Horizon 2020
FET HPC RIA Project #671553

ExaNeSt – European Exascale
System Interconnect and Storage
http://www.exanest.eu/

Deliverable **D2.2**

Requirement analysis (Network and Storage) and porting roadmap

version 1.1 – 12 Genuary 2017

EDITOR, CONTRIBUTORS

Partner	*Authors*
INAF	*Lead Editor: Luca Tornatore* *Other Contributors: Giuliano Taffoni, Giuseppe Murante, David Goz*
INFN	*Elena Pastorelli, Pier Stanislao Paolucci, Piero Vicini*
EXACT-LAB	*Giuseppe Piero Brandino, Stefano Cozzini*
ENGINSOFT	*Gino Perna, Stefano Bridi*
MDBS	*Ying Zhang (assisted by Panagiotis Koutsourakis, Richard Koopmanschap, Pedro Ferreira, Niels Nes, Martin Kersten and Sjoerd Mullender)*
REVIEWERS	*Stefano Borgani (INAF), Paul Carpenter (Barcelona Supercomputing Center), Dirk Pleiter (Forschungszentrum Jülich)*

<u>REVISION HISTORY</u>

Version	*Date*	*Description*
0.1	02 Nov 2016	The first skeleton of the document.
0.2	08 Nov 2016	First contribution by MonetDB inserted
0.3	11 Nov 2016	First contribution from INAF inserted
0.4	14 Nov 2016	Other contributions added by EnginSoft, Exact-LAB and INAF
0.5	15 Nov 2016	Other contributions added by EnginSoft, INAF and INFN; preliminary version of section 5 based on sections 2, 3 and 4
0.9	15 Nov 2016	All contributions added; GANTT view added in section 5
0.95	21 Nov 2016	Reviewers' comments received; document modified accordingly
1.0	28 Nov 2016	Version 1.0 released
1.1	12 Gen 2017	Version 1.1 released – same as version 1.0 but publicly available

Table of contents

LIST OF ABREVIATIONS

CSV	*Comma Separated Value*
CP	*CheckPoint*
ccNUMA	*cache-coherent Non-Uniform Memory Access*
DBMS	*Database Management System*
D*X.Y*	*Deliverable X.y*
GAS	*Global Address Space*
KVM	*Kernel-based Virtual Machine*
MA	*Mini-Applications*
MPI	*Message Passing Interface*
NUMA	*Non-Uniform Memory Access*
NVM	*Non-Volatile Memory*
OS	*Operating System*
PGAS	*Partitioned Global Address Space*
QFDB	*Quad FPGA Daughter Board*
RDMA	*Remote Direct Memory Access*
VOSYS	*Virtual Open System (MDBS partner)*
WP*X*	*Work Package X*
T*X.Y*	*Task X.y*

<u>SUMMARY</u>

The partners participating in WP2 have identified the applications suitable to contribute to development and evaluation of the technology platform, as reported in D2.1.

The core of T2.2 of WP2 consists in porting those identified applications to the ExaNeSt Prototype.
This document details the plan for this action.

As first, the preliminary basic requirements that the applications pose to the platform are assessed in terms of system attributes. Those are both on the hardware, such as storage and network requirements, and on system software requirements, such as operating system, compilers, extensions (e.g. vectorization capabilities or openMP/openCL API level), low level libraries (message passing, e.g. MPI, multithreading, e.g. pthreads or standard different than POSIX).

Then, for each application the partners will describe the porting strategy to adapt them,- or their core algorithms and tasks, to the new target platform.

The activity, in the co-design perspective, involves an incremental and iterative approach with continuous interactions with WP3, WP4 and WP5.

This document details this porting strategy and the foreseen roadmap for it, keeping in mind that the final deliverable of T2.2 consists of a series of performance benchmarks to validate the network and storage infrastructure as well as the performance of the overall infrastructure.

1. Introduction

This deliverable delineates the requirements analysis, porting strategy and the roadmap to port the applications identified in D2.1.

We address the reader to Section 2 of D2.1 as for applications' names and descriptions.

1.1 Porting Strategy approach

Applications identified by partners in D2.1 are categorizable in two distinct types – although all of them are open source and publicly available – as follows.

Some of the applications (namely: *GADGET, PINOCCHIO, DPSNN-STDP, MonetDB*), are actively developed by the participants that fully master their entire architecture, dataflow and algorithms.

Other applications (namely: *LAMMPS, RegCM, SWIFT, OpenFOAM, SailFish CFD*) are, instead, just professionally, although insightfully, known and used by the participants.

This reflects the real-world situation in which largely diversified users have access to the HPC platforms that must fit as much as possible with different needs and requirements without requiring the users to modify the codes.

In turn, the vast majority of current available codes for HPC have been designed and developed with a substantially different paradigm in mind – limited number of CPUs (i.e. ~10^3) each with a limited number of cores, distributed memory (few GB per CPU), almost exclusive use of MPI and little or no multi-threading – which in turn reflect on the "procedural" nature of the codes. Instead, exascale architecture and million-tasks paradigm with hierarchical and mixed shared-distributed memory model, intrinsically open new perspectives and compel to profoundly re-design and reengineer the codes.

Hence our ensemble of applications offers the opportunity of fully exploiting the co-design concept.

In fact, on the one hand we have the chance to conceive, develop and test new architectures and designs for some codes, natively tailoring them to the exascale perspective. On the other hand, we can define the engineering of the new platform keeping in mind key features and requirements that come from widely used, state-of-the-art codes.

At the same time, core algorithms and tasks will be extracted from all applications and set as mini-apps that will be used as benchmarking units. This means that single algorithms and implementation will be improved in such a way to increasingly fit to the exascale target system, while also being used to test, benchmark and validate the platform prototypes, in an incremental and iterative process.

It is worthwhile mentioning here that particular attention will be given to exploit *(1)* the acceleration potential due to FPGAs, through writing OpenCL kernels for the most computational intensive parts; and *(2)* the impact that UNIMEM concept can have on code's performance.

We denominate this aforementioned activity as *"porting"*.

This document is structured as follows.

In Sections 2 and 3 we will report about key features required by the applications in terms of inter-connet and storage.

In Section 2 we will summarize the system-level requisites that must be fulfilled on the basis of applications' needs in terms of e.g. operating system, compilers, compilers extensions and capabilities, low-level (e.g. message-passing, threads) and high-level (e.g. fft, hdf, scientific) libraries.

In Section 3 we will summarize the results of applications' analysis in terms of network and storage requirements.

Section 4, which is the core section of this document, will describe the details and roadmap for the porting activity for each application.

In Section 5 we will provide a comprehensive view of the main points in the report.

Finally, in Section 6 we recall the public links to all the code mentioned in this document.

1.2 Project milestones

Several actions listed in the porting roadmap depend on the progressive availability of new hardware and software features offered by ExaNeSt WP3, WP4 and WP5. For instance, the availability and usability of FPGA and NVM and the availability of OpenCL standards.

Moreover, as it will be detailed in the following sections, the optimization of the MPI library on the platform interconnect network and UNIMEM is crucial for the porting and development activity.

The availability of a first release of such optimized version of MPI should start from Month 22, as recently stated in a joint meeting among ExaNeSt, ECOScale and ExaNoDe projects that has been held in Dublin (2016, November 7th).

The precise terms of the agreement will be stated in the MOU that represents the outcome of that meeting.

Finally, we remind that some details about the exact timing of the planned porting roadmap are in fact based on the schedules of other WPs and on the release of the optimized MPI library.
Such dependencies will be explicitly highlighted where appropriate.

1.3 Contributions

All sections about "MATERIAL SCIENCE" and "CLIMATE SCIENCE" are contributed by EXACT-LAB.

All sections about "ASTROPHYSICS" are contributed by INAF.

All sections about "ENGINEERING (CFD)" are contributed by EnginSoft.

All sections about "NEUROSCIENCE" are contributed by INFN.

All sections about "DATABASE" are contributed by MonetDB Solutions.

2. System-level Requirements

A summary of the requirements from every single code will be presented in Section 5.1

2.1 MATERIAL SCIENCE

2.1.1 LAMMPS

LAMMPS is a Molecular Dynamics package that support a broad range of architectures and operating systems. It can be compiled on any Linux distributions and it supports both message passing paradigm (using MPI) and shared memory systems (using OpenMP) as well as accelerators through both CUDA and openCL.

The system level requirements are

Hardware: no specific hardware is required to run. Optional accelerators (OpenCL supported)

OS: *nix-like system, POSIX compliant.

Compilers: C++98 compliant compiler

Optional support for OpenMP 2.0

Low-level libraries required: MPI 1.2 compliant library (optional but suggested support to MPI 2.0 for parallel I/O)

2.2 CLIMATE SCIENCE

2.2.1 RegCM

RegCM is a Regional Climate code actively used in Climate research. It can be compiled on any Linux distribution and it is based on the message passing paradigm for parallel execution using MPI standard.

The system level requirements are

Hardware: no specific hardware is required to run.

OS: *nix-like system, POSIX compliant.

Compilers: Fortran 90 compliant compiler

Low-level libraries required: MPI 1.2 compliant library

High-level libraries required: NetCDF 4 and HDF5 for input-output

2.3 ASTROPHYSICS

2.3.1 GADGET

GADGET is a lagrangian code to perform numerical simulations of gravitationally interacting particles of both dark matter and baryonic matter. It also includes solvers for hydrodynamic and other several physical processes for baryonic particles. The evolution can be in either a cosmological background or not.

Hardware: no specific hardware is required, endianism does not affect the code functionality

OS: *nix-like system, POSIX compliant.

Compilers: C-compiler compliant with C11 standard, implementing the IEEE754 standard of floating-point representation and operations.
The code can take advantage from compiler's vectorization capabilities.
Support for OpenMP 3.1 is required.
Support for OpenMP 4.x and OpenCL 2.x are foreseen to be required.

Low-level libraries required: pthreads, MPI 3.x

High-level libraries required: standard libc, FFTW 2.1.5, HDF5.x, GNU Scientific Library (GSL). HDF is optional, all the others are strictly required.

2.3.2 SWIFT

SWIFT is a lagrangian code to perform numerical simulations of gravitationally interacting particles of both dark matter and baryonic matter. It also includes solvers for hydrodynamic and the radiative loss for baryonic particles. The evolution can be in either a cosmological background or not.

Hardware: no specific hardware is required, endianism does not affect the code functionality

OS: *nix-like system, POSIX compliant.

Compilers: C-compiler compliant with C11 standard, implementing the IEEE754 standard of floating-point representation and operations.

The code can take advantage from compiler's vectorization capabilities.
Requested support for OpenMP 3.1, support for OpenMP 4.x is recommended.

Low-level libraries required: pthreads, MPI 3.x

High-level libraries required: standard libc, FFTW 2.1.5, HDF5.x, Gnu Scientific Library (GSL). HDF is optional, all the others are strictly required.

2.3.3 PINOCCHIO

This code generates catalogues of collapsed dark-matter haloes in a cosmological volumes, using a perturbative approach.

Hardware: no specific hardware is required, endianism does not affect the code functionality

OS: *nix-like system, POSIX compliant.

Compilers: C-compiler compliant with C11 standard, implementing the IEEE754 standard of floating-point representation and operations.

The code can take advantage from compiler's vectorization capabilities.
Support for OpenMP 3.1 / 4.x and OpenCL 2.x will be required in the future.

Low-level libraries required: pthreads, MPI 2.x

High-level libraries required: standard libc, FFTW 3.x, Gnu Scientific Library (GSL).

2.3.4 HiGPUs

HiGPUs is a N-Body code with high order Hermite's integration scheme.

Hardware: GPUs are required, otherwise no specific hardware is required to run, endianism does not affect the code functionality

OS: known to run on *nix-like system, POSIX compliant.

Compilers: C-compiler compliant with C99 standard, implementing the IEEE754 standard of floating-point representation and operations.
Support for OpenCL 1.x is required.

Low-level libraries required: MPI 1.2

High-level libraries required: none.

2.4 ENGINEERING (CFD)

2.4.1 OpenFOAM

OpenFOAM is a generic software for computational fluid dynamics (CFD): it has an extensive range of features to solve any kind of CFD problem, including complex fluid flows involving chemical reactions, turbulence and heat transfer.

Hardware: no specific hardware is required to run, endianism does not impact on the code functionality

OS: *nix-like system, POSIX compliant.

Compilers: C-compiler compliant with C++11 standard, implementing the IEEE754 standard of floating-point representation and operations.
Vectorization (SSE2 / AVX512) ability is strongly recommended.
Support for OpenMP 3.1 is foreseen to be required in future
.

Low-level libraries required: pthreads, MPI 1.2

High-level libraries required: standard libc supported by C++11, Scotch 5.1.12b, GCAL 4.6.1, HDF5.x, GNU Scientific Library (GSL)2.1 . HDF is optional, all the others are strictly required, blas, Metis, Parmetis for domain decomposition..

2.4.2 SailFishCFD

SailfishCFD is a computational fluid dynamics solver based on the Lattice Boltzmann method.

Hardware: no specific hardware is required to run, endianism does not impact on the code functionality

OS: *nix-like system, POSIX compliant.

Compilers: Python 2.7+, C-compiler compliant with C99 standard,

Low-level libraries required: pthreads

High-level libraries required:numpy-1.7.0, scipy-0.13.0, sympy-0.7.3, mako-0.9.0, execnet-1.0.9, pyzmq-14.0.0, netifaces-0.8 (for distributed simulations), matplotlib to produce batch output graphs

2.5 NEUROSCIENCE

2.5.1 DPSNN

The Distributed and Polychronous Spiking Neural Network simulator (DPSNN) is a natively distributed mini-application benchmark representative of plastic spiking neural network simulators, coded as a network of C/C++ processes interfaced using a pure MPI implementation.

Processes describe synapses in input to cluster of neurons with an irregular interconnection topology, with complex inter-process traffic patterns broadly varying in time and per process; it can be used to gauge performances of existing computing platforms and drive development of dedicated future parallel/distributed computing systems.

The application was designed to be natively distributed and parallel, to be easily plugged to standard and custom software/hardware communication interfaces and, for a given configuration, to have output which is invariant against how many processes or hardware nodes it is run on, to simplify the scalability analysis on different architectures.

In the following, the system-level requirements for the DPSNN:

Hardware: no requirements for specific hardware in order to run. The code can currently be compiled and run on either Intel platforms (for instance Xeon CPU E5-2630 v2/v3) and ARM-based platforms (for instance the ARM cores in the NVIDIA Tegra K1/X1). Endianism does not affect code functionality.

OS: no known requirements besides those of a POSIX-compliant OS; our tests were performed on GNU/Linux derivatives as CentOS (for Intel) and Ubuntu (for ARM).

Compilers: Up to date GCC compiler, e.g. GCC version 4.4.7 (Red Hat 4.4.7-17) or later. Our tests were performed on GCC 4.4.7, 4.8.4 and 5.2.1. There are no known incompatibilities with all levels of optimization, standard ones available on the GNU compiler can be used.

Low-level libraries required: an MPI implementation compliant with MPI-2. The code is currently running both using OpenMPI (version 1.10.3) and MPICH (version 3.2).

High-level libraries required: standard C/C++ libraries: libc, libm, libstdc++, libgcc_s, libpthread.

2.6 DATABASE

2.6.1 MonetDB

Recall that MonetDB is a multi-threaded analytical database management system primarily implemented in C, its main system-level requirements include the following:

- Linux is the primarily supported OS of MonetDB. MonetDB operates on any of the conventional Linux distributions, such as RedHat, Ubuntu, CentOS, and Fedora.
- Up to date GCC compiler, e.g. gcc (GCC) 6.2.1 20160916 (Red Hat 6.2.1-2).
- MonetDB heavily relies on the *pthread* library and related multi-threading primitives.
- MonetDB heavily relies on efficient implementation of memory mapped files for I/O of persistent data and *malloc()* for temporary data.
- The following programming languages should be available in the ExaNeSt platform: C, Java (6+), R and Python (2.7+ and 3.4+).
- Although the *atomic_ops* library is not required for MonetDB to be fully functional, MonetDB relies on it for parallel processing. Therefore, this library is paramount for the performance.
- The list below includes the required libraries to build MonetDB from source code. A complete list of optional libraries can be found in Appendix III of Deliverable D4.1.
  - bison, bzip2-devel, gsl-devel, libatomic-ops-devel, libcurl-devel
  - libuuid-devel, libxml2-devel, openssl-devel, pcre-devel (4.5+), perl, pthread python-devel, readline-devel, system, unixODBC-devel, zlib-devel
- The MonetDB profiler Marvin is a web-based JavaScript+Python application. The following software are additionally needed:
  - The MonetDB native Python client API *pymonetdb*[1].
  - The Python web framework and network library *Tornado*[2].
  - Recent browsers compatible with ES5 (ECMAScript 5).

1 https://github.com/gijzelaerr/pymonetdb
2 https://www.tornadoweb.org

3. Requirements of Network and Storage

We recall that a detailed analysis of pattern of the MPI calls and of the size of the exchanged message has been performed by WP3 and reported in D3.1 *Specifications for Optical and Electrical Interconnects.*

Here below we summarize and refer to the figures discussed in that report, to which we address for further details and discussions.

A summary of the requirements will be presented in Section 5.2.

3.1 MATERIAL SCIENCE

3.1.1 LAMMPS

Network: LAMMPS is Molecular Dynamics code based on domain decomposition for message passing parallelization. Since only the data of the particles close to the boundary needs to be communicated, the size of each message is typically small, kB to MB, in a point-to-point fashion.

However, since the communication takes places several times a second, latency is a major concern. Test comparing the performance of native InfiniBand transport layer (typical latency ~1 us) and Ethernet TCP transport layer (typical latency ~30 us) yield a performance penalty that can reach a factor of 3 or 4 using high latency communication. This is reflected in the time spent in communication, that has been observed to take up to 75% of the walltime (below 25% in the case of low latency).

In conclusion, low latency network (few ~us) is necessary requirement, while large bandwidth (by which we mean >10Gbps) is not the main concern.

Storage: It is quite difficult to provide an accurate description of the kind or of the amount of I/O operations performed by LAMMPS as it depends heavily on how the user configures the execution.

As input, LAMMPS read a small script with all the parameters for the execution from standard input. This is usually a small ASCII file with less than a few hundred of lines. If specified in the input, the execution could restart from a binary "restart file". If this is the case, LAMMPS will get the initial position of the atoms from a binary file that contains 6 floating point numbers for each particle; the number of particles is a variable of the input script and fluctuates greatly depending on the kind of simulation the user is performing, from millions to billions, so the size of the input files can range from kB to several GB.

About the output, the footprint can be extremely small if only thermodynamics quantities are required (this are printed in the log file), or become quite large in the case the user chooses to save a dump file or a restart file of all the system during the execution. A restart file contains all the information needed to restart the simulation from a particular point of the execution. It is one (or more) binary file and its dimension is in the order of 6N floating point numbers where N is the number of particles. The restart file is the mean to handle failure re-covery.

For what concerns the "dump" command, we have a similar situation than before. The dump file as more flexible than the “restart file”. Indeed:

- It can be a binary (hdf5, xyz, ...) or a text file
- It can contains information about just a subset of all the particles

A particular case of dump is the one that produce images or a movie from the overall simula-tion.
So, overall, the size of the output can vary from few MB to several (hundreds) of GB.

Finally, we want to point out that, for a typical use, the I/O is not the most relevant bot-tleneck of the software.

3.2 CLIMATE SCIENCE

3.2.1 RegCM

Network: RegCM is regional climate model based on domain decomposition for paralleliza-tion. Domain decompositions is only in 2D due to the relative small number of points in the vertical direction. Since only the data of the grid point close to the boundary needs to be com-municated, the size of each message is typically small, kB to MB, in a point-to-point fashion.

However, since the communication takes place several times a second, latency is a major concern. Test comparing the performance of InfiniBand transport layer (typical latency ~1us) and TCP transport layer (typical latency ~30 us) yield a performance penalty that can reach a factor of 2 or 3 using high latency communication.

RegCM also features an extensive use of MPI collective operation for I/O operation: on the current version the spokesperson approach is used so one single node collects tempor-ary results on the node to be dumped on files.

In conclusion, low latency network (few ~us) is necessary requirement for P2P opera-tions while large bandwidth is mandatory for collective I/O operations.

Storage: A complete run of RegCM usually requires the following input set:

- A DOMAIN file to localise the model on a world region. This file contains the loc-alised topology and land-use databases, as well as projection information and land sea mask. This is generally negligible in size for the domains under consideration (200x200x25)

- An ICBC (Initial Condition Boundary Condition) set of files that contain the result of a global simulation over the region that RegCM will simulate for all the period of simula-tion.

In the ICBC files, the overall simulated time must be taken into account to have an idea of the dimension. In particular there can be from 4 to 6 fields that associate one floating point number to each cell for every time step. Therefore, we have that the dimension of the file is of the order of 6 × 200 × 200 × 25 × 4 = ~ 24 MB for each time step. A usual time step is 6 hours long and so for one month of simulation we need 1 GB of data. A complete, medium-size, simulation can cover 150 years, bringing the size of ICBC to reach more than 1 TB of data in input.

All the input files are NetCDF files. The access to them is completely serial. During the simulation, a complete dump of the status of the simulation is usually performed once a week. This allows restarting the simulation at that point if something goes wrong during the execution. If we still assume to be using a 200 × 200 × 25 grid, then a restart file is of about 350 MB. These files are written by process 0 that first collects all the data from the other processes and then dumps the data on the disk in a complete serial way

For what concerns the output of the simulation, this is also written in serial by the process 0 that, every 6 hours of the model, add a time step to the following NetCDF files:

- ATM file with the atmosphere status from the model
- RAD file that contains radiation fluxes information
- SRF file with the surface diagnostic variables
- STS file with information about the physical state of the system.

For a 150 years of simulated time, all these files can reach a dimension of about 15 TB.

We also note that from 2016 on the domain used by RegCM in the Cordex experiment (see: www.cordex.org) will be increased by a factor of 2 to 2.5 in the X and Y directions with the net result to increase the overall volume of data of a factor of 4 times at least.

3.3 ASTROPHYSICS

3.3.1 GADGET

Network: the communication weave of this code is extremely complex and made up of very different pattern used in different parts of the code, all of which are active in every time-step. Broadly, the code uses both all-to-all and point-to-point communications. The former are the most frequent while called for the smallest message size (from few to few hundreds bytes), whereas the latter are less in number but dominates the volume of data exchange.

However, due to the high number of small message-size communications, the network latency could be a major issue. Tests made comparing time-to-solution performance using the Infiniband transport layer and the TCP transport layer (that has 30 times larger latency), resulted in a 5× longer running time with the latter configuration, the discrepancy being all due to communication.

As a global figure, obtained studying scalasca traces and outcomes of purposely inserted checksums within the code, the code exchanges in each timestep about 5-7 MB per particle per MPI task. Then, for a typical problem addressed with 10^9 particles on 512 tasks, about $2{\times}10^{12}$ Bytes at each timestep transit over the machine's network.

Then, both low-latency and bandwidth are strong requirements.

Storage: the code writes 2 types of files: *(a)* the "snapshot" files, that store the status of the simulated physical system at desired epochs, and *(b)* the checkpoint files that save the status of the run at fixed time intervals (the exact timing in seconds is a parameter set by the user). Each particle requires on average about ~0.5kB, so that the typical size for snapshots (type *a*)

is of about 0.5 kB/particle × [10^8 - 10^9]particles ~ [5-10]GB, that can grow by at least an order of magnitude for ambitious projects. Snapshots can be splitted in several files to limit the single-file size.
Checkpoint files, that contains also informations on some memory and run-time structures, are ~10% larger than snapshot files. Each MPI task writes its own checkpoint file.

3.3.2 SWIFT

Network and Storage: we will use SWIFT as a test case to evaluate its scaling ability and the performances of its task-based workflow in the target environment, as well as against the incremental re-engineering of GADGET.

As such, we did not yet perform a detailed analysis of its network and storage imprint, that, however, we do not expect to be overall much different in volume than the GADGET's one. We do expect a qualitatively different pattern in network communications, which will be the object of profiling in the next future.

3.3.3 PINOCCHIO

Network: the code is divided in two main sections. In the first one it basically performs FFT, while in the second one it operates on clustered groups of particles. Communications then happen with two very different patterns.
In the first part the communication follows the 3D-FFT pattern, provided that the parallel execution currently adopts the FFTW 1D-slab decomposition. Then, grid data are divided among N_G tasks (where N_G is the size of the grid) to calculate the 1D transform, and subsequently the data are transposed in the other 2 directions to calculate the 2D transform. So, the network is basically interested by the communication of ~N^3 complex doubles among N tasks. Typical goal values for N_G are equal or larger than ~10^4.

In the second part, particles – that are as many as grid nodes, hence N_G^3 – whose physical properties have been updated accordingly to the results of the FFT, are re-shuffled among MPI tasks moving from a slab-based distribution to a cubic distribution. This requires a all-to-all communication that strongly depends on the problem size and the number N_T of MPI tasks, but can be estimated to be $O(N_G^3)$×S, where S ~ 100B is the amount of bytes exchanged per particle. This communication happens only once.

Storage: the output files contain the catalog of formed structures at desired epochs. The number of formed structures is larger for larger volumes and higher mass resolution (i,e. number of grid nodes in a fixed volume), and for a given problem (simulated volume with N_G^3 grid nodes) it grows with cosmic time. Hence, the size of the output files for a given problem will be larger while the run advances, and will scale with N_G^3 and roughly linearly with simulated volume V.

As an example, a volume representative of the visible universe with $N_G \sim 3800$ and 6 outputs produces a total amount of about ~32GB for catalogs and additional 11GB of further data. The goal is to simulate the same volume but with $N_G \gtrsim 10^4$ that should dump 20× larger files, and significantly more outputs.

3.3.4 HiGPUs

Network: given a simulated ensemble of N particles, the domain decomposition divides the computational domain evenly among the N_T MPI tasks, so that each has N/N_T particles, Each tasks communicates 4 doubles per particle to all other tasks through a MPI_Allreduce call, therefore sending $N/N_T×(N_T-1)$ chunks of data.
The network then must support $N/N_T×N_T×(N_T-1)$×sizeof(double)×4 bytes of communication.

Given that the typical problem-size ranges from 10^5 to 10^7 particles, this amounts to about [3-300]MB×(N_T-1) per step.

Storage: the code does not perform any parallel I/O, all the particles are dumped to storage devices by MPI Task 0, that writes about 100B per particle. A single data file is N×100B long, typically in the range [0.01-1]GB.

3.4 ENGINEERING (CFD)

3.4.1 OpenFOAM

Network: the communication pattern is complex and made up of very different patterns used in different parts of the code, all of which are active in every time-step or iteration for convergence, based also on the physics that has to be simulated. Both point-to-point and all-to-all communications are widely used in order to resync boundary equations to all threads participating at the solution.

As in any scientific parallel software, due to large number of relatively small message-size communications, the network latency is crucial point of any parallel solver. It is normal to expect 6x gain between RDMA and TCP connections on the same media for small problems, whilst to become practically unusable over TCP for large number of processes.

Sizing of a problem with respect to the number of cores you wish to involve strongly depends on the type of solver, type of problem and type of physics interaction of the simulation. Typical figures can range from 50000 cells up to 1M cells per core.

Storage: OpenFOAM generates many output files. Creating a checkpoint leads to the creation of a directory for each MPI process containing a file per parameter. When running in an exascale environment, it can create therefore several hundred files at each checkpoint, which has the potential to result in thousands or even tens of thousands of files. Those files are small in size but a critical point could be the filesystem performances.

The code produces several restart files per MPI task. As the simulation has timesteps and the physics of the problem cannot proceed independently between threads so much, in order to assure a correct solution checkpoint has to be done with synchronization.

In current production simulations, a checkpoint is produced every X simulation timesteps, trying to have a complete restart situation every 2 hours of calculation maximum (this is because either a crash could occur or the solution of the problem could diverge and thus the user has to take actions restarting with different relaxation parameters depending on the type of problem). Restarting from a CP could be done with different partitioning, but to do this, you need to reassembling the last configuration and repartitioning the entire system, which would take alot of time.

3.4.2 SailFish CFD

Network: SailFishCFD is a CFD code written for solving Lattice-Boltzmann equations, thus using a totally different approach with industrial CFD codes trying to gain performance (in order of accuracy of high-order vortex analysis). The program instantiates a master process for each physical node that subsequently spawns N subprocesses. Communication is done over Pthreads and thus over TCP. Message size depends on the domain being solved for each subprocess, varying around few kB to MB, in a point-to-point fashion.

Storage: A typical SailFishCFD run requires just the possibility to dump time steps in order to do post-processing after the analysis.

For a medium size problem of 10M lattices the typical dimension of output files are about 80M for each time step that have to be multiplied by the number of time steps. In turbulent flows and to study flow instabilities 1k steps are common thus corresponding to about 100 GB of data.

3.5 NEUROSCIENCE

3.5.1 DPSNN

Network: two main phases can be identified in the DPSNN application: an initialization phase and an iterative simulation phase. During the former, connections between pairs of processes are established according to a specific synaptic matrix interconnecting the cluster of neurons of the network. During this phase, the synaptic matrix itself is exchanged among processes. This phase is repeated only once during the whole simulation, and is implemented using a sequence of MPI_Alltoall() and MPI_Alltoallv().

During the second phase, instead, at each code iteration every process communicate to its subscribers (a subset of the network processes) the spikes produced in the last iteration. This communication step, repeated every 1 ms of simulated neural activity, is implemented in the present version of DPSNN using a sequence of two MPI_Alltoallv().

As an example, assuming a max memory of 4GB per ARM core, 1200 synapses per neuron, 40 byte per synapses, a maximum of 90K neurons can be allocated on each ARM core. At a mean firing rate of 3 Hz, this corresponds to 268 spikes emitted by each core per simulated ms. Each spike is represented by the ID of the spiking neuron and its emission time (AER, Address Event Representation). Therefore the communication involves small packets and is dominated by latency.

Storage: In first approximation, the total memory required by the DPSNN application is proportional to the total number of synapses in the system S=NxM, where N is the number of neurons and M the mean number of synapses projected by each neuron. Each synapse requires about 32-40 bytes. Checkpointing would require saving the whole set of synapses. For instance, in case of a simulation of a single cortical area described by a grid of 96 by 96 cortical columns, composed of 1250 neurons per column, projecting ~1200 synapses per neuron, the required amount of memory to be checkpointed is ~ 514 GB.

For what concern the storage of the simulation output, the size and the number of output files depends on several factors, among which, for instance, the dimension of the simulated network, the simulation parameters, the average network firing rate and the duration of the simulation itself. In these terms, the storage for the output files can easily span between a few MB up to several (hundreds of) GB. For instance, the simulation output of a grid constituted of 96 by 96 cortical columns, showing a Slow Waves behaviour with a mean firing rate of 3Hz, lasting for only 10 simulated seconds, is more than 6GB for the only file collecting the spikes. It's quite common to have a higher mean firing rates (few tens of Hz), longer simulation time, or several additional output files used to monitor the simulation behaviour.

For what concerns the input, the present version of DPSNN application relies on few kB of input data (less than 2/3 kB, usually): three ASCII files containing the simulation parameters.

3.6 DATABASE

3.6.1 MonetDB

As pointed out in earlier deliverables D2.1, D3.1 and D4.1, an analytical database management system such as MonetDB supports a broad spectrum of end user applications (from astronomy, remote sensing, to finance, healthcare and aviation), and highly diverse and unpredictably workloads (in terms of query complexity, data sizes and number of concurrent users). Therefore, MonetDB does not have a single set of network and storage requirements.

In this section, we try to divide the possibilities into several groups of scenarios. To simplify matters, we assume here load-once-read-many databases, which cover a major class of big data analytical applications. They will also be the use case scenarios we will use to evaluate ExaNeSt.

Network requirements

Network traffic typically happens in the following scenarios:

- *Initial data loading:* this rarely happens, but it would require big chunks of CSV data (10s ~ 100s GBs) to be read from the main storage system to the computing nodes. After the CVS data are processed and converted into MonetDB binary data, the binary data need to be written to the persistent storage (again 10s ~ 100s GBs).
- *Database startup:* this rarely happens, but it might require big chunks of data (10s ~ 100s GBs) to be loaded from main storage to the computing nodes.
- *Query execution on cold data:* unpredictable frequency, but expected to be much less frequent than query execution on hot data. Might require big chunks of data (10s GB) to be loaded from main storage to the computing nodes.
- *Query execution on hot data:* unpredictable frequency, but expected to be much more frequent than query execution on cold data. Data are expected to be (mostly) in MEM, or in the NVM device. If MEM + NVM is big enough to hold all hot data + intermediate data, no network traffic to the main storage will happen.
- *Database client-server communication:* one round-trip per query, i.e. sending query and getting results. The data size is mostly small, e.g. in 10s ~ 100s MBs. No network traffic to main storage, but among compute nodes, depending on where the client is.
- *During query execution:* each query is handled using all cores available within a coherent island. This implies fast exchange of large temporary data fragments (10s of MBs up to GBs) within a Daughter-Board. No network traffic to main storage.
- *MonetDB server-server communication (for distributed processing):* mostly small amount of data (10s ~ 100s MBs) among compute nodes. No network traffic to main storage.
- *Database migration/duplication:* this involves moving/copying a whole database (10s ~ 100s GBs) from one compute node to another. No network traffic to main storage.

Storage requirements

MonetDB has the following quantitative and qualitative requirements on the storage system:

- Per database 10 ~ 100 GB disk storage for MonetDB binary files. When loading CSV data into MonetDB binary format, data size reduction is ~50%. For databases in order of GBs or more, MonetDB storage overhead is negligible (<1%).
- During query execution, MonetDB will produce intermediate results, which might end up on disk. Intermediate results are generally not more than 25% of the data size, but worst-case scenario (e.g. computing a Cartesian product) can blow up disk usage.
- MonetDB will try to maximally utilise the NVM devices to avoid accessing main storage system.
- A rule of thumb from the database world. A database cache is typically 10% of the HDD size. In MonetDB, as a main-memory oriented system, likely 50% of the memory is allocated to persistent data and the rest for temporary data.
- Fast I/O by means of memory mapped files. This has been discussed in more details in earlier deliverables D4.1 and D4.2.
- A strong *malloc* library for management of intermediate objects:

  This is a particularly important area that should be addressed by the ExaNeSt OS. In MonetDB, we know the malloc-ed blocks will be neither persisted nor updated. Therefore, the malloc-ed objects may be anywhere where memory is available. This can be exploited by the ExaNest OS by making the malloc-ed structures nomadic using page stealing. Large malloc-ed objects may be split over several nodes, however, the OS should pull them together again on a local node before executing the database operators. Small malloc-ed objects may be duplicated to optimise data locality. This calls for an *madvise()*-like mechanism to give hints of the usage of malloc-ed blocks.

  Most operations in MonetDB either accept memory mapped files or malloc-ed structures. They are both considered large, e.g. from a few MBs to several GBs. The result of an operation is typically again a lightweight structure held in malloc-ed or memory mapped blocks.

  During query processing, a dataflow graph is traversed, whose sources are all persistent objects and whose intermediates are malloc-ed structures. There can be multiple workers active on the same dataflow graph and the code for the operations is small enough to consider running it close to where the data reside.

  The MonetDB interpreter would be helped if remote execution of an operator, i.e. near where the data live, could be as easy as to ‘pull’ intermediates towards the local memory, possibly using a swapping of elements.
- Profiling a distributed version of a DBMS calls for being able to gather streams of events from all nodes. A complicating factor is that most database operations that cause event records to be procured require less than a few millisecond. This leads to the requirement that ExaNeSt should think about how to efficiently gather massive amount of events from an exascale distributed platform. This requirement also pertains to the HPC applications, which most probably generate even more events to be gathered and monitored.

4. Porting details and roadmap

A comprehensive view of the roadmaps detailed in this section can be found in Section 5.3.

4.1 MATERIAL SCIENCE

4.1.1 LAMMPS

The porting strategy for LAMMPS is composed of three different, parallel steps

A. LAMMPS code base porting to ARM
   1. compilation and testing on ARM, with possibly an analysis of the micro-architectural features relevant to scientific applications.
   2. fine tuning, once the MPI support to the eXaNeSt architecture will be available.
   3. Extensive benchmarking and profiling will be carried one during all the phases of the porting, to provide to network and storage work packages insights on real-world applications.

B. LAMMPS OpenCL porting to FPGA accelerator: LAMMPS supports OpenCL through a set of wrappers that abstract the framework-specific implementation (CUDA, OpenCL). The porting will consist in optimizing and in some case re-engineer the openCL-specific routine, maintaining however the full stack of abstractions which LAMMPS is built upon. Re-engineering in particular will be required when some openCL features will not be available in the runtime provided by the FPGA

C. miniMD porting and optimization for the ExaNeSt architecture: miniMD is molecular dynamics (MD) mini app, that contains the main computational features of LAMMPS in a minimal code (~5.000 lines, compared to ~200.000 lines of the current LAMMPS implementation). It is intended as an agile development MD code, that can be easily modified and optimized to exploit architecture-specific features, without the burden of dealing with a large-scale, full-fledged production code. It was specifically developed with the co-design approach in mind.
   The porting strategy will include:
   1. Porting miniMD to ARM, with possibly an analysis of the micro-architectural features relevant to scientific applications
   2. Optimization of MPI communication: this will amount to re-engineer large portion of the code, following the guidelines of the network WP, possibly using MPI one-sided communications.
   3. Optimization of the openCL kernels: Since miniMD does not implement a complex abstraction of the accelerator, the optimization and reengineering can be much more radical, up to complete redesign if necessary.

   The optimization achieved in miniMD could be than ported to LAMMPS, if time permits and if they are compatible with the LAMMPS abstraction layers.

Point (A.1) is partially completed, and is expected to completed by M18.
Point (A.2) conclusion depends on the availability of the ExaNeSt-specific MPI implementation, and is expected to be completed by the end of the project, M36
Point (A.3) due to its intrinsic codesign nature, will last until the end of the project, M36

Point (B) conclusion depends on the availability of the OpenCL framework specific to ExaNeSt. and therefore it will last until the end of the project M36

Point (C.1) will be completed by M15
Point (C.2) will be completed by M36
Point (C.3) will be completed by M36

4.2 CLIMATE SCIENCE

4.2.1 RegCM

The porting strategy for RegCM is composed by

A. RegCM code base porting to ARM
    1. compilation and testing on ARM
    2. fine tuning, once the MPI support to the eXaNeSt architecture will be available.
    3. Extensive benchmarking and profiling will be carried one during all the phases of the porting, to provide to network and storage work packages insights on real-world applications.
B. Porting of the next major release (yet unreleased), which will support hybrid programming model (MPI + OpenMP) and will feature new communication patterns.

Point (A.1) is partially completed, and is expected to completed by M18.
Point (A.2) conclusion depends on the availability of the eXaNeSt-specific MPI implementation, and is expected to be completed by the end of the project, M36
Point (A.3) due to its intrinsic codesign nature, will last until the end of the project, M36

Point (B) depends on the availability of the new major release of the code. Its conclusion is expected by M36.

4.3 ASTROPHYSICS

4.3.1 GADGET

Firstly, to introduce our porting plan and roadmap, we recall some basic elements about GADGET, that we have chosen as our candidate code for a complete re-engineerization. This code is widely known and used in the community of numerical cosmology, and a release of a new version suited to run and scale on exascale platforms would greatly benefit many research groups in Europe.

This code is a full-fledged cosmological code that aims to solve the gravitational and hydrodynamical equations that rules the formation and evolution of cosmic structures. The scientific problem can be divided in three parts: gravity, a long-range interaction affecting all of the computational elements of the chosen domain; hydrodynamics, that is almost "local" and only affects normal matter; and "extra-physics", namely a number of very complex astrophysical processes – i.e. star formation, stellar feedback, black holes feedback, radiative transfer, chemistry, magnetic fields – that strongly affect the evolution of structures at scale lengths below the Jeans length. The code distributes the workflow among the MPI tasks and further exploits threads-coworking within each MPI task (using either OpenMP or pthreads), although this latter option is limited to some core loops where data races are avoided by perfect parallelism (for instance where local particle quantities are updated / calculated).

Some additional remarks on the code architecture will be helpful to define the porting strategy, as follows.

(1) As of today parallel machines are increasingly built with (cc)NUMA hierarchical architectures in which multiple cores have access to a hierarchy of memories. Multiple cache memories are placed at the lowest (fastest) levels of the hierarchy, and every core may have individual caches, while groups of cores may share higher level caches or single memory controllers.
For the sake of performance, on such systems it is crucial that codes take advantage of spatial and temporal locality of data, which also requires to be NUMA-aware. This concept, in the PGAS memory model provided by UNIMEM – which shall make it easier to deal with remote memory access – translates in the *affinity* that a portion of the shared memory may have with a particular process.

The GADGET memory model is not NUMA-aware and tries to exploit memory locality only with some basic stratagem, for instance trying to keep close in memory particles that are also close in space and are therefore likely to interact with each other. Hence, as a general remark, underlying the general re-design of the code there must be a re-design of the GADGET's memory model in order to make it NUMA-aware or, in other words, to consider the affinity to the process of different memory regions relatively to the data being processed.
Due to the extreme diversity of physical processes being modelled and algorithms implemented, this is quite a difficult task that does not have a unique solution and may require memory layout transformations in some points (for instance, from arrays of structures to structures of arrays). This task will last the whole activity as part of the code re-engineering.

(2) The code has been conceived as the parallel generalization of a serial code, in a pre-multi-thread-era. Hence the workflow is then rigidly procedural, meaning by this that all the tasks perform the same operations – possibly individually using more threads in local loops – with frequent synchronization via MPI message.
An analysis (see D3.1) of the network imprint reveals that the number of communications that exchange very small amount of data (few to hundreds of bytes) largely outnumber those that exchange large data chunks (0.1-10 or more MB), although the latter account for most of the total data traffic on the machine network. As a consequence, the network latency can represent a significant fraction of both the communication and running times.
In view of these two facts, and the strongly hierarchical architecture foreseen for the target machine, it is compulsory to adopt a different code design i.e. to decompose the work-

flow in as-small-as-possible single tasks with clear dependencies on, and conflicts between, each other from both the point of view of operations to be performed and data to be processed.
In such a way, the workflow would be translated in a queue system where idling threads perform the first available tasks on not-under-use data. Synchronization of operations should pass as much as possible through RDMA operations and the queue system itself. Moreover, "encouraging" threads that resides on the same group of cores to undergo similar tasks, or tasks operating on the same data, should lead to a more efficient exploitation of memory affinity and locality (even if redundancy of some data may be required).

This and the previous point 2) depict what is broadly the direction that we plan to take to re-engineer the GADGET code.

We will start from a stripped-down version of the publicly available code, that incorporates only the gravity and hydrodynamical solvers, and the star formation plus the stellar evolution modules. That version will be available by M13.

To undergo the general plan broadly described in previous points 1) and 2), we note that a cosmological code like GADGET is based on some well-defined pillars, namely: *(a)* a domain decomposition strategy, *(b)* a data-structure that organizes the particles in space and memory, *(c)* an algorithm for neighbours-searching, *(d)* a gravity solver and *(e)* an hydrodynamical solver.

Actually, *(a)* and *(b)* are strictly coupled and must be designed together, and strongly influences also *(c)*. The *(d)* and *(e)* algorithms follow from the previous ones (provided that *(d)* may also require the use of parallel FFT to solve long-range interactions).
As such, we intend to separately develop mini-apps for each of those algorithms, designed ab-initio as "atomic" inter-dependent tasks (with no circular dependencies) together with a task scheduler and data resource manager.

In this way we will be able to develop different algorithms and strategies for each of the "pillars" detailed above in a much easier way that in a unique monolithic code.

Namely, we plan to proceed with the following tasks:

T1 : domain decomposition
T2 : particle organization structure
T3 : neighbour search
T4 : gravity solver, further subdivided in

- T4.1 : direct summation + fast-multipole-like algorithm
- T4.2 : Particle-Mesh algorithm through use of FFTW (introducing 3D decomposition for FFT as detailed below in Section 4.3.3)

T5 : hydrodynamical solver
T6 : extra-physics (star formation + stellar evolution and feedback)

Moreover,

T7 : as soon as preliminary OpenCL will be available on the platform, we shall start to develop dedicated kernel to upload vector computations on FPGAs
T8: as soon as platform-optimized implementation of MPI is available, we shall start to profile the communication pattern in order to further optimize it

We plan to complete T1, T2 and T3 by M22, while to start T4 and T5 from M18 and complete them by M30, and to start T6 by M26 and complete it by M34.

Task T7 ad T8 will start depending on the availability of OpenCL, optimized MPI and FP-GAs, which depends on the schedule of WP3, WP4 and WP5.

4.3.2 SWIFT

The SWIFT code is the result of a re-engineering activity similar to that we described in the previous section.

It presents many of the qualities than a exascale-ready code is expected to have, although it is still in a preliminary test version. As it comes it its publicly available version, it includes basically only the gravity and hydrodynamical solvers.

We shall use SWIFT for test & evaluation purposes (task T1) on the subsequent testbed release of the platform, as well as to compare with the progressive re-design of the GADGET code.

Moreover, the SWIFT scheduler comes in a stand-alone version that we intend to use both as an alternative scheduler to test the mini-apps described in the previous section (T2), and to develop mini-app to make an intensive use of multi-threading and UNIMEM APIs, so as to intensively test and evaluate the target platform (task T3).

Task T1 and T3 will last until M30, whereas T2 will come along with task T1-T6 in the previous section.

4.3.3 PINOCCHIO

This code generates catalogues of collapsed dark-matter haloes in a cosmological volumes, using a perturbative approach that is much faster than a full N-Body simulation while preserving 10% accuracy in reproducing the evolution of dark-matter haloes.

It basically consists in 3 parts. In the first one (Step 1), initial conditions (IC) are generated on a cubic grid with $N_X \times N_Y \times N_Z$ grid points (although usually $N_X = N_Y = N_Z$) . This amounts to generate the power spectrum of pristine perturbations on an otherwise uniform and isotropic matter distribution. Such tiny perturbations lead to gravitational collapse of structures.

In the second part (Step 2) the code computes the instant at which matter elements are expected to gravitationally collapse (GC), i.e. get to high density. This is done by solving a set of differential equations, mainly through the use of parallel FFTs.

Then, in the last part (step 3), the code assembles mass elements into either collapsed haloes or filaments, a procedure that mimics the hierarchical formation of objects. This is the most computationally-demanding part of the code.

We propose the following roadmap to port the PINOCCHIO code on the target machine.

T 1) Porting PINOCCHIO to ARM. This action has been already completed as a preliminary test on the available testbed machine (Juno-board based).

T 2) The original code relies on FFTW library to perform FFT in parallel. As such, it is bound to the FFTW 1D-decomposition, so that the grid is decomposed in "slabs" among the MPI tasks. That means that the code scales at maximum as $N_X \times N_Y \times N_Z$ / max(N_X, N_Y, N_Z), and in the case $N_G < N_{TASKS}$ (let as assume for simplicity $N_X = N_Y = N_Z = N_G$), a fraction of the MPI tasks will just wait for the others to perform the calculations.
An alternative approach consists in decomposing the grid in 2D, i.e. in "pencils", so

that there will be, for instance, $N_X \times N_Y$ tasks performing the FFT instead of only N_X. The library PFFT, built on FFTW is able to perform such decomposition and even a 3D decomposition for which the data are evenly distributed among all the tasks and the number of "pencils" is as much as possible close to $N_X \times (N_Y \times N_Z)$. Actually, the 2D decomposition (the 3D is actually a refined 2D decomposition) requires more, and more complicated, all-to-all communications. Whether this represents an advantage in terms of time-to-solution could be strongly dependent on the case and the machine's architecture.
We will then modify the original algorithm in order to exploit 1D/2D or 3D decomposition, depending on the number of grid points and of MPI tasks, so as to be able to generate IC already on a 3D space decomposition instead of the current 1D one.
This should have two main advantages: the first is to decrease the time-to-solution for generating both the IC and the GC for all the tasks will perform the related calculations; the second is to partially avoid the re-distribution of the data to all tasks that now is mandatory at the begin of the third step of the code.

We plan to complete this first action by M18.

T 3) Once the GC have been calculated in Step 2, particles must be re-distributed to all the tasks, so that each single task will own a cubic subregion of the computational domain (that region should encloses the task's assigned particles plus a surrounding ghost regions).
This re-shuffling, which is extremely expensive from the point of view of communications as explained in section 2.3.3, will be partially avoided while both IC and GC will be calculated on a 3D decomposition, as a consequence of the previous action.
In this third action, we plan to re-design the workflow and memory model so as to *(a)* exploit much more efficiently the multi-threading, through either OpenMP or pthreads, *(b)* introduce as much as possible the exploitation of vector operations and, *(c)* individuate core operations to be inserted in OpenCL kernels in order to queue them on accelerator resources (FPGA).

We plan to complete this last action by M30.

4.3.4 HiGPUs

Note: HiGPUs[3] is a N-Body code with high order Hermite's integration scheme with block time steps, with a direct evaluation of the particle-particle forces. It is fully parallel and exploits both OpenMP and MPI as well as either CUDA or OpenCL to run on GPUs on hybrid systems.

We replaced the code CHANGA, that was originally included in our pool of codes pool and largely share characteristics with SWIFT, with HiGPUs that presents two main advantages: *(1)* it is significantly simpler and smaller – because it handles a specific astrophysical problem, i.e. the gravitational direct interaction of N-bodies; *(2)*, it is specifically designed and optimized to run on accelerators with OpenCL, whose intensive use is one of the project's targets.

We will use HiGPUs as a test-case and validation code for the target architecture, exploiting its extremely good scalability to stress the machine network and the fact that it takes full advantage of multi-threading (through MPI+OpenMP) and accelerators (through OpenCL kernels) to stress both the UNIMEM and the FPGA functionalities.

3 reference: R. Capuzzo-Dolcetta, , M. Spera , D. Punzo, A fully parallel, high precision, N-body code running on hybrid computing platforms, 2013, J. Comp. Phys., 236, 580

The code itself can basically be considered a mini-app, that we will start to use for testing and validation as soon as OpenCL standard is available (as it should be at month 13, as detailed in Section 5).

As a first task (T1), we expect to have a preliminary profiling of the code behaviour and architecture's performance by M18, so as to extrapolate hints on further improvements to be implemented in close collaborations with the HiGPUs' developers by M30 (task T2).

4.4 ENGINEERING (CFD)

4.4.1 OpenFOAM

OpenFOAM is a general purpose CFD code.It basically consists in 3 consecutive parts. In the first one (Step 1) the problem definition and IC are read and a domain decomposition is done according to number of processes needed.

The second steps is the solution process itself, done through MPI tasks and strongly tainted to the physics of the simulation that has to be done. At the end of the simulation each task has written a file for each of the unknowns .

The third step reconstruct the entire domain.

The most critical part is the second, that can play a crucial role in the time to solution of the problem and in the accuracy of the results. Thus a substantial amount of effort will be dedicated to trying to optimize and gain better performance for ExaNeSt architecture.

The porting strategy for OpenFOAM is divided into three different steps:

1) OpenFOAM code base porting along with all scientific libraries required to ARM
2) Compilation and testing on ARM

   We plan to complete this part for M15

3) Profiling of the code and partial rewrite of a reference solver, in order to increase the ratio between number of cells per core and overall solver efficiency.

   This task can proceed in parallel either on a different architecture in order to not block any other activity. We plan to complete this part for M33.

4) Fine tuning, once the MPI support to the ExaNeSt architecture will be available.

   We plan to complete this part for M33

5) Extensive benchmarking and profiling will be carried on during all the phases of the porting, to provide feedback for network and storage design and tuning. To be completed in the corresponding of the previous phases.

4.4.2 SailFish CFD

SailFishCFD is a demonstrator for CPU+GPU acceleration of a general purpose CFD code.

The solver is based on the Lattice Boltzmann Method, which is conceptually quite simple to understand and which scales very well with increasing computational resources.

SailfishCFD, unlike the majority of CFD packages, which are written in compiled languages such as C++ or Fortran, is implemented in Python and can be accelerated in CUDA C/OpenCL, where should express the major benefits.

The code is written itself with a strong python templating in mind, and is quite unique in its CFD approach.

The entire domain can be spread out on several CPUs or GPUs via a pthread model that is responsible of the boundary interactions between the lattices.

The most critical part of code porting will be the huge amount of system scientific libraries needed by the code and the interconnection model.

The porting strategy for SailFishCFD will be:

1) SailFishCFD code base porting along with all scientific libraries required to ARM
2) Compilation and testing on ARM

   We plan to complete this part for M18

1) Fine tuning, performances tests, different patterns of lattices subdivision tests and profiling as soon as eXaNeSt network and storage architecture will be available.

   We plan to complete this part for M33

6) Extensive benchmarking and profiling will be carried on during all the phases of the porting, to provide feedback for network and storage design and tuning. To be completed in the corresponding of the previous phases.

4.5 NEUROSCIENCE

4.5.1 DPSNN

DPSNN, the Distributed Plastic Spiking Neural Net application can be configured to stress either the networking or the computation features available on the execution platforms Indeed, when a neural net including a relatively low number of neurons, each one projecting thousands of synapses is distributed on a large number of hardware cores, DPSNN can be used to stress the networking component, because the produced traffic consists of many short packets (a few spikes), to be transmitted to many target processes. In addition, depending on the regime of imposed status of biological activity, the traffic can be more or less homogenous (for what concerns both its spatial and temporal workload and traffic characteristics).

For growing number of neurons per core the computation components grows more that the number of packets. and starts to be the dominating component.

According to project gantt, we foresee the following detailed roadmap:

1. Identification of the numerical parameters needed to drive DPSNN to stress either the network or the computation or the storage implementation.
2. Extraction of a mini-application benchmark to be executed on the ExaNeSt platforms, and implementation of alternative communication strategies (e.g. from MPI_Alltoallv collectives to explicit point-to-point communications)
3. Porting to ARM based ExaNeSt prototyping systems and we foresee the following steps:
   a. Generic MPI interconnected ARM base system

   b. Unit: quad ARM, 16 GB total (Trenz platform or equivalent FPGA development kit);
   c. Node: 16 ARM, 64GB QFDB (initially on a cluster of 4 Trenz, then on the QFDB)
   d. Multi-Node system based on multiple blade partition: 2 (or more) fully populated blades (~128 ARM cores, 512 GB) allowing for evaluation of application performances on systems characterized by different network topologies (n-dimensional Torus vs DragonFly vs switch-based).
   e. ExaNeSt large prototype (about 1024 cores).
4. Depending on the progressive availability of new hardware and software features offered by ExaNeSt WP3, WP5 and WP6 (e.g RDMA zero copy point-to-point communication, completed and optimized MPI porting), the code will be enhanced and the resulting improvements will be assessed.
5. Finally, a projection of the performances for million core systems based on ExaNeSt foreground will be produced.

Point (1) and (2) are partially completed, and expected to be completed by M18.

Point (3.a) has been completed on two quad-core ARM-based nodes (NVidia Tegra) MPI interconnected on Ethernet.

Point (3.b) is in progress and expected to be completed at M18.

Points (3.c), (3.d) strongly depend on the availability of the ExaNeSt hardware platforms, full software stack and (at least preliminary) MPI implementation. According to gantt the activities are expected to be completed not before M24. Finally activities related to point (3.e) will be carried on in the period M30-M36

While, point (4) activities will be performed in parallel to point (3.x) activities evaluation, point (5) will be completed by M36.

4.6 DATABASE

4.6.1 MonetDB

MonetDB porting roadmap is divided into two stages. In the first stage, until M18 of the project, we will concentrate on porting MonetDB to the hardware and system software used in ExaNeSt:

- Continue and finish porting MonetDB to the ARM architecture. This includes porting all features included in the official MonetDB releases; the complete MonetDB test suite[4]; and the TPC-H and Air Traffic benchmarks.
- Extending MonetDB to make use of the NVM layer to cache temporary data.
- Improve MonetDB's Resilience: we will continue and mature MonetDB's transaction replication features. This includes, among others, extending the replication service to use BeeGFS on ARM.
- Porting MonetDB to run on KVM, including preliminary performance study to fine-tune system configurations.
- Preliminary performance study of TPC-H and Air Traffic benchmarks on Juno boards, and/or on on-premises hardware similar to the ExaNeSt architecture.

4 http://monetdb.cwi.nl/testweb/web/status.php

- Continue and finish the development of version 1 of *Marvin*, the database profiler for single MonetDB instance. This includes improving variable lifetime visualisation, and extending Marvin visualisation with CPU and MEM usage info.

In the second stage, until M33 of the project, we will concentrate on MonetDB feature extensions for the ExaNeSt architecture:

- Evaluate and help improving the enhanced *mmap()* implementation provided by FORTH. To give *mmap()* hints how certain pages are used, we will extend the most expensive relational operators, such as some relational projections, selections, joins, and group by, with *madvise()* kind of features (more information can be found in Deliverable D4.1, Section 7.6).
- Evaluate and help improving the *malloc()* library in a distributed setting. This calls for support from partners at the OS and/or VM levels.
- In MonetDB, the relational operator *sql.bind()* is responsible for binding the actual data of a relational column to a variable for further execution. Currently, this operator load data directly from disks to the MEM. In ExaNeSt, this operator can be extended to also make use of the NVM cache to improve query performance.
- Improve MonetDB client-to-server and MonetDB-to-MonetDB communication. Currently, a text-based row-wise protocol *MAPI* (MonetDB API) is used, which is far from optimal. An efficient communication protocol is particularly important in ExaNeSt. We have started studying alternative protocols and are working on replacing MAPI with a more efficient, binary-based protocol.
- Redesign the TPC-H and Air Traffic benchmarks to assist the MonetDB-on-ExaNeSt scale-out exploration. This will help us evaluate the ExaNeSt architecture in supporting state-of-the-art big data analytics applications.
    - Currently, both benchmarks only address scale-up, i.e. a single database instance. For both benchmarks, we will develop a distributed version. This requires us to design the testing architecture and scenarios.
    - We will design and evaluate different distributed MonetDB execution strategies to explore the ExaNeSt coherent islands. The challenge is finding optimal data placements, while taking into account UNIMEM within a daughter-board.
- We will evaluate VM accelerations provided by VOSYS. Particularly, we are interested in the live migration feature to see how much that can help fast (re)start of a whole database server. For big data analytical applications, this is useful not only to increase the resilience level of the database server, but also to promptly react to sudden changes in the workloads.
- We will design and develop a version 2 of the MonetDB profiler Marvin, so that we can use it to monitor distributed MonetDB instances. This includes evaluating Allinea profiling tool to see if and how it can be used in Marvin2; and extending Marvin visualisation for distributed MonetDB, which in turn requires us to redesign both the Marvin front-end GUI and the back-end information passing.

5. Summary

5.1 Summary of System-level requirements

- **Hardware:** no intrinsic dependency on any particular hardware has emerged. Many of the applications require vector accelerator, like GPUs or FPGAs: the exploitation of such devices is however one of the main goal of the project.

- **Operating System:** All the applications have been developed in *nix-like, POSIX compliant OS.
  Pthreads library comes up to be mandatory or a plus for many applications.

- **Compilers:** dependencies are on language standards rather than on particular compiler or interpreter.
  - Languages and relative standards required are:
    C99, C11, C++98 or higher, F90, python 2.7+ and 3.4+, Java 6+, Perl, Bison
  - Compliance with IEEE754 standard is required.
  - Vectorization abilities are used if available by some codes (SSE2/AVX).
  - OpenCL 1.2 is required by some codes
  - OpenMP 2.x and 3.x are required by some codes, possibly 4.x

- **Low-level Libraries:** pthreads, MPI v 1.2, 2.x, 3.1, atomic_ops(-devel), bzip2-devel, libcurl-devel, libuuid-devel, libxml2-devel, openssl-devel, pcre-devel (4.5+), perl, python-devel, readline-devel, system, unixODBC-devel, zlib-devel
  If no version is specified it is intended that current up-to-date version in reference linux distribution (for instance RedHat, CentOS, ...) are adopted.

- **High-level Libraries:**
  Strictly required
  - Blas3.6
  - Execnet 1.0.9
  - Gcal
  - GSL 2.2.x
  - HDF 5
  - FFTW 2.1.5, FFTW 3.x
  - Mako 0.9.0
  - METIS 5.1
  - NetCDF 4
  - NumPY 1.7.0
  - ParaMETIS 5.1
  - PYZMQ 14.0
  - SciPY 0.13
  - SCOTCH 6.0
  - SymPY 0.7.3

  Optional

   - Matplotlib
   - Netifaces 0.8
   - The MonetDB native Python client API pymonetdb
   - The Python web framework and network library Tornado
   - Recent browsers compatible with ES5 (ECMAScript 5)

  If no version is specified, it is intended that current up-to-date versions are suitable to be adopted.

5.2 Summary of Network and Storage requirements

Network requirements

The following facts emerge when considering all the network requirements stated in Section 2.

- Most of the messages exchanged among MPI tasks are collective and of small size (few bytes up to kBs).
- Message sizes range from few bytes up to several MBs. While the upper bound also depends on the combination of the number of MPI tasks and of the size of the physical problem simulated, since the problems used to profile the network imprint have been chosen carefully it seems that it might lies around O(10) MBs.
- Smaller packets are more frequently sent by collective communications, while the largest ones are typically sent by point-to-point routines.
- Applications uses a complex pattern of both collective and point-to-point operations. However, most of the communications (considered by number of calls to MPI routines) are collective, and most of them are all-to-all.
- Since most of the messages have small size, below O(10) kBs, the latency of the network is highly crucial and growing it from ~1.5 µs (characteristics of Infiniband connections) to ~30 µs (characteristics of TCP over Infiniband) can dilate the run time by 5-6 times.
  This fact strongly advocates for a highly customized version of MPI, with respect to both the transport layer over the platform's network and the full exploitation of zero-copy RDMA capabilities of UNIMEM.
- Network bandwidth turns out to be also of crucial importance, at least for some of the applications that either use collective communications for parallel I/O, or exchange a very large amount of data through the network (available estimates range up to some 10^3 GBs per second).

Storage requirements

The following facts emerge when considering all the storage requirements discussed in Section 3.

- Checkpointing may be a crucial activity for the file system. Many applications let each MPI process to write its own part of the checkpoint data dump, so that several thousands or tens of thousands files will be written simultaneously, occupying a total amount of few to hundreds of GBs.
  In the perspective of adopting a frequent checkpointing as a general policy to ensure resiliency, the file system performances are then a primary concern.

Also, the presence of NVM close to computing cores with high-bandwidth connections, coupled with a “off-line” transfer to storage, seems to be mandatory.

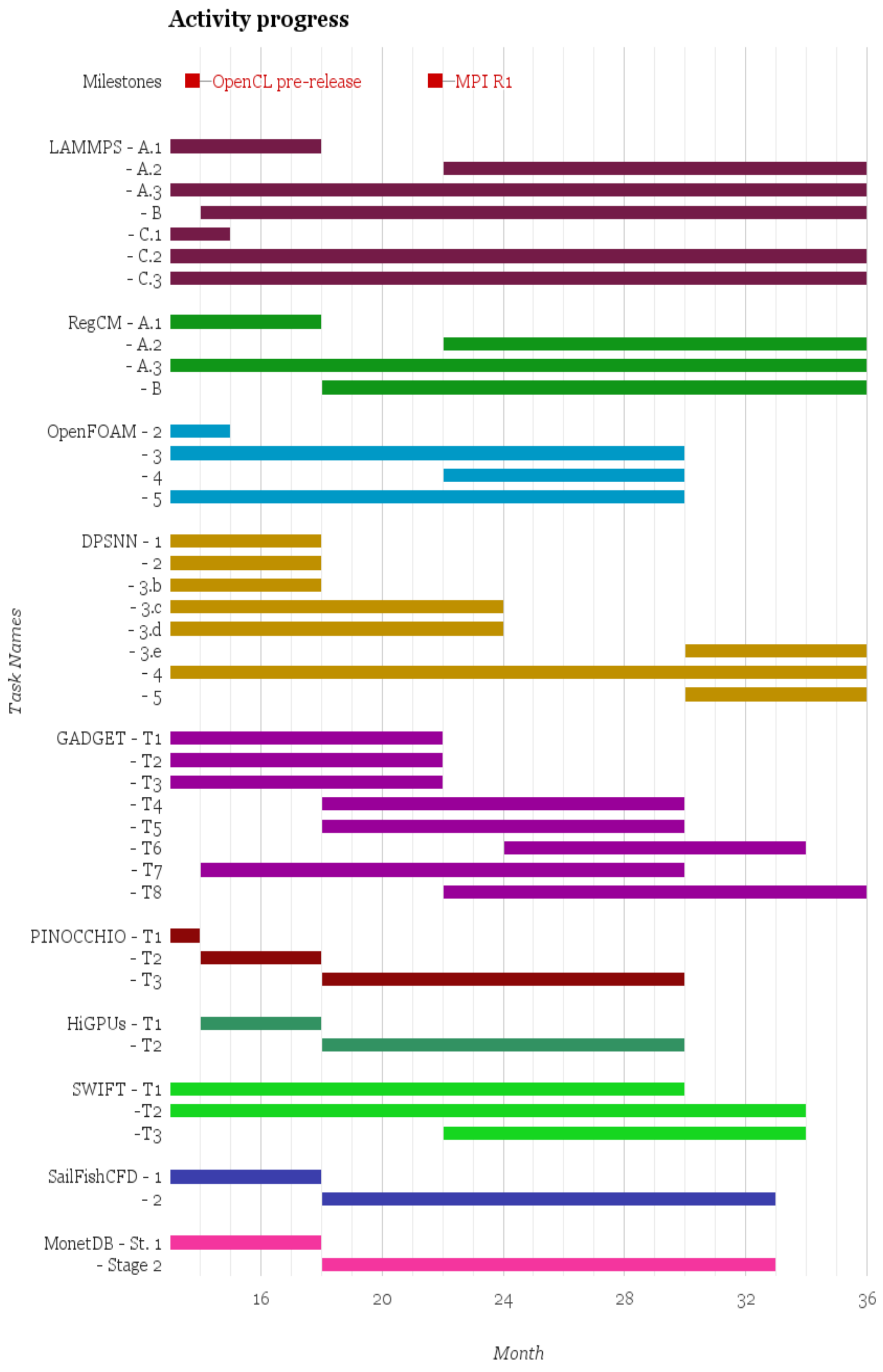
- The volume of single output files range from few MBs up to O(100) GBs, while the volume of the entire output of a single run can reach several tens of TBs.
- Outputs are saved either as ascii files or (preferably) as binary files that can have HDF, NetCDF or proprietary formats.
- The platform must then be capable to “absorb” a data flow of TB/s and the creations of (tens of) thousands files without becoming a bottleneck for the runs.

5.3 Comprehensive view of roadmaps

In this section we summarize in a GANTT view the roadmap described in this document for all the codes selected by WP2 partners for the co-design activity.

In the figure below we mark the two primary milestones of the project upon which some of the porting tasks depend, namely the release of OpenCL – scheduled for Month 13 –, and the release of a preliminary version of the MPI libraries – scheduled for Month 22.

The bars starts at the planned month of beginning and last as specified in paragraphs of Section 4.


Activity progress
Milestones
OpenCL pre-release
MPI R1
LAMMPS - A.1
- A.2
- A.3
- B
- C.1
- C.2
- C.3
RegCM - A.1
- A.2
- A.3
- B
OpenFOAM - 2
- 3
- 4
- 5
DPSNN - 1
- 2
- 3.b
- 3.c
- 3.d
- 3.e
- 4
- 5
GADGET - T1
- T2
- T3
- T4
- T5
- T6
- T7
- T8
PINOCCHIO - T1
- T2
- T3
HiGPUs - T1
- T2
SWIFT - T1
-T2
-T3
SailFishCFD - 1
- 2
MonetDB - St. 1
- Stage 2
Task Names
16
20
24
28
32
36
Month

6. References

In this section we report the public links for every code mentioned in this document.

DPSNN

https://arxiv.org/ftp/arxiv/papers/1310/1310.8478.pdf

https://arxiv.org/ftp/arxiv/papers/1512/1512.05264.pdf

HiGPUs

http://astrowww.phys.uniroma1.it/dolcetta/HPCcodes/HiGPUs.html

GADGET

http://wwwmpa.mpa-garching.mpg.de/gadget/

LAMMPS

http://lammps.sandia.gov

MonetDB

http://www.monetdb.org

OpenFOAM

http://www.openfoam.org

PINOCCHIO

http://adlibitum.oats.inaf.it/monaco/Homepage/Pinocchio/

RegCM

http://www.ictp.it/research/esp/models/regcm4.aspx

SailFISHCFD

http://sailfish.us.edu.pl

SWIFT

http://icc.dur.ac.uk/swift/