding-window cross-correlation analysis and pattern detection using the ASSET algorithm [16].

Elephant will be fed spikes from selected populations of the Potjans-Diesmann model or multiarea model using the MUSIC library. In this use case, NEST will be running on the CM, while Elephant will be executed on the DAM.

2.2.2 Benchmarking metrics

From a user perspective, the most essential metric is the application turnaround time, since work in neuronal network simulation entails a high proportion of exploratory simulations. Besides queuing time, which largely is beyond the user's control, simulation execution time is therefore essential.

It is assumed that overall runtimes for all use cases will be determined by the speed of the NEST simulation. The runtime for a NEST simulation has two distinct components: the network construction time and the network simulation time (see [9], for details). For short exploratory simulations, the network construction time can be significant, for long-running simulations such as long multiarea model simulations (100 s simulated time), network construction is less relevant.

It is also required that simulations are numerically accurate within the limits set by IEEE754 floating point arithmetic (double precision).

2.2.3 Scalability

NEST shows almost linear strong scaling using MPI and OpenMP parallelisation, up to large numbers of processes. An important scaling for NEST is maximum-filling scaling [2]: For each number of MPI ranks and OpenMP threads, simulate the largest network that can be represented in the available memory. This is similar to weak scaling, but problem sizes grow slower than linear due to administrative overhead. Maximum-filling scaling is particularly relevant for systems with relatively little memory per CPU core. Close to linear maximum-filling scaling up to the full size of the K and JUQUEEN supercomputers [2] was observed.

Arbor is parallelised using Intel Thread Building Blocks, showing strong scaling up to 1024 nodes (one MPI rank per node for inter-node communications and 64 threads on JUQUEEN and 256 threads on JULIA) and somewhat slower scaling beyond this point.

2.2.4 Modularity

Experience so far indicates that NEST simulations achieve best performance when run on powerful general-purpose CPUs, i.e., the DEEP-EST CM. Arbor is designed to benefit from modern high-core-count architectures and will therefore be executed on the ESB, while data analysis with Elephant will be performed on the DAM.

2.2.5 Communication

NEST divides the network to be simulated into a fixed number of MPI ranks, each with the same, fixed number of OpenMP threads. Threads and MPI ranks interact only at well-defined synchronisation points, when they exchange spikes. Ranks currently exchange data using `MPI_Allgather()`, but a transition to `MPI_Alltoall()` is under development. The amount of data to be exchanged is limited: The multiarea model generates in total approximately 6000 spikes per communication interval, each represented by a single 64-bit integer, resulting in a total data volume well below 100 KByte per communication. Each rank contributes approximately equally, i.e., approximately 6 KByte per rank when running on 16 ranks on the SDV cluster.

2.2.6 Compute

NEST mainly performs arithmetic operations on double precision floats, plus pointer and 64-bit integer arithmetic, including modulo. Especially for modelling synaptic plasticity, fast computations of the exponential function are required (individual, not vectorised; also `expm1()`).

Current experience indicates that the most crucial performance bottleneck for most NEST simulations is spike delivery: This requires the random traversal of large hierarchical data structures up to several GByte in size, followed by update (read, add and write back) into neurons' input buffer data structures.

For Arbor, the heavy computational performance is in solving the segment-coupling ODEs for neurons, with updates of spike events that have been channelled to the accelerators. This is a very sparse matrix, coupling each segment to 1 (or a few) up and downstream segments. The neurons themselves are grouped for parallel computation: thousands of neurons have the same computation on their own matrices applied simultaneously, making the problem accelerator friendly. The heavy memory access portion on the CPU is searching the global

spike buffer for events that are associated with each group of neurons to be updated, and then creating a data structure to be transferred to the accelerators. This search is an iterated binary search to find “chunks” of relevant spikes, which will reduce in the worst case to a single contiguous spike followed by a binary search forward from that point.

Arbor makes use of Thread Building Blocks, CUDA, KNL directives, and AVX, and overlaps GPU with CPU operations. All memory exchanges with the accelerators are done explicitly. However, the large spike buffer is handled on the CPU to be organised into events to be sent to computational units on the accelerator. This goes into global RAM, which may end up being accessed by any core for the event wrangling step. So Arbor would benefit from shared memory.

2.2.7 Memory

NEST simulations typically use a large part of the memory available on current supercomputers with relatively little memory per compute core. In order to achieve sufficiently fast simulation times on memory-rich machines such as the SDV cluster nodes (Intel Xeon E5-2680v3 processors), less than the available memory might be used. Running the full multiarea model, which can fit into the memory of just 6 SDV cluster nodes across all 16 nodes would use only about half the memory available on each node, while exploiting the compute power of all cores.

While Arbor mostly accesses data contiguously and through reduction operations, NEST has highly unstructured memory access patterns during spike delivery which vary over time as they depend on the stochastic activity of the neurons. The probability distribution of the access patterns over time will be roughly stable, although there can be variations on shorter time scales. Therefore memory latency is significantly more important for NEST than memory bandwidth.

For some use cases, especially those involving synaptic plasticity, it would be useful to be able to take a snapshot of an entire NEST simulation so that one could re-start it at a later time. Due to the lack of checkpointing in NEST, this would require operating level support for copying an entire simulation to non-volatile memory (NVM), from which the simulation could later be restored. Such restoration would also have to restore relations between MPI processes in parallel simulations. The amount of NVM required for this should be a few times the memory required for the actual simulation, so that one could store at least one, preferably multiple snapshots.

2.2.8 I/O

NEST and Arbor read compact specifications from files and provide output as files. Currently, all output is in form of text files using the POSIX interface. For highly parallel simulations, NEST has internal mechanisms to concentrate file output to a small number of MPI processes. Fully parallel output to binary format based on SIONlib is currently under development and will be integrated into NEST and Arbor in the near future.

The total amount of output data even from large, long-running NEST simulations is limited: A simulation of 100 seconds of activity of the full multiarea model, saving every spike fired, would generate less than 100 GByte of data.

2.2.9 Elasticity

Both NEST and Arbor can be executed on different numbers of cores. The number of parallel processes can be chosen at startup for both applications, but is fixed afterwards (mouldable, not malleable). NEST requires that all MPI processes have the same number of OpenMP threads and has a design guarantee that any split between MPI processes and threads resulting in the same total number of threads will yield identical simulations results.

2.2.10 Resiliency

Neither NEST nor Arbor support checkpointing at present. For NEST, checkpointing has proven challenging to implement due to the widespread dependence on pointers and hierarchical data structures, and the lack of introspection capabilities.

NEST could benefit from system-level techniques that would allow saving a simulation to and restoring from snapshot images.

3 Task 1.3: Molecular dynamics (Task leader: NCSA)

Molecular dynamics (MD) is a theoretical approach widely used in the field of material sciences, life sciences, chemistry, etc. for studies of processes and phenomena on spatial and temporal scales unreachable by experimental techniques nowadays. MD consists of solving the classical equations of motion numerically to calculate the time evolution of a system of particles (atoms, molecules or generalised particle-like objects). The potential energy of the systems under investigation is usually parametrised by an empirically constructed and determined function called force field. The quantities (properties) of interest are calculated using the information taken from the resulting trajectories (time evolution of the system).

The simulation tracks the trajectories of many particles (atoms) as the simulated system evolves in time. It solves differential equations of motion at each time step. 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.

Gromacs is one of the best world's molecular dynamics software packages. Its development started in the last decade of the 20th century and the algorithm implementations and the code progress is well described in [19], [20], [21], [22], [23], [24] and [25].

3.1 Application structure

Gromacs is a toolbox allowing users to prepare the structure they want to simulate, run the simulation and analyse the results at the end. Usually, the following steps are needed for an end-to-end simulation:

- Data preparation (pre-processing) – construction of the system to be simulated, force field assignment, setting up the molecular dynamics parameters, etc.
- Simulation run.
- Post-processing (data analysis).
- Optionally: visualisation.

It is proposed to base the use case on already prepared data and focus on the simulation proper and post-processing. Visualisation is not included in the use case. The simulation step is parallel. Post-processing tools are mostly serial with occasional OpenMP parallelisation. The code is written in C and C++ and the most important libraries to be linked are those for fast Fourier transform (FFT). In the current release one can use either MKL or FFTW depending on the compilers and libraries' performance.

GROMACS [25] uses multi-level parallelism that distributes computational work across domains, multiple cores working on each domain and even instruction-level parallelism across those cores. Molecular dynamics step is repeated as many times as long the simulated time required. One of the most frequently used algorithms for long range interaction treatment is the Particle Mesh Ewald algorithm [26], which uses calculations in the Fourier space to estimate the contribution to the energy and forces of the particles beyond the cut-off radius for direct calculations. To ensure good performance scalability, the MPI

ranks are divided in two groups, one does calculations in the real space and the other one in the reciprocal space. The main workflow of a typical simulation step is depicted in Figure 2, where one can see the key steps and communications needed.

The idea is to use the Modular Supercomputing Architecture (MSA) by running short-range interactions and bonded interaction on the Extreme Scale Booster (ESB) and long-range interaction on the HPC Cluster Module (CM). An example workflow is shown in Figure 3. As the main bottleneck is the FFT part of the code the main goal is to improve the strong scaling of the application (Figure 3). The size of the use cases will be increased in order to meet the user community requirements and fully exploit the increasing computing power.

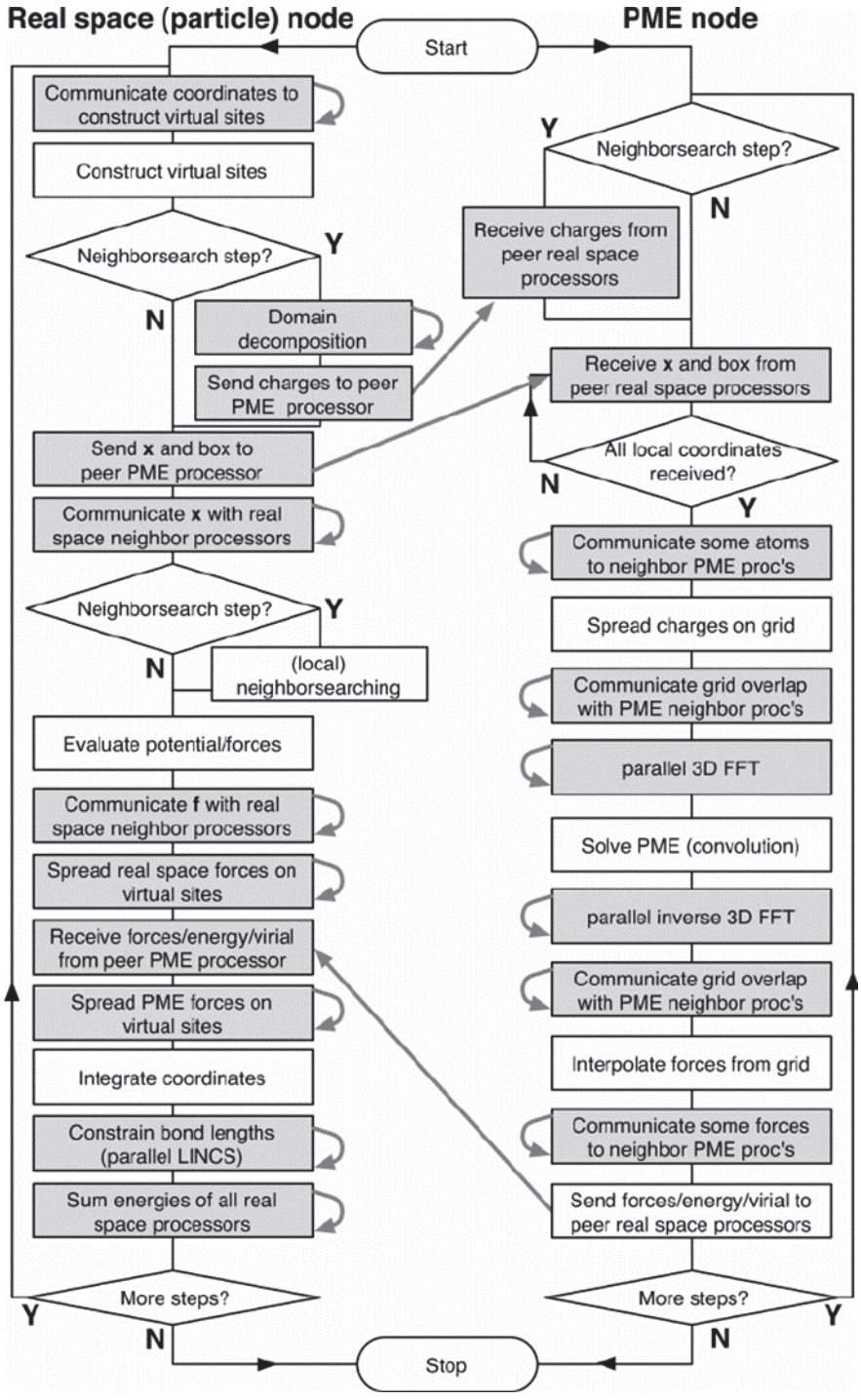


Figure 2: Flowchart for a typical simulation step for both particle and PME nodes [22] (NCSA)

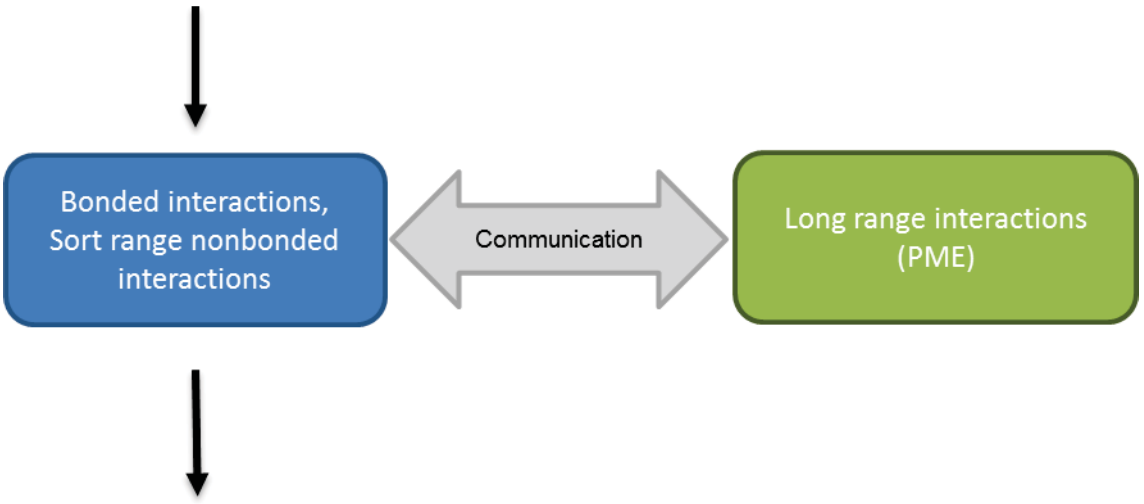


Figure 3: Suggested workflow of Gromacs suitable for the MSA (NCSA)

3.2 Application requirements

Gromacs is written in C/C++ in a hybrid MPI/OpenMP manner. Minimal software requirements are: C compiler, C++ compiler that supports C++11 standards, MPI, OpenMP, either FFTW or MKL library. From the hardware point of view, the use cases defined below need around 100 Bytes of RAM per atom per thread and for master thread about 100 KByte per atom. The total arithmetic density for the defined used cases is 0.25 FLOP/byte. The nodes interconnect network should have low latency and bandwidth of order of hundreds of GByte/s or higher for good MPI scalability.

3.2.1 Use case description

Three use cases will be used for benchmarking. They are chosen to represent small, average and big atomic systems and consist of approximately 34k, 325k and 2.2M atoms respectively.

3.2.2 Benchmarking metrics

Given the atomic system and algorithms, the performance of a molecular dynamics program is measured by the simulated time per unit wall-clock time. Nowadays, the most convenient units are ns/day (nanoseconds per day). Another important metric for a machine benchmarking is the application turnaround.

3.2.3 Scalability

Gromacs has a hybrid MPI/OpenMP parallel code. OpenMP is used for intra-node parallelization. The parallel efficiency reaches up to 80% with 64 OpenMP threads and it is shown in Figure 4.

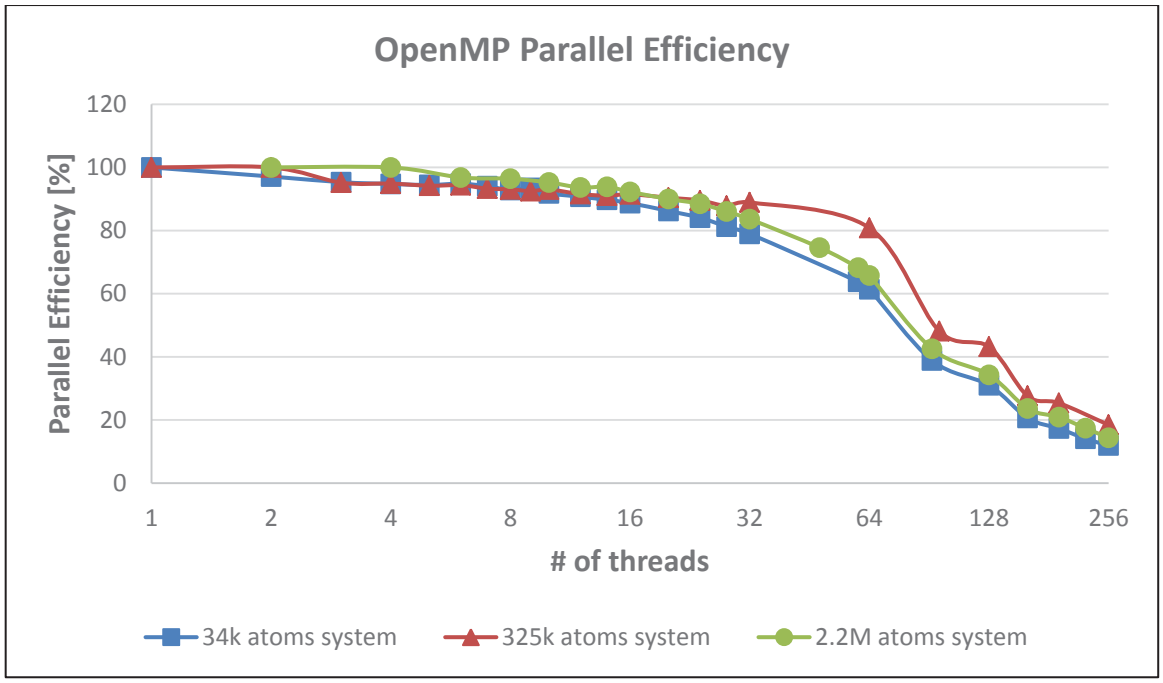


Figure 4: Parallel efficiency of Gromacs 2013.3 running on one KNL node (NCSA)

MPI is used for distributed-memory parallelisation. According to the literature the simulation program (mdrun) scales well up to thousands of nodes, but it depends on the simulated system size. One can find some results on the performance scalability of older Gromacs version in [27].

For big systems consisting of millions of atoms (particles) it has good strong scaling. The weak scaling is good for more than 1000 atoms (particles) per node.

3.2.4 Modularity

The current implementation certainly benefits from modular scheme by offloading part of the heavier calculations to a GPU. It is expected that a similar approach may be beneficial if the CM is used, but heavier calculations are uploaded to the ESB. Also, in the light of discussing the possibility to have FPGA installed on the Data Analytics Module (DAM), it might be worth trying to offload FFT calculations if a suitable FFT FPGA is implemented. Finally the CM will be used for post-processing.

3.2.5 Communication

Data parallelism – the domain of the problem is split between all participating MPI processes and each process gets its share of particles to deal with.

MPI is used for inter-node communication. The use cases combine both distributed-memory and shared-memory parallelism implemented using MPI and OpenMP, respectively.

In [28] it was shown that the increasing communications are the main reason for the performance deterioration of the MD simulation packages on large numbers of cores – typical for the case of Petascale computations. The FFT calculation requires all-to-all communication between the nodes involved in performing it (this setting can be controlled by configuration settings).

3.2.6 Compute

Gromacs uses a threading model with OpenMP implementation. Running on a typical number of threads (as developers suggest max 6 threads) the code has parallel efficiency about 90%. There are architecture specific kernels which use SIMD instructions for most of the computationally demanding routines. Total arithmetic density for the use cases are:

- Magainin ~34k atoms – 0.23
- Bombinin ~325k atoms – 0.25
- Ribosome ~2.2M atoms – 0.26

Figure 5 - Figure 7 show the performance vs arithmetic density for simulations running on a single KNL node. This analysis includes the most heavily used functions and loops on the critical path of the simulation.

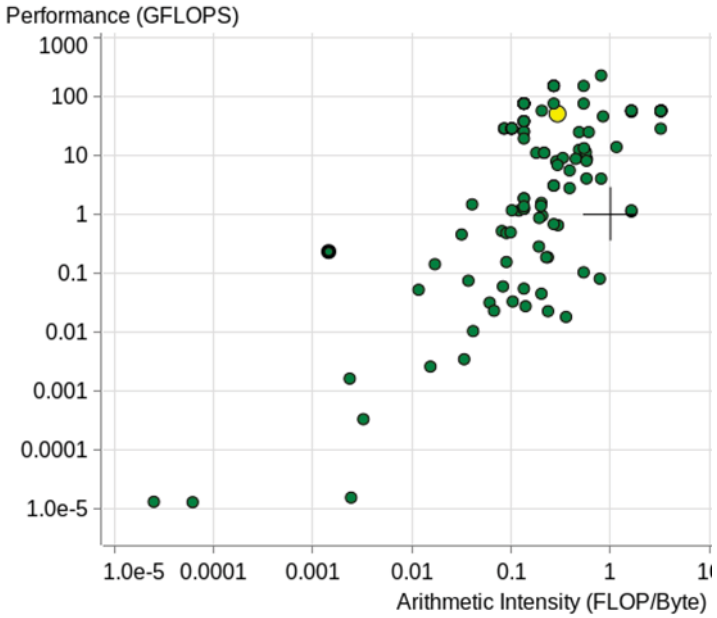


Figure 5: Magainin ~ 34k atoms benchmark (NCSA)

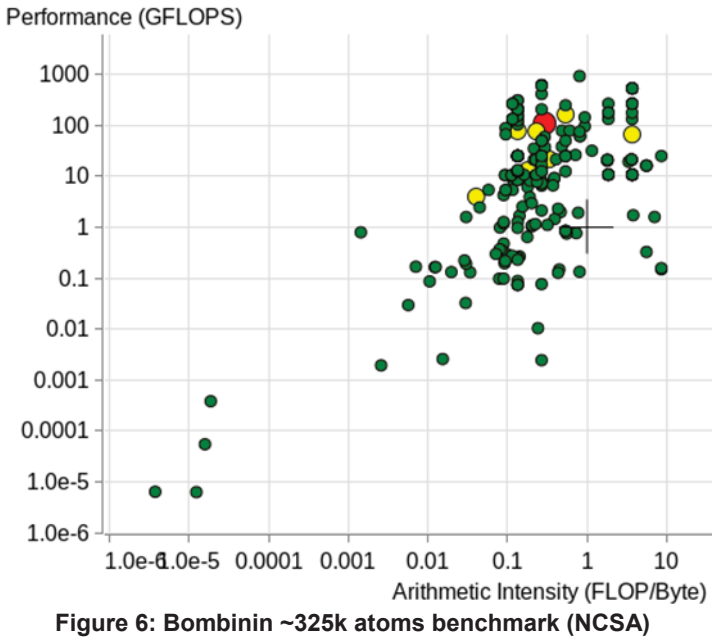


Figure 6: Bombinin ~325k atoms benchmark (NCSA)

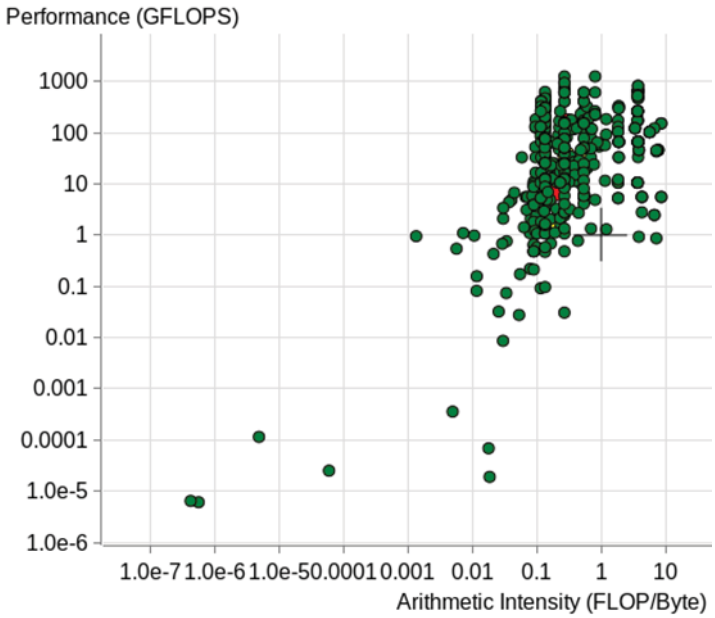


Figure 7: Ribosome ~2.2M atoms benchmark (NCSA)

3.2.7 Memory

There are no minimal memory requirements considering present computing systems, however larger bandwidth would be quite beneficial. The application is scalable enough and the memory per node limits the size of the domains. Each MPI process (calculating interactions in the assigned domain) can use OpenMP parallelisation as it was already pointed out and each thread needs its own buffers to be allocated. Therefore the higher the number of threads, the larger the amount of memory to be allocated. One can see the memory allocated by the application running on one KNL node with different number of threads for the three already mentioned use cases in Figure 8.

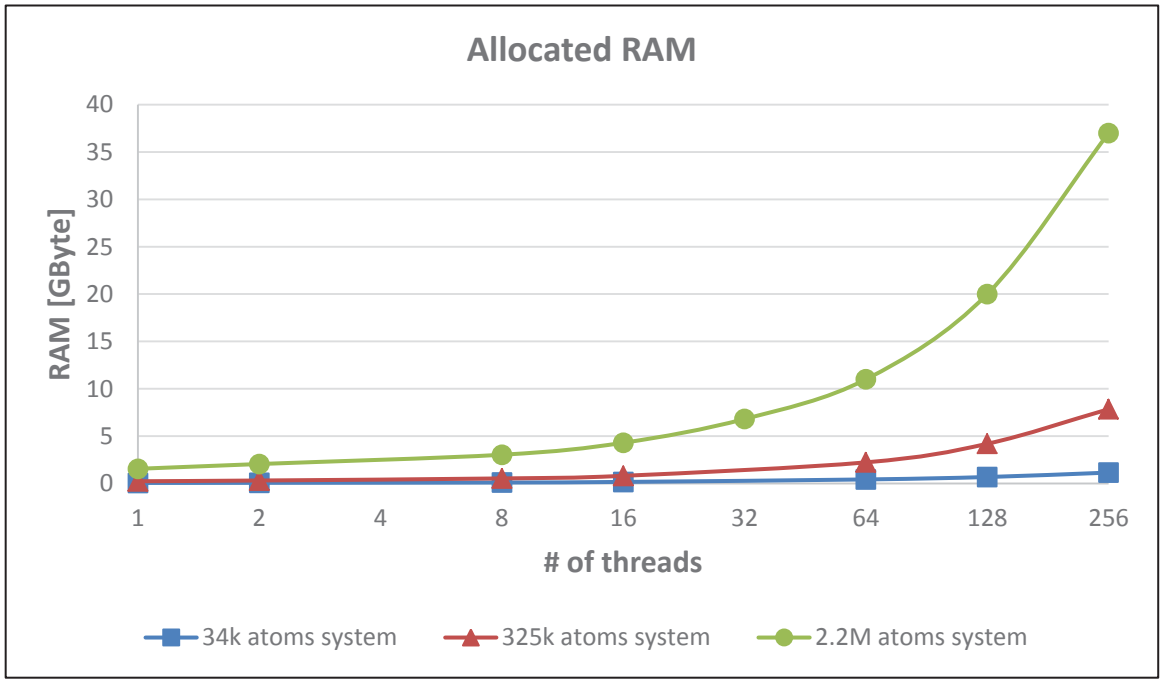


Figure 8: The memory allocated by Gromacs for the three atomic systems of different sizes (NCSA)

When running Gromacs in hybrid MPI/OpenMP mode the maximum performance is gained using between 4 and 6 OpenMP threads per MPI process. The memory allocated by the MPI process with rank 0 is shown in Figure 9 and the memory allocated by the other MPI ranks – in Figure 10. The values plotted are approximate and they can slightly vary during the simulation due to dynamic load balancing algorithm with can change the size of the domains.

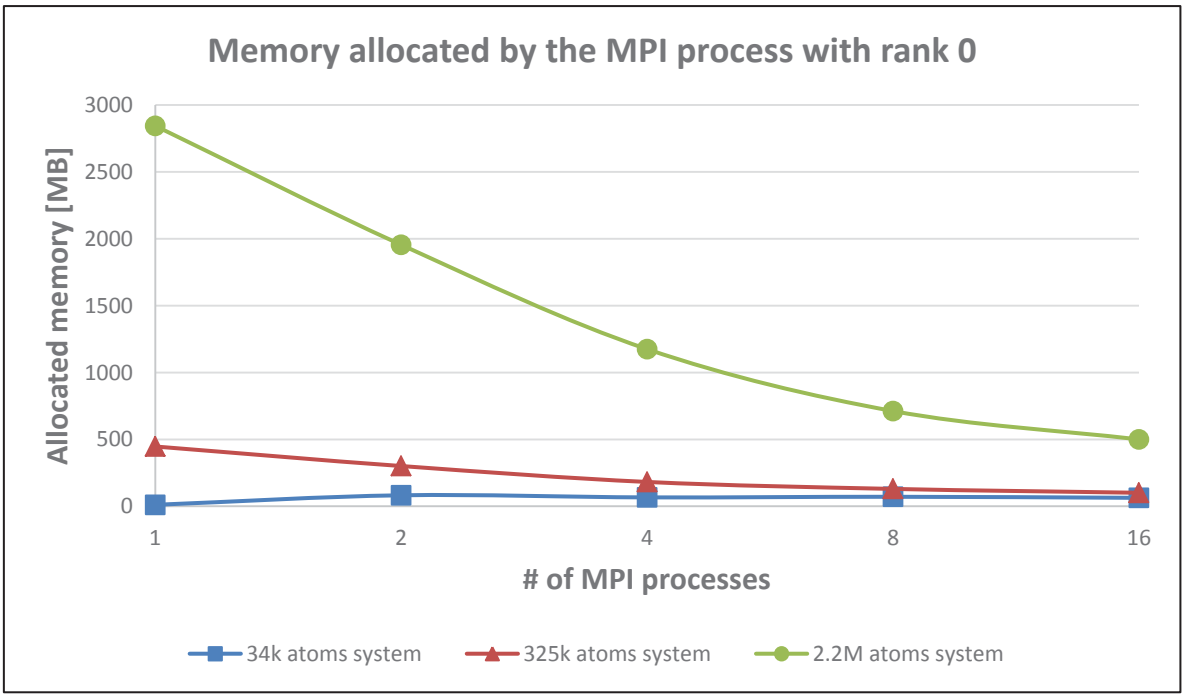


Figure 9: The memory allocated by the MPI process with rank 0 of Gromacs for the three atomic systems of different sizes (NCSA)

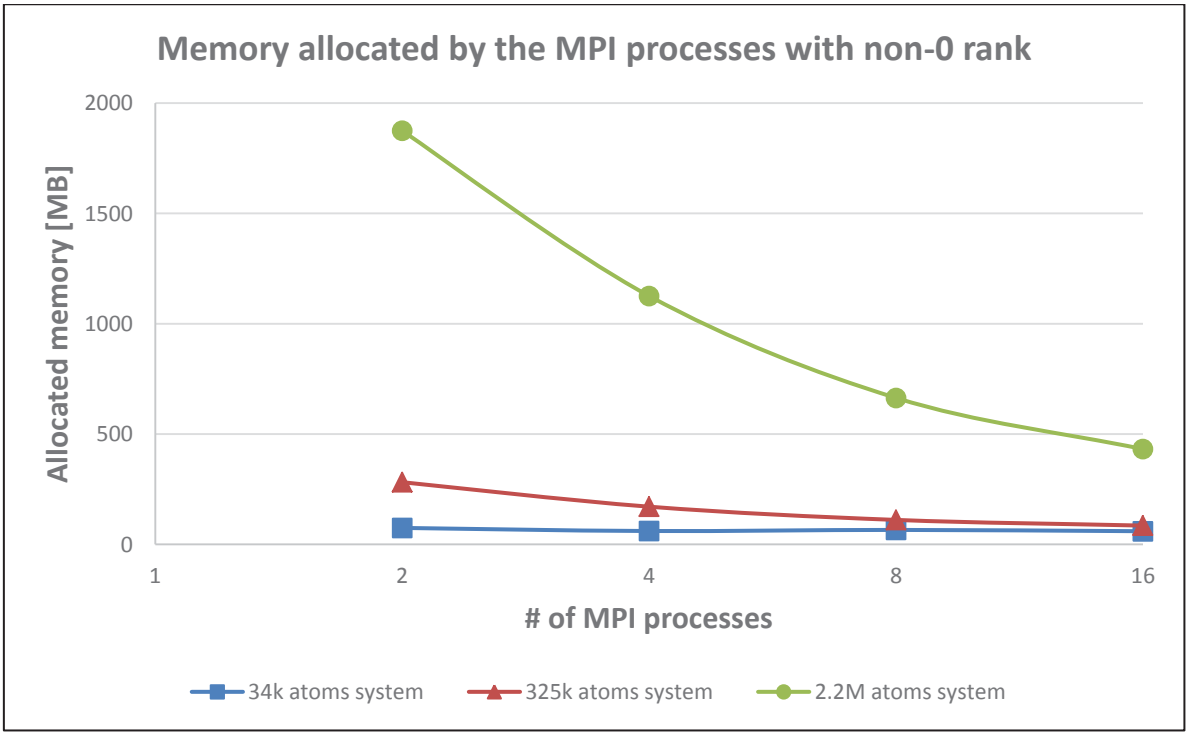


Figure 10: The memory allocated by the MPI processes with non-0 rank of Gromacs for the three atomic systems of different sizes (NCSA)

3.2.8 I/O

The application reads the input information from files and writes the output to files. The output of the simulation run is a trajectory file, which contains the coordinates, and possibly velocities, and forces, of all the atoms for each predefined number of time steps set by the user. So the size of the output file depends on the number of atoms and number of time steps being simulated, as well as on the precision used. Somewhere between 25 and 100 bytes can be written for each atom per trajectory frame. The data for all the atoms is written to the trajectory file by MPI process with rank 0, which gathers the data from the rest of the MPI processes. A usual user's request is writing frame every 1000-10000 steps, which corresponds to writing operations on seconds bases. Another usual user request is writing energy, temperature, pressure, box size, etc. to a separate file with almost the same or a bit higher frequency, but the amount of data is of the order of tents to hundreds of bytes per record.

3.2.9 Elasticity

The application can be executed on any number of resources, but they can't be changed dynamically. This is controlled via the MPI runtime and through configuration directives (e.g. for example how many nodes will be used to perform FFT calculations). It is mouldable, with no limits.

3.2.10 Resiliency

Restarting of a simulation is ensured by resuming its state from one checkpoint file. The frequency of writing such a file is set by the user with default value 15 min. The checkpoint file is written by MPI process 0 which gathers all required information from the rest of the MPI ranks.

4 Task 1.4: Radio astronomy (Task leader: ASTRON)

Within DEEP-EST, parts of the imaging pipeline of a radio telescope will be studied. This is a collection of applications that transforms raw telescope data into calibrated sky images. Figure 11 depicts this pipeline. On the left, the signals from antennas in the field are digitised and locally combined. The data are sent over Wide-Area Network links to a central location, where the correlator application filters and combines all data in real time, and writes it output to disk. After the observation has finished, bad data (due to interference) are detected and removed, and the remaining data are calibrated and imaged. During the project the focus will be on the two computationally most intensive applications in this pipeline: the correlator and the imager. The correlator's main task is to combine the data from all receivers, and the imager's main task is to create sky images. These applications are described in more detail in the application description document [29].

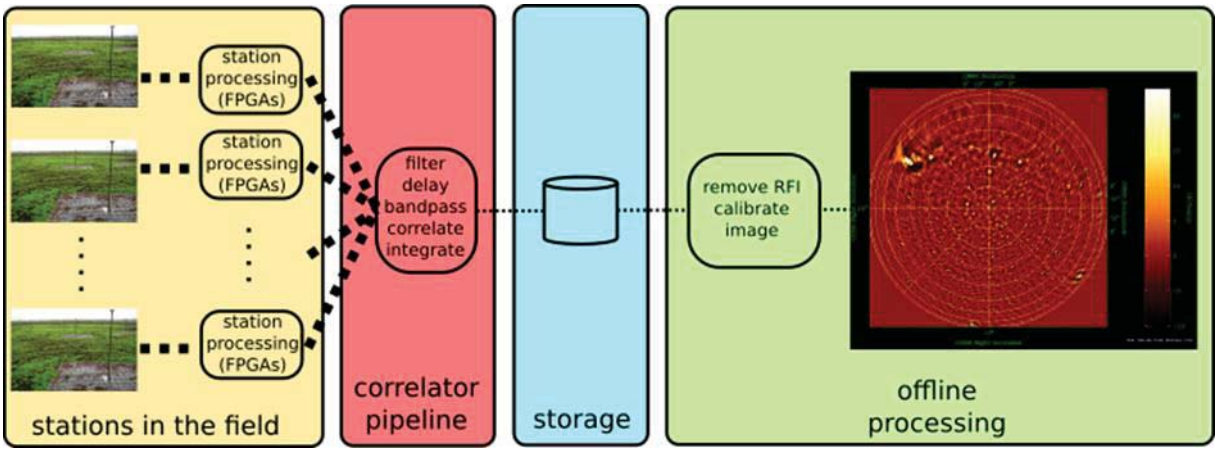


Figure 11: Workflow of the imaging pipeline (ASTRON)

4.1 Application structure

As (or are porting) the correlator and imager implementations were ported to several accelerator platforms (AMD and NVIDIA GPUs, the Xeon Phi, a DSP and an FPGA), we can play with various mappings of the pipeline components onto different DEEP-EST module types. This way, it can be analysed which modules are the most (energy) efficient for our applications, and study the impact of moving intermediate data between the different module types. Figure 12 shows an example of such a mapping, where the correlator and imaging applications are offloaded to the data-analytic modules, but this is not the only possible mapping.

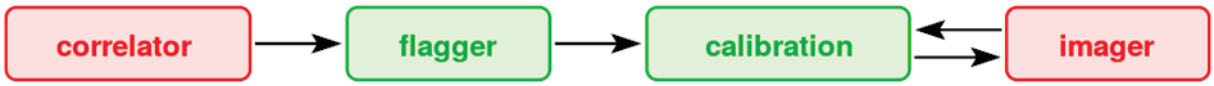


Figure 12: Possible mapping of the imaging pipeline components (ASTRON)

4.2 Application requirements

4.2.1 Use case description

The telescope data undergo a long chain of operations, as each of the applications themselves consists of a series of (signal-processing) algorithms. The last steps normally form a repeated process that subtracts the strong sources from a sky image to reveal the weaker ones.

All processing steps are parallel in frequency or in receiver (-pair); most steps are also parallel in time. As the axis along which parallelism is exploited changes several times, data needs to be redistributed, sometimes on a coarse scale (distributed memory), sometimes on a fine scale (in memory or cache).

A real observation normally takes hours (not just to increase sensitivity, but also to get a good coverage of the Fourier-transformed image, taking advantage of the earth rotation). Most performance measurements can be done on much shorter time scales, though.

The applications are programmed in C++11. OpenMP and `std::threads` are used to exploit thread-level parallelism, and OpenMP pragmas and (sometimes) intrinsics for vector-level parallelism. For NVIDIA GPUs, a CUDA back-end is used, and for AMD GPUs, an OpenCL back-end. For FPGAs, also OpenCL is used, but these kernel implementations are quite different from GPU kernel implementations. The applications depend on language-dependent FFT libraries, the Vector Math Library or Short Vector Math Library (Xeon / Xeon Phi only), MPI (older versions of the correlator only), power measurement tools or libraries like LIKWID or Intel Performance Counter Monitor, or the radio-astronomical CasaCore library [30].

All levels of parallelism (distributed memory, multi-threading, vectorization) can be exploited in all applications, but so far, little reason was found to exploit distributed-memory parallelism in most of the applications, as most applications are embarrassingly parallel. Hence, the work can be split over many independent processes. Some applications have already been optimised very well (e.g., the correlator, imager) while others are not that well optimised.

4.2.2 Benchmarking metrics

The correlator application processes data in real time; here the goal is to minimise resource and power usage for the workload of a given telescope configuration. For the other applications (including the imager), the plan is to try to minimise run time. Important metrics are operations per second (in TFLOPS), and energy efficiency (in GFLOPS/W). Currently, the best results are obtained on NVIDIA GPUs. Table 1 lists performance and energy efficiency numbers for important kernels on the Titan X (Pascal); performance on other platforms for the correlator is given in [31] and for the imager in [32].

The numbers shown in Table 1 are obtained with single-precision floating point; the correlate kernel achieves no less than 40,700,000,000,000 integer operations per second (157 Giga-ops/W) on 8-bit data.

		TFLOPS	GFLOPS/W
correlator	FIR filter	2.04	10.9
	FFT	0.505	2.75
	bandpass correction/delay compensation/transpose	0.216	1.28
	correlate	9.98	37.7
		TFLOPS	GFLOPS/W
imager	gridder	8.36	27.6
	degridder	6.80	24.4

	subgrid-FFT	0.347	1.37
	subgrid-iFFT	0.348	1.31
	grid-FFT	0.609	3.65
	adder	0.062	0.325
	splitter	0.067	0.291

Table 1: Performance measurements of correlator and imager (ASTRON)

4.2.3 Scalability

The applications exploit parallelism at several levels. In theory, most kernels (except FFTs) are well vectorisable, but in practice, this does not always work well (e.g., the Xeon and Xeon Phi cannot vectorise sine/cosine instructions in hardware, reducing imager performance). With respect to multi-threading and distributed scaling, there was no scaling bound determined; normally there are sufficient amounts of frequency bands that can all be processed independently. Whether this remains true in the future is not clear yet: there are ideas about transferring calibration results across frequency bands; frequency subbands cannot be processed independently then any more.

As the correlator is an application that runs within real-time constraints, terms like strong scaling or weak scaling do not really apply; the scaling metric here is the amount of resources (compute, energy) needed to meet the real-time constraints of a particular telescope setup. For the other applications, strong scaling applies to cases where telescope parameters do not change, and weak scaling applies to situations where processing requirements increase, e.g., after a telescope upgrade, so both scaling types are justifiable.

Datasets (artificial and real) will be used that are representative for LOFAR and for the Square Kilometre Array (SKA). Especially for the latter one, the data sizes can be large, but normally performance measurements are done on much smaller subsets.

4.2.4 Modularity

As the pipeline consists of a series of applications, the software is modular by nature. One of the things that has to be figured out is the best way to map pipeline components to DEEP-EST Modules: should each application run on the module type where it runs most efficiently, or is data transport between them too expensive, if the individual applications run best on different architectures? At least the different module types like FGPA- and GPU-accelerated modules could be explored.

4.2.5 Communication

Different methods are used to transfer data: UDP as correlator input, MPI within the correlator (if used at all; see below), TCP as correlator output to some storage node, and intermediate files between other pipeline components. These files are stored in a domain-specific format, often called Measurement Sets, and supported by the radio-astronomical CasaCore library.

Software correlators typically use MPI to redistribute input data, as each input network link contains all frequency subbands of a single receiver, while a single subband frequency from all receivers is needed to correlate the data. However, a telescope can be designed in such a way that the data redistribution is done by switching incoming UDP packets, eliminating the

need to use MPI. This complicates the FPGA firmware near the receivers, but simplifies the correlator and significantly reduces the amount of network hardware needed.

Threading and shared memory are heavily used by most applications, distributed memory only in the correlator, and tasking is used mostly to control threads that handle input, output, and GPU streams etc. Note that it would be difficult to manage NUMA domains efficiently, mostly because the programming models, libraries, and system calls do not combine well (e.g., one cannot enforce a pinned host buffer to be allocated in some particular NUMA domain using the OpenCL runtime).

Data rates can be high. The AARTFAAC correlator [33] (a LOFAR appendage) already handles 70 Gbit/s per machine. As next-generation GPUs will provide much more compute power than current GPUs, it can be foreseen that data rates between 200 and 300 Gbit/s per machine are necessary for next-generation correlators. Similarly, the imager almost hits PCIe bandwidth limits (about 100 Gbit/s) right now; hence I/O is something that will be considered within DEEP-EST.

4.2.6 Compute

An in-depth study of the computational performance of the most-important compute kernels on various (accelerator) architectures [31], [32] was already provided in earlier works. In summary, the computationally most intensive kernels are generally compute bound, while others were bound by memory bandwidth or something else (e.g., instruction latency). Examples of roofline plots for some kernels on various GPUs from AMD (S10000, Fury-X) and NVidia (Titan X, GTX 1080), an Intel Xeon Phi (7120X), a DSP from Texas Instruments, and a regular dual Xeon CPU (2xE5-26697v3) are shown in Figure 13 and Figure 14; the plots show that the bounds are highly dependent on compute kernel and architecture.

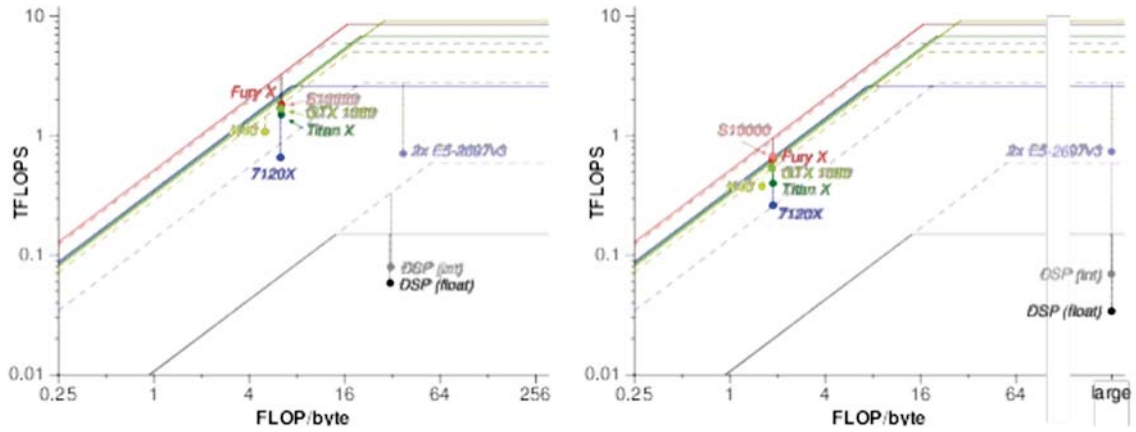


Figure 13: Roofline plot for the FIR filter (left) and the FFT (right) (ASTRON)

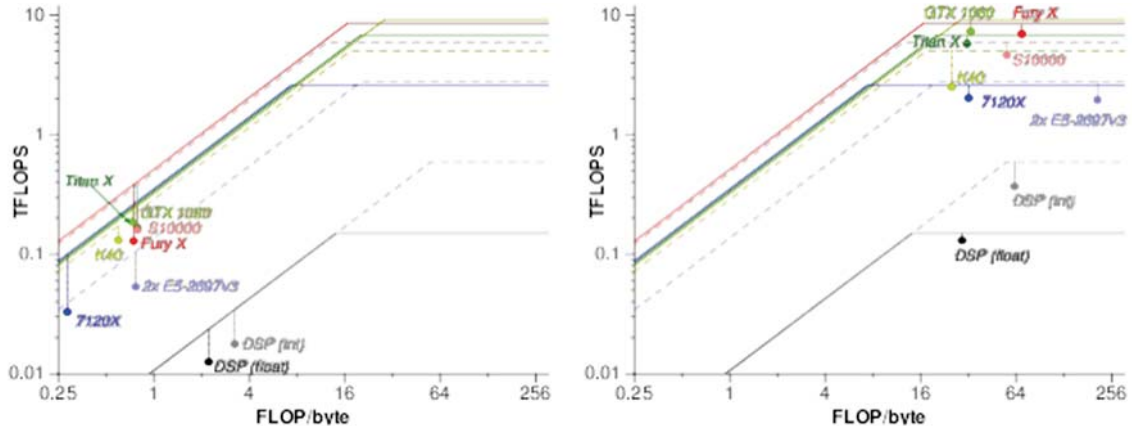


Figure 14: Roofline plot for the Bandpass Correction / Delay Compensation / Transpose (left) and the correlation (right) (ASTRON)

Table 2 shows some typical run times on a Titan X (Pascal) GPU; the "%" column shows which fraction of the total time is spent in a kernel. In the correlator application, most of the time is spent in the correlator compute kernel. In the imager, most of the time is spent in the gridding and degridding kernels.

		runtime (ms)	%
correlator	FIR filter	7.1	5.15
	FFT	13.6	9.87
	bandpass correction / delay compensation / transpose	12.0	8.71
	correlate	105	76.2
imager	gridder	3778	38.1
	degridder	4759	47.9
	subgrid-FFT	529	5.32
	subgrid-iFFT	501	5.04
	grid-FFT	7	0.07
	adder	237	2.39
	splitter	116	1.17

Table 2: Runtime measurements for correlator and imager (ASTRON)

Most of the computations are done in single-precision floating point, but for the correlator 8 or 16 bits precision is usually sufficient, and this can be exploited on architectures that provide better performance for these types. The imager might need double precision floating point in the final image, but this is not in the critical path of the computations. Most kernels of both the correlator and the imager map well to Fused Multiply Add (FMA) operations, FFTs being a notable exception. The imager also relies heavily on good sine/cosine performance (in the critical path, for every 17 FMAs, one sine and one cosine is computed). Especially NVIDIA GPUs support this very well, as they overlap these sin/cos computations with FMA

operations. Xeon and Xeon Phi processors use the SVML library to perform (vectorised) sin/cos computations in software.

These applications are currently ported to Intel FPGAs, using the OpenCL/FPGA toolkit. The main goals are to demonstrate a significant reduction in development time (compared to a Hardware Description Language (HDL)) for the things already done on FPGAs (filtering, correlating, beam forming, etc.), and to demonstrate an even higher energy efficiency than GPUs for complex applications like the imager, that were previously deemed too complex to implement in a HDL.

Apart from FPGAs, future research will include exploring new GPU features. The “tensor cores” of upcoming Volta GPUs promise a huge performance jump for Deep Learning algorithms, but it seems that the dedicated hardware that enables this boost (mixed-precision matrix multiplication) will boost performance of signal-processing algorithms like correlating and beam forming similarly. Another new feature (using device memory as a paged hardware cache for host memory) seems very useful when creating large sky images.

4.2.7 Memory

Memory requirements vary for the different applications in the imaging pipeline. While the correlator needs tens of Gigabytes (mostly for buffering), sets of large sky images can easily occupy hundreds of Gigabytes, hence 384-512 GByte of main memory is not excessive. Large-capacity 3D XPoint DIMMs would allow to experiment with even larger images; DRAM as cache before the 3D XPoint is something that may actually work well for this application. In addition, as 3D XPoint is non-volatile, it may be used for checkpointing. The working set size of most algorithms is not that large though, so we can also profit from memories (e.g. HBM) that have higher bandwidth and are smaller in size (say, 16 GByte).

Memory access pattern are different for the different compute kernels; unit-stride access is performed as often as possible, but larger strides cannot always be avoided. The imager operates on blobs that draw ellipses around the image (one ellipse per thread), so there is locality, but the access pattern is not regular.

4.2.8 I/O

Radio telescopes produce much data. Receiving the UDP data (in real time) is quite a challenge; it requires careful management of threads that receive packets, NUMA domains, and interrupts handlers. A contemporary correlator like that of AARTFAAC already receives no less than 60 Gbit/s per machine; it is necessary to experiment with 200-300 Gbit/s input data rates to keep up with the much higher compute power of next-generation GPUs and FPGAs, and to demonstrate readiness for next-generation instruments.

The I/O requirements of the imager depend on whether intermediate data sets are kept local on a machine or if they are moved across different module types between pipeline components. In the latter case, data rates of 100 Gbit/s will be necessary, as the current GPU imager is not far from hitting similar PCIe bandwidth limits.

The correlator fully overlaps external I/O, host–GPU transfers, and GPU computations. The imager overlaps host–GPU transfers and GPU computations, but not (yet) external I/O.

4.2.9 Monitoring

Performance (in TFLOPS) and energy efficiency (in GFLOPS/W) are important metrics for the applications, and several tools are used to analyse these in detail. On CPUs, tools like LIKWID and Intel Performance Counter Monitor are used. On accelerators like (PCIe) GPUs and FPGAs, PowerSensor (see Figure 15) is used, a custom-built, micro-controller-based tool that measures the current drawn by the device at very high time resolution (8.6 kHz). The microcontroller reports the sensor readings via USB back to the host, and the application can use a simple library to measure the energy consumption of individual compute kernels, due to the high time resolution. Figure 16 shows that the power consumption can vary significantly for the different compute kernels that are executed consecutively. It would be useful if similar high-time-resolution measurements could be made within DEEP-EST as well.

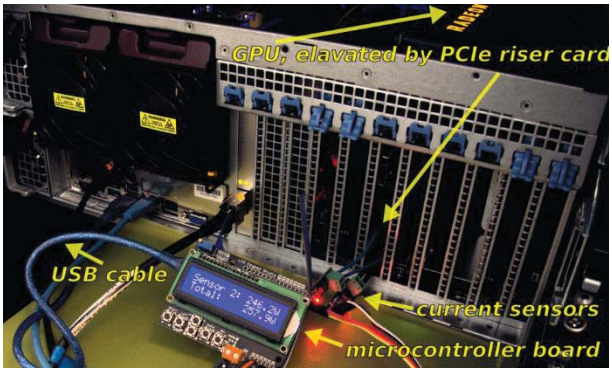


Figure 15: PowerSensor (ASTRON)

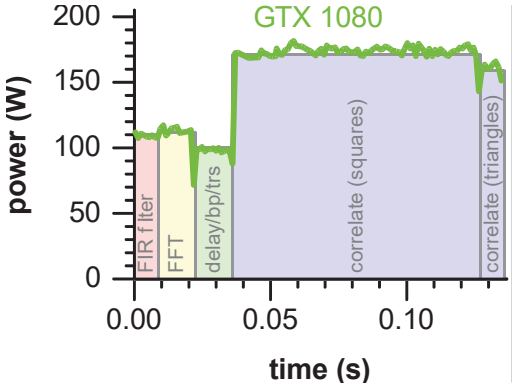


Figure 16: Power consumption for different kernels (ASTRON)

4.2.10 Elasticity

The applications are mouldable (i.e., the number of used cores can be set at program start), but the number of used cores cannot be changed during runtime. The correlator needs a minimum amount of resources to run in real time.

4.2.11 Resiliency

The correlator contains several mechanisms to continue operating even if (part of) the input data is missing, or if (part of) the output cannot be written to disk fast enough.

The mockup imager used in DEEP-ER uses SCR for checkpointing, but this mechanism is not (yet) implemented in the WSClean imager [34] used by LOFAR. Checkpointing was quite straightforward to implement, as the image under construction in the only main data structure that needs to be stored. The runtime overhead is small, as in practice, the amount of data written per hour would be small.

5 Task 1.5: Space Weather (Task leader: KU Leuven)

The Space Weather application (**SWA**) is composed of two complementary sections:

- **DLMOS**: A Deep Learning Model of the Solar Wind to forecast the plasma conditions at the orbit of the Earth from images of the Sun.

- **xPic**: A Particle-in-Cell code for the detailed simulation of the plasma environment of the planets.

The SWA will demonstrate that a coupling of Machine Learning and HPC can improve our understanding of the Sun-Earth interactions, with the long-term goal of performing accurate forecasting of the effects of solar activity on the Earth plasma environment.

5.1 Application structure

The general structure of the SWA is presented in Figure 17. The code xPic can be used to simulate the plasma flow around planets in the solar system, in particular the Earth and Mercury. Plasma is a flow of charged particles that respond dynamically to the ambient electromagnetic field. To accurately predict the plasma dynamics, two systems are coupled: a field solver performs the calculation of Maxwell equations of electromagnetism in a Cartesian grid, and a particle solver calculates the motion of billions of charged particles (ions and electrons) using Newton's equations of motion. Data is interpolated between these two solvers, in one direction by projecting the electric and magnetic field data on each particle, and in the other direction by integration of particle information, by statistical moment gathering, into the Cartesian grid.

The initial and boundary conditions for the xPic simulations of the magnetosphere of the planets are given by the DLMOS model. Simulations of the transport of plasma from the Sun to the Earth have not produced very accurate results in the prediction of the solar wind conditions, mainly due to the impossibility of stationing satellites between these two objects to perform data assimilation and to the many unknowns in the initial conditions. Machine Learning techniques are used in this project to forecast solar wind conditions from solar images. The images are downloaded from data servers via Internet and are used to train the DLMOS model for a better forecasting of the solar wind conditions used to launch the xPic simulations.

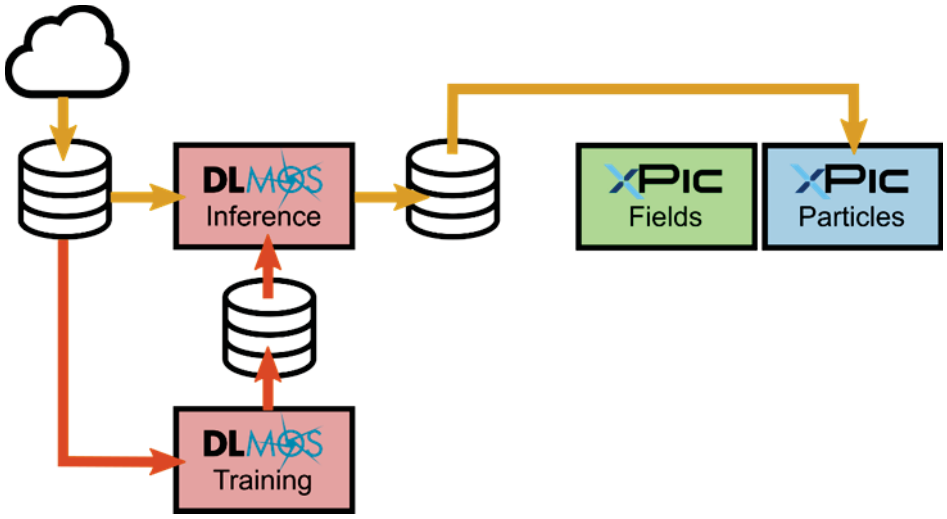


Figure 17: Workflow of the Space Weather Application (SWA) (KU Leuven)

5.1.1 DLMOS

The DLMOS model is coded in Python. Multiple frameworks are available to code Machine Learning (ML) models using Python. The efforts will concentrate on using TensorFlow/Keras, but the support of other frameworks such as PyTorch, and the use of Matlab for quick model testing is encouraged.

As with another ML algorithm, DLMOS needs to be deployed in two modes: training and scoring (also called inference). The first mode takes large amounts of input data and trains the model. The second mode uses the trained model to predict a singular input sequence. While scoring a Deep Learning (DL) model can be generally performed quickly in regular CPU architectures, training the model requires important resources in disk access, memory size and computing power. TensorFlow can be deployed in multiple architectures, including GPUs, CPUs and mobile devices. In addition, Intel has developed an optimized version for Xeon Phi processors. Hence, it is proposed to perform scoring in the HPC Cluster Module (CM) or the Extreme Scale Booster (ESB), and training in the specialised Data Analytics Module (DAM).

Defining the structure of a ML model is very complex, as this structure changes with time, type of data and objectives. We are still in the exploratory phase of the DLMOS model, but we can define the following main phases of the model. Figure 18 shows the general structure of a ML algorithm. It consists of two main parts: 1) a data pre-processing pipeline, and 2) a Neural Network (NN). For high performance of the ML model, the input data of the NN has to be as clean and homogeneous as possible. So during the development of a ML project, most of the time is spent testing different methods to clean the raw input data. The pre-processing pipeline is the final result of this process.

The second important decision is the topology of the NN itself. How many layers and how many neurons to use is still a matter of debate. Currently the most used option is to perform brute force search of the optimal parameters.

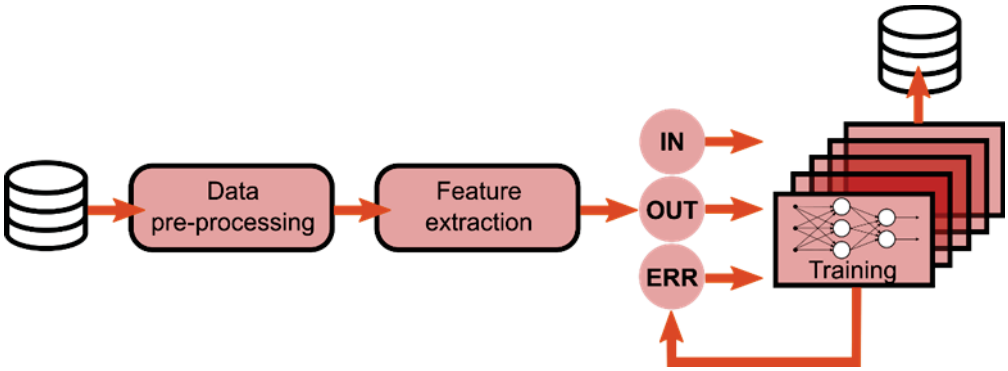


Figure 18: Training mode of the DLMOS model (KU Leuven)

The application workflow of the DLMOS model presented in Figure 18 will require multiple iterations in each one of the two main phases. Finding a pre-processing algorithm requires low to medium resources in memory and compute power. Using the final pre-processing pipeline for training requires very strong resources. Training the NN also requires very high resources. Although there is the possibility to split both phases in different modules, it makes more sense to maintain the large amounts of data generated as close as possible to the NN. It is planned then to deploy the DLMOS model in the DAM.

5.1.2 xPic

The xPic code is deployed in the modular system by dividing the problem in its two main components: a field solver is loaded in the CM, while a particle solver is run in the ESB. Electromagnetic fields are transferred from the CM to the ESB using MPI message passing. Statistical moments are sent from the ESB to the CM at each time step. The interpolation between grids and particles is performed on the particle solver, in the ESB.

5.2 Application requirements

5.2.1 DLMOS

This is a model composed of multiple parts. Those parts change in time as the model is optimised and the data is processed. The model runs at two paces: a) the human data analytics pace is slow and requires low resources, it is very interactive and requires access to different data and computing tools, and b) the second pace is the training of the NN, which requires HPC resources without human intervention. It is suggested to create two types of partitions, one for each pace of the ML models.

The hardware required for both paces is similar, what changes is the intensity and the amount of resources needed. ML models would require a hardware that has fast access to disk, high memory capacity and efficient computing power.

In terms of software, it is planned to use the TensorFlow/Keras framework, but it is strongly suggested to install a scientific computing python distribution like Anaconda, which allows to also install other ML packages like Scikit-learn and PyTorch. We also suggest that it could be important to support the use of prototyping tools like Matlab for the first human data analytics pace.

5.2.2 xPic

The code is based on three levels of parallelism using vectorisation (SIMD), multi-threading (OpenMP), and domain decomposition (MPI with multithreading support). The code is written in C++11, and uses multiple I/O libraries inherited from the DEEP-ER project, including SIONlib, SCR and parallel HDF5. The code uses the PETSc library. The CMake building tool is used for the compilation of the code.

There are two critical parts in this code: the particle mover requires high memory bandwidth and massively parallel power, and the field solver requires fast network interconnection to accelerate its multiple MPI communications.

5.2.3 Use case description

The full SWA system will work in sequence: satellite data will be passed as an input to the DLMOS model. This application will perform the scoring of the input, using the CM or the DAM, and will generate a forecast of the plasma conditions used as an input for the xPic code. Data from DLMOS needs to be transformed into valid initial and boundary conditions for xPic. The simulations of xPic will be performed in parallel using the CM-ESB division.

At the same time, multiple instances of the DLMOS model will be running in parallel in the DAM, updating the training of the model and improving its forecasting accuracy. Two levels of parallelism are predicted: the multiple instances will run independently, and each instance will use parallel computing to train its own NN. It is planned to incorporate techniques to extract the best model from the population of multiple instances in order to perform the scoring.

5.2.3.1 DLMOS

By the end of the project it is expected to operate in the following workflow: multiple DLMOS instances run in parallel in the DAM using large amounts of data for the training. Periodically, the best model is selected and saved to disk.

When an xPic run is requested, the DLMOS scoring mode is launched in the CM. It loads the best available NN model and performs the scoring the input satellite data. The output of the scoring is transformed into the initial and boundary conditions for xPic. This means the creation of an initial HDF5.

5.2.3.2 xPic

The xPic code loads an initial file created by the DLMOS model and imposes the proper boundary conditions. The code is launched in the same way as in DEEP-ER: the field solver runs in the CM and the particle solver in the ESB. The simulation is run using large resources to reach results as fast as possible.

5.2.4 Benchmarking metrics

5.2.4.1 DLMOS

High scalability is not expected, but a metric is required that measures how fast a training cycle is performed. Accuracy of the algorithm is also an important metric. In general, such cycles represent the feed-forward and feed-backward of data (two times multiple matrix multiplications) composed by multiple elements of the training set (one element in our case is

11 images of 1 MByte each, a “batch” of elements is in general composed between 50 and 100 elements).

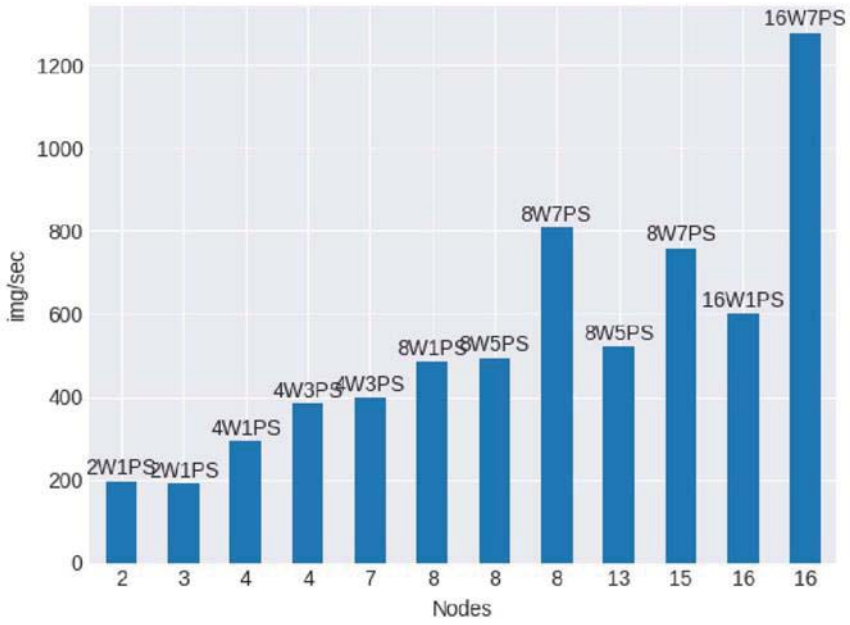


Figure 19: Throughput (images/sec) of a parallel TensorFlow application with multiple GPU nodes [35] (KU Leuven)

There is no standard metric for the performance of ML applications, but it can be thought about creating new metrics like training time to accuracy, input data processed per second, scalability with the number of processors, among others. Figure 19 shows the reported throughput of a Convolutional Neural Network (CNN) for up to 16 nodes, each node with 2 NVidia K80 cards, where the updating of the NN parameters (PS) played an important role [35]. A training time equivalent to two or three execution times of a queue is targeted (three consecutive jobs should be enough to train one model from scratch). This of course depends on the system, but in general we think of production jobs of 24 hours.

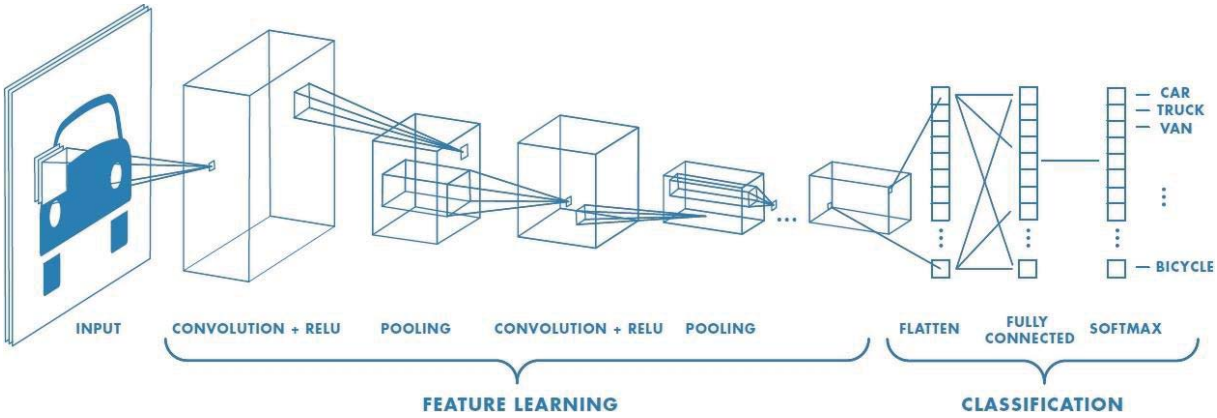


Figure 20: Typical topology of a Convolutional Neural Network (KU Leuven)

The problem size to train depends on the type of layers in the network (convolutional, fully connected, pooling, recurring, etc.). It is the result of a series of matrix multiplications: one matrix multiplication is performed per each layer of the NN. In Figure 20 a typical CNN for image classification is presented. The convolutions are matrix multiplications of the size of the layer filter, typically 3x3x3x32 (height, width, channels, and kernels). This multiplication is done multiple times per sample, almost as many times as pixels in the image. If the image is

512 x 512, the number of multiplications is around 8.38 million. The first internal layer will be the input for the next layer, its size is in general equal to the size of the image times the number of kernels of the layer, which in this case would be 512 x 512 x 32. A pooling procedure is in general performed to reduce the amount of data to 256 x 256 x 32. Now the procedure is repeated in the next layer, only this time with 32 channels instead of just 3. This second layer would require also around 2M matrix multiplications. A third layer with only 8 kernels and an additional pooling would reduce the data to around 20k nodes, using an additional 1M matrix multiplications. At this point the data is flattened and a final matrix multiplication between two fully connected layers is performed. If the final layer contains 10 neurons, it would be a 20k x 10 matrix multiplication.

This is the order of magnitude of a typical CNN application. One image is processed using around 10 to 15 million matrix multiplications, and the NN weights can comprise around 300k values (almost all of them in the fully connected layers). These values are typically stored using low precision floating points. In a typical run, multiple images are processed at the same time, in a group called mini-batch. Typical mini-batch sizes are composed of up to 100 images. A perfect weak scaling of the CNN would allow to double the number of processed images by doubling the number of nodes.

The inference step is equivalent to the propagation of a single input trough the NN. The time taken by the inference is very small compared to the training. It requires as many matrix multiplications as layers available in the NN. This procedure can be critical in real-time applications in smartphones or self-driving cars, but is not critical for our application.

5.2.4.2 xPic

The best metric of performance is good scalability. An almost perfect weak scalability is targeted, over many processors. It is needed to perform small simulations in small systems and large simulations in large supercomputers. Also it is needed to show that in all cases the code gives a good performance. Strong scalability is also important for us to obtain faster times to solution, but it has a lower priority over weak scalability. Above 80% weak scalability is targeted for runs on 100000 cores.

For xPic it was found that memory bandwidth is the most important requirement. Running in KNL processors it could be noticed that there is a huge difference between loading the code in the MCDRAM and the external memory. The core of the application is very simple, and thus requires the “feeding” of large amounts of data at the same time.

5.2.5 Scalability

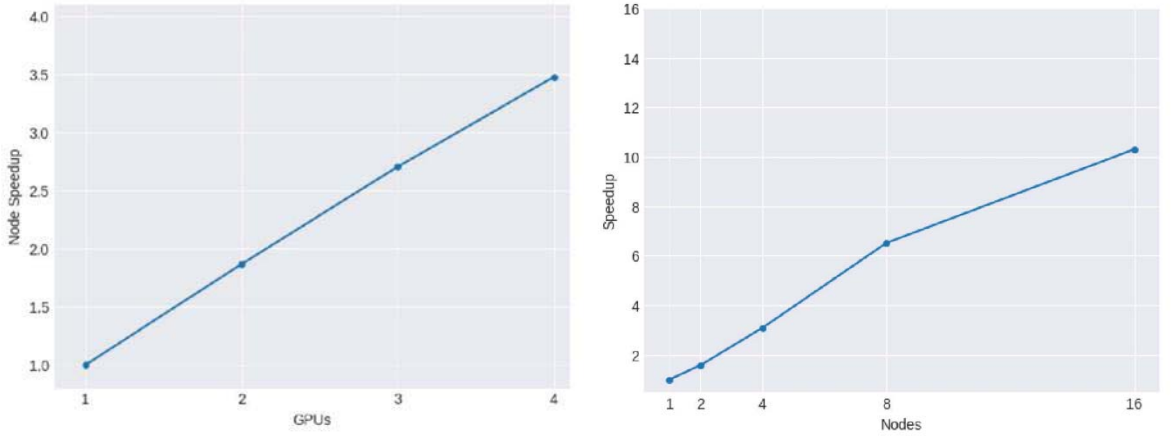
5.2.5.1 DLMOS

Our model will work using the TensorFlow/Keras framework. Very little literature is available on the parallel performance of this framework. This is due to multiple reasons, including the relative youth of the framework and the diverse nature of the different applications. The NN architecture, the type of input data and the detailed options of the algorithm are factors that can influence the scalability of the framework.

Recently Campos et al. [35], from BSC, performed a study on the scalability of the framework. Figure 21 left shows the scalability of their model on a single node. The compute

node used 2 NVidia K80 cards, giving access to 4 GK220 GPU processors. They obtain a linear scaling with an efficiency of 87% for 4 GPU cards.

Figure 21 right shows the result of their scalability test on up to 16 nodes (64 GPUs) with an almost linear scalability and an efficiency of around 63% for 16 nodes.



(a) Parallelism speedup inside a node using different number of GPUs.

(b) Distribution speedup using several nodes, with 4 GPUs, with the optimal configuration. The baseline is one node with 4 GPUs.

Figure 21: Speedup of a TensorFlow application in one GPU node (left) and multiple GPU nodes (right) [35] (KU Leuven)

The parallel efficiency of individual nodes is an interesting avenue of research.

Based on these results, our project will initially perform runs on a small number of nodes. The focus will be on optimising the performance up to 8 nodes. Large scale parallelism will be obtained using a second parallel layer.

Multiple copies of the DLMOS model will be used and they will run in parallel in multiple different nodes. Each one of the models will have different characteristics, from the number of layers and neurons to the type of input data used. An overlaying genetic algorithm will e developed that will evolve the parameters of the different copies of the model in order to achieve faster convergence to an optimal solution and avoid local minima and saddle points.

Our goal is to show that an alternative training method can be used to attain faster results using massively parallel supercomputers, with a very good efficiency.

5.2.5.2 xPic

The code xPic has been programmed using C++11. It features three levels of parallelism: vectorisation (SIMD), multi-threading (OpenMP) and domain decomposition (MPI). The particle solver uses #pragma calls to achieve high scalability. The field solver is based on the PETSc library and is limited by the scalability of such library.

The code can run completely on a single architecture or can be launched in a Cluster-Booster Mode (CBMODE) that deploys the field solver and the particle solver in different architectures, typically a CPU cluster and a Xeon Phi booster. The code division is achieved using a call to MPI_Comm_spawn. An MPI interconnect is created in order to move data from/to each solver.

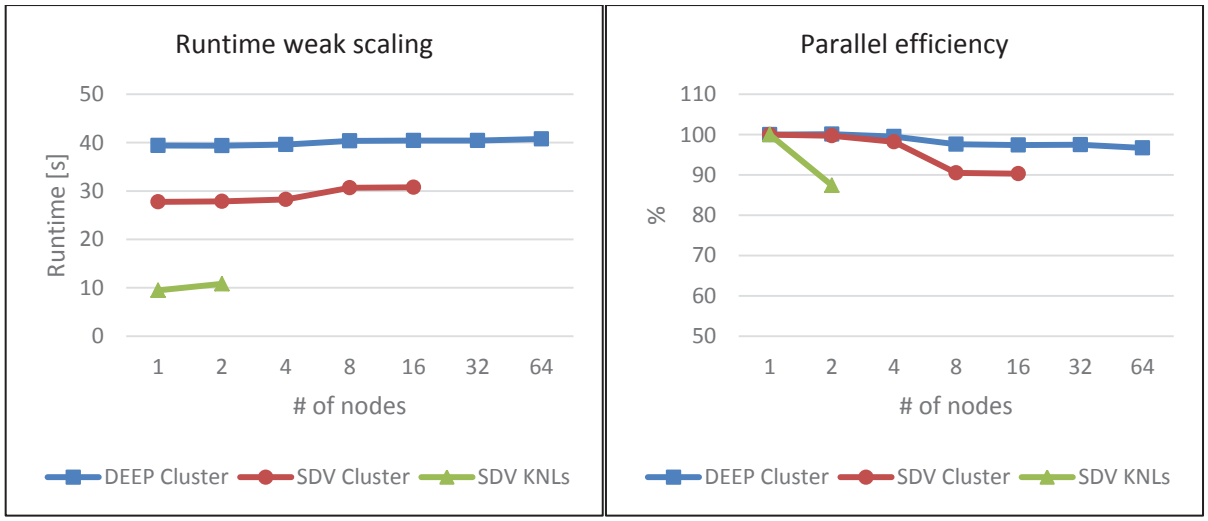


Figure 22: Weak scaling of the xPic code on different DEEP(-ER) systems (KU Leuven)

Figure 22 and Figure 23 show the performances of the code (weak and strong scaling respectively) reported in the DEEP-ER project. We already show a very good weak scaling performance, but it is still needed to perform optimisations in the code to maintain a parallel efficiency as close as possible to 100%, in particular in such small number of nodes. The degradation in the performances is mainly due to the field solver. We need to check and optimise the PETSc field solver and, if it is necessary, the solver will be replaced for an alternative (or home tailored) solution in order to attain optimal performance. It is expected to perform this analysis using the Paraver and Dimemas tools from BSC.

Our final goal is to develop an efficient code that can present perfect weak scaling (>90%) for up to 100000 cores.

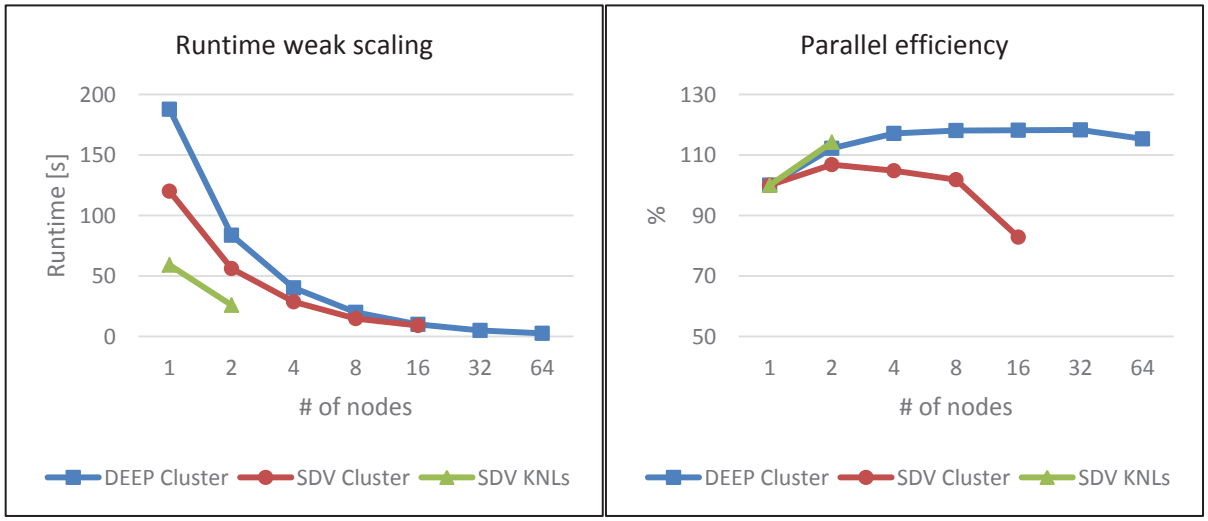


Figure 23: Strong scaling of the xPic code on different DEEP(-ER) systems (KU Leuven)

5.2.6 Modularity

5.2.6.1 DLMOS

This model is based on the TensorFlow framework. This allows offloading different parts of the algorithm to different architectures. The current idea is to use the DAM to perform the

input data pre-processing and the NN training. However, we can envision to use the modularity of the system to offload data intensive procedures, like feed forward matrix computations, to the ESB, while the DAM processors perform the back-propagation gradient descent training. A similar approach was taken by Campos et al. [35] to achieve the performances presented before in GPU clusters.

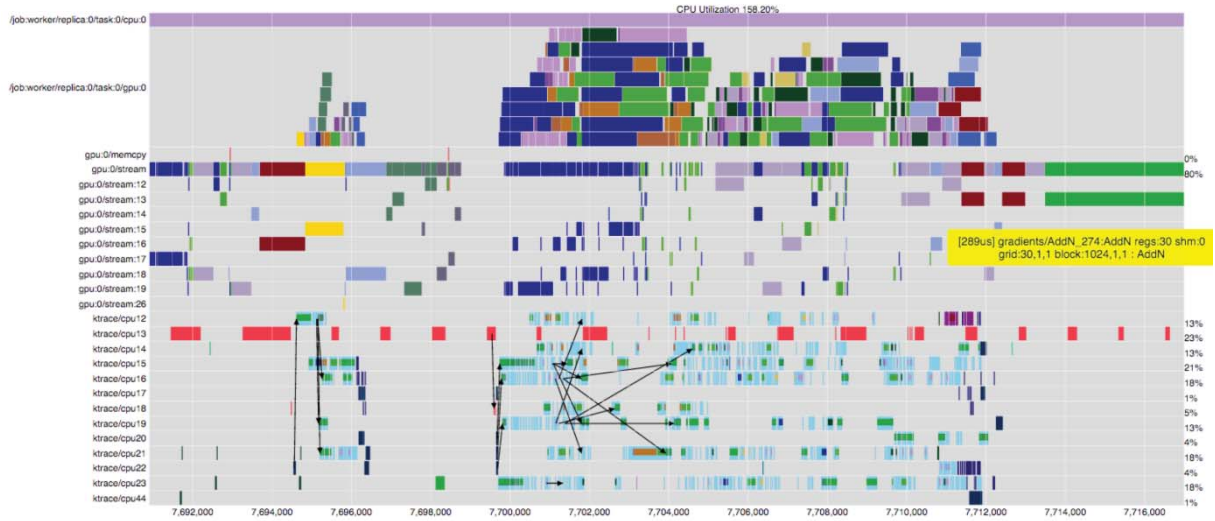


Figure 24: Traces of a TensorFlow application running in a cluster of GPUs [36] (KU Leuven)

An additional possibility is to perform image pre-processing in the DAM, feed-forward propagation in the ESB, and back-propagation in the CM. This intricately use of multiple architectures is not recommended at the moment: the parallelisation strategy of TensorFlow might lead to long spinning times in the different nodes. Figure 24, published by Abadi et al. [36], shows a trace of the execution of a TensorFlow model on a heterogeneous CPU/GPU architecture. The GPU is used as an accelerator of particular tasks. These tasks are launched by a scheduling system from the CPU. At the moment it does not seem that there is an intention to optimise the use of the accelerators, but only to use them as auxiliary fast compute power. This leads to the current limitations in scalability of the framework.

5.2.6.2 xPic

The code xPic has been designed with the Cluster-Booster separation in mind. The code can run on a single module (CM, ESB) or in Cluster-Booster mode. Since the KNL processors allow loading libraries compiled for Xeon processors, the code can run in multiple combinations of modules: CM+ESB, CM+CM, and ESB+ESB. We imagine that if there is support for the Intel compiler and the necessary libraries, we will be able to run one of the solvers (or both) in the DAM.

The code is agnostic to the architecture, but depends on the support of the Intel compiler for such platform.

5.2.7 Communication

5.2.7.1 DLMOS

As shown in Figure 24 above, the TensorFlow framework uses a rather complex data flow scheme. These communications are not necessarily regular and might be blocking for sections of the model that require a logical sequence. The size of the messages passed

among the components might also be irregular, depending on the kind of information exchanged between the different “nodes” of the graph.

It will be necessary to study in more detail the performance characteristics of the framework in general, using benchmark cases, and extracting detailed traces of the different parts of the model.

The persistent memory of the NAM technology could be potentially used to store information about the topology and/or parameters of the NN that is trained, keeping “the best” module available for multiple applications to run.

5.2.7.2 xPic

The code performs memory exchanges among the processors that belong to a single MPI process (we could have one or multiple MPI processes per node), and it performs message passing among MPI processes. The size of the MPI communications depends on the size of the problem.

In the particle solver, an MPI process is responsible for the computation of a 3D cubic block of cells. Each one of these cubes is subdivided in multiple cubic blocks. Only the blocks located at the interfaces between MPI processes perform MPI communications through that face. Multiple blocks in a single MPI process are allowed to perform MPI communications in parallel. Communications are performed first in the faces along the x direction, then along the y direction and finally along the z direction. The size of the messages is then dependent on the ratio between the volumes of the blocks and its faces. The bigger the cubic domain, the lower is the ratio of surface/volume, so communications become less intensive. But there is a fine balance that requires small volumes to fit the maximum amount of data in cache memory.

The communication patterns of the field solver are controlled by the PETSc library. A Krylov space method is used to solve the linear system associated with the Maxwell equations. These methods require the calculation of a global residual and the computation of first order differential operators. The former requires a global collective operation and the later requires the exchange of ghost node data among neighbouring processors.

Regarding the use of new technologies available in DEEP-EST, we think that it is possible to use the NAM to perform non-critical computations of auxiliary data. For example, we would like to calculate the total energy of the fields, the kinetic energy of the particles or velocity distributions of particles in given zones of the domain. To do so, it is needed to load the data into the NAM, and then the NAM needs to perform basic mathematical operations, including addition, power and square root.

Memory requirements for the NAM from xPic (KU Leuven)

Use of the NAM: calculate the kinetic energy of the particles: Ktot = SUM_allparticles[sqrt(u^2 + v^2 + w^2)]

# of cells / node	200000
Particles. / cell	2000
# field vectors	3
# vectors / particle	3
Precision (b)	8

# nodes	1	4	16	64	256	1024
# particles	4.00E+008	1.60E+009	6.40E+009	2.56E+010	1.02E+011	4.10E+011
T. mem. use (GByte)	9.6	38.4	153.6	614.4	2457.6	9830.4

Use of the NAM: calculate the field energy: Ebtot = SUM_allnodes[sqrt(B^2)]

# nodes	1	4	16	64	256	1024
T. mem. use (GByte)	0.0048	0.0192	0.0768	0.3072	1.2288	4.9152

Table 3: Memory requirements for the proposed use of the NAMs in xPic (KU Leuven)

5.2.8 Compute

5.2.8.1 DLMOS

A first look at the performances of the TensorFlow framework has been presented before. For our applications it is expected to use single (or low) precision floating point operations for the training of the NN, but it is required to use double precision operations for the pre-processing of the high-fidelity images used as input. We will progressively rise the resolution of the input images, depending on the computing system available, from 512 x 512 pixels to 4k x 4k pixels. A single image batch can be composed of 16 images, with 4 channels, for a total of 4k x 4k x 4 x 16 = 8.6 GByte, using 8-byte floats. Inference will use only one image, reducing the data use to 530 MByte.

TensorFlow suffers from low parallel efficiency. The problem of suboptimal efficiency in one or multiple nodes has not been raised in the community, and is a possible research avenue for future developments. There is still room for improvements. It is expected that the model will be limited by the data transfer rates between the disks and the memory, and by the computing efficiency of the pre-processing pipeline.

During the training sequence, multiple instances of the model will run in parallel in different nodes. Non-blocking data exchanges will be performed among the models to improve their accuracy. This procedure will be managed by a supervising node and will be designed using non-blocking communications.

5.2.8.2 xPic

The application takes advantage of highly vectorised machines, like the Xeon Phi processors. It requires the use of double precision floating point operations for accuracy on the physics results. It uses SIMD instructions (from SSE, AVX to AVX-512 instruction set), OpenMP and MPI. The field solver requires the use of PETSc which is based solely on MPI

for its parallelization. The particle solver is limited by the memory bandwidth, as shown by the results of the runs performed on KNL cards, with and without the MCDRAM. But we still need to perform more detailed analysis of the performance of the code to be sure of these results.

The code uses multiple arithmetic and trigonometric functions for the resolution of the particle movement and the linear system of Maxwell equations. In addition, for the initialisation of the particle population we require the use of a good random number generator.

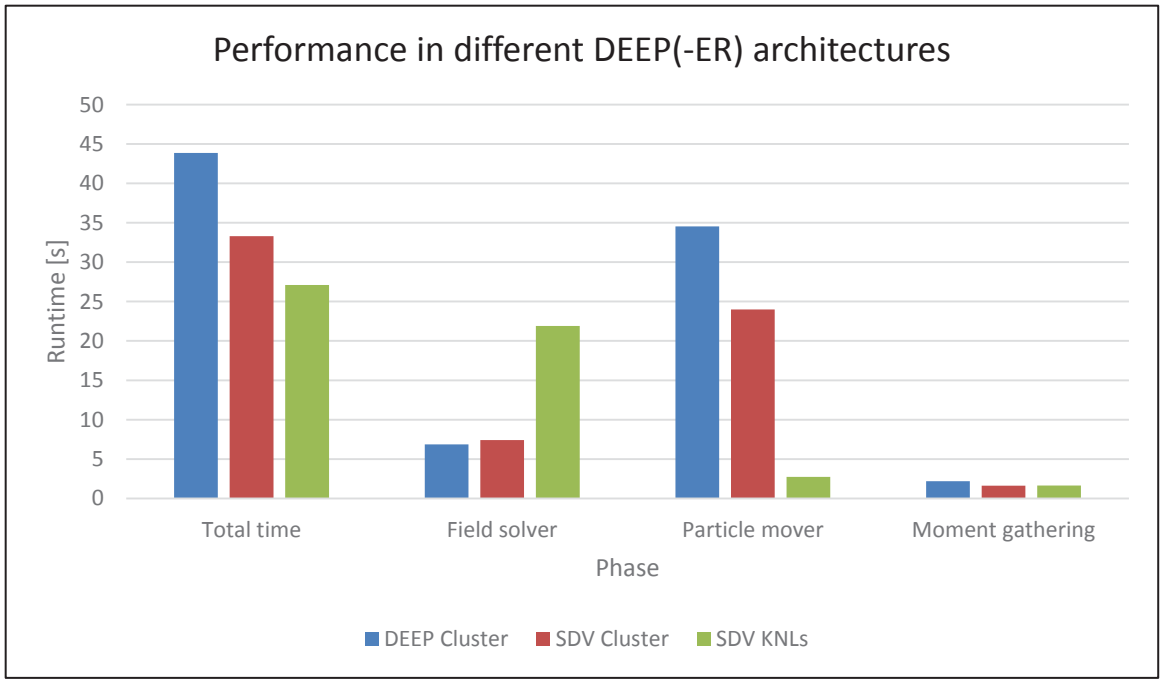


Figure 25: Performance of the phases of xPic on different DEEP(-ER) architectures (KU Leuven)

Figure 25 shows the performance of the different phases of the code in three different architectures. From this graphic it is clear that the field solver performs much better on the Xeon processors, while the particle mover gives much faster runtimes using the Xeon Phi processors. Moment gathering is also performed faster in the KNL nodes. A clear speed up is possible by using the Cluster-Booster mode (CBMODE) taking the best of both worlds.

5.2.9 Memory

5.2.9.1 DLMOS

Following I/O considerations presented in the next section, we estimate that the amount of data to load in memory, per training batch, could be around 16 GByte per core. Larger memory sizes allow to perform more computations at once, but the good ration between the size of the input data and the speed of the processing (throughput) still needs to be found and might heavily depend on the particular DL application. Using larger batches of images could accelerate the computing time, and minimise data I/O and pre-processing times, but the price to pay is a higher demand on memory size and performance.

5.2.9.2 xPic

It is expected to have a minimum of 16 GByte of memory per node. This was a hard limit due to the MCDRAM size in KNL nodes in the DEEP-ER Project. But the code would benefit from much larger systems with at least 100 GByte of memory per node. Most of the memory

access is contiguous, but sections of the code require gather/scatter, and random access. Currently there are no subroutines that pre-load information from the DDR4/DRAM to the High Bandwidth Memory (HBM), but such routines can be implemented during the project. In all cases, this code works better with access to HBM.

5.2.10 I/O

5.2.10.1 DLMOS

The data used as input correspond to multiple images of the Sun and 1D temporal information of the plasma conditions in front of the Earth. By far the largest use of disk space will be the input solar images. These images will also be pre-processed in a pipeline that will create additional “clean” images. The solar images are obtained from the Solar Dynamics Observatory (SDO) spacecraft. This mission produces around 1 TByte of data every day. Each image can have a resolution of 4k x 4k, for a total size per image of around 16 MByte. Lower resolution images are available. It is planned to use the largest possible resolution to capture CME and solar flare physics in the small scales. But we will limit the resolution depending on the system performance, as the number of dimensions of the problem increases exponentially with the resolution. The cadence of the 4k images, i.e. the time between two images, can be as low as 10 seconds. It is planned to use a lower cadence for our analysis, with around three images per hour. Ten channels are available per image, for a total of 720 images per day, or 96.6 GByte of data per day.

These images are passed through the pre-processing pipeline, which generates additional and/or auxiliary data used as input for the NN. For the moment we have not an exact estimation of the total number of files generated, but we predict a maximum of 2x increase in the data (the data input to the NN should be a smart simplification of the raw data).

Depending on the quality of the databases used for the solar images, these are available from 2 to 5 years. This is equivalent to 70 to 176 TByte of data. In addition, satellite measurements of the plasma conditions in front of the Earth should add around 100 GByte of data for the same period of time. Not all data need to be loaded in memory, but a large “batch” of images has to be used at the same time for training. We think that five hours of data (8.6 GByte) is a batch size that can be useful. But in addition, the memory must hold the NN weights, the NN topology, and all the auxiliary data. Deep Neural Networks (DNN) can be composed of thousands of neurons, generating multiple large dense matrices that are distributed among all the available processors. The data in these matrices can be stored in half or low precision floats. We hope that memory sizes of around 16 GByte per core could be enough to train the NN.

5.2.10.2 xPic

The code creates two types of pHDF5 files: one file containing the Maxwell fields of the simulation in a 3D Cartesian grid, and one file containing the totality of the ions and electrons in the simulation. The former has an average size of 30 MByte and the later can grow to very large sizes up to 1 TByte. This last file is only kept in disk to perform simulation restarts and it is expected to keep only a couple of them in disk at the same time. Field files are saved once every 500 iterations. Each “typical” (for the cases we plan to use in DEEP-EST) iteration is usually performed in 1 second. So one field (of 30 MByte) file can be stored every 8 to 10 minutes. Each particle file is stored only once per simulation.

The code xPic performs I/O routines in the particle solver while the field solver is running. The particle solver is generally attached to the Booster node, so it would benefit from a direct link from these accelerators to the file server, or the NVMe. I/O is a very small part of the total compute time of the code, so it is not very sensitive to the performance of the file system.

5.2.11 Monitoring

5.2.11.1 DLMOS

Currently there is no clear need to include in the models information about the hardware, so monitoring the system is not a priority for our application. But it would be very interesting to obtain a summary of the statistics of power consumption, IPC, FLOPs, instructions per second, or any other measurement that can help the project and can be presented to the community.

5.2.11.2 xPic

Power measurements for the code were performed during the DEEP-ER Project, but it required a complex interaction with the administrators of the system. We launched the application in the system, and in the meantime the administrators launched a power measurement script. The data was then sent via email for analysis. It would be an interesting improvement to have a more direct method to measure the power consumption of any application.

5.2.12 Elasticity

5.2.12.1 DLMOS

Due to the nature of the TensorFlow framework, the model is very malleable. Resources are assigned using a task and queue system depending on the needs of the model. However, this dynamic strategy can also lead to bad load balancing. We are not aware of current techniques to change the resources distribution on the fly, while the training is taking place, but we can explore the possibility of performing restarts of the model using different resources.

The limitations of the model reside on the speed data that is transferred from persistent memory or disk, to the processing units, and the computing speed of these units to perform matrix operations and gradient computation.

5.2.12.2 xPic

Unfortunately, for this code, the number of MPI processes must be divisible by the number of threads and the number of cells used in the code. This is checked during runtime and creates an error if is not fulfilled, halting the execution of the code. There is currently no method implemented to automatically adapt to different resources on the fly, but we would like to explore the possibility of including new algorithms that distribute better the available resources.

This code benefits from large memory, but more importantly, from HMB memory in the Booster side (particle solver).

5.2.13 Resiliency

5.2.13.1 DLMOS

TensorFlow supports consistent checkpointing and recovery of its execution state on a restart using in-house functions. Each variable in the model is connected to a ”save” procedure, which is periodically executed, every N iterations or once every N seconds. When the save function is executed the contents of variables are written to persistent storage, from which each variable can be read on restart.

We are not aware if checkpointing libraries, like SCR (Scalable Checkpoint-Restart), can give support to additional checkpointing and restart of Python scripts. If such is available, and its implementation is simple, we could test our model with that tool.

5.2.13.2 xPic

The code has already been equipped with all the available tools for resiliency from the DEEP-ER project. We use a coupling of SIONlib and SCR for automatic checkpointing. The size of these checkpoints is equal to the size of a single field and particle files. For large simulations the particle file checkpoint can reach size of the order of half a TByte.

It was found that it is not straightforward to use different checkpointing methods proposed by SCR: BUDDY vs. PARTNER vs. LOCAL. They should be just code words to change. Instead we have had to duplicate many parts of the code to be able to pass from one mode to the other. The input for such selection is also located in multiple places at the same time: from an input file, to environment variables, to sections inside our own code. Sometimes we are not sure where one option should be placed.

The benefits of having a resiliency tool installed in our code cannot be underestimated: we have obtained improvements in write times, and we can perform quick restarts of simulations that have stalled or that were kicked out of queue before the expected end of the simulation. We plan to work in a co-design collaboration with WP6 to improve the usability of these I/O, resiliency and checkpointing tools.

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

Our data analytics within the realm of Earth sciences consists of three mutually exclusive Machine Learning applications, namely:

- Highly parallel DBSCAN (HPDBSCAN) covered in sub-chapter 6.1.
- Support vector machines with a specific parallel implementation (piSVM) in sub-chapter 6.2.
- Deep Learning with TensorFlow, sub-chapter 6.3.

Highly parallel DBSCAN (HPDBSCAN) is a clustering algorithm developed by researchers from the University of Iceland and the Jülich Supercomputing Centre. Subchapter 6.1 below is dedicated to this algorithm.

6.1 HPDBSCAN

Point clouds are datasets consisting of points with multiple dimensions, such as the 'Inner City of Bremen' [37] point cloud dataset created by 3D laser scans of the city. HPDBSCAN is a highly parallel implementation of the established DBSCAN clustering algorithm that takes points of an arbitrary dimension as one of its input argument and correctly labels each point to a cluster ID. For the Bremen dataset described previously, HPDBSCAN can be used to classify points as a building or filters it out as noise. The correctness is based on two required input values: the minimal number of points required to form a cluster and the maximum neighbourhood search radius.

6.1.1 Application Structure

HPDBSCAN´s execution is divided into six steps:

1. The entire point-cloud dataset is divided into equal-size chunks and loaded by all processors.
2. The data is pre-processed where each point is sorted into a virtual, unique, spatial cell corresponding to their location within the data space.
3. The sorted data workload is balanced via a heuristic and redistributed to distinct number of cells from the hyper grid.
4. Clustering of the assigned cells is performed locally by each processor.
5. Local results are merged into one global result.
6. Cluster relabelling rules from the previous step are broadcasted and applied locally.

There are two innovative approaches possible in optimising the algorithm for DEEP-EST with expected performance improvements. These approaches are explained in detail in Figure 26 and Figure 27.

6.1.1.1 HPDBSCAN Workload A

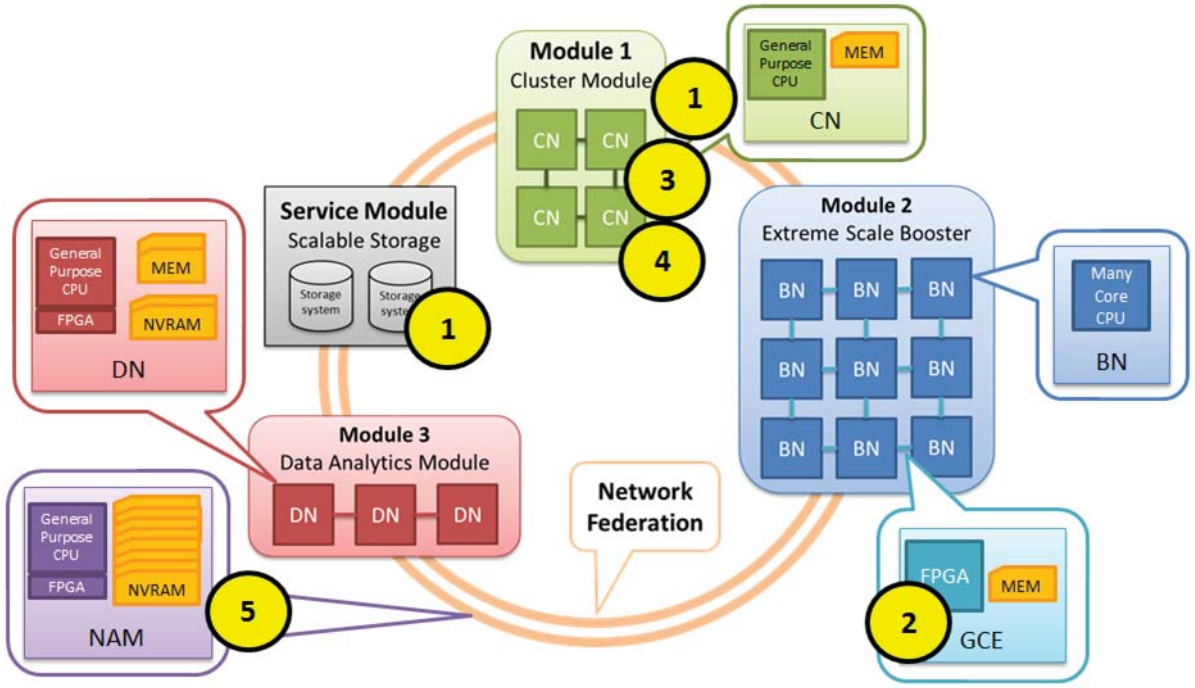


Figure 26: DEEP-EST Workflow for HPDBSCAN (Workload A) (UoI)

Steps of HPDBSCAN Workload A:

(1) The point cloud dataset is loaded with parallel I/O using HDF5 in the DEEP-EST Scalable Storage Service Module (SSSM) and the DEEP-EST HPC Cluster Module (CM).

(2) The indexing through sorting and cost heuristic small computing elements can take advantage of the FPGA in the DEEP-EST Global Collective Engine (GCE) module in combination with a MPI collective `MPI_Alltoall()` function to distribute the points equally among the DEEP-EST CM (index might be re-used by other mining algorithms).

(3) Clustering with HPDBSCAN is performed on the DEEP-EST CM locally (OpenMP) for shared memory elements and the load given by the MPI collective of the `MPI_Alltoall()` function in (2).

(4) Merging the different computed clusters on chunk edges according to specific rules using halos across nodes is performed on the DEEP-EST CM globally (MPI).

(5) Cluster ID and noise ID are usually added to the HDF5 file but could reside in the Network Attached Memory (NAM) for scientific studies, e.g. level of detail (LoD).

6.1.1.2 HPDBSCAN Workload B

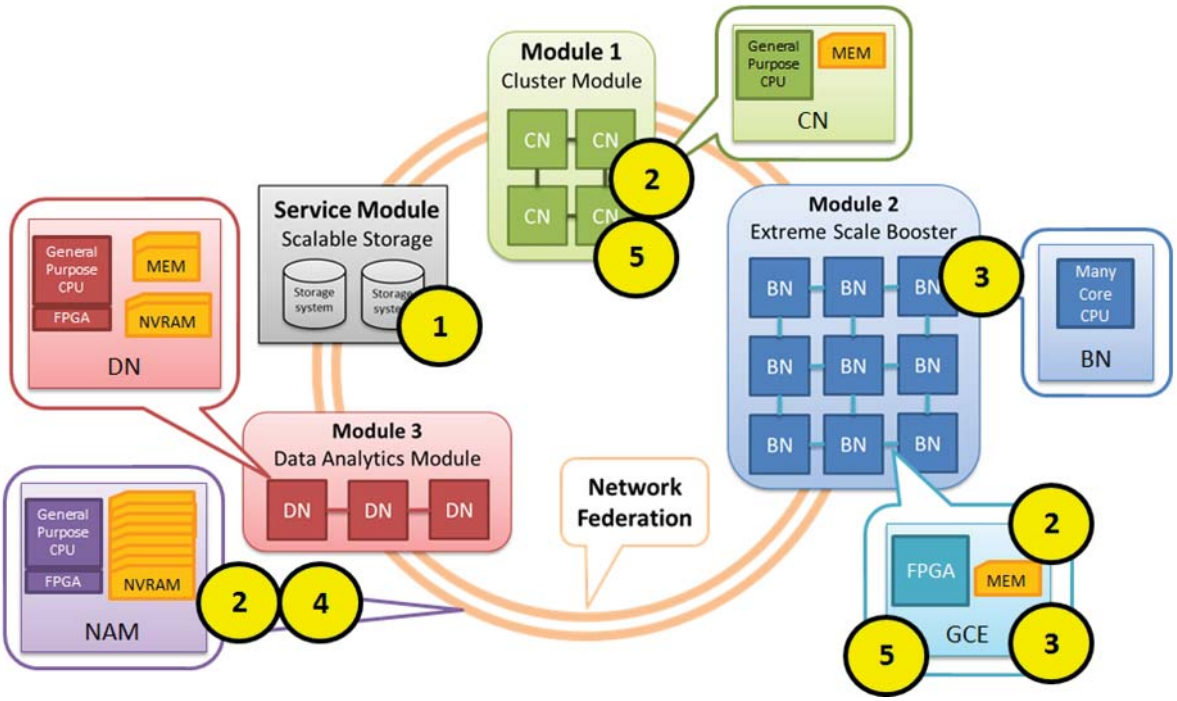


Figure 27: DEEP-EST Workflow for HPDBSCAN (Workload B)

Steps of HPDBSCAN Workload B:

(1) The point cloud dataset is loaded with parallel I/O using HDF5 in the DEEP-EST SSSM and the DEEP-EST Extreme Scale Booster (ESB).
(2) After clustering following Workload A the data reside in the DEEP-EST NAM as a fixed number of levels of importance (w.r.t. detail/scale) introducing serious limitations (w.r.t. level choices and data density jumps between levels).
(3) Selected point cloud LoD studies require continuous instead of fixed levels of importance using importance values of a point regarded as an added dimension to space and time using n-D space filling curves or tree structures whereby the latter may take advantage of MPI collectives using the DEEP-EST GCE enabling small computing modifications to the original clustered datasets in combinations with the highly scalable DEEP-EST ESB.
(4) The different data set results of the various modifications towards continuous levels of importance (CLoI) can be placed in different sections of the DEEP-EST NAM in order to take advantage of a variety of (semi-) continuous and spatio-temporal representations (e.g. zoom-in/out) for scientific studies.
(5) Based on the CLoI data available in the DEEP-EST NAM the DEEP-EST CM could use this information to re-cluster the data using the process of Workload A again but on a modified dataset (e.g. towards space and time) or using different parameters (i.e. MinPoints, or Epsilon-neighbourhood) always avoiding parallel I/O by using intensively memory & networking.

6.1.2 Application requirements

The application is written in C++ (GNU version 4.9.x and up). It mostly uses standard C/C++ libraries but in addition is also requires:

- HDF5 C/C++ library

- OpenMP
- MPI
- ArrayFire

6.1.2.1 Use case description

Perform clustering on very large multi-dimensional point-clouds, e.g. a three dimensional laser scanned point cloud from urban and/or rural areas, given certain input criteria.

6.1.2.2 Benchmarking metric

Time-to-solution is the most significant metric of HPDBSCAN, speedup of that metric is calculated with a varying core count. The only valid measurements are those that produce the correct result, i.e. where each point receives the correct cluster labelling [38].

Additional performance analysis will be performed using the Paraver and Dimemas tools developed by the Barcelona Supercomputing Center (BSC).

6.1.2.3 Scalability

HPDBSCAN has an optimised thread scalability implementation within a node using OpenMP and uses MPI to scale the application across arbitrary nodes. The application scales strongly with a near-linear speedup expected given additional nodes/cores.

Some scalability plots are available in previous publications [38].

6.1.2.4 Modularity

Due to its significant parallel workload HPDBSCAN can profit from the modular structure of DEEP-EST and what the different modules have to offer. The application is moving large chunks of memory between nodes at different points of its execution where some parts can be re-used to speed its execution on different modules. This is all explained in detail in Figure 26 and Figure 27 in section 6.1.1.

6.1.2.5 Communication

As previously mentioned, HPDBSCAN uses both MPI and OpenMP to communicate across nodes and cores, respectively. The communication profile is heavily dependent on the makeup of the dataset that is being processed. Specifically, data is distributed among all nodes using `MPI_Alltoall` and gathered with `MPI_Gather`.

6.1.2.6 Compute

The single program, multiple data (SPMD) approach is used for the computation. The application consists of a single program that distributes tasks to multiple processors that run simultaneously. Additionally, HPDBSCAN will be integrated into the Juelich Machine Learning Library (JUML) where it will use the open source ArrayFire library [39] for additional speedup.

6.1.2.7 Memory

The memory requirements depend on the initial execution parameters, size and make of the dataset. In general, a “standard amount” of memory and bandwidth should suffice. More specifically, the actual memory footprint is O(log(n)) for the indexing step plus O(n/p) for

redistribution where n is the total number of points in the dataset and p is the number of processors used.

HPDBSCAN accesses the memory contiguously and requires at least 2 GByte of RAM per core to process large point-cloud datasets such as the Bremen point-cloud dataset mentioned previously. In order to process even larger datasets HPDBSCAN requires 4 GByte, or even 8 GByte and more RAM per core. The memory access pattern depends heavily on whether the input data set is spatially sorted or not.

6.1.2.8 I/O

HPDBSCAN uses datasets stored in files with the HDF5 format. The size of these files can range from a few hundred MBs to tens of GBytes, depending on the number of dimensions, size and resolution of the point-cloud data. Each point in the dataset uses 4 bytes per dimension (F32), i.e. 12 bytes are required for a single point in three dimensions. At the end of its execution, the application appends the cluster ID labels to the dataset file which increases it by approximately 4 bytes per point in the dataset, i.e. a 3D dataset is turned into a 4D dataset (with the cluster ID label being the added dimension).

6.1.2.9 Monitoring

HPDBSCAN uses system time measurements. Other system monitoring methods are not used to control, steer or optimise its execution. However, additional monitoring methods could possibly be used to analyse the performance.

6.1.2.10 Elasticity

HPDBSCAN is rather a rigid application and will not be able to utilise additional resources that become available during its execution. Moreover, a reduction of cores will be fatal for its execution (see 6.1.2.11). However, it can be executed with an arbitrary number of resources with no practical limit.

6.1.2.11 Resiliency

HPDBSCAN is not a resilient application. It expects the hardware and its access to it to remain static throughout its execution. The cores that are assigned to HPDBSCAN during the start of its execution must remain unchanged until the application finishes its execution, otherwise an error will occur and the allocated resource time will be wasted.

6.2 piSVM

This application employs piSVM, a parallel implementation of the Support Vector Machine (SVM) data classification approach, to classify in our use case hyper-spectral data of natural and man-made land covers. Multiple supervised training models are trained using different kernels and/or datasets, with data preparation and feature engineering, with the goal of acquiring a model with high classification accuracy and a low error-rate. It is expected that a large number of iterations will be necessary before an accurate model is reached, making this a computationally difficult task.

6.2.1 Application Structure

There are basically six steps for a SVM like piSVM:

1. Preparing data and perform feature engineering when necessary.

2. Experiment with the datasets by trying out several kernels and/or datasets and train a SVM model.
3. Test the model in order to check accuracy and/or error rates of the experiment models and go back to (2) if not satisfied.
4. Once settled on a relatively good setup, the SVM model performs cross-validation in order to perform concrete model selection (i.e. pick kernel parameter, regularisation parameters, etc.). Typically the parameters with the best accuracy or lowest error rate are set – note that the dataset gets biased (i.e. slightly contaminated) since we have performed a decision on data.
5. Train the final model again on all data with the selected parameters (i.e. slightly contaminated is ok since we do cross-validation).
6. Test the accuracy to be expected out of sample once deploying the solutions.

The application workflow on the DEEP-EST System will be divided into two parts, workflow parts A and B where the former represents the training/test pipeline and the latter its cross-validation. These workflows (or workloads respectively) are explained in detail in the two following subsections.

6.2.1.1 piSVM Workload A

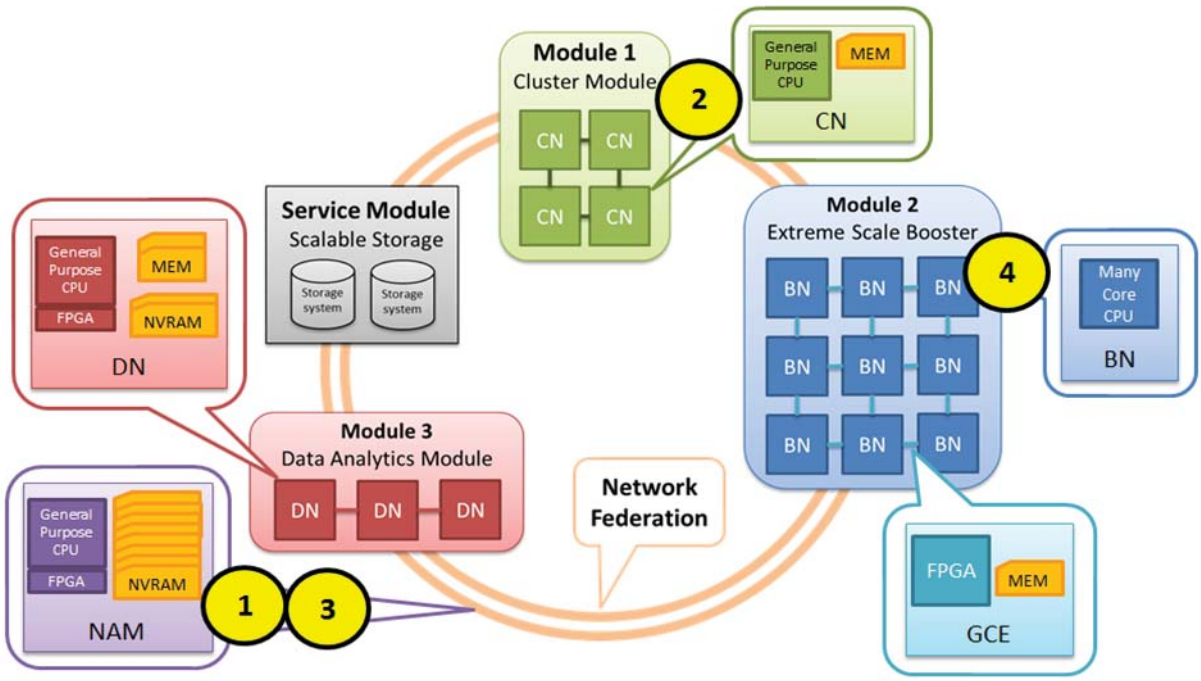


Figure 28: DEEP-EST Workflow for piSVM (Workload A, Training/Test pipeline) (UoI)

Training/Test pipeline, piSVM:

(1) The training dataset and testing dataset of the remote sensing application is used many times in the process and make sense to put into the DEEP-EST NAM.
(2) Training with piSVM in order to generate a model requires powerful CPUs with good interconnection for the inherent optimisation process and thus can take advantage of the DEEP-EST CM (use of training dataset, requires piSVM parameters for kernel and cost).
(3) Instead of saving the trained SVM model (i.e. file with support vectors) to disk it makes sense to put this model into the DEEP-EST NAM.

(4) Testing with piSVM in order to evaluate the model accuracy requires not powerful CPUs and not a good interconnection but scales perfectly (i.e. nicely parallel) and thus can take advantage of the ESB (use of testing dataset & model file residing in NAM).
(5) In case of insufficient accuracy, go to step 2.

6.2.1.2 piSVM Workload B

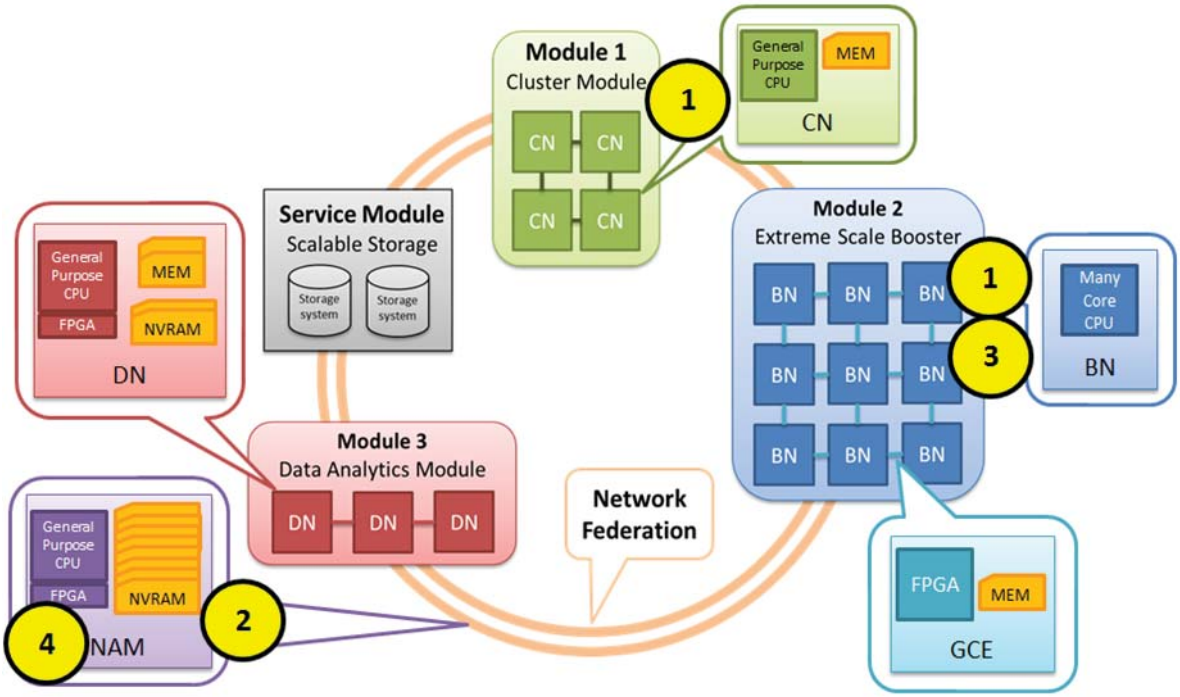


Figure 29: DEEP-EST Workflow for piSVM (Workload B, cross-validation) (UoI)

Cross-validation, piSVM:

(1) Initial experiments performed with training and testing shows that a parameter space search is required in order to perform model selection (i.e. validation).
(2) Validation requires a validation dataset or again the training dataset when using cross-validation (low bias) but instead of reading the training data from a file again and again it can be placed in the DEEP-EST NAM.
(3) n-fold cross-validation over a grid of parameters (kernel, cost) performs an estimate of the out-of-sample performance and performs n-times independent training process on a “folded” subset of the dataset (use of training data in folds). n-fold Cross-Validation (e.g. 10-fold often used) with piSVM is partly computational intensive whereby each fold can be nicely parallelised without requiring a good interconnection and thus can take advantage of the DEEP-EST CM (use of training data in folds) whereby results of each fold per parameter can be put in the DEEP-EST NAM module.
(4) The best parameters w.r.t. MAXIMUM accuracy in all the folds across all the parameter spaces can be computed using the DEEP-EST NAM (FPGA computing maximum).
(5) The best parameter set that resides in the DEEP-EST NAM is given as input to the training/test pipeline (see Workload A).

6.2.2 Application requirements

The application is written in C++ (GNU version 4.9.x and up). It mostly uses standard C/C++ libraries but also requires MPI.

6.2.2.1 Use case description

This application can be used to determine the best parameter set when analysing remote sensing hyperspectral land-cover data from the field of Earth sciences with multiple features selected.

6.2.2.2 Benchmarking metric

It is important that the training produces good model accuracy with a low error rate.
Additionally, the benchmarked code should have a low time-to-solution.

6.2.2.3 Scalability

The node scalability depends on the volume and complexity of the datasets, 80 nodes have been used with results published in [40], but also higher numbers are possible. The scalability also depends on the inherent SVM model (cascade SVM vs. full matrix) that will be explored in DEEP-EST with bigger datasets.

6.2.2.4 Modularity

As explained in Figure 28 and Figure 29 the application can benefit greatly from the modular structure of DEEP-EST and its specific modules. Refer to 6.2.1 and the figures in its sub-chapters for more information.

6.2.2.5 Communication

piSVM uses MPI for inter-process communication, e.g. `MPI_Send` and `MPI_Recv`, with no particular extension. The code is not hybrid at the moment (MPI & OpenMP) but we do intend to support it through the use of OmpSs, which is based on OpenMP, to fully utilise the hardware acceleration possibilities offered by the DEEP-EST Project.

6.2.2.6 Compute

The single program, multiple data (SPMD) approach is used for the computation to train multiple modules in parallel. The application consists of a single program that distributes training tasks among an arbitrary number of cores with no practical upper limit. Computational time is largely dependent on the initial dataset and the acceptable accuracy of the trained model. There is often a trade-off between the number of required computations vs. the model accuracy and/or failure rate (cascade SVM vs. full matrix).

The training phase is massively parallel and both communication- and compute-intensive. Sequential minimal optimisations (SMO) via the inherent serial libSVM are involved to get those Lagrange multipliers that are not approximately zero in order to find the support vectors of a very large matrix that depends on the number of `N` samples. The testing phase is simply inference that follows a nicely parallel pattern. It is therefore less compute intensive and requires almost no synchronisation.

6.2.2.7 Memory

The memory requirement is dependent on the initial data, e.g. number of land cover classes, and the selected number of features being optimised. The application accesses the memory contiguously and requires at least 2 GByte of RAM per core, with even more preferred, i.e. 4 GByte or 8 GByte per core.

The application can benefit from the sense of performing different learning tasks on different modules or using innovative hardware elements (e.g. NAM for datasets that can be re-used during optimisation tasks).

6.2.2.8 I/O

This application uses raw multispectral data stored in files, where values are often stored as space separated strings. Similarly, the output is stored in files. An example dataset used it the Indian Pines AVIRIS dataset [41] over an agricultural site filled with fields with regular geometry (200 spectral bands, 1417x617 pixels, spatial resolution of 20 meter, 52 classes of different land cover). The raw AVIRIS dataset is 830 MByte in size and it is split into two parts: Training data, which is typically 10% of the initial dataset, and testing data containing the remaining 90%.

6.2.2.9 Monitoring

The Dimemas tool developed by BSC will be employed to collect and visualise detailed data about the application's performance.

6.2.2.10 Elasticity

In its current form the application is not elastic. It will not be able to utilise additional resources that become available during its execution, and a reduction of cores will be fatal for its execution. However, it can be executed with an arbitrary number of resources, with no practical limit.

6.2.2.11 Resiliency

The application is not a resilient in its current form. It expects the hardware and its access to it to remain static throughout the training. Due to its iterative form, piSVM, can be non-trivially extended to be more resilient but no such plans exist at the moment. The topic of this possible extension might be re-visited later during the project lifetime if it proves difficult to generate good quality data due to the non-resilient nature of the application.

6.3 TensorFlow

This Machine Learning application employs Deep Learning techniques to classify and extract features automatically from the earth sciences dataset. It´s use is similar to piSVM but more advanced. It is applied to the same datasets and should produce better results.

6.3.1 Application Structure

Generally speaking, TensorFlow performs the following steps:

1. A training graph is built.
2. Training data is loaded.
3. Data is trained.

4. 2-3 is iterated an arbitrary number of times (e.g. 1000).
5. Model is evaluated.

For DEEP-EST two Deep Learning techniques are used, Convoluted Neural Networks (CNN) and transfer learning. These are explained in detail in the two following subsections.

6.3.1.1 Convolutional Neural Network

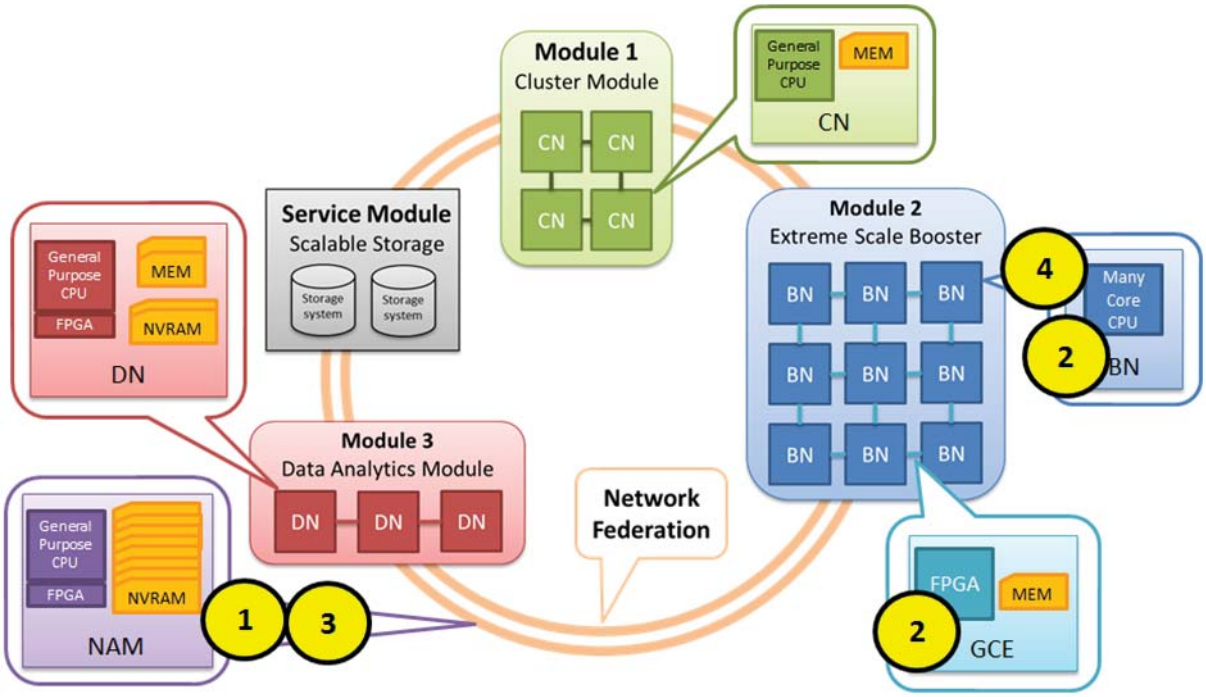


Figure 30: Convolutional Neural Network (UoI)

CNN, TensorFlow:

(1) The training and testing datasets of the remote sensing application are used multiple times in the process and thus it makes sense to put into the DEEP-EST NAM.
(2) Advantage is taken of the MPI collective operations available in the DEEP-EST GCE, and therefore the DEEP-EST ESB, to optimise the process of training the CNNs. Training in TensorFlow works best on CPUs with multiple cores because of the inherent optimisation process based on Stochastic Gradient Descent (SDG).
(3) Trained models of selected architectural CNN setups need to be compared and thus can be put in the DEEP-EST NAM.
(4) Testing with TensorFlow to evaluate the model accuracy works also quite well for many-core architectures and scales satisfactorily. Hence we can take advantage of the ESB to test datasets & CNN models residing in NAM.

6.3.1.2 Transfer Learning

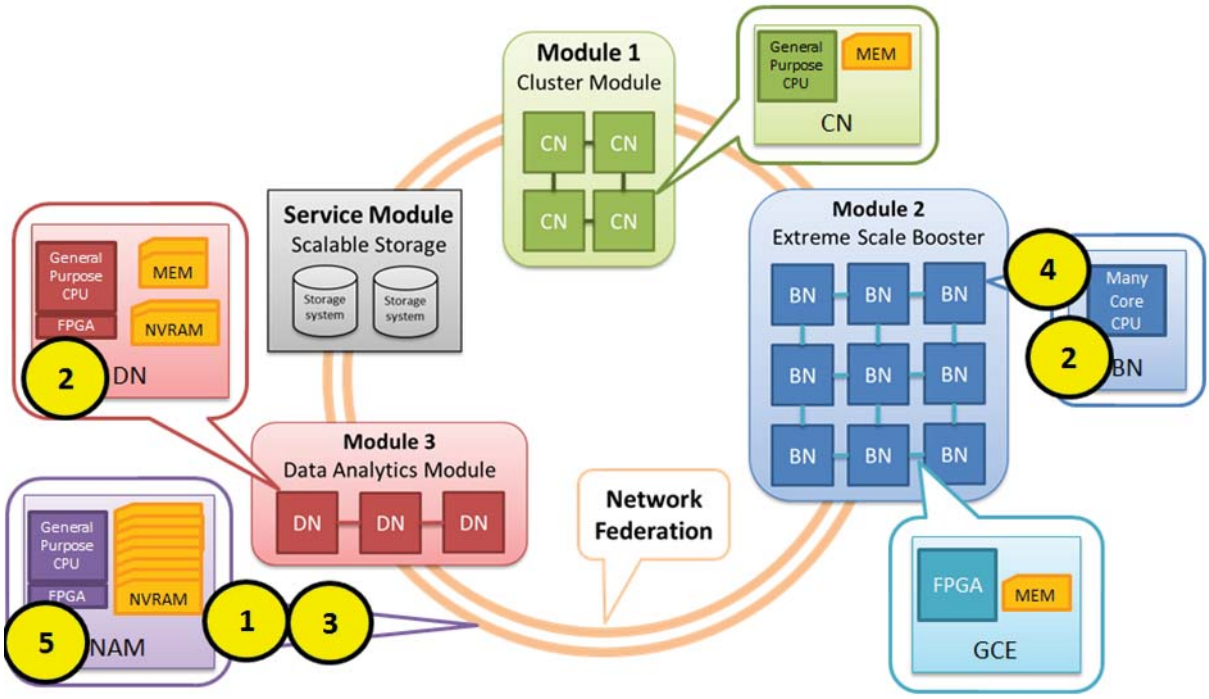


Figure 31: Transfer Learning (UoI)

Transfer Learning, TensorFlow:

(1) Studies have shown that Transfer Learning works especially well for remote sensing data without ground truth or labelled data (i.e. unsupervised) and pre-trained networks trained on general images like ImageNet (e.g. like Overfeat) are available and are put into the DEEP-EST NAM to be re-used for different unsupervised Deep Learning CNN training processes.

(2) Based on pre-trained features another CNN architectural setup is trained with the real remote sensing data whereby the DEEP-EST Data Analytics Module (DAM) is an interesting approach since the FPGA might be used to compute the transformation from pre-trained features as suitable inputs to the real training process of the CNN based on remote sensing data.

(3) Trained models of selected architectural CNN setups that have been used with pre-trained features need to be compared and thus can be put in the DEEP-EST NAM.

(4) Testing with TensorFlow in order to evaluate the model accuracy works also quite well for many-core architectures and scales perfectly (i.e. nicely parallel) and thus can take advantage of the ESB (use of testing dataset & CNN models residing in NAM).

(5) Testing results are written back to the DEEP-EST NAM per CNN architectural design, the FPGA in the NAM can compute the best obtained accuracy for all the different setups.

(6) If accuracy is too low, consider to move back to step (1) in order to change the pre-trained network or step (2) in order to create a better CNN architectural setup based on (another set of pre-trained features).

6.3.2 Application requirements

TensorFlow uses Python to control the execution of a highly-optimised C++ core. The core also uses CUDA when applicable. In most cases researchers only need to use Python.

6.3.2.1 Use case description

The use case is the same as described in sub-chapter 6.2.1 with piSVM. Both applications provide the basis for classifying hyperspectral datasets from airborne and satellite sensors over time (i.e. time series). Deep Learning with TensorFlow can learn features automatically instead of having to rely on manual selection as piSVM.

6.3.2.2 Benchmarking metric

The time it takes to train Deep Learning networks combined with its accuracy will be used as an indicator of good performance. Additionally, speedup between code iterations and the error rate are also performance factors to consider.

6.3.2.3 Scalability

In general, TensorFlow and other tools for Deep Learning do not scale with an increased number of cores. This is, however, improving rapidly as the underlying methods of this recent topic improve with further research. It is expected the application to scale much better in three years' time than is currently possible, both due to the general improvement of the tool and our work on customising it for DEEP-EST.

6.3.2.4 Modularity

The illustrations in section 6.3.1 demonstrate how the application can utilise the DEEP-EST Modules to improve the application.

6.3.2.5 Communication

TensorFlow supports MPI and built-in methods for distributed computing.

6.3.2.6 Compute

The single program, multiple data (SPMD) approach is used for the computation. The application consists of a single program that distributes training tasks among an arbitrary number of cores. Both CPUs and GPUs can be used to train models simultaneously with no practical upper limit. Any BLAS library can be supported as a backend when compiling a TensorFlow application since Eigen version 3.3. or later which TensorFlow extends.

6.3.2.7 Memory

TensorFlow supports SIMD operations on GPU's and Intel CPU's alike which the application can use to improve performance. The overall memory requirements depend heavily on the dataset and its corresponding training model.

6.3.2.8 I/O

The application supports a number of formats for I/O operations. Most likely we will utilise a technique similar to our first application, HPDBSCAN, and employ HDF5 files for both the training and the cross validation data. The size of these files can vary immensely.

TensorFlow itself uses the DataSet API which supports data aggregation from distributed file systems, a feature that we will likely use to further improve the time it takes to train models.

The current CNN's raw input amounts to 11.9 GByte of data and produces a 2.4 MByte output. Concerning possible NAM requirements, the following three-step use-case is produced which also emphasises that TensorFlow training is a repeated process:

(a) Assuming 10 further feature engineering tunings would be done (so 10 datasets) it could be said that the NAM should take 10 x 105 MByte + 10 x 11 MByte for our processed case: ~1.1 GByte requirement for NAM:

(b) Assuming 100 runs would be done, roughly for research and trying out the parameters before doing cross-validation, the NAM should store 100 x 17 MByte in the process from training to testing: ~1.7 GByte.

(c) Cross-validation has in principle no direct file output except an estimate of the sample performance, hence saves not a model file. However, the grid search as a whole should be stored in NAM (e.g. 1 MByte), which is negligible.

6.3.2.9 Monitoring

There exist custom monitoring tools that work exclusively with TensorFlow, such as the "Guild AI" tool that we would use in addition to the more standard HPC monitoring tools such as Paraver and Dimemas developed by BSC. These monitoring tools will be used to collect and visualise detailed data about the application's performance in order to make informed decisions with the purpose of improvement.

6.3.2.10 Elasticity

It is possible but not trivial to adapt the application to a different number of resources after it starts. The application performs the same operations thousands of times, i.e. loading data and applying it to the training model, and it is conceivable to adapt the application between these iterations.

6.3.2.11 Resiliency

TensorFlow offers a built-in fault tolerance mechanism based on checkpoints. Failures in a distributed execution can be detected in a variety of places, such as:

- In a communication between a Send and Receive pair node within the distributed system.
- During periodic health checks.

When a failure is detected, the entire graph execution is aborted and restarted from scratch by default. This loss of work can be mitigated by restoring from a saved checkpoint rather than starting from scratch.

7 Task 1.7: High Energy Physics (Task leader: CERN)

CMSSW

Firstly, CMSSW is the software framework for the Compact Muon Solenoid (CMS) Experiment at CERN. It provides a common foundation for applications, a standard API for all of the subsystems (High Level Trigger, Tracker, Calorimeters, Muon Detectors, etc…) across the CMS experiment for data processing, analysis and monitoring. It also provides a set of abstractions upon which each individual subsystem can build on as well as a common infrastructure for data flow and persistency.

Secondly, CMSSW defines a set of standard workflows that are executed centrally by the experiment as a whole and not on the user basis. However, users can rerun all or parts of these workflows when necessary. Typically, these workflows include the following steps:

- Digitization - RAW Data comes in a very efficiently packed format. Typically RAW data corresponds to the raw detector reading and it requires a special treatment (transformation) in order to obtain actual signals (in units of charge, femto-coulomb or similar) that have been read out by sensors. This step is called DIGI.
- Reconstruction - electrical signals that are being read in and digitized are not the primary physics quantities of interest, however the energy or the location/position are. Therefore, a certain calibration step is needed to convert (transform) DIGI signals into, so called, RECO quantities which carry actual meaningful physics information. For certain subsystems this step involves certain regression procedures, like fitting of the data points to a known functional form.
- “Clusterisation” / Particle Flow - High Energy Physics (HEP) physicists operate on elementary particles, not on charge or energy readings (abstraction on abstraction). Therefore, it is important to provide them with means for clustering a set of readings into a particular elementary particle type:
    - Each subsystem can cluster on its own. For instance the electromagnetic calorimeter can build a photon or electron objects by itself.
    - Information from various subsystems can be combined - Particle Flow. For instance, an electron will leave a track in the tracker system, whereas a photon will not. This provides additional identification/classification information.
    - This step includes possible Machine Learning (ML) classification.
    - The Clustering step can be rerun. Some physics groups choose non-standard algorithms for clustering, therefore the first step of the data analysis might involve re-clustering.
- Triggering - The concept of a trigger is the corner stone of the HEP system as it selects the events that are going to be read out. Typically this part is extremely high-performing and optimised. The range of algorithms that can be used in this step varies significantly, from simple computations to regression and inference.

To summarise all of the above, CMS software framework, code-wise, is responsible for simulation, data processing, analytics and high-performance triggering algorithms.

Data Analytics with Apache Spark

The second piece that will be tested is the physics data analytics with Apache Spark and Hadoop-stack. By Hadoop-stack we refer to various open source Big Data solutions available for data processing/queuing/storage/analytics/etc. This use case involves only data analytics pipelines: Data/Feature Engineering + ML, and query systems. It does not involve the processing of standard CMSSW workflows - the idea is to take the data that has already been prepared centrally and provide an analytics platform for the final user - a physicist.

7.1 Application structure

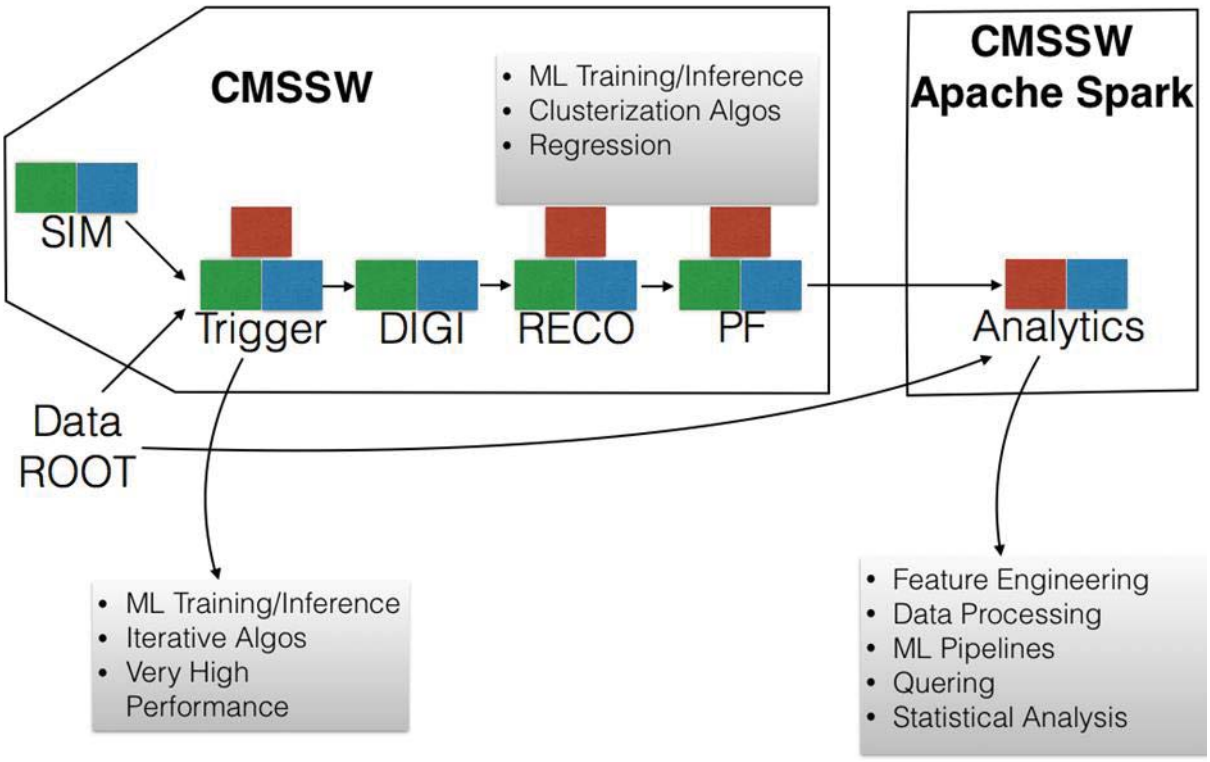


Figure 32: Workflow CMSSW (CERN)

The CMS software framework is a multi-threaded framework, not a multi-processing one, therefore the communication among different physical nodes is avoided by definition. Within the CMS experiment, a different platform is responsible for distributing the workflows among the physical nodes and communicating the results of the execution/processing. The workload distribution (per node) is done on an event-basis, which means that different nodes process completely different sets of events.

The initial idea is to distribute the computations on the DEEP-EST Prototype targeting a particular hardware component with a particular algorithm/processing step, and benefiting from the specialised capabilities of the DEEP-EST Architecture. In Figure 32, the high level overview of various steps involved is provided. This graph should be read from left to right. These steps show the flow of data as it gets transformed from raw sensor/simulated data into high level objects used for the actual analysis. Various algorithms are involved at all steps,

from simple calculations to regression and ML pipelines (data transforms + inference). The “Trigger” step requires ultimate computation performance and can utilise various kinds of accelerators (GPUs/FPGAs), with a possibility to fall back on CPUs. It is important to note that, in general, software is going to be flexible to optimally select the hardware to run a particular algorithm on. The analytics step is I/O bound (depending on what is done exactly), so that caching techniques + huge memory would allow faster in-memory data processing. It will benefit from the Data Analytics Module (DAM) and the ESB.

Currently CMS has approximately 3000 users that run various data analysis workflows on the CERN grid. It is important to mention that one of the goals at CERN is to generalise the approach and create a reduced amount of common data-analysis applications for CMS’ users. For benchmarking, it is planned to select a few different particular analysis-cases (Higgs Analyses), to reflect the fact that HEP-data analysis is always a multi-user scenario. Moreover, one of the main goals of this application software is to be able to optimally handle the multi-user scheduling.

There are, however, particular workflows that are typically run centrally under a single user, such as the Track Fitting Algorithm at the Trigger level or Particle Flow Algorithm, which will utilise both the ESB and HPC Cluster Module (CM) (and eventually the DAM, too) and should benefit from such heterogeneity.

Finally, it is planned to test Apache Spark as a platform targeting the final data analysis. The idea is to explore ways to separate the structured, central high performance data processing (CMSSW), from the analytics pipelines. The usage of the ESB and DAM is foreseen for this part of the application. Again, this depends on the particular algorithm being run. For instance, statistical analysis tends to be more CPU bound, but Query Systems are generally I/O bound.

CMS data analysis workflows will be improved combining HPC and Big Data components. The goal is to demonstrate a proof-of-concept in which analysis objects can be dynamically refreshed when detector calibration parameters are modified. This would require subsections of CMSSW to be executed in a deterministic way, and data objects to be updated when accessed during an analysis workflow. This concept is referred to as *data transformation*, and is in the exploration phase. It has the potential to make data processing more dynamic and more efficient as only objects that need to be accessed would be updated.

7.2 Application requirements

7.2.1 Use case description

HEP Data Analytics for the Compact Muon Solenoid (CMS) experiment on heterogeneous resources is the primary objective of our use case. The precise configuration parameters and characteristics will be described in detail in D1.2. This application performs various data transformations and applies clustering, regression and classification techniques to draw conclusions about high level physics objects.

7.2.2 Benchmarking metrics

The primary metric of interest for both CMSSW and Spark workloads which is planned to be optimised, is the throughput - the number of events processed per unit of time (second) without loss of precision.

In addition, minimising energy consumption implies lower overall computing costs and constitutes therefore another metric of interest.

7.2.3 Scalability

Overall, since no communication across nodes is required (at the moment), node scalability is trivially established for CMSSW workloads. For thread scalability, currently CMSSW jobs run on the grid with 4 threads. There are two reasons for not going to higher counts: first, a given node will certainly have other CMSSW processes running at the same time; second, heavy I/O from/to a single file should be avoided. The latter point is currently being addressed. In summary, the load per node will consist of multiple independent identical CMSSW processes. The only difference in them is that each will be processing different rows of data.

For the Apache Spark analytics applications, scalability strongly depends on the workload (i.e. what the actual query is doing) and the amount of shuffling needed. Figure 33 shows preliminary results obtained from using a Hadoop Cluster of 14 nodes with 36 physical cores each. To perform this test 1.2 TByte of CMS public data was used distributed across more than 1000 files. The execution time was measured for six different queries for two different cases: first, varying the number of executors, but loading each one with two tasks (threads) per core; second, keeping the total number of executors at maximum and varying the number of threads per node. Although these results are preliminary, basic strong scalability properties can be observed.

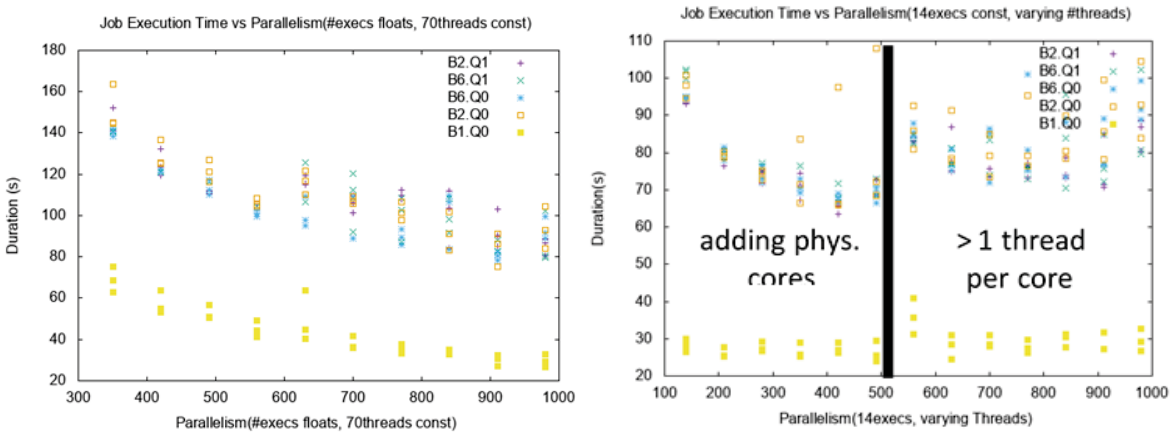


Figure 33: Scaling behaviour of the Apache Spark analytics (CERN)

7.2.4 Modularity

The graphical representation of the Data Flow in Figure 32 explicitly shows the modularity of the framework. CMSSW incorporates a plugin management framework that loads only plugins (shared libraries) that are specified.

7.2.5 Communication

At this point, no communication across physical nodes within CMSSW is done. The workload is distributed across different nodes and processes are independent from each other. However, targeting heterogeneous resources is considered moving (updating) to a framework that is aware of the resources available and can potentially offload tasks (kernel) to an accelerator (not on the same node).

For the case of analytics with Apache Spark, the exact numbers on the amount of communication are not yet known. Further analysis will provide more insight, to be documented in upcoming deliverables (D1.2 and D1.3).

7.2.6 Compute

The compute requirements are not particularly demanding. CMSSW can be used on standard Intel CPUs built with gcc compiler. However, since the goal is to optimise the heterogeneous workload, accelerator units are required (both FPGAs and GPUs are possible), aiming at using the available processing units efficiently.

In the CMSSW workloads, algorithms of varying complexity will be employed. For a given LHC collision, simple loop iterations with float operations, simple vector or matrix operations, regression procedures (like fitting), or a complex cellular automaton can all be used. The reason for such a wide range of algorithms and operations is the varying complexity of the components that constitute the CMS detector.

For the Apache Spark analytics applications, workloads can be split into two categories. First, the data preparation with the feature engineering step in which a range of “simple” queries will be benchmarked. These mainly consist of simple manipulations of collections (filter, map, groupBy, reduce), vector or matrix operations. Although each of them is simple, once combined, these components create a complicated query plan. Second category builds up on the first one and includes the Extract Transform Load (ETL) pipelines together with Machine Learning (ML) algorithms. As a baseline, two ML algorithms will be used: a Boosted Decision Tree and a Deep Learning model.

7.2.7 Memory

Each CMSSW process requires at least 1-2 GByte memory. This number goes up depending on the collision environment and can go as high as 4-8 GByte for higher pile up values (number of simultaneous bunch crossings). For Data Analytics with Apache Spark, which run in-memory, caching the whole dataset would be the optimal solution.

For the CMSSW workloads, most frequently memory access patterns are sequential and linear. Most of the data types are heap allocated collections (typically vectors) of composite objects and the most basic query to perform is to sweep through the whole array and filter out what is not needed.

Precise access patterns for Apache Spark workloads are not known yet, but the data structures are the same as in the case of CMSSW.

7.2.8 I/O

ROOT [42] is a column data format and is a standard within the HEP community. Comparing to Big Data solutions, it provides similar functionality to Apache Parquet, Apache Avro, or HDF5. Reading separate columns of data independently and being self-descriptive are among the ROOT's features.

Typical size of a single input ROOT-file for both CMSSW and Apache Spark data processing fall in the range 0.5 GByte to 3-4 GByte. The size of the datasets (a dataset is a logical collection of files) to be processed range from several TBytes to PBytes. The size of a single event (on disk) is on the order of 1 MByte.

In DEEP-EST, it is foreseen to use in total up to tens of TBytes for the CMSSW workloads, by using smaller benchmarks. The workload here will be more computationally expensive.

For the Apache Spark workloads, it is planned to use at most several hundreds of TBytes of data. Again, the same kind and size of input file is used, this time with just more files.

There is one important feature of the addressed data: most of Big Data data formats are column and therefore allow for column pruning. That means that a column that a specific query does not need to read will not be read at all. Therefore, the input file sizes can be reduced by removing the irrelevant columns from the datasets. This approach will be targeted by applying a pre-processing step before copying data to the Scalable Storage Support Module (SSSM) at JUELICH. In the production scenario CMS jobs are not always scheduled to run where data is located and often jobs will be streaming large amounts of data from some other site on the grid. However, to simplify the workflow, in DEEP-EST the pre-processed data is planned to be stored directly on the SSSM.

7.2.9 Elasticity

Apache Spark utilised with resource managers like Hadoop Yarn or Apache Mesos is, by definition, elastic in terms of resource usage. It remains to be decided whether and how these can be combined with the SLURM scheduler used in DEEP-EST.

For the CMSSW processes that is not quite the case. The current implementation sets up an Intel TBB scheduler and uses a pool of threads of constant size.

7.2.10 Resiliency

No checkpointing is currently being utilised.

8 Global conclusion

The objective of this document is to collect all important application requirements in order to give the hardware and system-software work packages sufficient information to start taking their architecture and design decisions. This conclusion gives a brief summary of the most important requirements.

From the current analysis, it is clear that the applications will have specific and different use cases, most of them involving multi-step workflows. Some applications even propose multiple use cases. A full and complete specification of these will be contained in Deliverable D1.2 in project month 9.

Each application proposes ways to distribute the work over the modules of the DEEP-EST MSA as described in the “Application structure” and “Modularity” sections. Some of them even presented multiple alternative ways to do so. This shows that across the application space, the specific value of the MSA is seen and appreciated. For each application and use case, the detailed specification of the distribution strategy will depend on a deeper application analysis, and the results will be reported in D1.3 in project month 12.

For most applications, the primary metric of success is application turnaround time – or synonymously time to a correct solution. Applications with real-time needs such as the ASTRON correlator must achieve a certain throughput and will use the number of nodes or energy required to achieve this as a success metric. Machine Learning use cases (Deep Learning and UoI’s piSVM) require good accuracy (in training and scoring) and strive to optimise time and energy needed to achieve this accuracy.

It is obvious that scaling behaviour and targets strongly depend on the application and the specific use case. Some use cases will use strong scaling, others exploit weak scaling, and for some Deep Learning workloads, ensemble scaling[2] is interesting. Objectives of scaling are also application-specific, as shown by maximum-filling scaling for NEST and the need to provide the required throughput for ASTRON and CERN use cases.

The applications use a variety of languages (C, C++ and Python) and high-level programming frameworks or libraries, for example TensorFlow (for Deep Learning), PETSc (for the field calculation of xPic) and MUSIC (for coordinating NEST and Arbor simulations in the NMBU use cases). CERN proposes to build their Data Analytics use case using the Apache Spark framework, which introduces its own communication, execution and management stacks.

The requirements in terms of inter-process communication are different. A large group of applications relies on MPI, which is the standard communication interface for HPC. Other applications like ASTRON's correlator and CERN's CMSSW process streaming data (and can use TCP or UDP), and the CERN Spark use case relies on the ZeroMQ messaging layer running on top of IP. Frameworks like PETSc and MUSIC rely on MPI. Finally, to find the best way to strongly scale training using the TensorFlow Deep Learning framework is still an active research topic – first MPI versions are available, yet performance has to be improved.

[2] Ensembles are sets of independent simulations (f.i. with varying parameters or constraints) , which are combined to create a desired result; this approach is common practice in fields like weather forecasting and computational engineering, and is also evaluated as a way to speed up DL training. Ensemble scaling refers to increasing the number of simulations run in parallel while computing an ensemble.

Quantitative data on communication and I/O depends strongly on the different use cases; this data will become available during the targeted application characterisation exercise for the benefit of D3.1 and then in full detail after the performance tools training in late November 2017.

A general characterisation of the computational and memory requirements is available for most applications, with quantitative data scheduled to become available towards the end of 2017.

Since there exist some initial ideas on how to distribute the applications over the DEEP-EST MSA, a first set of suggestions on how the modules could look like were presented by the application developers:

- Several applications were already ported and (partly) optimised for KNL (e.g. KULeuven, NMBU, and ASTRON). These optimisations (like hybrid parallelism and use of SIMD) will bring benefits for the CM and ESB in DEEP-EST.
- Parts of other codes, on the other hand, are already ported to GPUs (Gromacs, the correlator and imager, TensorFlow …). Therefore, this technology also could be a candidate to equip either the DAM or the ESB.
- For some applications and its software components the potential use of FPGA based acceleration might be beneficial (e.g. ASTRON, NCSA, ML/DL) and will be evaluated during the project.
- For the CM, all requirements could be satisfied with CPUs providing a high single thread performance.
- The required memory per core/node seems to be achievable with all technologies that are in discussion for the DEEP-EST Prototype.

For ASTRON the per-node data ingestion bandwidth is important. For the correlator, 200-300 Gbit/s are expected to be required in the future. DEEP-EST will investigate whether the full bandwidth can indeed be integrated on the prototype under the given budget constraints and without compromising other applications. If necessary, a scaled-down use case can be employed.

Regarding the storage capacity (e.g. for input and output files) the requirements start at a few MBytes reaching up to a few TBytes. More detailed data will be available with the complete use case definition.

Several applications have expressed specific interest in making use of innovative DEEP-EST System features like NVM, NAM and GCE. As an example, some Machine Learning use cases (HPDBSCAN, piSVM and TensorFlow from UoI) will explore to utilise both the NAM and GCE components, and NVM (as attached I/O devices or as storage class memory) can serve to store application snapshots (NEST from NMBU) or checkpoints. Since these applications will run on modules across the full system, it is important to have fast access to the NAM from anywhere in the DEEP-EST Prototype.

Regarding I/O functionality, applications do either use the standard POSIX I/O interfaces, or rely on higher-level libraries such as SIONlib, pHDF5, ROOT I/O and CasaCore.

The DEEP-EST Applications employ different resiliency strategies. TensorFlow and Gromacs use in-house checkpointing. Others such as xPic and the imaging pipeline of ASTRON use

the resiliency features further developed in the DEEP-ER Project (e.g. SCR). Finally, other applications like CMSSW or NEST do not employ resiliency mechanisms as of today. The former has expressed interest to used storage-class memory to store application snapshots (i.e. memory images) on storage-class memory. The CERN Data Analytics use case can profit from Apache Spark's built-in resiliency features.

Most of the applications are flexible regarding the number of nodes/cores they can run on, yet they do expect the resource allocation to stay constant while they are running. In particular, they cannot profit from resources being added during execution, and would fail if resources are taken from them. Possible exceptions are those applications that use multithreading on each node and do not depend on the actual number of cores available. Applications using OpenMP tasking, TBB or OmpSs are examples.

At the co-design workshop (end of September 2017), the concept of elasticity was discussed in terms of workflows, which graphs of steps, each one potentially being a highly complex, parallel application. As such a workflow is progressing, the resource requirements may vary, and allocations can be changed accordingly. To take advantage of this, the execution of steps could be orchestrated by the DEEP-EST System workload manager (SLURM), with exactly matching resources being handed out to each step for its duration. An alternative approach would use SLURM to give a set of resources to a separate workflow orchestration component, which in turn executes the workflow, giving subsets of these resources to the steps as they are executed. The decision on which way to proceed with regard to the DEEP-EST Prototype will be reached in WP5.

8.1 Next steps

The next step is to carry out an in-depth analysis of all applications, using a set of proven profiling and tracing tools (e.g. BSCs Extrae/Paraver). In order to familiarise the application partners with these tools a training event at BSC is planned for end of November 2017. The overall plan for additional trainings and workshops for other topics is described in [43].

In addition, JUELICH, BSC and Intel are working with the code developers to perform a first application characterisation during November, relying on first traces obtained from the applications. This data will influence Deliverable D3.1 at project month 6.

The analysis phase of WP1 will provide traces of the applications and benchmark use cases to WP2 (this is also the topic of Deliverable D1.2 at project month 9). As a result of this detailed analysis, each application partner will develop its strategy to use the DEEP-EST Prototype in the most efficient way to be presented in D1.3 (project month 12).

9 Bibliography

[1] DEEP Projects Consortium, “DEEP Projects,” [Online]. Available: http://www.deep-projects.eu/. [Accessed 20 October 2017].

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] H. E. Plesser, J. M. Eppler, A. Morrison, M. Diesmann and M.-O. Gewaltig, “Hybrid MPI-thread parallelization: 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.

[9] 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.

[10] A. Morrison, A. Aertsen and M. Diesmann, “Spike-timing dependent plasticity in balanced recurrent networks,” in *Neural Comput. 19, 1437–1467. doi: 10.1162/neco.2007.19.6.1437*, 2007.

[11] E. Hagen, D. Dahmen, M. L. Stavrinou, H. Lindén, T. Tetzlaff, S. J. van Albada, S. Grün, M. Diesmann and G. T. Einevoll, “Hybrid Scheme for Modeling Local Field Potentials from Point-Neuron Networks,” in *Cereb. Cortex, doi:10.1093/cercor/bhw237*, 2016.

[12] “HybridLFPy,” [Online]. Available: https://github.com/LFPy/hybridLFPy.

[13] “Arbor: The Arbor multi-compartment neural network simulation library. [Online],” [Online]. Available: https://github.com/eth-cscs/arbor. [Accessed 18 10 2017].

[14] “MUSIC: The Multisimulation Coordinaor,,” [Online]. Available: https://github.com/INCF/MUSIC.

[15] “Elephant—Electrophysiology Analysis Toolkit,” [Online]. Available: http://elephant.readthedocs.io/en/latest/.

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

[17] T. C. Potjans and M. Diesmann, “The cell-type specific cortical microcircuit: relating structure and activity in a full-scale spiking network model,” in *Cereb. Cortex 24, 785–806. doi: 10.1093/cercor/bhs358*, 2014.

[18] M. Schmidt, R. Bakker, K. Shen, G. Bezgin, C. Hilgetag, M. Diesmann and S. J. van Albada, “Full-density multi-scale account of structure and dynamics of macaque visual cortex,” in *arXiv:1511.09364v3*, 2016.

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

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

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

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

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

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

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

[26] U. Essmann, L. Perera, M. L. Berkowitz, T. Darden, H. Lee and L. G. Pedersen, “A smooth particle mesh Ewald method,” 1995.

[27] C. Kutzner, R. Apostolov, B. Hess and H. Grubmüller, “Scaling of the GROMACS 4.6 molecular dynamics code on SuperMUC,” [Online]. Available: https://www.mpibpc.mpg.de/14613164/Kutzner_2014_ParCo-conf2013.pdf.

[28] F. Affinito, A. Emerson, L. Litov, P. Petkov, R. Apostolov, L. Axner, B. Hess, E. Lindahl and M. F. Iozzi, “Performance Analysis and Petascaling Enabling of GROMACS”.

[29] J. W. Romein, “Application description ASTRON,” 2017. [Online]. Available: https://bscw.zam.kfa-juelich.de/bscw/bscw.cgi/2411360?op=preview&back_url=2337350.

[30] “CasaCore,” [Online]. Available: https://github.com/casacore/casacore.

[31] J. W. Romein, “A Comparison of Accelerator Architectures for Radio-Astronomical Signal-Processing Algorithms,” in *Int. Conf. on Parallel Processing (ICPP'16)*, Philadelphia, PA, 2016, pp. 484-489.

[32] B. Veenboer, M. Petschow and J. W. Romein, “Image-Domain Gridding on GPUs,” in *IEEE International Parallel and Distributed Processing Symposium (IPDPS'17)*, Orlando, FL, 2017, pp. 545-554.

[33] P. Prasad, F. Huizinga, E. Kooistra, D. van der Schuur, A. Gunst, J. Romein, M. Kuiack, G. Molenaar, A. Rowlinson, J. D. Swinbank and R. A. M. J. Wijers, “The AARTFAAC All-Sky Monitor: System Design and Implementation,” *JAI,* vol. 5, no. 4, pp. 1641008-1 - 1641008-17, 2016.

[34] A. R. Offringa and et al., “WSClean: an Implementation of a Fast, Generic Wide-Field Imager for Radio Astronomy,” *Monthly Notices of the Royal Astronomical Society,* vol. 444, no. 1, pp. 606-619, 2014.

[35] V. Campos, F. Sastre, M. Yagües, J. Torres and X. Giró-i-Nieto, “Scaling a Convolutional Neural Network for classification of Adjective Noun Pairs with TensorFlow on GPU Clusters,” in *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (pp. 677-682)*, 2017.

[36] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, ... and S. Ghemawat, “Tensorflow: Large-scale machine learning on heterogeneous distributed systems,” *arXiv preprint,* vol. arXiv:1603.04467, 2016.

[37] “HPDBSCAN Benchmark test files,” B2SHARE, 13 January 2017. [Online]. Available: https://b2share.eudat.eu/records/7f0c22ba9a5a44ca83cdf4fb304ce44e.

[38] M. Goetz, C. Bodenstein and M. Riedel, “HPDBSCAN – Highly Parallel DBSCAN,” in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC 2015, Machine Learning in HPC Environments (MLHPC) Workshop.

[39] [Online]. Available: https://github.com/arrayfire/arrayfire.

[40] G. Cavallaro, M. Riedel, M. Richerzhagen, J. A. Benediktsson and A. Plaza, “On Understanding Big Data Impacts in Remotely Sensed Image Classification Using Support Vector Machine Methods,” in *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, Issue 99*, 2015, pp. 1-13.

[41] M. Riedel, “piSVM1.2.1 Analytics Training (10x-CV optimized) Indian Pines Images Processed 30 Features 52 Classes,” B2SHARE, 22 December 2016. [Online]. Available: https://b2share.eudat.eu/records/37c30138588f4ae6a806f3142d696f81.

[42] “ROOT,” CERN, [Online]. Available: https://root.cern.ch/. [Accessed 17 10 2017].

[43] S. Eisenreich, “Communication plan, toolkit and owned channels,” in *Deliverable 7.1, DEEP-EST Project*, 2017.

[44] J. Schmidt, “Report on projections and improvements for the DEEP/DEEP-ER concept,” in *Deliverable 7.2, DEEP-ER Project*, 2017.

[45] Hagen et al, “Cereb Cortex,” in *doi: 10.1093/cercor/bhw237*, 2016.

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

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

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